

The L^AT_EX3 Sources

The L^AT_EX3 Project*

Released 2020-08-07

Abstract

This is the reference documentation for the `expl3` programming environment. The `expl3` modules set up an experimental naming scheme for L^AT_EX commands, which allow the L^AT_EX programmer to systematically name functions and variables, and specify the argument types of functions.

The T_EX and ϵ -T_EX primitives are all given a new name according to these conventions. However, in the main direct use of the primitives is not required or encouraged: the `expl3` modules define an independent low-level L^AT_EX3 programming language.

At present, the `expl3` modules are designed to be loaded on top of L^AT_EX 2 ϵ . In time, a L^AT_EX3 format will be produced based on this code. This allows the code to be used in L^AT_EX 2 ϵ packages *now* while a stand-alone L^AT_EX3 is developed.

While `expl3` is still experimental, the bundle is now regarded as broadly stable. The syntax conventions and functions provided are now ready for wider use. There may still be changes to some functions, but these will be minor when compared to the scope of `expl3`.

New modules will be added to the distributed version of `expl3` as they reach maturity.

*E-mail: latex-team@latex-project.org

Contents

I	Introduction to <code>expl3</code> and this document	1
1	Naming functions and variables	1
1.1	Terminological inexactitude	3
2	Documentation conventions	4
3	Formal language conventions which apply generally	5
4	<code>TeX</code> concepts not supported by <code>LaTeX3</code>	6
II	The <code>l3bootstrap</code> package: Bootstrap code	7
1	Using the <code>LaTeX3</code> modules	7
III	The <code>l3names</code> package: Namespace for primitives	8
1	Setting up the <code>LaTeX3</code> programming language	8
IV	The <code>l3basics</code> package: Basic definitions	9
1	No operation functions	9
2	Grouping material	9
3	Control sequences and functions	10
3.1	Defining functions	10
3.2	Defining new functions using parameter text	11
3.3	Defining new functions using the signature	12
3.4	Copying control sequences	15
3.5	Deleting control sequences	15
3.6	Showing control sequences	15
3.7	Converting to and from control sequences	16
4	Analysing control sequences	17
5	Using or removing tokens and arguments	18
5.1	Selecting tokens from delimited arguments	20
6	Predicates and conditionals	21
6.1	Tests on control sequences	22
6.2	Primitive conditionals	22
7	Starting a paragraph	24
7.1	Debugging support	24

V	The <code>l3expan</code> package: Argument expansion	25
1	Defining new variants	25
2	Methods for defining variants	26
3	Introducing the variants	27
4	Manipulating the first argument	29
5	Manipulating two arguments	31
6	Manipulating three arguments	32
7	Unbraced expansion	33
8	Preventing expansion	33
9	Controlled expansion	35
10	Internal functions	37
VI	The <code>l3quark</code> package: Quarks	38
1	Quarks	38
2	Defining quarks	38
3	Quark tests	39
4	Recursion	39
5	An example of recursion with quarks	40
6	Scan marks	41
VII	The <code>l3tl</code> package: Token lists	43
1	Creating and initialising token list variables	43
2	Adding data to token list variables	44
3	Modifying token list variables	45
4	Reassigning token list category codes	45
5	Token list conditionals	46
6	Mapping to token lists	48
7	Using token lists	51

8	Working with the content of token lists	51
9	The first token from a token list	53
10	Using a single item	56
11	Viewing token lists	58
12	Constant token lists	58
13	Scratch token lists	59
VIII	The <code>l3str</code> package: Strings	60
1	Building strings	60
2	Adding data to string variables	61
3	Modifying string variables	62
4	String conditionals	63
5	Mapping to strings	64
6	Working with the content of strings	66
7	String manipulation	69
8	Viewing strings	70
9	Constant token lists	71
10	Scratch strings	71
IX	The <code>l3str-convert</code> package: string encoding conversions	72
1	Encoding and escaping schemes	72
2	Conversion functions	72
3	Conversion by expansion (for PDF contexts)	74
4	Creating 8-bit mappings	74
5	Possibilities, and things to do	74
X	The <code>l3seq</code> package: Sequences and stacks	76
1	Creating and initialising sequences	76
2	Appending data to sequences	77

3	Recovering items from sequences	77
4	Recovering values from sequences with branching	79
5	Modifying sequences	80
6	Sequence conditionals	81
7	Mapping to sequences	81
8	Using the content of sequences directly	84
9	Sequences as stacks	85
10	Sequences as sets	86
11	Constant and scratch sequences	87
12	Viewing sequences	88
XI	The <code>l3int</code> package: Integers	89
1	Integer expressions	90
2	Creating and initialising integers	91
3	Setting and incrementing integers	92
4	Using integers	93
5	Integer expression conditionals	93
6	Integer expression loops	95
7	Integer step functions	97
8	Formatting integers	98
9	Converting from other formats to integers	99
10	Random integers	100
11	Viewing integers	101
12	Constant integers	101
13	Scratch integers	101
	13.1 Direct number expansion	102
14	Primitive conditionals	102
XII	The <code>l3flag</code> package: Expandable flags	104

1	Setting up flags	104
2	Expandable flag commands	105
XIII	The <code>l3prg</code> package: Control structures	106
1	Defining a set of conditional functions	106
2	The boolean data type	108
3	Boolean expressions	110
4	Logical loops	112
5	Producing multiple copies	113
6	Detecting \TeX 's mode	113
7	Primitive conditionals	114
8	Nestable recursions and mappings	114
	8.1 Simple mappings	114
9	Internal programming functions	115
XIV	The <code>l3sys</code> package: System/runtime functions	116
1	The name of the job	116
2	Date and time	116
3	Engine	116
4	Output format	117
5	Platform	117
6	Random numbers	117
7	Access to the shell	118
	7.1 Loading configuration data	119
	7.2 Final settings	119
XV	The <code>l3clist</code> package: Comma separated lists	120
1	Creating and initialising comma lists	120
2	Adding data to comma lists	122
3	Modifying comma lists	122

4	Comma list conditionals	124
5	Mapping to comma lists	124
6	Using the content of comma lists directly	126
7	Comma lists as stacks	127
8	Using a single item	128
9	Viewing comma lists	128
10	Constant and scratch comma lists	129
XVI	The <code>l3token</code> package: Token manipulation	130
1	Creating character tokens	130
2	Manipulating and interrogating character tokens	132
3	Generic tokens	135
4	Converting tokens	135
5	Token conditionals	136
6	Peeking ahead at the next token	139
7	Description of all possible tokens	142
XVII	The <code>l3prop</code> package: Property lists	145
1	Creating and initialising property lists	145
2	Adding entries to property lists	146
3	Recovering values from property lists	146
4	Modifying property lists	147
5	Property list conditionals	147
6	Recovering values from property lists with branching	148
7	Mapping to property lists	149
8	Viewing property lists	150
9	Scratch property lists	150
10	Constants	151

XVIII	The <code>l3msg</code> package: Messages	152
1	Creating new messages	152
2	Contextual information for messages	153
3	Issuing messages	154
3.1	Expandable error messages	156
4	Redirecting messages	157
XIX	The <code>l3file</code> package: File and I/O operations	159
1	Input–output stream management	159
1.1	Reading from files	160
1.2	Writing to files	163
1.3	Wrapping lines in output	165
1.4	Constant input–output streams, and variables	166
1.5	Primitive conditionals	166
2	File operation functions	166
XX	The <code>l3skip</code> package: Dimensions and skips	171
1	Creating and initialising <code>dim</code> variables	171
2	Setting <code>dim</code> variables	172
3	Utilities for dimension calculations	172
4	Dimension expression conditionals	173
5	Dimension expression loops	175
6	Dimension step functions	176
7	Using <code>dim</code> expressions and variables	177
8	Viewing <code>dim</code> variables	178
9	Constant dimensions	179
10	Scratch dimensions	179
11	Creating and initialising <code>skip</code> variables	179
12	Setting <code>skip</code> variables	180
13	Skip expression conditionals	181
14	Using <code>skip</code> expressions and variables	181

15	Viewing skip variables	181
16	Constant skips	182
17	Scratch skips	182
18	Inserting skips into the output	182
19	Creating and initialising muskip variables	183
20	Setting muskip variables	183
21	Using muskip expressions and variables	184
22	Viewing muskip variables	184
23	Constant muskips	185
24	Scratch muskips	185
25	Primitive conditional	185
XXI	The l3keys package: Key–value interfaces	186
1	Creating keys	187
2	Sub-dividing keys	191
3	Choice and multiple choice keys	192
4	Setting keys	194
5	Handling of unknown keys	194
6	Selective key setting	195
7	Utility functions for keys	196
8	Low-level interface for parsing key–val lists	197
XXII	The l3intarray package: fast global integer arrays	199
1	l3intarray documentation	199
1.1	Implementation notes	200
XXIII	The l3fp package: Floating points	201
1	Creating and initialising floating point variables	202
2	Setting floating point variables	203

3	Using floating points	203
4	Floating point conditionals	205
5	Floating point expression loops	206
6	Some useful constants, and scratch variables	208
7	Floating point exceptions	209
8	Viewing floating points	210
9	Floating point expressions	211
9.1	Input of floating point numbers	211
9.2	Precedence of operators	212
9.3	Operations	212
10	Disclaimer and roadmap	219
 XXIV The <code>l3fparray</code> package: fast global floating point arrays		222
1	<code>l3fparray</code> documentation	222
 XXV The <code>l3cctab</code> package: Category code tables		223
1	Creating and initialising category code tables	223
2	Using category code tables	223
3	Category code table conditionals	224
4	Constant category code tables	224
 XXVI The <code>l3sort</code> package: Sorting functions		225
1	Controlling sorting	225
 XXVII The <code>l3tl-analysis</code> package: Analysing token lists		226
1	<code>l3tl-analysis</code> documentation	226
 XXVIII The <code>l3regex</code> package: Regular expressions in \TeX		227
1	Syntax of regular expressions	227
2	Syntax of the replacement text	232
3	Pre-compiling regular expressions	234

4	Matching	234
5	Submatch extraction	235
6	Replacement	236
7	Constants and variables	236
8	Bugs, misfeatures, future work, and other possibilities	237
 XXIX The l3box package: Boxes		240
1	Creating and initialising boxes	240
2	Using boxes	240
3	Measuring and setting box dimensions	241
4	Box conditionals	242
5	The last box inserted	242
6	Constant boxes	242
7	Scratch boxes	243
8	Viewing box contents	243
9	Boxes and color	243
10	Horizontal mode boxes	243
11	Vertical mode boxes	245
12	Using boxes efficiently	246
13	Affine transformations	247
14	Primitive box conditionals	250
 XXX The l3coffins package: Coffin code layer		251
1	Creating and initialising coffins	251
2	Setting coffin content and poles	251
3	Coffin affine transformations	253
4	Joining and using coffins	253
5	Measuring coffins	254

6	Coffin diagnostics	254
7	Constants and variables	255
XXXI	The l3color-base package: Color support	256
1	Color in boxes	256
XXXII	The l3luatex package: Lua _{TEX} -specific functions	257
1	Breaking out to Lua	257
2	Lua interfaces	258
XXXIII	The l3unicode package: Unicode support functions	259
XXXIV	The l3text package: text processing	260
1	l3text documentation	260
1.1	Expanding text	260
1.2	Case changing	262
1.3	Removing formatting from text	263
1.4	Control variables	263
XXXV	The l3legacy package: Interfaces to legacy concepts	264
XXXVI	The l3candidates package: Experimental additions to l3kernel	265
1	Important notice	265
2	Additions to l3box	265
2.1	Viewing part of a box	265
3	Additions to l3expan	266
4	Additions to l3fp	266
5	Additions to l3file	266
6	Additions to l3flag	267
7	Additions to l3intarray	267
7.1	Working with contents of integer arrays	267
8	Additions to l3msg	268

9	Additions to <code>l3prg</code>	268
10	Additions to <code>l3prop</code>	269
11	Additions to <code>l3seq</code>	270
12	Additions to <code>l3sys</code>	271
13	Additions to <code>l3tl</code>	272
14	Additions to <code>l3token</code>	273
XXXVII	Implementation	274
1	<code>l3bootstrap</code> implementation	274
1.1	LuaTeX-specific code	274
1.2	The <code>\pdfstrcmp</code> primitive in XeTeX	275
1.3	Loading support Lua code	275
1.4	Engine requirements	276
1.5	Extending allocators	277
1.6	The LaTeX3 code environment	277
2	<code>l3names</code> implementation	279
3	Internal kernel functions	303
4	Kernel backend functions	309
5	<code>l3basics</code> implementation	310
5.1	Renaming some TeX primitives (again)	310
5.2	Defining some constants	312
5.3	Defining functions	313
5.4	Selecting tokens	313
5.5	Gobbling tokens from input	315
5.6	Debugging and patching later definitions	315
5.7	Conditional processing and definitions	316
5.8	Dissecting a control sequence	322
5.9	Exist or free	324
5.10	Preliminaries for new functions	326
5.11	Defining new functions	327
5.12	Copying definitions	329
5.13	Undefining functions	329
5.14	Generating parameter text from argument count	330
5.15	Defining functions from a given number of arguments	331
5.16	Using the signature to define functions	332
5.17	Checking control sequence equality	334
5.18	Diagnostic functions	334
5.19	Decomposing a macro definition	336
5.20	Doing nothing functions	336
5.21	Breaking out of mapping functions	337

5.22	Starting a paragraph	337
6	l3expan implementation	338
6.1	General expansion	338
6.2	Hand-tuned definitions	341
6.3	Last-unbraced versions	345
6.4	Preventing expansion	347
6.5	Controlled expansion	348
6.6	Emulating e-type expansion	348
6.7	Defining function variants	355
6.8	Definitions with the automated technique	366
7	l3quark implementation	367
7.1	Quarks	367
7.2	Scan marks	376
8	l3tl implementation	377
8.1	Functions	377
8.2	Constant token lists	378
8.3	Adding to token list variables	379
8.4	Internal quarks and quark-query functions	380
8.5	Reassigning token list category codes	381
8.6	Modifying token list variables	384
8.7	Token list conditionals	387
8.8	Mapping to token lists	393
8.9	Using token lists	394
8.10	Working with the contents of token lists	395
8.11	Token by token changes	398
8.12	The first token from a token list	400
8.13	Using a single item	404
8.14	Viewing token lists	407
8.15	Internal scan marks	408
8.16	Scratch token lists	408
9	l3str implementation	409
9.1	Internal auxiliaries	409
9.2	Creating and setting string variables	410
9.3	Modifying string variables	411
9.4	String comparisons	412
9.5	Mapping to strings	415
9.6	Accessing specific characters in a string	417
9.7	Counting characters	421
9.8	The first character in a string	423
9.9	String manipulation	424
9.10	Viewing strings	425

10	l3str-convert implementation	426
10.1	Helpers	426
10.1.1	Variables and constants	426
10.2	String conditionals	427
10.3	Conversions	429
10.3.1	Producing one byte or character	429
10.3.2	Mapping functions for conversions	430
10.3.3	Error-reporting during conversion	431
10.3.4	Framework for conversions	431
10.3.5	Byte unescape and escape	436
10.3.6	Native strings	437
10.3.7	clist	438
10.3.8	8-bit encodings	438
10.4	Messages	441
10.5	Escaping definitions	442
10.5.1	Unescape methods	442
10.5.2	Escape methods	447
10.6	Encoding definitions	449
10.6.1	UTF-8 support	449
10.6.2	UTF-16 support	454
10.6.3	UTF-32 support	459
10.7	PDF names and strings by expansion	462
10.7.1	ISO 8859 support	463
11	l3seq implementation	478
11.1	Allocation and initialisation	480
11.2	Appending data to either end	483
11.3	Modifying sequences	483
11.4	Sequence conditionals	486
11.5	Recovering data from sequences	488
11.6	Mapping to sequences	491
11.7	Using sequences	495
11.8	Sequence stacks	496
11.9	Viewing sequences	497
11.10	Scratch sequences	498
12	l3int implementation	498
12.1	Integer expressions	499
12.2	Creating and initialising integers	501
12.3	Setting and incrementing integers	503
12.4	Using integers	504
12.5	Integer expression conditionals	504
12.6	Integer expression loops	508
12.7	Integer step functions	509
12.8	Formatting integers	511
12.9	Converting from other formats to integers	516
12.10	Viewing integer	519
12.11	Random integers	519
12.12	Constant integers	520
12.13	Scratch integers	520

12.14	Integers for earlier modules	520
13	l3flag implementation	521
13.1	Non-expandable flag commands	521
13.2	Expandable flag commands	522
14	l3prg implementation	523
14.1	Primitive conditionals	523
14.2	Defining a set of conditional functions	523
14.3	The boolean data type	523
14.4	Internal auxiliaries	524
14.5	Boolean expressions	526
14.6	Logical loops	530
14.7	Producing multiple copies	531
14.8	Detecting T _E X's mode	533
14.9	Internal programming functions	534
15	l3sys implementation	534
15.1	Kernel code	534
15.1.1	Detecting the engine	534
15.1.2	Randomness	535
15.1.3	Platform	535
15.1.4	Configurations	536
15.1.5	Access to the shell	537
15.2	Dynamic (every job) code	539
15.2.1	The name of the job	539
15.2.2	Time and date	540
15.2.3	Random numbers	540
15.2.4	Access to the shell	541
15.2.5	Held over from l3file	542
15.3	Last-minute code	542
15.3.1	Detecting the output	542
15.3.2	Configurations	543
16	l3clist implementation	544
16.1	Removing spaces around items	545
16.2	Allocation and initialisation	546
16.3	Adding data to comma lists	548
16.4	Comma lists as stacks	549
16.5	Modifying comma lists	551
16.6	Comma list conditionals	554
16.7	Mapping to comma lists	555
16.8	Using comma lists	558
16.9	Using a single item	559
16.10	Viewing comma lists	561
16.11	Scratch comma lists	561

17	l3token implementation	562
17.1	Internal auxiliaries	562
17.2	Manipulating and interrogating character tokens	562
17.3	Creating character tokens	564
17.4	Generic tokens	573
17.5	Token conditionals	574
17.6	Peeking ahead at the next token	581
18	l3prop implementation	587
18.1	Internal auxiliaries	588
18.2	Allocation and initialisation	589
18.3	Accessing data in property lists	591
18.4	Property list conditionals	595
18.5	Recovering values from property lists with branching	596
18.6	Mapping to property lists	597
18.7	Viewing property lists	598
19	l3msg implementation	599
19.1	Internal auxiliaries	599
19.2	Creating messages	599
19.3	Messages: support functions and text	601
19.4	Showing messages: low level mechanism	601
19.5	Displaying messages	604
19.6	Kernel-specific functions	613
19.7	Expandable errors	620
20	l3file implementation	623
20.1	Input operations	623
20.1.1	Variables and constants	623
20.1.2	Stream management	624
20.1.3	Reading input	626
20.2	Output operations	629
20.2.1	Variables and constants	629
20.2.2	Internal auxiliaries	630
20.3	Stream management	630
20.3.1	Deferred writing	632
20.3.2	Immediate writing	632
20.3.3	Special characters for writing	633
20.3.4	Hard-wrapping lines to a character count	633
20.4	File operations	643
20.4.1	Internal auxiliaries	644
20.5	GetIfInfo	661
20.6	Messages	662
20.7	Functions delayed from earlier modules	663

21	l3skip implementation	664
21.1	Length primitives renamed	664
21.2	Internal auxiliaries	664
21.3	Creating and initialising <code>dim</code> variables	664
21.4	Setting <code>dim</code> variables	665
21.5	Utilities for dimension calculations	666
21.6	Dimension expression conditionals	667
21.7	Dimension expression loops	669
21.8	Dimension step functions	670
21.9	Using <code>dim</code> expressions and variables	671
21.10	Viewing <code>dim</code> variables	673
21.11	Constant dimensions	674
21.12	Scratch dimensions	674
21.13	Creating and initialising <code>skip</code> variables	674
21.14	Setting <code>skip</code> variables	675
21.15	Skip expression conditionals	676
21.16	Using <code>skip</code> expressions and variables	676
21.17	Inserting skips into the output	677
21.18	Viewing <code>skip</code> variables	677
21.19	Constant skips	677
21.20	Scratch skips	677
21.21	Creating and initialising <code>muskip</code> variables	678
21.22	Setting <code>muskip</code> variables	679
21.23	Using <code>muskip</code> expressions and variables	679
21.24	Viewing <code>muskip</code> variables	680
21.25	Constant muskips	680
21.26	Scratch muskips	680
22	l3keys Implementation	680
22.1	Low-level interface	680
22.2	Constants and variables	686
22.2.1	Internal auxiliaries	689
22.3	The key defining mechanism	689
22.4	Turning properties into actions	691
22.5	Creating key properties	697
22.6	Setting keys	702
22.7	Utilities	710
22.8	Messages	713
23	l3intarray implementation	713
23.1	Allocating arrays	714
23.2	Array items	715
23.3	Working with contents of integer arrays	717
23.4	Random arrays	718
24	l3fp implementation	720

25	l3fp-aux implementation	720
25.1	Access to primitives	720
25.2	Internal representation	720
25.3	Using arguments and semicolons	722
25.4	Constants, and structure of floating points	722
25.5	Overflow, underflow, and exact zero	725
25.6	Expanding after a floating point number	725
25.7	Other floating point types	726
25.8	Packing digits	729
25.9	Decimate (dividing by a power of 10)	732
25.10	Functions for use within primitive conditional branches	734
25.11	Integer floating points	735
25.12	Small integer floating points	736
25.13	Fast string comparison	737
25.14	Name of a function from its l3fp-parse name	737
25.15	Messages	737
26	l3fp-traps Implementation	738
26.1	Flags	738
26.2	Traps	738
26.3	Errors	742
26.4	Messages	742
27	l3fp-round implementation	743
27.1	Rounding tools	743
27.2	The round function	747
28	l3fp-parse implementation	750
28.1	Work plan	751
28.1.1	Storing results	752
28.1.2	Precedence and infix operators	753
28.1.3	Prefix operators, parentheses, and functions	756
28.1.4	Numbers and reading tokens one by one	757
28.2	Main auxiliary functions	758
28.3	Helpers	759
28.4	Parsing one number	760
28.4.1	Numbers: trimming leading zeros	766
28.4.2	Number: small significand	768
28.4.3	Number: large significand	770
28.4.4	Number: beyond 16 digits, rounding	772
28.4.5	Number: finding the exponent	774
28.5	Constants, functions and prefix operators	777
28.5.1	Prefix operators	777
28.5.2	Constants	781
28.5.3	Functions	782
28.6	Main functions	782
28.7	Infix operators	784
28.7.1	Closing parentheses and commas	786
28.7.2	Usual infix operators	788
28.7.3	Juxtaposition	789

	28.7.4 Multi-character cases	789
	28.7.5 Ternary operator	790
	28.7.6 Comparisons	790
	28.8 Tools for functions	792
	28.9 Messages	794
29	l3fp-assign implementation	795
	29.1 Assigning values	795
	29.2 Updating values	796
	29.3 Showing values	797
	29.4 Some useful constants and scratch variables	797
30	l3fp-logic Implementation	798
	30.1 Syntax of internal functions	798
	30.2 Tests	798
	30.3 Comparison	799
	30.4 Floating point expression loops	802
	30.5 Extrema	805
	30.6 Boolean operations	807
	30.7 Ternary operator	808
31	l3fp-basics Implementation	809
	31.1 Addition and subtraction	809
	31.1.1 Sign, exponent, and special numbers	810
	31.1.2 Absolute addition	812
	31.1.3 Absolute subtraction	814
	31.2 Multiplication	818
	31.2.1 Signs, and special numbers	818
	31.2.2 Absolute multiplication	819
	31.3 Division	822
	31.3.1 Signs, and special numbers	822
	31.3.2 Work plan	823
	31.3.3 Implementing the significand division	825
	31.4 Square root	830
	31.5 About the sign and exponent	837
	31.6 Operations on tuples	838
32	l3fp-extended implementation	839
	32.1 Description of fixed point numbers	840
	32.2 Helpers for numbers with extended precision	840
	32.3 Multiplying a fixed point number by a short one	841
	32.4 Dividing a fixed point number by a small integer	842
	32.5 Adding and subtracting fixed points	843
	32.6 Multiplying fixed points	844
	32.7 Combining product and sum of fixed points	845
	32.8 Extended-precision floating point numbers	847
	32.9 Dividing extended-precision numbers	850
	32.10 Inverse square root of extended precision numbers	853
	32.11 Converting from fixed point to floating point	855

33	l3fp-expo implementation	857
33.1	Logarithm	857
33.1.1	Work plan	857
33.1.2	Some constants	858
33.1.3	Sign, exponent, and special numbers	858
33.1.4	Absolute ln	858
33.2	Exponential	866
33.2.1	Sign, exponent, and special numbers	866
33.3	Power	870
33.4	Factorial	876
34	l3fp-trig Implementation	878
34.1	Direct trigonometric functions	879
34.1.1	Filtering special cases	879
34.1.2	Distinguishing small and large arguments	882
34.1.3	Small arguments	883
34.1.4	Argument reduction in degrees	883
34.1.5	Argument reduction in radians	885
34.1.6	Computing the power series	892
34.2	Inverse trigonometric functions	895
34.2.1	Arctangent and arccotangent	896
34.2.2	Arcsine and arccosine	901
34.2.3	Arccosecant and arcsecant	903
35	l3fp-convert implementation	904
35.1	Dealing with tuples	904
35.2	Trimming trailing zeros	905
35.3	Scientific notation	905
35.4	Decimal representation	906
35.5	Token list representation	908
35.6	Formatting	909
35.7	Convert to dimension or integer	910
35.8	Convert from a dimension	910
35.9	Use and eval	911
35.10	Convert an array of floating points to a comma list	912
36	l3fp-random Implementation	913
36.1	Engine support	913
36.2	Random floating point	917
36.3	Random integer	917
37	l3fparray implementation	922
37.1	Allocating arrays	922
37.2	Array items	923

38	l3cctab implementation	926
38.1	Variables	926
38.2	Allocating category code tables	927
38.3	Saving category code tables	928
38.4	Using category code tables	929
38.5	Category code table conditionals	934
38.6	Constant category code tables	935
38.7	Messages	937
39	l3sort implementation	938
39.1	Variables	938
39.2	Finding available \toks registers	939
39.3	Protected user commands	941
39.4	Merge sort	942
39.5	Expandable sorting	946
39.6	Messages	951
40	l3tl-analysis implementation	952
40.1	Internal functions	952
40.2	Internal format	952
40.3	Variables and helper functions	953
40.4	Plan of attack	955
40.5	Disabling active characters	956
40.6	First pass	956
40.7	Second pass	961
40.8	Mapping through the analysis	964
40.9	Showing the results	965
40.10	Messages	967
41	l3regex implementation	968
41.1	Plan of attack	968
41.2	Helpers	969
41.2.1	Constants and variables	971
41.2.2	Testing characters	972
41.2.3	Internal auxiliaries	972
41.2.4	Character property tests	976
41.2.5	Simple character escape	977
41.3	Compiling	983
41.3.1	Variables used when compiling	983
41.3.2	Generic helpers used when compiling	985
41.3.3	Mode	986
41.3.4	Framework	988
41.3.5	Quantifiers	991
41.3.6	Raw characters	994
41.3.7	Character properties	995
41.3.8	Anchoring and simple assertions	996
41.3.9	Character classes	997
41.3.10	Groups and alternations	1000
41.3.11	Catcodes and csnames	1003
41.3.12	Raw token lists with \u	1007

41.3.13 Other	1009
41.3.14 Showing regexes	1009
41.4 Building	1013
41.4.1 Variables used while building	1013
41.4.2 Framework	1014
41.4.3 Helpers for building an NFA	1015
41.4.4 Building classes	1017
41.4.5 Building groups	1018
41.4.6 Others	1023
41.5 Matching	1024
41.5.1 Variables used when matching	1025
41.5.2 Matching: framework	1027
41.5.3 Using states of the NFA	1031
41.5.4 Actions when matching	1031
41.6 Replacement	1034
41.6.1 Variables and helpers used in replacement	1034
41.6.2 Query and brace balance	1035
41.6.3 Framework	1036
41.6.4 Submatches	1038
41.6.5 Csnames in replacement	1040
41.6.6 Characters in replacement	1041
41.6.7 An error	1045
41.7 User functions	1045
41.7.1 Variables and helpers for user functions	1047
41.7.2 Matching	1048
41.7.3 Extracting submatches	1048
41.7.4 Replacement	1052
41.7.5 Storing and showing compiled patterns	1054
41.8 Messages	1054
41.9 Code for tracing	1060
42 l3box implementation	1061
42.1 Support code	1061
42.2 Creating and initialising boxes	1061
42.3 Measuring and setting box dimensions	1062
42.4 Using boxes	1063
42.5 Box conditionals	1063
42.6 The last box inserted	1063
42.7 Constant boxes	1064
42.8 Scratch boxes	1064
42.9 Viewing box contents	1064
42.10 Horizontal mode boxes	1065
42.11 Vertical mode boxes	1067
42.12 Affine transformations	1070

43	l3coffins Implementation	1079
43.1	Coffins: data structures and general variables	1079
43.2	Basic coffin functions	1080
43.3	Measuring coffins	1086
43.4	Coffins: handle and pole management	1086
43.5	Coffins: calculation of pole intersections	1089
43.6	Affine transformations	1092
43.7	Aligning and typesetting of coffins	1100
43.8	Coffin diagnostics	1105
43.9	Messages	1111
44	l3color-base Implementation	1111
45	l3luatex implementation	1113
45.1	Breaking out to Lua	1113
45.2	Messages	1114
45.3	Lua functions for internal use	1114
46	l3unicode implementation	1118
47	l3text implementation	1122
47.1	Internal auxiliaries	1122
47.2	Utilities	1122
47.3	Configuration variables	1125
47.4	Expansion to formatted text	1126
48	l3text-case implementation	1133
48.1	Case changing	1133
48.2	Case changing data for 8-bit engines	1150
49	l3text-purify implementation	1157
49.1	Purifying text	1157
49.2	Accent and letter-like data for purifying text	1163
50	l3legacy Implementation	1169
51	l3candidates Implementation	1170
51.1	Additions to l3box	1170
51.1.1	Viewing part of a box	1170
51.2	Additions to l3flag	1172
51.3	Additions to l3msg	1173
51.4	Additions to l3prg	1174
51.5	Additions to l3prop	1175
51.6	Additions to l3seq	1175
51.7	Additions to l3sys	1177
51.8	Additions to l3file	1178
51.9	Additions to l3tl	1178
51.9.1	Building a token list	1178
51.9.2	Other additions to l3tl	1182
51.10	Additions to l3token	1182

52	l3deprecation implementation	1184
52.1	Helpers and variables	1184
52.2	Patching definitions to deprecate	1185
52.3	Removed functions	1188
52.4	Loading the patches	1192
52.5	Deprecated l3box functions	1193
52.6	Deprecated l3str functions	1193
52.7	Deprecated l3seq functions	1193
52.7.1	Deprecated l3tl functions	1194
52.8	Deprecated l3token functions	1195
52.9	Deprecated l3file functions	1195
	Index	1196

Part I

Introduction to `expl3` and this document

This document is intended to act as a comprehensive reference manual for the `expl3` language. A general guide to the L^AT_EX3 programming language is found in [expl3.pdf](#).

1 Naming functions and variables

L^AT_EX3 does not use `@` as a “letter” for defining internal macros. Instead, the symbols `_` and `:` are used in internal macro names to provide structure. The name of each *function* is divided into logical units using `_`, while `:` separates the *name* of the function from the *argument specifier* (“arg-spec”). This describes the arguments expected by the function. In most cases, each argument is represented by a single letter. The complete list of arg-spec letters for a function is referred to as the *signature* of the function.

Each function name starts with the *module* to which it belongs. Thus apart from a small number of very basic functions, all `expl3` function names contain at least one underscore to divide the module name from the descriptive name of the function. For example, all functions concerned with comma lists are in module `clist` and begin `\clist_`.

Every function must include an argument specifier. For functions which take no arguments, this will be blank and the function name will end `:`. Most functions take one or more arguments, and use the following argument specifiers:

- N and n** These mean *no manipulation*, of a single token for `N` and of a set of tokens given in braces for `n`. Both pass the argument through exactly as given. Usually, if you use a single token for an `n` argument, all will be well.
- c** This means *csname*, and indicates that the argument will be turned into a *csname* before being used. So `\foo:c {ArgumentOne}` will act in the same way as `\foo:N \ArgumentOne`.
- V and v** These mean *value of variable*. The `V` and `v` specifiers are used to get the content of a variable without needing to worry about the underlying T_EX structure containing the data. A `V` argument will be a single token (similar to `N`), for example `\foo:V \MyVariable`; on the other hand, using `v` a *csname* is constructed first, and then the value is recovered, for example `\foo:v {MyVariable}`.
- o** This means *expansion once*. In general, the `V` and `v` specifiers are favoured over `o` for recovering stored information. However, `o` is useful for correctly processing information with delimited arguments.
- x** The `x` specifier stands for *exhaustive expansion*: every token in the argument is fully expanded until only unexpandable ones remain. The T_EX `\edef` primitive carries out this type of expansion. Functions which feature an `x`-type argument are *not* expandable.
- e** The `e` specifier is in many respects identical to `x`, but with a very different implementation. Functions which feature an `e`-type argument may be expandable. The drawback is that `e` is extremely slow (often more than 200 times slower) in older engines, more precisely in non-LuaT_EX engines older than 2019.

- f** The **f** specifier stands for *full expansion*, and in contrast to **x** stops at the first non-expandable token (reading the argument from left to right) without trying to expand it. If this token is a *space token*, it is gobbled, and thus won't be part of the resulting argument. For example, when setting a token list variable (a macro used for storage), the sequence

```
\tl_set:Nn \l_my_a_tl { A }
\tl_set:Nn \l_my_b_tl { B }
\tl_set:Nf \l_my_a_tl { \l_my_a_tl \l_my_b_tl }
```

will leave `\l_my_a_tl` with the content `A\l_my_b_tl`, as `A` cannot be expanded and so terminates expansion before `\l_my_b_tl` is considered.

- T and F** For logic tests, there are the branch specifiers **T** (*true*) and **F** (*false*). Both specifiers treat the input in the same way as **n** (no change), but make the logic much easier to see.
- p** The letter **p** indicates *TeX parameters*. Normally this will be used for delimited functions as `expl3` provides better methods for creating simple sequential arguments.
- w** Finally, there is the **w** specifier for *weird* arguments. This covers everything else, but mainly applies to delimited values (where the argument must be terminated by some specified string).
- D** The **D** specifier means *do not use*. All of the *TeX* primitives are initially `\let` to a **D** name, and some are then given a second name. Only the kernel team should use anything with a **D** specifier!

Notice that the argument specifier describes how the argument is processed prior to being passed to the underlying function. For example, `\foo:c` will take its argument, convert it to a control sequence and pass it to `\foo:N`.

Variables are named in a similar manner to functions, but begin with a single letter to define the type of variable:

- c** Constant: global parameters whose value should not be changed.
- g** Parameters whose value should only be set globally.
- l** Parameters whose value should only be set locally.

Each variable name is then build up in a similar way to that of a function, typically starting with the module¹ name and then a descriptive part. Variables end with a short identifier to show the variable type:

clist Comma separated list.

dim "Rigid" lengths.

fp Floating-point values;

int Integer-valued count register.

¹The module names are not used in case of generic scratch registers defined in the data type modules, e.g., the **int** module contains some scratch variables called `\l_tmpa_int`, `\l_tmpb_int`, and so on. In such a case adding the module name up front to denote the module and in the back to indicate the type, as in `\l_int_tmpa_int` would be very unreadable.

muskip “Rubber” lengths for use in mathematics.

seq “Sequence”: a data-type used to implement lists (with access at both ends) and stacks.

skip “Rubber” lengths.

str String variables: contain character data.

tl Token list variables: placeholder for a token list.

Applying V-type or v-type expansion to variables of one of the above types is supported, while it is not supported for the following variable types:

bool Either true or false.

box Box register.

coffin A “box with handles” — a higher-level data type for carrying out **box** alignment operations.

flag Integer that can be incremented expandably.

farray Fixed-size array of floating point values.

intarray Fixed-size array of integers.

ior/iow An input or output stream, for reading from or writing to, respectively.

prop Property list: analogue of dictionary or associative arrays in other languages.

regex Regular expression.

1.1 Terminological inexactitude

A word of warning. In this document, and others referring to the `expl3` programming modules, we often refer to “variables” and “functions” as if they were actual constructs from a real programming language. In truth, `TeX` is a macro processor, and functions are simply macros that may or may not take arguments and expand to their replacement text. Many of the common variables are *also* macros, and if placed into the input stream will simply expand to their definition as well — a “function” with no arguments and a “token list variable” are almost the same.² On the other hand, some “variables” are actually registers that must be initialised and their values set and retrieved with specific functions.

The conventions of the `expl3` code are designed to clearly separate the ideas of “macros that contain data” and “macros that contain code”, and a consistent wrapper is applied to all forms of “data” whether they be macros or actually registers. This means that sometimes we will use phrases like “the function returns a value”, when actually we just mean “the macro expands to something”. Similarly, the term “execute” might be used in place of “expand” or it might refer to the more specific case of “processing in `TeX`’s stomach” (if you are familiar with the `TeXbook` parlance).

If in doubt, please ask; chances are we’ve been hasty in writing certain definitions and need to be told to tighten up our terminology.

²`TeX`nically, functions with no arguments are `\long` while token list variables are not.

2 Documentation conventions

This document is typeset with the experimental `l3doc` class; several conventions are used to help describe the features of the code. A number of conventions are used here to make the documentation clearer.

Each group of related functions is given in a box. For a function with a “user” name, this might read:

```
\ExplSyntaxOn
\ExplSyntaxOff
```

```
\ExplSyntaxOn ... \ExplSyntaxOff
```

The textual description of how the function works would appear here. The syntax of the function is shown in mono-spaced text to the right of the box. In this example, the function takes no arguments and so the name of the function is simply reprinted.

For programming functions, which use `_` and `:` in their name there are a few additional conventions: If two related functions are given with identical names but different argument specifiers, these are termed *variants* of each other, and the latter functions are printed in grey to show this more clearly. They will carry out the same function but will take different types of argument:

```
\seq_new:N
\seq_new:c
```

```
\seq_new:N <sequence>
```

When a number of variants are described, the arguments are usually illustrated only for the base function. Here, `<sequence>` indicates that `\seq_new:N` expects the name of a sequence. From the argument specifier, `\seq_new:c` also expects a sequence name, but as a name rather than as a control sequence. Each argument given in the illustration should be described in the following text.

Fully expandable functions Some functions are fully expandable, which allows them to be used within an `x`-type or `e`-type argument (in plain `TeX` terms, inside an `\edef` or `\expanded`), as well as within an `f`-type argument. These fully expandable functions are indicated in the documentation by a star:

```
\cs_to_str:N ☆
```

```
\cs_to_str:N <cs>
```

As with other functions, some text should follow which explains how the function works. Usually, only the star will indicate that the function is expandable. In this case, the function expects a `<cs>`, shorthand for a `<control sequence>`.

Restricted expandable functions A few functions are fully expandable but cannot be fully expanded within an `f`-type argument. In this case a hollow star is used to indicate this:

```
\seq_map_function:NN ☆
```

```
\seq_map_function:NN <seq> <function>
```

Conditional functions Conditional (`if`) functions are normally defined in three variants, with `T`, `F` and `TF` argument specifiers. This allows them to be used for different “true”/“false” branches, depending on which outcome the conditional is being used to test. To indicate this without repetition, this information is given in a shortened form:

<code>\sys_if_engine_xetex:<i><u>TF</u></i> *</code>	<code>\sys_if_engine_xetex:TF {\langle true code \rangle} {\langle false code \rangle}</code>
--	---

The underlining and italic of TF indicates that three functions are available:

- `\sys_if_engine_xetex:T`
- `\sys_if_engine_xetex:F`
- `\sys_if_engine_xetex:TF`

Usually, the illustration will use the TF variant, and so both $\langle true code \rangle$ and $\langle false code \rangle$ will be shown. The two variant forms T and F take only $\langle true code \rangle$ and $\langle false code \rangle$, respectively. Here, the star also shows that this function is expandable. With some minor exceptions, *all* conditional functions in the `expl3` modules should be defined in this way.

Variables, constants and so on are described in a similar manner:

<code>\l_tmpa_tl</code>	
-------------------------	--

A short piece of text will describe the variable: there is no syntax illustration in this case.

In some cases, the function is similar to one in $\text{\LaTeX} 2_{\epsilon}$ or plain \TeX . In these cases, the text will include an extra “ **\TeX hackers note**” section:

<code>\token_to_str:N *</code>	<code>\token_to_str:N \langle token \rangle</code>
--------------------------------	--

The normal description text.

\TeX hackers note: Detail for the experienced \TeX or $\text{\LaTeX} 2_{\epsilon}$ programmer. In this case, it would point out that this function is the \TeX primitive `\string`.

Changes to behaviour When new functions are added to `expl3`, the date of first inclusion is given in the documentation. Where the documented behaviour of a function changes after it is first introduced, the date of the update will also be given. This means that the programmer can be sure that any release of `expl3` after the date given will contain the function of interest with expected behaviour as described. Note that changes to code internals, including bug fixes, are not recorded in this way *unless* they impact on the expected behaviour.

3 Formal language conventions which apply generally

As this is a formal reference guide for $\text{\LaTeX} 3$ programming, the descriptions of functions are intended to be reasonably “complete”. However, there is also a need to avoid repetition. Formal ideas which apply to general classes of function are therefore summarised here.

For tests which have a TF argument specification, the test is evaluated to give a logically TRUE or FALSE result. Depending on this result, either the $\langle true code \rangle$ or the $\langle false code \rangle$ will be left in the input stream. In the case where the test is expandable, and a predicate (`_p`) variant is available, the logical value determined by the test is left in the input stream: this will typically be part of a larger logical construct.

4 \TeX concepts not supported by $\text{\LaTeX}3$

The \TeX concept of an “`\outer`” macro is *not supported* at all by $\text{\LaTeX}3$. As such, the functions provided here may break when used on top of $\text{\LaTeX}2_{\epsilon}$ if `\outer` tokens are used in the arguments.

Part II

The l3bootstrap package

Bootstrap code

1 Using the L^AT_EX3 modules

The modules documented in `source3` are designed to be used on top of L^AT_EX 2_ε and are loaded all as one with the usual `\usepackage{expl3}` or `\RequirePackage{expl3}` instructions.

As the modules use a coding syntax different from standard L^AT_EX 2_ε it provides a few functions for setting it up.

`\ExplSyntaxOn`
`\ExplSyntaxOff`
 Updated: 2011-08-13

`\ExplSyntaxOn` *<code>* `\ExplSyntaxOff`

The `\ExplSyntaxOn` function switches to a category code régime in which spaces are ignored and in which the colon (:) and underscore (_) are treated as “letters”, thus allowing access to the names of code functions and variables. Within this environment, ~ is used to input a space. The `\ExplSyntaxOff` reverts to the document category code régime.

`\ProvidesExplPackage`
`\ProvidesExplClass`
`\ProvidesExplFile`
 Updated: 2017-03-19

`\RequirePackage{expl3}`
`\ProvidesExplPackage` {*<package>*} {*<date>*} {*<version>*} {*<description>*}

These functions act broadly in the same way as the corresponding L^AT_EX 2_ε kernel functions `\ProvidesPackage`, `\ProvidesClass` and `\ProvidesFile`. However, they also implicitly switch `\ExplSyntaxOn` for the remainder of the code with the file. At the end of the file, `\ExplSyntaxOff` will be called to reverse this. (This is the same concept as L^AT_EX 2_ε provides in turning on `\makeatletter` within package and class code.) The *<date>* should be given in the format *<year>/<month>/<day>*. If the *<version>* is given then it will be prefixed with v in the package identifier line.

`\GetIdInfo`
 Updated: 2012-06-04

`\RequirePackage{l3bootstrap}`
`\GetIdInfo` \$Id: *<SVN info field>* \$ {*<description>*}

Extracts all information from a SVN field. Spaces are not ignored in these fields. The information pieces are stored in separate control sequences with `\ExplFileName` for the part of the file name leading up to the period, `\ExplFileDate` for date, `\ExplFileVersion` for version and `\ExplFileDescription` for the description.

To summarize: Every single package using this syntax should identify itself using one of the above methods. Special care is taken so that every package or class file loaded with `\RequirePackage` or similar are loaded with usual L^AT_EX 2_ε category codes and the L^AT_EX3 category code scheme is reloaded when needed afterwards. See implementation for details. If you use the `\GetIdInfo` command you can use the information when loading a package with

```
\ProvidesExplPackage{\ExplFileName}
  {\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
```


Part III

The l3names package Namespace for primitives

1 Setting up the L^AT_EX3 programming language

This module is at the core of the L^AT_EX3 programming language. It performs the following tasks:

- defines new names for all T_EX primitives;
- switches to the category code régime for programming;

This module is entirely dedicated to primitives, which should not be used directly within L^AT_EX3 code (outside of “kernel-level” code). As such, the primitives are not documented here: *The T_EXbook*, *T_EX by Topic* and the manuals for pdfT_EX, X_YT_EX, LuaT_EX, pT_EX and upT_EX should be consulted for details of the primitives. These are named `\tex_<name>:D`, typically based on the primitive’s *<name>* in pdfT_EX and omitting a leading **pdf** when the primitive is not related to pdf output.

Part IV

The l3basics package

Basic definitions

As the name suggest this package holds some basic definitions which are needed by most or all other packages in this set.

Here we describe those functions that are used all over the place. With that we mean functions dealing with the construction and testing of control sequences. Furthermore the basic parts of conditional processing are covered; conditional processing dealing with specific data types is described in the modules specific for the respective data types.

1 No operation functions

`\prg_do_nothing: *`

`\prg_do_nothing:`

An expandable function which does nothing at all: leaves nothing in the input stream after a single expansion.

`\scan_stop:`

`\scan_stop:`

A non-expandable function which does nothing. Does not vanish on expansion but produces no typeset output.

2 Grouping material

`\group_begin:`

`\group_begin:`**`\group_end:`**

`\group_end:`

These functions begin and end a group for definition purposes. Assignments are local to groups unless carried out in a global manner. (A small number of exceptions to this rule will be noted as necessary elsewhere in this document.) Each `\group_begin:` must be matched by a `\group_end:`, although this does not have to occur within the same function. Indeed, it is often necessary to start a group within one function and finish it within another, for example when seeking to use non-standard category codes.

`\group_insert_after:N`

`\group_insert_after:N` $\langle token \rangle$

Adds $\langle token \rangle$ to the list of $\langle tokens \rangle$ to be inserted when the current group level ends. The list of $\langle tokens \rangle$ to be inserted is empty at the beginning of a group: multiple applications of `\group_insert_after:N` may be used to build the inserted list one $\langle token \rangle$ at a time. The current group level may be closed by a `\group_end:` function or by a token with category code 2 (close-group), namely a `}` if standard category codes apply.

3 Control sequences and functions

As \TeX is a macro language, creating new functions means creating macros. At point of use, a function is replaced by the replacement text (“code”) in which each parameter in the code ($\#1$, $\#2$, *etc.*) is replaced the appropriate arguments absorbed by the function. In the following, *code* is therefore used as a shorthand for “replacement text”.

Functions which are not “protected” are fully expanded inside an \mathbf{x} expansion. In contrast, “protected” functions are not expanded within \mathbf{x} expansions.

3.1 Defining functions

Functions can be created with no requirement that they are declared first (in contrast to variables, which must always be declared). Declaring a function before setting up the code means that the name chosen is checked and an error raised if it is already in use. The name of a function can be checked at the point of definition using the `\cs_new...` functions: this is recommended for all functions which are defined for the first time.

There are three ways to define new functions. All classes define a function to expand to the substitution text. Within the substitution text the actual parameters are substituted for the formal parameters ($\#1$, $\#2$, ...).

new Create a new function with the **new** scope, such as `\cs_new:Npn`. The definition is global and results in an error if it is already defined.

set Create a new function with the **set** scope, such as `\cs_set:Npn`. The definition is restricted to the current \TeX group and does not result in an error if the function is already defined.

gset Create a new function with the **gset** scope, such as `\cs_gset:Npn`. The definition is global and does not result in an error if the function is already defined.

Within each set of scope there are different ways to define a function. The differences depend on restrictions on the actual parameters and the expandability of the resulting function.

nopar Create a new function with the **nopar** restriction, such as `\cs_set_nopar:Npn`. The parameter may not contain `\par` tokens.

protected Create a new function with the **protected** restriction, such as `\cs_set_protected:Npn`. The parameter may contain `\par` tokens but the function will not expand within an \mathbf{x} -type or \mathbf{e} -type expansion.

Finally, the functions in Subsections 3.2 and 3.3 are primarily meant to define *base functions* only. Base functions can only have the following argument specifiers:

N and n No manipulation.

T and F Functionally equivalent to **n** (you are actually encouraged to use the family of `\prg_new_conditional:` functions described in Section 1).

p and w These are special cases.

The `\cs_new:` functions below (and friends) do not stop you from using other argument specifiers in your function names, but they do not handle expansion for you. You should define the base function and then use `\cs_generate_variant:Nn` to generate custom variants as described in Section 2.

3.2 Defining new functions using parameter text

<code>\cs_new:Npn</code>	<code>\cs_new:Npn <function> <parameters> {<code>}</code>
<code>\cs_new:cpn</code>	Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_new:Npx</code>	<code><parameters></code> (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_new:cpx</code>	definition is global and an error results if the <code><function></code> is already defined.

<code>\cs_new_nopar:Npn</code>	<code>\cs_new_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_new_nopar:cpn</code>	Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_new_nopar:Npx</code>	<code><parameters></code> (#1, #2, etc.) will be replaced by those absorbed by the function. When the
<code>\cs_new_nopar:cpx</code>	<code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The definition
	is global and an error results if the <code><function></code> is already defined.

<code>\cs_new_protected:Npn</code>	<code>\cs_new_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_new_protected:cpn</code>	Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_new_protected:Npx</code>	<code><parameters></code> (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_new_protected:cpx</code>	<code><function></code> will not expand within an x-type argument. The definition is global and an
	error results if the <code><function></code> is already defined.

<code>\cs_new_protected_nopar:Npn</code>	<code>\cs_new_protected_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_new_protected_nopar:cpn</code>	
<code>\cs_new_protected_nopar:Npx</code>	
<code>\cs_new_protected_nopar:cpx</code>	

Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (#1, #2, etc.) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The `<function>` will not expand within an x-type or e-type argument. The definition is global and an error results if the `<function>` is already defined.

<code>\cs_set:Npn</code>	<code>\cs_set:Npn <function> <parameters> {<code>}</code>
<code>\cs_set:cpn</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_set:Npx</code>	<code><parameters></code> (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_set:cpx</code>	assignment of a meaning to the <code><function></code> is restricted to the current \TeX group level.

<code>\cs_set_nopar:Npn</code>	<code>\cs_set_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_set_nopar:cpn</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_set_nopar:Npx</code>	<code><parameters></code> (#1, #2, etc.) will be replaced by those absorbed by the function. When the
<code>\cs_set_nopar:cpx</code>	<code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The assignment
	of a meaning to the <code><function></code> is restricted to the current \TeX group level.

<code>\cs_set_protected:Npn</code>	<code>\cs_set_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_set_protected:cpn</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_set_protected:Npx</code>	<code><parameters></code> (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_set_protected:cpx</code>	assignment of a meaning to the <code><function></code> is restricted to the current \TeX group level.
	The <code><function></code> will not expand within an x-type or e-type argument.

<code>\cs_set_protected_nopar:Npn</code>	<code>\cs_set_protected_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_set_protected_nopar:cpn</code>	
<code>\cs_set_protected_nopar:Npx</code>	
<code>\cs_set_protected_nopar:cpx</code>	

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current \TeX group level. The $\langle function \rangle$ will not expand within an **x**-type or **e**-type argument.

<code>\cs_gset:Npn</code>	<code>\cs_gset:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset:cpn</code>	
<code>\cs_gset:Npx</code>	
<code>\cs_gset:cpx</code>	

Globally sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the $\langle function \rangle$ is *not* restricted to the current \TeX group level: the assignment is global.

<code>\cs_gset_nopar:Npn</code>	<code>\cs_gset_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_nopar:cpn</code>	
<code>\cs_gset_nopar:Npx</code>	
<code>\cs_gset_nopar:cpx</code>	

Globally sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The assignment of a meaning to the $\langle function \rangle$ is *not* restricted to the current \TeX group level: the assignment is global.

<code>\cs_gset_protected:Npn</code>	<code>\cs_gset_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_protected:cpn</code>	
<code>\cs_gset_protected:Npx</code>	
<code>\cs_gset_protected:cpx</code>	

Globally sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the $\langle function \rangle$ is *not* restricted to the current \TeX group level: the assignment is global. The $\langle function \rangle$ will not expand within an **x**-type or **e**-type argument.

<code>\cs_gset_protected_nopar:Npn</code>	<code>\cs_gset_protected_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_protected_nopar:cpn</code>	
<code>\cs_gset_protected_nopar:Npx</code>	
<code>\cs_gset_protected_nopar:cpx</code>	

Globally sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The assignment of a meaning to the $\langle function \rangle$ is *not* restricted to the current \TeX group level: the assignment is global. The $\langle function \rangle$ will not expand within an **x**-type argument.

3.3 Defining new functions using the signature

<code>\cs_new:Nn</code>	<code>\cs_new:Nn <function> {<code>}</code>
<code>\cs_new:(cn Nx cx)</code>	

Creates $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The definition is global and an error results if the $\langle function \rangle$ is already defined.

`\cs_new_nopar:Nn`
`\cs_new_nopar:(cn|Nx|cx)`

`\cs_new_nopar:Nn <function> {<code>}`

Creates $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The definition is global and an error results if the $\langle function \rangle$ is already defined.

`\cs_new_protected:Nn`
`\cs_new_protected:(cn|Nx|cx)`

`\cs_new_protected:Nn <function> {<code>}`

Creates $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The $\langle function \rangle$ will not expand within an x-type argument. The definition is global and an error results if the $\langle function \rangle$ is already defined.

`\cs_new_protected_nopar:Nn`
`\cs_new_protected_nopar:(cn|Nx|cx)`

`\cs_new_protected_nopar:Nn <function> {<code>}`

Creates $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The $\langle function \rangle$ will not expand within an x-type or e-type argument. The definition is global and an error results if the $\langle function \rangle$ is already defined.

`\cs_set:Nn`
`\cs_set:(cn|Nx|cx)`

`\cs_set:Nn <function> {<code>}`

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current T_EX group level.

`\cs_set_nopar:Nn`
`\cs_set_nopar:(cn|Nx|cx)`

`\cs_set_nopar:Nn <function> {<code>}`

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current T_EX group level.

`\cs_set_protected:Nn`
`\cs_set_protected:(cn|Nx|cx)`

`\cs_set_protected:Nn <function> {<code>}`

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The $\langle function \rangle$ will not expand within an x-type argument. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current T_EX group level.

<code>\cs_set_protected_nopar:Nn</code>	<code>\cs_set_protected_nopar:Nn <function> {<code>}</code>
<code>\cs_set_protected_nopar:(cn Nx cx)</code>	

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The $\langle function \rangle$ will not expand within an *x*-type or *e*-type argument. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current T_EX group level.

<code>\cs_gset:Nn</code>	<code>\cs_gset:Nn <function> {<code>}</code>
<code>\cs_gset:(cn Nx cx)</code>	

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the $\langle function \rangle$ is global.

<code>\cs_gset_nopar:Nn</code>	<code>\cs_gset_nopar:Nn <function> {<code>}</code>
<code>\cs_gset_nopar:(cn Nx cx)</code>	

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The assignment of a meaning to the $\langle function \rangle$ is global.

<code>\cs_gset_protected:Nn</code>	<code>\cs_gset_protected:Nn <function> {<code>}</code>
<code>\cs_gset_protected:(cn Nx cx)</code>	

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The $\langle function \rangle$ will not expand within an *x*-type argument. The assignment of a meaning to the $\langle function \rangle$ is global.

<code>\cs_gset_protected_nopar:Nn</code>	<code>\cs_gset_protected_nopar:Nn <function> {<code>}</code>
<code>\cs_gset_protected_nopar:(cn Nx cx)</code>	

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The $\langle function \rangle$ will not expand within an *x*-type or *e*-type argument. The assignment of a meaning to the $\langle function \rangle$ is global.

<code>\cs_generate_from_arg_count:NNnn</code>	<code>\cs_generate_from_arg_count:NNnn <function> <creator> {<number>}</code>
<code>\cs_generate_from_arg_count:(cNnn Ncnn)</code>	<code>{<code>}</code>

Updated: 2012-01-14

Uses the $\langle creator \rangle$ function (which should have signature Npn , for example `\cs_new:Npn`) to define a $\langle function \rangle$ which takes $\langle number \rangle$ arguments and has $\langle code \rangle$ as replacement text. The $\langle number \rangle$ of arguments is an integer expression, evaluated as detailed for `\int_eval:n`.

3.4 Copying control sequences

Control sequences (not just functions as defined above) can be set to have the same meaning using the functions described here. Making two control sequences equivalent means that the second control sequence is a *copy* of the first (rather than a pointer to it). Thus the old and new control sequence are not tied together: changes to one are not reflected in the other.

In the following text “cs” is used as an abbreviation for “control sequence”.

```
\cs_new_eq:NN
\cs_new_eq:(Nc|cN|cc)
```

```
\cs_new_eq:NN <cs1> <cs2>
\cs_new_eq:NN <cs1> <token>
```

Globally creates $\langle control\ sequence_1 \rangle$ and sets it to have the same meaning as $\langle control\ sequence_2 \rangle$ or $\langle token \rangle$. The second control sequence may subsequently be altered without affecting the copy.

```
\cs_set_eq:NN
\cs_set_eq:(Nc|cN|cc)
```

```
\cs_set_eq:NN <cs1> <cs2>
\cs_set_eq:NN <cs1> <token>
```

Sets $\langle control\ sequence_1 \rangle$ to have the same meaning as $\langle control\ sequence_2 \rangle$ (or $\langle token \rangle$). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the $\langle control\ sequence_1 \rangle$ is restricted to the current T_EX group level.

```
\cs_gset_eq:NN
\cs_gset_eq:(Nc|cN|cc)
```

```
\cs_gset_eq:NN <cs1> <cs2>
\cs_gset_eq:NN <cs1> <token>
```

Globally sets $\langle control\ sequence_1 \rangle$ to have the same meaning as $\langle control\ sequence_2 \rangle$ (or $\langle token \rangle$). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the $\langle control\ sequence_1 \rangle$ is *not* restricted to the current T_EX group level: the assignment is global.

3.5 Deleting control sequences

There are occasions where control sequences need to be deleted. This is handled in a very simple manner.

```
\cs_undefine:N
\cs_undefine:c
```

```
\cs_undefine:N <control sequence>
```

Sets $\langle control\ sequence \rangle$ to be globally undefined.

Updated: 2011-09-15

3.6 Showing control sequences

```
\cs_meaning:N ★
\cs_meaning:c ★
```

```
\cs_meaning:N <control sequence>
```

This function expands to the *meaning* of the $\langle control\ sequence \rangle$ control sequence. For a macro, this includes the $\langle replacement\ text \rangle$.

Updated: 2011-12-22

T_EXhackers note: This is T_EX’s `\meaning` primitive. For tokens that are not control sequences, it is more logical to use `\token_to_meaning:N`. The `c` variant correctly reports undefined arguments.

`\cs_show:N`
`\cs_show:c`

Updated: 2017-02-14

`\cs_show:N` $\langle control\ sequence \rangle$
Displays the definition of the $\langle control\ sequence \rangle$ on the terminal.

T_EXhackers note: This is similar to the T_EX primitive `\show`, wrapped to a fixed number of characters per line.

`\cs_log:N`
`\cs_log:c`

New: 2014-08-22
Updated: 2017-02-14

`\cs_log:N` $\langle control\ sequence \rangle$
Writes the definition of the $\langle control\ sequence \rangle$ in the log file. See also `\cs_show:N` which displays the result in the terminal.

3.7 Converting to and from control sequences

`\use:c` ★

`\use:c` $\{ \langle control\ sequence\ name \rangle \}$

Expands the $\langle control\ sequence\ name \rangle$ until only characters remain, and then converts this into a control sequence. This process requires two expansions. As in other `c`-type arguments the $\langle control\ sequence\ name \rangle$ must, when fully expanded, consist of character tokens, typically a mixture of category code 10 (space), 11 (letter) and 12 (other).

T_EXhackers note: Protected macros that appear in a `c`-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

As an example of the `\use:c` function, both

`\use:c { a b c }`

and

```
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { a b c }
\use:c { \tl_use:N \l_my_tl }
```

would be equivalent to

`\abc`

after two expansions of `\use:c`.

`\cs_if_exist_use:N` ★
`\cs_if_exist_use:c` ★
`\cs_if_exist_use:NTF` ★
`\cs_if_exist_use:cTF` ★

`\cs_if_exist_use:N` $\langle control\ sequence \rangle$
`\cs_if_exist_use:NTF` $\langle control\ sequence \rangle$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$
Tests whether the $\langle control\ sequence \rangle$ is currently defined according to the conditional `\cs_if_exist:NTF` (whether as a function or another control sequence type), and if it is inserts the $\langle control\ sequence \rangle$ into the input stream followed by the $\langle true\ code \rangle$. Otherwise the $\langle false\ code \rangle$ is used.

New: 2012-11-10

<code>\cs:w</code>	★	<code>\cs:w</code> \langle <i>control sequence name</i> \rangle <code>\cs_end:</code>
<code>\cs_end:</code>	★	

Converts the given \langle *control sequence name* \rangle into a single control sequence token. This process requires one expansion. The content for \langle *control sequence name* \rangle may be literal material or from other expandable functions. The \langle *control sequence name* \rangle must, when fully expanded, consist of character tokens which are not active: typically of category code 10 (space), 11 (letter) or 12 (other), or a mixture of these.

T_EXhackers note: These are the T_EX primitives `\csname` and `\endcsname`.

As an example of the `\cs:w` and `\cs_end:` functions, both

`\cs:w a b c \cs_end:`

and

```
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { a b c }
\cs:w \tl_use:N \l_my_tl \cs_end:
```

would be equivalent to

`\abc`

after one expansion of `\cs:w`.

<code>\cs_to_str:N</code>	★	<code>\cs_to_str:N</code> \langle <i>control sequence</i> \rangle
---------------------------	---	---

Converts the given \langle *control sequence* \rangle into a series of characters with category code 12 (other), except spaces, of category code 10. The result does *not* include the current escape token, contrarily to `\token_to_str:N`. Full expansion of this function requires exactly 2 expansion steps, and so an x-type or e-type expansion, or two o-type expansions are required to convert the \langle *control sequence* \rangle to a sequence of characters in the input stream. In most cases, an f-expansion is correct as well, but this loses a space at the start of the result.

4 Analysing control sequences

<code>\cs_split_function:N</code>	★	<code>\cs_split_function:N</code> \langle <i>function</i> \rangle
-----------------------------------	---	---

New: 2018-04-06

Splits the \langle *function* \rangle into the \langle *name* \rangle (*i.e.* the part before the colon) and the \langle *signature* \rangle (*i.e.* after the colon). This information is then placed in the input stream in three parts: the \langle *name* \rangle , the \langle *signature* \rangle and a logic token indicating if a colon was found (to differentiate variables from function names). The \langle *name* \rangle does not include the escape character, and both the \langle *name* \rangle and \langle *signature* \rangle are made up of tokens with category code 12 (other).

The next three functions decompose T_EX macros into their constituent parts: if the \langle *token* \rangle passed is not a macro then no decomposition can occur. In the latter case, all three functions leave `\scan_stop:` in the input stream.

\cs_prefix_spec:N ★

New: 2019-02-27

\cs_prefix_spec:N $\langle token \rangle$

If the $\langle token \rangle$ is a macro, this function leaves the applicable \TeX prefixes in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1~y #2 }
\cs_prefix_spec:N \next:nn
```

leaves $\backslash\text{long}$ in the input stream. If the $\langle token \rangle$ is not a macro then $\backslash\text{scan_stop}$: is left in the input stream.

\TeX hackers note: The prefix can be empty, $\backslash\text{long}$, $\backslash\text{protected}$ or $\backslash\text{protected}\backslash\text{long}$ with backslash replaced by the current escape character.

\cs_argument_spec:N ★

New: 2019-02-27

\cs_argument_spec:N $\langle token \rangle$

If the $\langle token \rangle$ is a macro, this function leaves the primitive \TeX argument specification in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1 y #2 }
\cs_argument_spec:N \next:nn
```

leaves $\text{\#1}\text{\#2}$ in the input stream. If the $\langle token \rangle$ is not a macro then $\backslash\text{scan_stop}$: is left in the input stream.

\TeX hackers note: If the argument specification contains the string \rightarrow , then the function produces incorrect results.

\cs_replacement_spec:N ★

New: 2019-02-27

\cs_replacement_spec:N $\langle token \rangle$

If the $\langle token \rangle$ is a macro, this function leaves the replacement text in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1~y #2 }
\cs_replacement_spec:N \next:nn
```

leaves $\text{x}\text{\#1}_\text{y}\text{\#2}$ in the input stream. If the $\langle token \rangle$ is not a macro then $\backslash\text{scan_stop}$: is left in the input stream.

\TeX hackers note: If the argument specification contains the string \rightarrow , then the function produces incorrect results.

5 Using or removing tokens and arguments

Tokens in the input can be read and used or read and discarded. If one or more tokens are wrapped in braces then when absorbing them the outer set is removed. At the same time, the category code of each token is set when the token is read by a function (if it

is read more than once, the category code is determined by the situation in force when first function absorbs the token).

<code>\use:n</code>	*	<code>\use:n</code>	<code>{\langle group_1 \rangle}</code>
<code>\use:nn</code>	*	<code>\use:nn</code>	<code>{\langle group_1 \rangle} {\langle group_2 \rangle}</code>
<code>\use:nnn</code>	*	<code>\use:nnn</code>	<code>{\langle group_1 \rangle} {\langle group_2 \rangle} {\langle group_3 \rangle}</code>
<code>\use:nnnn</code>	*	<code>\use:nnnn</code>	<code>{\langle group_1 \rangle} {\langle group_2 \rangle} {\langle group_3 \rangle} {\langle group_4 \rangle}</code>

As illustrated, these functions absorb between one and four arguments, as indicated by the argument specifier. The braces surrounding each argument are removed and the remaining tokens are left in the input stream. The category code of these tokens is also fixed by this process (if it has not already been by some other absorption). All of these functions require only a single expansion to operate, so that one expansion of

`\use:nn { abc } { { def } }`

results in the input stream containing

`abc { def }`

i.e. only the outer braces are removed.

T_EXhackers note: The `\use:n` function is equivalent to L^AT_EX 2_ε's `\@firstofone`.

<code>\use_i:nn</code>	*	<code>\use_i:nn</code>	<code>{\langle arg_1 \rangle} {\langle arg_2 \rangle}</code>
<code>\use_ii:nn</code>	*		

These functions absorb two arguments from the input stream. The function `\use_i:nn` discards the second argument, and leaves the content of the first argument in the input stream. `\use_ii:nn` discards the first argument and leaves the content of the second argument in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

T_EXhackers note: These are equivalent to L^AT_EX 2_ε's `\@firstoftwo` and `\@secondoftwo`.

<code>\use_i:nnn</code>	*	<code>\use_i:nnn</code>	<code>{\langle arg_1 \rangle} {\langle arg_2 \rangle} {\langle arg_3 \rangle}</code>
<code>\use_ii:nnn</code>	*		
<code>\use_iii:nnn</code>	*		

These functions absorb three arguments from the input stream. The function `\use_i:nnn` discards the second and third arguments, and leaves the content of the first argument in the input stream. `\use_ii:nnn` and `\use_iii:nnn` work similarly, leaving the content of second or third arguments in the input stream, respectively. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

<code>\use_i:nnnn</code>	*	<code>\use_i:nnnn</code>	<code>{\langle arg_1 \rangle} {\langle arg_2 \rangle} {\langle arg_3 \rangle} {\langle arg_4 \rangle}</code>
<code>\use_ii:nnnn</code>	*		
<code>\use_iii:nnnn</code>	*		
<code>\use_iv:nnnn</code>	*		

These functions absorb four arguments from the input stream. The function `\use_i:nnnn` discards the second, third and fourth arguments, and leaves the content of the first argument in the input stream. `\use_ii:nnnn`, `\use_iii:nnnn` and `\use_iv:nnnn` work similarly, leaving the content of second, third or fourth arguments in the input stream, respectively. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

<code>\use_i_ii:nnn</code>	★	<code>\use_i_ii:nnn {\langle arg_1 \rangle} {\langle arg_2 \rangle} {\langle arg_3 \rangle}</code>
----------------------------	---	--

This function absorbs three arguments and leaves the content of the first and second in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect. An example:

```
\use_i_ii:nnn { abc } { { def } } { ghi }
```

results in the input stream containing

```
abc { def }
```

i.e. the outer braces are removed and the third group is removed.

<code>\use_ii_i:nn</code>	★	<code>\use_ii_i:nn {\langle arg_1 \rangle} {\langle arg_2 \rangle}</code>
---------------------------	---	---

New: 2019-06-02

This function absorbs two arguments and leaves the content of the second and first in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect.

<code>\use_none:n</code>	★	<code>\use_none:n {\langle group_1 \rangle}</code>
--------------------------	---	--

<code>\use_none:nn</code>	★
---------------------------	---

<code>\use_none:nnn</code>	★
----------------------------	---

<code>\use_none:nnnn</code>	★
-----------------------------	---

<code>\use_none:nnnnn</code>	★
------------------------------	---

<code>\use_none:nnnnnn</code>	★
-------------------------------	---

<code>\use_none:nnnnnnn</code>	★
--------------------------------	---

<code>\use_none:nnnnnnnn</code>	★
---------------------------------	---

<code>\use_none:nnnnnnnnn</code>	★
----------------------------------	---

These functions absorb between one and nine groups from the input stream, leaving nothing on the resulting input stream. These functions work after a single expansion. One or more of the `n` arguments may be an unbraced single token (*i.e.* an `N` argument).

TeXhackers note: These are equivalent to L^AT_EX 2_ε's `\@gobble`, `\@gobbletwo`, *etc.*

<code>\use:e</code>	★	<code>\use:e {\langle expandable tokens \rangle}</code>
---------------------	---	---

New: 2018-06-18

Fully expands the `\langle token list \rangle` in an `x`-type manner, *but* the function remains fully expandable, and parameter character (usually `#`) need not be doubled.

TeXhackers note: `\use:e` is a wrapper around the primitive `\expanded` where it is available: it requires two expansions to complete its action. When `\expanded` is not available this function is very slow.

<code>\use:x</code>		<code>\use:x {\langle expandable tokens \rangle}</code>
---------------------	--	---

Updated: 2011-12-31

Fully expands the `\langle expandable tokens \rangle` and inserts the result into the input stream at the current location. Any hash characters (`#`) in the argument must be doubled.

5.1 Selecting tokens from delimited arguments

A different kind of function for selecting tokens from the token stream are those that use delimited arguments.

<code>\use_none_delimit_by_q_nil:w</code>	<code>*</code>	<code>\use_none_delimit_by_q_nil:w <balanced text> \q_nil</code>
<code>\use_none_delimit_by_q_stop:w</code>	<code>*</code>	<code>\use_none_delimit_by_q_stop:w <balanced text> \q_stop</code>
<code>\use_none_delimit_by_q_recursion_stop:w</code>	<code>*</code>	<code>\use_none_delimit_by_q_recursion_stop:w <balanced text></code>

Absorb the *<balanced text>* from the input stream delimited by the marker given in the function name, leaving nothing in the input stream.

<code>\use_i_delimit_by_q_nil:nw</code>	<code>*</code>	<code>\use_i_delimit_by_q_nil:nw {<inserted tokens>} <balanced text></code>
<code>\use_i_delimit_by_q_stop:nw</code>	<code>*</code>	<code>\q_nil</code>
<code>\use_i_delimit_by_q_recursion_stop:nw</code>	<code>*</code>	<code>\use_i_delimit_by_q_stop:nw {<inserted tokens>} <balanced text> \q_stop</code>
		<code>\use_i_delimit_by_q_recursion_stop:nw {<inserted tokens>}</code>
		<code><balanced text> \q_recursion_stop</code>

Absorb the *<balanced text>* from the input stream delimited by the marker given in the function name, leaving *<inserted tokens>* in the input stream for further processing.

6 Predicates and conditionals

L^AT_EX3 has three concepts for conditional flow processing:

Branching conditionals Functions that carry out a test and then execute, depending on its result, either the code supplied as the *<true code>* or the *<false code>*. These arguments are denoted with T and F, respectively. An example would be

`\cs_if_free:cTF {abc} {<true code>} {<false code>}`

a function that turns the first argument into a control sequence (since it's marked as c) then checks whether this control sequence is still free and then depending on the result carries out the code in the second argument (true case) or in the third argument (false case).

These type of functions are known as “conditionals”; whenever a TF function is defined it is usually accompanied by T and F functions as well. These are provided for convenience when the branch only needs to go a single way. Package writers are free to choose which types to define but the kernel definitions always provide all three versions.

Important to note is that these branching conditionals with *<true code>* and/or *<false code>* are always defined in a way that the code of the chosen alternative can operate on following tokens in the input stream.

These conditional functions may or may not be fully expandable, but if they are expandable they are accompanied by a “predicate” for the same test as described below.

Predicates “Predicates” are functions that return a special type of boolean value which can be tested by the boolean expression parser. All functions of this type are expandable and have names that end with `_p` in the description part. For example,

`\cs_if_free_p:N`

would be a predicate function for the same type of test as the conditional described above. It would return “true” if its argument (a single token denoted by N) is still free for definition. It would be used in constructions like

```

\bool_if:nTF {
  \cs_if_free_p:N \l_tmpz_tl || \cs_if_free_p:N \g_tmpz_tl
} {\true code} {\false code}

```

For each predicate defined, a “branching conditional” also exists that behaves like a conditional described above.

Primitive conditionals There is a third variety of conditional, which is the original concept used in plain \TeX and $\text{\LaTeX 2}_{\epsilon}$. Their use is discouraged in `expl3` (although still used in low-level definitions) because they are more fragile and in many cases require more expansion control (hence more code) than the two types of conditionals described above.

```

\c_true_bool
\c_false_bool

```

Constants that represent `true` and `false`, respectively. Used to implement predicates.

6.1 Tests on control sequences

```

\cs_if_eq_p:NN *
\cs_if_eq:NNTF *

```

```

\cs_if_eq_p:NN <cs1> <cs2>
\cs_if_eq:NNTF <cs1> <cs2> {\true code} {\false code}

```

Compares the definition of two *<control sequences>* and is logically `true` if they are the same, *i.e.* if they have exactly the same definition when examined with `\cs_show:N`.

```

\cs_if_exist_p:N *
\cs_if_exist_p:c *
\cs_if_exist:NTF *
\cs_if_exist:cTF *

```

```

\cs_if_exist_p:N <control sequence>
\cs_if_exist:NNTF <control sequence> {\true code} {\false code}

```

Tests whether the *<control sequence>* is currently defined (whether as a function or another control sequence type). Any definition of *<control sequence>* other than `\relax` evaluates as `true`.

```

\cs_if_free_p:N *
\cs_if_free_p:c *
\cs_if_free:NNTF *
\cs_if_free:cTF *

```

```

\cs_if_free_p:N <control sequence>
\cs_if_free:NNTF <control sequence> {\true code} {\false code}

```

Tests whether the *<control sequence>* is currently free to be defined. This test is `false` if the *<control sequence>* currently exists (as defined by `\cs_if_exist:N`).

6.2 Primitive conditionals

The ϵ - \TeX engine itself provides many different conditionals. Some expand whatever comes after them and others don’t. Hence the names for these underlying functions often contains a `:w` part but higher level functions are often available. See for instance `\int_compare_p:nNn` which is a wrapper for `\if_int_compare:w`.

Certain conditionals deal with specific data types like boxes and fonts and are described there. The ones described below are either the universal conditionals or deal with control sequences. We prefix primitive conditionals with `\if_`.

<code>\if_true:</code>	★	<code>\if_true: <true code> \else: <false code> \fi:</code>
<code>\if_false:</code>	★	<code>\if_false: <true code> \else: <false code> \fi:</code>
<code>\else:</code>	★	<code>\reverse_if:N <primitive conditional></code>
<code>\fi:</code>	★	<code>\if_true:</code> always executes <i><true code></i> , while <code>\if_false:</code> always executes <i><false code></i> .
<code>\reverse_if:N</code>	★	<code>\reverse_if:N</code> reverses any two-way primitive conditional. <code>\else:</code> and <code>\fi:</code> delimit the branches of the conditional. The function <code>\or:</code> is documented in <code>l3int</code> and used in case switches.

TeXhackers note: These are equivalent to their corresponding TeX primitive conditionals; `\reverse_if:N` is ε -TeX's `\unless`.

<code>\if_meaning:w</code>	★	<code>\if_meaning:w <arg₁> <arg₂> <true code> \else: <false code> \fi:</code>
----------------------------	---	---

`\if_meaning:w` executes *<true code>* when *<arg₁>* and *<arg₂>* are the same, otherwise it executes *<false code>*. *<arg₁>* and *<arg₂>* could be functions, variables, tokens; in all cases the *unexpanded* definitions are compared.

TeXhackers note: This is TeX's `\ifx`.

<code>\if:w</code>	★	<code>\if:w <token₁> <token₂> <true code> \else: <false code> \fi:</code>
<code>\if_charcode:w</code>	★	<code>\if_catcode:w <token₁> <token₂> <true code> \else: <false code> \fi:</code>
<code>\if_catcode:w</code>	★	These conditionals expand any following tokens until two unexpandable tokens are left. If you wish to prevent this expansion, prefix the token in question with <code>\exp_not:N</code> . <code>\if_catcode:w</code> tests if the category codes of the two tokens are the same whereas <code>\if:w</code> tests if the character codes are identical. <code>\if_charcode:w</code> is an alternative name for <code>\if:w</code> .

<code>\if_cs_exist:N</code>	★	<code>\if_cs_exist:N <cs> <true code> \else: <false code> \fi:</code>
<code>\if_cs_exist:w</code>	★	<code>\if_cs_exist:w <tokens> \cs_end: <true code> \else: <false code> \fi:</code>

Check if *<cs>* appears in the hash table or if the control sequence that can be formed from *<tokens>* appears in the hash table. The latter function does not turn the control sequence in question into `\scan_stop:!` This can be useful when dealing with control sequences which cannot be entered as a single token.

<code>\if_mode_horizontal:</code>	★	<code>\if_mode_horizontal: <true code> \else: <false code> \fi:</code>
<code>\if_mode_vertical:</code>	★	Execute <i><true code></i> if currently in horizontal mode, otherwise execute <i><false code></i> . Similar for the other functions.
<code>\if_mode_math:</code>	★	
<code>\if_mode_inner:</code>	★	

7 Starting a paragraph

`\mode_leave_vertical:`

New: 2017-07-04

`\mode_leave_vertical:`

Ensures that `\TeX` is not in vertical (inter-paragraph) mode. In horizontal or math mode this command has no effect, in vertical mode it switches to horizontal mode, and inserts a box of width `\parindent`, followed by the `\everypar` token list.

`\TeX`hackers note: This results in the contents of the `\everypar` token register being inserted, after `\mode_leave_vertical:` is complete. Notice that in contrast to the `\LaTeX 2ε` `\leavevmode` approach, no box is used by the method implemented here.

7.1 Debugging support

`\debug_on:n`
`\debug_off:n`

New: 2017-07-16

Updated: 2017-08-02

`\debug_on:n { <comma-separated list> }`
`\debug_off:n { <comma-separated list> }`

Turn on and off within a group various debugging code, some of which is also available as `expl3` load-time options. The items that can be used in the `<list>` are

- **`check-declarations`** that checks all `expl3` variables used were previously declared and that local/global variables (based on their name or on their first assignment) are only locally/globally assigned;
- **`check-expressions`** that checks integer, dimension, skip, and muskip expressions are not terminated prematurely;
- **`deprecation`** that makes soon-to-be-deprecated commands produce errors;
- **`log-functions`** that logs function definitions;
- **`all`** that does all of the above.

Providing these as switches rather than options allows testing code even if it relies on other packages: load all other packages, call `\debug_on:n`, and load the code that one is interested in testing. These functions can only be used in `\LaTeX 2ε` package mode loaded with `enable-debug` or another option implying it.

`\debug_suspend:`
`\debug_resume:`

New: 2017-11-28

`\debug_suspend: ... \debug_resume:`

Suppress (locally) errors and logging from `debug` commands, except for the **`deprecation`** errors or warnings. These pairs of commands can be nested. This can be used around pieces of code that are known to fail checks, if such failures should be ignored. See for instance `l3coffins`.

Part V

The l3expan package

Argument expansion

This module provides generic methods for expanding T_EX arguments in a systematic manner. The functions in this module all have prefix `exp`.

Not all possible variations are implemented for every base function. Instead only those that are used within the L^AT_EX3 kernel or otherwise seem to be of general interest are implemented. Consult the module description to find out which functions are actually defined. The next section explains how to define missing variants.

1 Defining new variants

The definition of variant forms for base functions may be necessary when writing new functions or when applying a kernel function in a situation that we haven't thought of before.

Internally preprocessing of arguments is done with functions of the form `\exp_....`. They all look alike, an example would be `\exp_args:NNo`. This function has three arguments, the first and the second are a single tokens, while the third argument should be given in braces. Applying `\exp_args:NNo` expands the content of third argument once before any expansion of the first and second arguments. If `\seq_gpush:No` was not defined it could be coded in the following way:

```
\exp_args:NNo \seq_gpush:Nn
  \g_file_name_stack
  { \l_tmpa_tl }
```

In other words, the first argument to `\exp_args:NNo` is the base function and the other arguments are preprocessed and then passed to this base function. In the example the first argument to the base function should be a single token which is left unchanged while the second argument is expanded once. From this example we can also see how the variants are defined. They just expand into the appropriate `\exp_` function followed by the desired base function, *e.g.*

```
\cs_generate_variant:Nn \seq_gpush:Nn { No }
```

results in the definition of `\seq_gpush:No`

```
\cs_new:Npn \seq_gpush:No { \exp_args:NNo \seq_gpush:Nn }
```

Providing variants in this way in style files is safe as the `\cs_generate_variant:Nn` function will only create new definitions if there is not already one available. Therefore adding such definition to later releases of the kernel will not make such style files obsolete.

The steps above may be automated by using the function `\cs_generate_variant:Nn`, described next.

2 Methods for defining variants

We recall the set of available argument specifiers.

- `N` is used for single-token arguments while `c` constructs a control sequence from its name and passes it to a parent function as an `N`-type argument.
- Many argument types extract or expand some tokens and provide it as an `n`-type argument, namely a braced multiple-token argument: `V` extracts the value of a variable, `v` extracts the value from the name of a variable, `n` uses the argument as it is, `o` expands once, `f` expands fully the front of the token list, `e` and `x` expand fully all tokens (differences are explained later).
- A few odd argument types remain: `T` and `F` for conditional processing, otherwise identical to `n`-type arguments, `p` for the parameter text in definitions, `w` for arguments with a specific syntax, and `D` to denote primitives that should not be used directly.

`\cs_generate_variant:Nn`
`\cs_generate_variant:cn`

Updated: 2017-11-28

`\cs_generate_variant:Nn` $\langle parent\ control\ sequence \rangle$ $\{ \langle variant\ argument\ specifiers \rangle \}$

This function is used to define argument-specifier variants of the $\langle parent\ control\ sequence \rangle$ for L^AT_EX3 code-level macros. The $\langle parent\ control\ sequence \rangle$ is first separated into the $\langle base\ name \rangle$ and $\langle original\ argument\ specifier \rangle$. The comma-separated list of $\langle variant\ argument\ specifiers \rangle$ is then used to define variants of the $\langle original\ argument\ specifier \rangle$ if these are not already defined. For each $\langle variant \rangle$ given, a function is created that expands its arguments as detailed and passes them to the $\langle parent\ control\ sequence \rangle$. So for example

```
\cs_set:Npn \foo:Nn #1#2 { code here }
\cs_generate_variant:Nn \foo:Nn { c }
```

creates a new function `\foo:cn` which expands its first argument into a control sequence name and passes the result to `\foo:Nn`. Similarly

```
\cs_generate_variant:Nn \foo:Nn { NV , cV }
```

generates the functions `\foo:NV` and `\foo:cV` in the same way. The `\cs_generate_variant:Nn` function can only be applied if the $\langle parent\ control\ sequence \rangle$ is already defined. If the $\langle parent\ control\ sequence \rangle$ is protected or if the $\langle variant \rangle$ involves any `x` argument, then the $\langle variant\ control\ sequence \rangle$ is also protected. The $\langle variant \rangle$ is created globally, as is any `\exp_args:N` $\langle variant \rangle$ function needed to carry out the expansion.

Only `n` and `N` arguments can be changed to other types. The only allowed changes are

- `c` variant of an `N` parent;
- `o`, `V`, `v`, `f`, `e`, or `x` variant of an `n` parent;
- `N`, `n`, `T`, `F`, or `p` argument unchanged.

This means the $\langle parent \rangle$ of a $\langle variant \rangle$ form is always unambiguous, even in cases where both an `n`-type parent and an `N`-type parent exist, such as for `\tl_count:n` and `\tl_count:N`.

For backward compatibility it is currently possible to make `n`, `o`, `V`, `v`, `f`, `e`, or `x`-type variants of an `N`-type argument or `N` or `c`-type variants of an `n`-type argument. Both are deprecated. The first because passing more than one token to an `N`-type argument will typically break the parent function's code. The second because programmers who use that most often want to access the value of a variable given its name, hence should use a `V`-type or `v`-type variant instead of `c`-type. In those cases, using the lower-level `\exp_args:No` or `\exp_args:Nc` functions explicitly is preferred to defining confusing variants.

3 Introducing the variants

The `V` type returns the value of a register, which can be one of `tl`, `clist`, `int`, `skip`, `dim`, `muskip`, or built-in T_EX registers. The `v` type is the same except it first creates a control sequence out of its argument before returning the value.

In general, the programmer should not need to be concerned with expansion control. When simply using the content of a variable, functions with a `V` specifier should be used. For those referred to by $(cs)name$, the `v` specifier is available for the same purpose. Only

when specific expansion steps are needed, such as when using delimited arguments, should the lower-level functions with `o` specifiers be employed.

The `e` type expands all tokens fully, starting from the first. More precisely the expansion is identical to that of T_EX’s `\message` (in particular `#` needs not be doubled). It was added in May 2018. In recent enough engines (starting around 2019) it relies on the primitive `\expanded` hence is fast. In older engines it is very much slower. As a result it should only be used in performance critical code if typical users will have a recent installation of the T_EX ecosystem.

The `x` type expands all tokens fully, starting from the first. In contrast to `e`, all macro parameter characters `#` must be doubled, and omitting this leads to low-level errors. In addition this type of expansion is not expandable, namely functions that have `x` in their signature do not themselves expand when appearing inside `x` or `e` expansion.

The `f` type is so special that it deserves an example. It is typically used in contexts where only expandable commands are allowed. Then `x`-expansion cannot be used, and `f`-expansion provides an alternative that expands the front of the token list as much as can be done in such contexts. For instance, say that we want to evaluate the integer expression $3 + 4$ and pass the result 7 as an argument to an expandable function `\example:n`. For this, one should define a variant using `\cs_generate_variant:Nn \example:n { f }`, then do

```
\example:f { \int_eval:n { 3 + 4 } }
```

Note that `x`-expansion would also expand `\int_eval:n` fully to its result 7, but the variant `\example:x` cannot be expandable. Note also that `o`-expansion would not expand `\int_eval:n` fully to its result since that function requires several expansions. Besides the fact that `x`-expansion is protected rather than expandable, another difference between `f`-expansion and `x`-expansion is that `f`-expansion expands tokens from the beginning and stops as soon as a non-expandable token is encountered, while `x`-expansion continues expanding further tokens. Thus, for instance

```
\example:f { \int_eval:n { 1 + 2 } , \int_eval:n { 3 + 4 } }
```

results in the call

```
\example:n { 3 , \int_eval:n { 3 + 4 } }
```

while using `\example:x` or `\example:e` instead results in

```
\example:n { 3 , 7 }
```

at the cost of being protected (for `x` type) or very much slower in old engines (for `e` type). If you use `f` type expansion in conditional processing then you should stick to using TF type functions only as the expansion does not finish any `\if... \fi`: itself!

It is important to note that both `f`- and `o`-type expansion are concerned with the expansion of tokens from left to right in their arguments. In particular, `o`-type expansion applies to the first *token* in the argument it receives: it is conceptually similar to

```
\exp_after:wN <base function> \exp_after:wN { <argument> }
```

At the same time, `f`-type expansion stops at the *first* non-expandable token. This means for example that both

```
\tl_set:No \l_tmpa_tl { { \g_tmpb_tl } }
```

and

```
\tl_set:Nf \l_tmpa_tl { { \g_tmpb_tl } }
```

leave `\g_tmpb_tl` unchanged: `{` is the first token in the argument and is non-expandable. It is usually best to keep the following in mind when using variant forms.

- Variants with `x`-type arguments (that are fully expanded before being passed to the `n`-type base function) are never expandable even when the base function is. Such variants cannot work correctly in arguments that are themselves subject to expansion. Consider using `f` or `e` expansion.
- In contrast, `e` expansion (full expansion, almost like `x` except for the treatment of `#`) does not prevent variants from being expandable (if the base function is). The drawback is that `e` expansion is very much slower in old engines (before 2019). Consider using `f` expansion if that type of expansion is sufficient to perform the required expansion, or `x` expansion if the variant will not itself need to be expandable.
- Finally `f` expansion only expands the front of the token list, stopping at the first non-expandable token. This may fail to fully expand the argument.

When speed is essential (for functions that do very little work and whose variants are used numerous times in a document) the following considerations apply because internal functions for argument expansion come in two flavours, some faster than others.

- Arguments that might need expansion should come first in the list of arguments.
- Arguments that should consist of single tokens `N`, `c`, `V`, or `v` should come first among these.
- Arguments that appear after the first multi-token argument `n`, `f`, `e`, or `o` require slightly slower special processing to be expanded. Therefore it is best to use the optimized functions, namely those that contain only `N`, `c`, `V`, and `v`, and, in the last position, `o`, `f`, `e`, with possible trailing `N` or `n` or `T` or `F`, which are not expanded. Any `x`-type argument causes slightly slower processing.

4 Manipulating the first argument

These functions are described in detail: expansion of multiple tokens follows the same rules but is described in a shorter fashion.

```
\exp_args:Nc ★ \exp_args:Nc <function> {<tokens>}  
\exp_args:cc ★
```

This function absorbs two arguments (the `<function>` name and the `<tokens>`). The `<tokens>` are expanded until only characters remain, and are then turned into a control sequence. The result is inserted into the input stream *after* reinsertion of the `<function>`. Thus the `<function>` may take more than one argument: all others are left unchanged.

The `:cc` variant constructs the `<function>` name in the same manner as described for the `<tokens>`.

T_EXhackers note: Protected macros that appear in a `c`-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

`\exp_args:No` ★ `\exp_args:No` $\langle function \rangle$ $\{\langle tokens \rangle\}$...

This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are expanded once, and the result is inserted in braces into the input stream *after* reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.

`\exp_args:NV` ★ `\exp_args:NV` $\langle function \rangle$ $\langle variable \rangle$

This function absorbs two arguments (the names of the $\langle function \rangle$ and the $\langle variable \rangle$). The content of the $\langle variable \rangle$ are recovered and placed inside braces into the input stream *after* reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.

`\exp_args:Nv` ★ `\exp_args:Nv` $\langle function \rangle$ $\{\langle tokens \rangle\}$

This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are expanded until only characters remain, and are then turned into a control sequence. This control sequence should be the name of a $\langle variable \rangle$. The content of the $\langle variable \rangle$ are recovered and placed inside braces into the input stream *after* reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.

TeXhackers note: Protected macros that appear in a v-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

`\exp_args:Ne` ★ `\exp_args:Ne` $\langle function \rangle$ $\{\langle tokens \rangle\}$

New: 2018-05-15

This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$) and exhaustively expands the $\langle tokens \rangle$. The result is inserted in braces into the input stream *after* reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.

TeXhackers note: This relies on the `\expanded` primitive when available (in LuaTeX and starting around 2019 in other engines). Otherwise it uses some fall-back code that is very much slower. As a result it should only be used in performance-critical code if typical users have a recent installation of the TeX ecosystem.

`\exp_args:Nf` ★ `\exp_args:Nf` $\langle function \rangle$ $\{\langle tokens \rangle\}$

This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are fully expanded until the first non-expandable token is found (if that is a space it is removed), and the result is inserted in braces into the input stream *after* reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.

<code>\exp_args:Nx</code>	<code>\exp_args:Nx</code>	$\langle function \rangle$	$\{\langle tokens \rangle\}$
---------------------------	---------------------------	----------------------------	------------------------------

This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$) and exhaustively expands the $\langle tokens \rangle$. The result is inserted in braces into the input stream *after* reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.

5 Manipulating two arguments

<code>\exp_args:NNc</code>	<code>\exp_args:NNc</code>	$\langle token_1 \rangle$	$\langle token_2 \rangle$	$\{\langle tokens \rangle\}$
----------------------------	----------------------------	---------------------------	---------------------------	------------------------------

These optimized functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments.

`\exp_args:NNv` *
`\exp_args:NNe` *
`\exp_args:NNf` *
`\exp_args:Ncc` *
`\exp_args:Nco` *
`\exp_args:NcV` *
`\exp_args:Ncv` *
`\exp_args:Ncf` *
`\exp_args:NVV` *

Updated: 2018-05-15

<code>\exp_args:Nnc</code>	<code>\exp_args:Noo</code>	$\langle token \rangle$	$\{\langle tokens_1 \rangle\}$	$\{\langle tokens_2 \rangle\}$
----------------------------	----------------------------	-------------------------	--------------------------------	--------------------------------

These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments. These functions need slower processing.

`\exp_args:Noc` *
`\exp_args:Noo` *
`\exp_args:Nof` *
`\exp_args:NVo` *
`\exp_args:Nfo` *
`\exp_args:Nff` *
`\exp_args:Nee` *

Updated: 2018-05-15

<code>\exp_args:NNx</code>	<code>\exp_args:NNx</code>	$\langle token_1 \rangle$	$\langle token_2 \rangle$	$\{\langle tokens \rangle\}$
----------------------------	----------------------------	---------------------------	---------------------------	------------------------------

These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments. These functions are not expandable due to their x-type argument.

6 Manipulating three arguments

<code>\exp_args:NNNo</code>	*	<code>\exp_args:NNNo</code>	$\langle token_1 \rangle$	$\langle token_2 \rangle$	$\langle token_3 \rangle$	$\{\langle tokens \rangle\}$
<code>\exp_args:NNNV</code>	*	These optimized functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, <i>etc.</i>				
<code>\exp_args:Nccc</code>	*					
<code>\exp_args:NcNc</code>	*					
<code>\exp_args:NcNo</code>	*					
<code>\exp_args:Ncco</code>	*					

<code>\exp_args:NNcf</code>	*	<code>\exp_args:NNoo</code>	$\langle token_1 \rangle$	$\langle token_2 \rangle$	$\{\langle token_3 \rangle\}$	$\{\langle tokens \rangle\}$
<code>\exp_args:NNno</code>	*	These functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, <i>etc.</i> These functions need slower processing.				
<code>\exp_args:NNnV</code>	*					
<code>\exp_args:NNoo</code>	*					
<code>\exp_args:NNVV</code>	*					
<code>\exp_args:Ncno</code>	*					
<code>\exp_args:NcnV</code>	*					
<code>\exp_args:Ncoo</code>	*					
<code>\exp_args:NcVV</code>	*					
<code>\exp_args:Nnnc</code>	*					
<code>\exp_args:Nnno</code>	*					
<code>\exp_args:Nnnf</code>	*					
<code>\exp_args:Nnff</code>	*					
<code>\exp_args:Nooo</code>	*					
<code>\exp_args:Noof</code>	*					
<code>\exp_args:Nffo</code>	*					
<code>\exp_args:Neee</code>	*					

<code>\exp_args:NNNx</code>	<code>\exp_args:NNNx</code>	$\langle token_1 \rangle$	$\langle token_2 \rangle$	$\{\langle tokens_1 \rangle\}$	$\{\langle tokens_2 \rangle\}$
<code>\exp_args:NNnx</code>	These functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, <i>etc.</i>				
<code>\exp_args:NNox</code>					
<code>\exp_args:Nccx</code>					
<code>\exp_args:Ncnx</code>					
<code>\exp_args:NNnx</code>					
<code>\exp_args:Nnox</code>					
<code>\exp_args:Noox</code>					

New: 2015-08-12

7 Unbraced expansion

```

\exp_last_unbraced:No  *
\exp_last_unbraced:NV  *
\exp_last_unbraced:Nv  *
\exp_last_unbraced:Ne  *
\exp_last_unbraced:Nf  *
\exp_last_unbraced:NNo *
\exp_last_unbraced:NNV *
\exp_last_unbraced:NNf *
\exp_last_unbraced:Nco *
\exp_last_unbraced:NcV *
\exp_last_unbraced:Nno *
\exp_last_unbraced:Noo *
\exp_last_unbraced:Nfo *
\exp_last_unbraced:NNNo *
\exp_last_unbraced:NNNV *
\exp_last_unbraced:NNNf *
\exp_last_unbraced:NnNo *
\exp_last_unbraced:NNNNo *
\exp_last_unbraced:NNNNf *

```

Updated: 2018-05-15

```
\exp_last_unbraced:Nno <token> {\tokens_1} {\tokens_2}
```

These functions absorb the number of arguments given by their specification, carry out the expansion indicated and leave the results in the input stream, with the last argument not surrounded by the usual braces. Of these, the :Nno, :Noo, :Nfo and :NnNo variants need slower processing.

T_EXhackers note: As an optimization, the last argument is unbraced by some of those functions before expansion. This can cause problems if the argument is empty: for instance, `\exp_last_unbraced:Nf \foo_bar:w { } \q_stop` leads to an infinite loop, as the quark is f-expanded.

```
\exp_last_unbraced:Nx \exp_last_unbraced:Nx <function> {\tokens}
```

This function fully expands the `<tokens>` and leaves the result in the input stream after reinsertion of the `<function>`. This function is not expandable.

```
\exp_last_two_unbraced:Noo * \exp_last_two_unbraced:Noo <token> {\tokens_1} {\tokens_2}
```

This function absorbs three arguments and expands the second and third once. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments, which are not wrapped in braces. This function needs special (slower) processing.

```
\exp_after:wN * \exp_after:wN <token_1> <token_2>
```

Carries out a single expansion of `<token_2>` (which may consume arguments) prior to the expansion of `<token_1>`. If `<token_2>` has no expansion (for example, if it is a character) then it is left unchanged. It is important to notice that `<token_1>` may be *any* single token, including group-opening and -closing tokens (`{` or `}` assuming normal T_EX category codes). Unless specifically required this should be avoided: expansion should be carried out using an appropriate argument specifier variant or the appropriate `\exp_arg:N` function.

T_EXhackers note: This is the T_EX primitive `\expandafter` renamed.

8 Preventing expansion

Despite the fact that the following functions are all about preventing expansion, they're designed to be used in an expandable context and hence are all marked as being 'expand-

able' since they themselves disappear after the expansion has completed.

`\exp_not:N` ★ `\exp_not:N` $\langle token \rangle$

Prevents expansion of the $\langle token \rangle$ in a context where it would otherwise be expanded, for example an **x**-type argument or the first token in an **o** or **e** or **f** argument.

T_EXhackers note: This is the T_EX `\noexpand` primitive. It only prevents expansion. At the beginning of an **f**-type argument, a space $\langle token \rangle$ is removed even if it appears as `\exp_not:N \c_space_token`. In an **x**-expanding definition (`\cs_new:Npx`), a macro parameter introduces an argument even if it appears as `\exp_not:N # 1`. This differs from `\exp_not:n`.

`\exp_not:c` ★ `\exp_not:c` $\{\langle tokens \rangle\}$

Expands the $\langle tokens \rangle$ until only characters remain, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited using `\exp_not:N`.

T_EXhackers note: Protected macros that appear in a **c**-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

`\exp_not:n` ★ `\exp_not:n` $\{\langle tokens \rangle\}$

Prevents expansion of the $\langle tokens \rangle$ in an **e** or **x**-type argument. In all other cases the $\langle tokens \rangle$ continue to be expanded, for example in the input stream or in other types of arguments such as **c**, **f**, **v**. The argument of `\exp_not:n` *must* be surrounded by braces.

T_EXhackers note: This is the ε -T_EX `\unexpanded` primitive. In an **x**-expanding definition (`\cs_new:Npx`), `\exp_not:n {\#1}` is equivalent to `##1` rather than to `#1`, namely it inserts the two characters `#` and `1`. In an **e**-type argument `\exp_not:n {\#}` is equivalent to `#`, namely it inserts the character `#`.

`\exp_not:o` ★ `\exp_not:o` $\{\langle tokens \rangle\}$

Expands the $\langle tokens \rangle$ once, then prevents any further expansion in **x**-type or **e**-type arguments using `\exp_not:n`.

`\exp_not:V` ★ `\exp_not:V` $\langle variable \rangle$

Recovers the content of the $\langle variable \rangle$, then prevents expansion of this material in **x**-type or **e**-type arguments using `\exp_not:n`.

<hr/> <hr/>	<code>\exp_not:v *</code>	<code>\exp_not:v {<tokens>}</code>	Expands the <i><tokens></i> until only characters remains, and then converts this into a control sequence which should be a <i><variable></i> name. The content of the <i><variable></i> is recovered, and further expansion in <i>x</i> -type or <i>e</i> -type arguments is prevented using <code>\exp_not:n</code> .
<hr/> <hr/>	<code>\exp_not:e *</code>	<code>\exp_not:e {<tokens>}</code>	Expands <i><tokens></i> exhaustively, then protects the result of the expansion (including any tokens which were not expanded) from further expansion in <i>e</i> or <i>x</i> -type arguments using <code>\exp_not:n</code> . This is very rarely useful but is provided for consistency.
<hr/> <hr/>	<code>\exp_not:f *</code>	<code>\exp_not:f {<tokens>}</code>	Expands <i><tokens></i> fully until the first unexpandable token is found (if it is a space it is removed). Expansion then stops, and the result of the expansion (including any tokens which were not expanded) is protected from further expansion in <i>x</i> -type or <i>e</i> -type arguments using <code>\exp_not:n</code> .
<hr/> <hr/>	<code>\exp_stop_f: *</code>	<code>\foo_bar:f { <tokens> \exp_stop_f: <more tokens> }</code>	This function terminates an <i>f</i> -type expansion. Thus if a function <code>\foo_bar:f</code> starts an <i>f</i> -type expansion and all of <i><tokens></i> are expandable <code>\exp_stop_f:</code> terminates the expansion of tokens even if <i><more tokens></i> are also expandable. The function itself is an implicit space token. Inside an <i>x</i> -type expansion, it retains its form, but when typeset it produces the underlying space (␣).

Updated: 2011-06-03

9 Controlled expansion

The `expl3` language makes all efforts to hide the complexity of `TeX` expansion from the programmer by providing concepts that evaluate/expand arguments of functions prior to calling the “base” functions. Thus, instead of using many `\expandafter` calls and other trickery it is usually a matter of choosing the right variant of a function to achieve a desired result.

Of course, deep down `TeX` is using expansion as always and there are cases where a programmer needs to control that expansion directly; typical situations are basic data manipulation tools. This section documents the functions for that level. These commands are used throughout the kernel code, but we hope that outside the kernel there will be little need to resort to them. Instead the argument manipulation methods document above should usually be sufficient.

While `\exp_after:wN` expands one token (out of order) it is sometimes necessary to expand several tokens in one go. The next set of commands provide this functionality. Be aware that it is absolutely required that the programmer has full control over the tokens to be expanded, i.e., it is not possible to use these functions to expand unknown input as part of *<expandable-tokens>* as that will break badly if unexpandable tokens are encountered in that place!

<code>\exp:w</code>	★
<code>\exp_end:</code>	★

New: 2015-08-23

`\exp:w <expandable tokens> \exp_end:`

Expands `<expandable-tokens>` until reaching `\exp_end:` at which point expansion stops. The full expansion of `<expandable tokens>` has to be empty. If any token in `<expandable tokens>` or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result `\exp_end:` will be misinterpreted later on.³

In typical use cases the `\exp_end:` is hidden somewhere in the replacement text of `<expandable-tokens>` rather than being on the same expansion level than `\exp:w`, e.g., you may see code such as

`\exp:w \@@_case:NnTF #1 {#2} { } { }`

where somewhere during the expansion of `\@@_case:NnTF` the `\exp_end:` gets generated.

T_EXhackers note: The current implementation uses `\romannumeral` hence ignores space tokens and explicit signs + and - in the expansion of the `<expandable tokens>`, but this should not be relied upon.

<code>\exp:w</code>	★
<code>\exp_end_continue_f:w</code>	★

New: 2015-08-23

`\exp:w <expandable-tokens> \exp_end_continue_f:w <further-tokens>`

Expands `<expandable-tokens>` until reaching `\exp_end_continue_f:w` at which point expansion continues as an `f`-type expansion expanding `<further-tokens>` until an unexpandable token is encountered (or the `f`-type expansion is explicitly terminated by `\exp_stop_f:`). As with all `f`-type expansions a space ending the expansion gets removed.

The full expansion of `<expandable-tokens>` has to be empty. If any token in `<expandable-tokens>` or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result `\exp_end_continue_f:w` will be misinterpreted later on.⁴

In typical use cases `<expandable-tokens>` contains no tokens at all, e.g., you will see code such as

`\exp_after:wN { \exp:w \exp_end_continue_f:w #2 }`

where the `\exp_after:wN` triggers an `f`-expansion of the tokens in `#2`. For technical reasons this has to happen using two tokens (if they would be hidden inside another command `\exp_after:wN` would only expand the command but not trigger any additional `f`-expansion).

You might wonder why there are two different approaches available, after all the effect of

`\exp:w <expandable-tokens> \exp_end:`

can be alternatively achieved through an `f`-type expansion by using `\exp_stop_f:`, i.e.

`\exp:w \exp_end_continue_f:w <expandable-tokens> \exp_stop_f:`

The reason is simply that the first approach is slightly faster (one less token to parse and less expansion internally) so in places where such performance really matters and where we want to explicitly stop the expansion at a defined point the first form is preferable.

³Due to the implementation you might get the character in position 0 in the current font (typically “”) in the output without any error message!

⁴In this particular case you may get a character into the output as well as an error message.

<code>\exp:w</code>	★
<code>\exp_end_continue_f:nw</code>	★

New: 2015-08-23

`\exp:w` *<expandable-tokens>* `\exp_end_continue_f:nw` *<further-tokens>*

The difference to `\exp_end_continue_f:w` is that we first we pick up an argument which is then returned to the input stream. If *<further-tokens>* starts with space tokens then these space tokens are removed while searching for the argument. If it starts with a brace group then the braces are removed. Thus such spaces or braces will not terminate the f-type expansion.

10 Internal functions

`\::n` `\cs_new:Npn \exp_args:Ncof { \::c \::o \::f \::: }`

`\::N` Internal forms for the base expansion types. These names do *not* conform to the general \LaTeX 3 approach as this makes them more readily visible in the log and so forth. They should not be used outside this module.

`\::c`
`\::o`
`\::e`
`\::f`
`\::x`
`\::v`
`\::V`
`\:::`

`\::o_unbraced` `\cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \::: }`

`\::e_unbraced` Internal forms for the expansion types which leave the terminal argument unbraced. These names do *not* conform to the general \LaTeX 3 approach as this makes them more readily visible in the log and so forth. They should not be used outside this module.

`\::f_unbraced`
`\::x_unbraced`
`\::v_unbraced`
`\::V_unbraced`

Part VI

The l3quark package

Quarks

Two special types of constants in L^AT_EX3 are “quarks” and “scan marks”. By convention all constants of type quark start out with `\q_`, and scan marks start with `\s_`.

1 Quarks

Quarks are control sequences that expand to themselves and should therefore *never* be executed directly in the code. This would result in an endless loop!

They are meant to be used as delimiter in weird functions, the most common use case being the ‘stop token’ (*i.e.* `\q_stop`). For example, when writing a macro to parse a user-defined date

```
\date_parse:n {19/June/1981}
```

one might write a command such as

```
\cs_new:Npn \date_parse:n #1 { \date_parse_aux:w #1 \q_stop }
\cs_new:Npn \date_parse_aux:w #1 / #2 / #3 \q_stop
{ <do something with the date> }
```

Quarks are sometimes also used as error return values for functions that receive erroneous input. For example, in the function `\prop_get:NnN` to retrieve a value stored in some key of a property list, if the key does not exist then the return value is the quark `\q_no_value`. As mentioned above, such quarks are extremely fragile and it is imperative when using such functions that code is carefully written to check for pathological cases to avoid leakage of a quark into an uncontrolled environment.

Quarks also permit the following ingenious trick when parsing tokens: when you pick up a token in a temporary variable and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary variable to the quark using `\tl_if_eq:NNTF`. A set of special quark testing functions is set up below. All the quark testing functions are expandable although the ones testing only single tokens are much faster. An example of the quark testing functions and their use in recursion can be seen in the implementation of `\clist_map_function:NN`.

2 Defining quarks

`\quark_new:N`

`\quark_new:N <quark>`

Creates a new `<quark>` which expands only to `<quark>`. The `<quark>` is defined globally, and an error message is raised if the name was already taken.

`\q_stop`

Used as a marker for delimited arguments, such as

```
\cs_set:Npn \tmp:w #1#2 \q_stop {#1}
```

<u><u>\q_mark</u></u>	Used as a marker for delimited arguments when <code>\q_stop</code> is already in use.
<u><u>\q_nil</u></u>	Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself need to be tested (in contrast to <code>\q_stop</code> , which is only ever used as a delimiter).
<u><u>\q_no_value</u></u>	A canonical value for a missing value, when one is requested from a data structure. This is therefore used as a “return” value by functions such as <code>\prop_get:NnN</code> if there is no data to return.

3 Quark tests

The method used to define quarks means that the single token (N) tests are faster than the multi-token (n) tests. The latter should therefore only be used when the argument can definitely take more than a single token.

<u><u>\quark_if_nil_p:N</u></u> *	<code>\quark_if_nil_p:N <token></code>
<u><u>\quark_if_nil:NTF</u></u> *	<code>\quark_if_nil:NTF <token> {\true code} {\false code}</code>
	Tests if the <code><token></code> is equal to <code>\q_nil</code> .
<u><u>\quark_if_nil_p:n</u></u> *	<code>\quark_if_nil_p:n {\token list}</code>
<u><u>\quark_if_nil_p:(o V)</u></u> *	<code>\quark_if_nil:nTF {\token list} {\true code} {\false code}</code>
<u><u>\quark_if_nil:nTF</u></u> *	Tests if the <code><token list></code> contains only <code>\q_nil</code> (distinct from <code><token list></code> being empty or
<u><u>\quark_if_nil:(o V)TF</u></u> *	containing <code>\q_nil</code> plus one or more other tokens).
<u><u>\quark_if_no_value_p:N</u></u> *	<code>\quark_if_no_value_p:N <token></code>
<u><u>\quark_if_no_value_p:c</u></u> *	<code>\quark_if_no_value:NTF <token> {\true code} {\false code}</code>
<u><u>\quark_if_no_value:NTF</u></u> *	Tests if the <code><token></code> is equal to <code>\q_no_value</code> .
<u><u>\quark_if_no_value:cTF</u></u> *	
<u><u>\quark_if_no_value_p:n</u></u> *	<code>\quark_if_no_value_p:n {\token list}</code>
<u><u>\quark_if_no_value:nTF</u></u> *	<code>\quark_if_no_value:nTF {\token list} {\true code} {\false code}</code>
	Tests if the <code><token list></code> contains only <code>\q_no_value</code> (distinct from <code><token list></code> being empty or containing <code>\q_no_value</code> plus one or more other tokens).

4 Recursion

This module provides a uniform interface to intercepting and terminating loops as when one is doing tail recursion. The building blocks follow below and an example is shown in Section 5.

<u><u>\q_recursion_tail</u></u>	This quark is appended to the data structure in question and appears as a real element there. This means it gets any list separators around it.
---------------------------------	---

\q_recursion_stop

This quark is added *after* the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.

\quark_if_recursion_tail_stop:N ★ **\quark_if_recursion_tail_stop:N** $\langle token \rangle$

Tests if $\langle token \rangle$ contains only the marker **\q_recursion_tail**, and if so uses **\use_none_delimit_by_q_recursion_stop:w** to terminate the recursion that this belongs to. The recursion input must include the marker tokens **\q_recursion_tail** and **\q_recursion_stop** as the last two items.

\quark_if_recursion_tail_stop:n ★ **\quark_if_recursion_tail_stop:n** $\{\langle token list \rangle\}$
\quark_if_recursion_tail_stop:o ★

Updated: 2011-09-06

Tests if the $\langle token list \rangle$ contains only **\q_recursion_tail**, and if so uses **\use_none_delimit_by_q_recursion_stop:w** to terminate the recursion that this belongs to. The recursion input must include the marker tokens **\q_recursion_tail** and **\q_recursion_stop** as the last two items.

\quark_if_recursion_tail_stop_do:Nn ★ **\quark_if_recursion_tail_stop_do:Nn** $\langle token \rangle \{\langle insertion \rangle\}$

Tests if $\langle token \rangle$ contains only the marker **\q_recursion_tail**, and if so uses **\use_i_delimit_by_q_recursion_stop:w** to terminate the recursion that this belongs to. The recursion input must include the marker tokens **\q_recursion_tail** and **\q_recursion_stop** as the last two items. The $\langle insertion \rangle$ code is then added to the input stream after the recursion has ended.

\quark_if_recursion_tail_stop_do:nn ★ **\quark_if_recursion_tail_stop_do:nn** $\{\langle token list \rangle\} \{\langle insertion \rangle\}$
\quark_if_recursion_tail_stop_do:on ★

Updated: 2011-09-06

Tests if the $\langle token list \rangle$ contains only **\q_recursion_tail**, and if so uses **\use_i_delimit_by_q_recursion_stop:w** to terminate the recursion that this belongs to. The recursion input must include the marker tokens **\q_recursion_tail** and **\q_recursion_stop** as the last two items. The $\langle insertion \rangle$ code is then added to the input stream after the recursion has ended.

\quark_if_recursion_tail_break:NN ★ **\quark_if_recursion_tail_break:nN** $\{\langle token list \rangle\}$
\quark_if_recursion_tail_break:nN ★ **\langle type \rangle_map_break:**

New: 2018-04-10

Tests if $\langle token list \rangle$ contains only **\q_recursion_tail**, and if so terminates the recursion using **\langle type \rangle_map_break:.** The recursion end should be marked by **\prg_break_point:Nn \langle type \rangle_map_break:.**

5 An example of recursion with quarks

Quarks are mainly used internally in the `expl3` code to define recursion functions such as `\tl_map_inline:nn` and so on. Here is a small example to demonstrate how to

use quarks in this fashion. We shall define a command called `\my_map_dbl:nn` which takes a token list and applies an operation to every *pair* of tokens. For example, `\my_map_dbl:nn {abcd} {[--#1--#2--]~}` would produce “[-a-b-] [-c-d-]”. Using quarks to define such functions simplifies their logic and ensures robustness in many cases.

Here’s the definition of `\my_map_dbl:nn`. First of all, define the function that does the processing based on the inline function argument `#2`. Then initiate the recursion using an internal function. The token list `#1` is terminated using `\q_recursion_tail`, with delimiters according to the type of recursion (here a pair of `\q_recursion_tail`), concluding with `\q_recursion_stop`. These quarks are used to mark the end of the token list being operated upon.

```
\cs_new:Npn \my_map_dbl:nn #1#2
{
  \cs_set:Npn \__my_map_dbl_fn:nn ##1 ##2 {#2}
  \__my_map_dbl:nn #1 \q_recursion_tail \q_recursion_tail
  \q_recursion_stop
}
```

The definition of the internal recursion function follows. First check if either of the input tokens are the termination quarks. Then, if not, apply the inline function to the two arguments.

```
\cs_new:Nn \__my_map_dbl:nn
{
  \quark_if_recursion_tail_stop:n {#1}
  \quark_if_recursion_tail_stop:n {#2}
  \__my_map_dbl_fn:nn {#1} {#2}
}
```

Finally, recurse:

```
\__my_map_dbl:nn
}
```

Note that contrarily to L^AT_EX3 built-in mapping functions, this mapping function cannot be nested, since the second map would overwrite the definition of `__my_map_dbl_fn:nn`.

6 Scan marks

Scan marks are control sequences set equal to `\scan_stop:`, hence never expand in an expansion context and are (largely) invisible if they are encountered in a typesetting context.

Like quarks, they can be used as delimiters in weird functions and are often safer to use for this purpose. Since they are harmless when executed by T_EX in non-expandable contexts, they can be used to mark the end of a set of instructions. This allows to skip to that point if the end of the instructions should not be performed (see l3regex).

`\scan_new:N`

New: 2018-04-01

`\scan_new:N` *<scan mark>*

Creates a new *<scan mark>* which is set equal to `\scan_stop:`. The *<scan mark>* is defined globally, and an error message is raised if the name was already taken by another scan mark.

<hr/> \s_stop <hr/>	Used at the end of a set of instructions, as a marker that can be jumped to using \use_
New: 2018-04-01	none_delimit_by_s_stop:w.

<hr/> \use_none_delimit_by_s_stop:w ★	\use_none_delimit_by_s_stop:w <i><tokens></i> \s_stop
--	---

New: 2018-04-01

Removes the *<tokens>* and **\s_stop** from the input stream. This leads to a low-level T_EX error if **\s_stop** is absent.

Part VII

The l3tl package

Token lists

T_EX works with tokens, and L^AT_EX3 therefore provides a number of functions to deal with lists of tokens. Token lists may be present directly in the argument to a function:

```
\foo:n { a collection of \tokens }
```

or may be stored in a so-called “token list variable”, which have the suffix `tl`: a token list variable can also be used as the argument to a function, for example

```
\foo:N \l_some_tl
```

In both cases, functions are available to test and manipulate the lists of tokens, and these have the module prefix `tl`. In many cases, functions which can be applied to token list variables are paired with similar functions for application to explicit lists of tokens: the two “views” of a token list are therefore collected together here.

A token list (explicit, or stored in a variable) can be seen either as a list of “items”, or a list of “tokens”. An item is whatever `\use:n` would grab as its argument: a single non-space token or a brace group, with optional leading explicit space characters (each item is thus itself a token list). A token is either a normal `N` argument, or `␣`, `{`, or `}` (assuming normal T_EX category codes). Thus for example

```
{ Hello } ~ world
```

contains six items (Hello, w, o, r, l and d), but thirteen tokens (`{`, H, e, l, l, o, `}`, `␣`, w, o, r, l and d). Functions which act on items are often faster than their analogue acting directly on tokens.

1 Creating and initialising token list variables

```
\tl_new:N \tl_new:N <tl var>
```

```
\tl_new:c
```

Creates a new `<tl var>` or raises an error if the name is already taken. The declaration is global. The `<tl var>` is initially empty.

```
\tl_const:Nn \tl_const:Nn <tl var> {<token list>}
```

```
\tl_const:(Nx|cn|cx)
```

Creates a new constant `<tl var>` or raises an error if the name is already taken. The value of the `<tl var>` is set globally to the `<token list>`.

```
\tl_clear:N \tl_clear:N <tl var>
```

```
\tl_clear:c
```

```
\tl_gclear:N
```

```
\tl_gclear:c
```

Clears all entries from the `<tl var>`.

<hr/>	
<code>\tl_clear_new:N</code>	<code>\tl_clear_new:N <tl var></code>
<code>\tl_clear_new:c</code>	
<code>\tl_gclear_new:N</code>	Ensures that the <code><tl var></code> exists globally by applying <code>\tl_new:N</code> if necessary, then applies
<code>\tl_gclear_new:c</code>	<code>\tl_(g)clear:N</code> to leave the <code><tl var></code> empty.
<hr/>	
<code>\tl_set_eq:NN</code>	<code>\tl_set_eq:NN <tl var_1> <tl var_2></code>
<code>\tl_set_eq:(cN Nc cc)</code>	Sets the content of <code><tl var_1></code> equal to that of <code><tl var_2></code> .
<code>\tl_gset_eq:NN</code>	
<code>\tl_gset_eq:(cN Nc cc)</code>	
<hr/>	
<code>\tl_concat:NNN</code>	<code>\tl_concat:NNN <tl var_1> <tl var_2> <tl var_3></code>
<code>\tl_concat:ccc</code>	
<code>\tl_gconcat:NNN</code>	Concatenates the content of <code><tl var_2></code> and <code><tl var_3></code> together and saves the result in
<code>\tl_gconcat:ccc</code>	<code><tl var_1></code> . The <code><tl var_2></code> is placed at the left side of the new token list.
<hr/>	
New: 2012-05-18	
<hr/>	
<code>\tl_if_exist_p:N *</code>	<code>\tl_if_exist_p:N <tl var></code>
<code>\tl_if_exist_p:c *</code>	<code>\tl_if_exist:NTF <tl var> {<true code>} {<false code>}</code>
<code>\tl_if_exist:N\overline{TF} *</code>	
<code>\tl_if_exist:c\overline{TF} *</code>	Tests whether the <code><tl var></code> is currently defined. This does not check that the <code><tl var></code> really is a token list variable.
<hr/>	
New: 2012-03-03	

2 Adding data to token list variables

<hr/>	
<code>\tl_set:Nn</code>	<code>\tl_set:Nn <tl var> {<tokens>}</code>
<code>\tl_set:(NV Nv No Nf Nx cn cV cv co cf cx)</code>	
<code>\tl_gset:Nn</code>	
<code>\tl_gset:(NV Nv No Nf Nx cn cV cv co cf cx)</code>	
<hr/>	
Sets <code><tl var></code> to contain <code><tokens></code> , removing any previous content from the variable.	
<hr/>	
<code>\tl_put_left:Nn</code>	<code>\tl_put_left:Nn <tl var> {<tokens>}</code>
<code>\tl_put_left:(NV No Nx cn cV co cx)</code>	
<code>\tl_gput_left:Nn</code>	
<code>\tl_gput_left:(NV No Nx cn cV co cx)</code>	
<hr/>	
Appends <code><tokens></code> to the left side of the current content of <code><tl var></code> .	
<hr/>	
<code>\tl_put_right:Nn</code>	<code>\tl_put_right:Nn <tl var> {<tokens>}</code>
<code>\tl_put_right:(NV No Nx cn cV co cx)</code>	
<code>\tl_gput_right:Nn</code>	
<code>\tl_gput_right:(NV No Nx cn cV co cx)</code>	
<hr/>	
Appends <code><tokens></code> to the right side of the current content of <code><tl var></code> .	

3 Modifying token list variables

```
\tl_replace_once:Nnn
\tl_replace_once:cnn
\tl_greplace_once:Nnn
\tl_greplace_once:cnn
```

Updated: 2011-08-11

```
\tl_replace_once:Nnn <tl var> {{<old tokens>}} {{<new tokens>}}
```

Replaces the first (leftmost) occurrence of *<old tokens>* in the *<tl var>* with *<new tokens>*. *<Old tokens>* cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

```
\tl_replace_all:Nnn
\tl_replace_all:cnn
\tl_greplace_all:Nnn
\tl_greplace_all:cnn
```

Updated: 2011-08-11

```
\tl_replace_all:Nnn <tl var> {{<old tokens>}} {{<new tokens>}}
```

Replaces all occurrences of *<old tokens>* in the *<tl var>* with *<new tokens>*. *<Old tokens>* cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern *<old tokens>* may remain after the replacement (see `\tl_remove_all:Nn` for an example).

```
\tl_remove_once:Nn
\tl_remove_once:cn
\tl_gremove_once:Nn
\tl_gremove_once:cn
```

Updated: 2011-08-11

```
\tl_remove_once:Nn <tl var> {{<tokens>}}
```

Removes the first (leftmost) occurrence of *<tokens>* from the *<tl var>*. *<Tokens>* cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

```
\tl_remove_all:Nn
\tl_remove_all:cn
\tl_gremove_all:Nn
\tl_gremove_all:cn
```

Updated: 2011-08-11

```
\tl_remove_all:Nn <tl var> {{<tokens>}}
```

Removes all occurrences of *<tokens>* from the *<tl var>*. *<Tokens>* cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern *<tokens>* may remain after the removal, for instance,

```
\tl_set:Nn \l_tmpa_tl {abbccd} \tl_remove_all:Nn \l_tmpa_tl {bc}
```

results in `\l_tmpa_tl` containing `abcd`.

4 Reassigning token list category codes

These functions allow the rescanning of tokens: re-apply T_EX's tokenization process to apply category codes different from those in force when the tokens were absorbed. Whilst this functionality is supported, it is often preferable to find alternative approaches to achieving outcomes rather than rescanning tokens (for example construction of token lists token-by-token with intervening category code changes or using `\char_generate:nn`).

<code>\tl_set_rescan:Nnn</code>	<code>\tl_set_rescan:Nnn <tl var> {<setup>} {<tokens>}</code>
<code>\tl_set_rescan:(Nno Nnx cnn cno cnx)</code>	
<code>\tl_gset_rescan:Nnn</code>	
<code>\tl_gset_rescan:(Nno Nnx cnn cno cnx)</code>	
Updated: 2015-08-11	

Sets $\langle tl\ var \rangle$ to contain $\langle tokens \rangle$, applying the category code régime specified in the $\langle setup \rangle$ before carrying out the assignment. (Category codes applied to tokens not explicitly covered by the $\langle setup \rangle$ are those in force at the point of use of `\tl_set_rescan:Nnn`.) This allows the $\langle tl\ var \rangle$ to contain material with category codes other than those that apply when $\langle tokens \rangle$ are absorbed. The $\langle setup \rangle$ is run within a group and may contain any valid input, although only changes in category codes, such as uses of `\cctab_select:N`, are relevant. See also `\tl_rescan:nn`.

T_EXhackers note: The $\langle tokens \rangle$ are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\endlinechar` unless that is equal to 32, before the user $\langle setup \rangle$), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

<code>\tl_rescan:nn</code>	<code>\tl_rescan:nn {<setup>} {<tokens>}</code>
Updated: 2015-08-11	

Rescans $\langle tokens \rangle$ applying the category code régime specified in the $\langle setup \rangle$, and leaves the resulting tokens in the input stream. (Category codes applied to tokens not explicitly covered by the $\langle setup \rangle$ are those in force at the point of use of `\tl_rescan:nn`.) The $\langle setup \rangle$ is run within a group and may contain any valid input, although only changes in category codes, such as uses of `\cctab_select:N`, are relevant. See also `\tl_set_rescan:Nnn`, which is more robust than using `\tl_set:Nn` in the $\langle tokens \rangle$ argument of `\tl_rescan:nn`.

T_EXhackers note: The $\langle tokens \rangle$ are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\endlinechar` unless that is equal to 32, before the user $\langle setup \rangle$), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

5 Token list conditionals

<code>\tl_if_blank_p:n</code> *	<code>\tl_if_blank_p:n {<token list>}</code>	
<code>\tl_if_blank_p:(e V o)</code> *	<code>\tl_if_blank:nTF {<token list>} {<true code>} {<false code>}</code>	
<code>\tl_if_blank:nTF</code> *		Tests if the $\langle token\ list \rangle$ consists only of blank spaces (<i>i.e.</i> contains no item). The test is
<code>\tl_if_blank:(e V o)TF</code> *		true if $\langle token\ list \rangle$ is zero or more explicit space characters (explicit tokens with character code 32 and category code 10), and is false otherwise.
Updated: 2019-09-04		

<code>\tl_if_empty_p:N</code>	★	<code>\tl_if_empty_p:N <tl var></code>
<code>\tl_if_empty_p:c</code>	★	<code>\tl_if_empty:NTF <tl var> {<true code>} {<false code>}</code>
<code>\tl_if_empty:nTF</code>	★	Tests if the <i><token list variable></i> is entirely empty (<i>i.e.</i> contains no tokens at all).
<code>\tl_if_empty:cTF</code>	★	

<code>\tl_if_empty_p:n</code>	★	<code>\tl_if_empty_p:n {<token list>}</code>
<code>\tl_if_empty_p:(V o)</code>	★	<code>\tl_if_empty:NNTF {<token list>} {<true code>} {<false code>}</code>
<code>\tl_if_empty:nTF</code>	★	Tests if the <i><token list></i> is entirely empty (<i>i.e.</i> contains no tokens at all).
<code>\tl_if_empty:(V o)TF</code>	★	

New: 2012-05-24
Updated: 2012-06-05

<code>\tl_if_eq_p:NN</code>	★	<code>\tl_if_eq_p:NN <tl var₁> <tl var₂></code>
<code>\tl_if_eq_p:(Nc cN cc)</code>	★	<code>\tl_if_eq:NNTF <tl var₁> <tl var₂> {<true code>} {<false code>}</code>
<code>\tl_if_eq:NNTF</code>	★	Compares the content of two <i><token list variables></i> and is logically true if the two contain the same list of tokens (<i>i.e.</i> identical in both the list of characters they contain and the category codes of those characters). Thus for example
<code>\tl_if_eq:(Nc cN cc)TF</code>	★	

```

\tl_set:Nn \l_tmpa_tl { abc }
\tl_set:Nx \l_tmpb_tl { \tl_to_str:n { abc } }
\tl_if_eq:NNTF \l_tmpa_tl \l_tmpb_tl { true } { false }

```

yields **false**. See also `\str_if_eq:nnTF` for a comparison that ignores category codes.

<code>\tl_if_eq:NnTF</code>		<code>\tl_if_eq:NnTF <tl var₁> {<token list₂>} {<true code>} {<false code>}</code>
<code>\tl_if_eq:cnTF</code>		Tests if the <i><token list variable₁></i> and the <i><token list₂></i> contain the same list of tokens, both in respect of character codes and category codes. This conditional is not expandable: see <code>\tl_if_eq:NNTF</code> for an expandable version when both token lists are stored in variables, or <code>\str_if_eq:nnTF</code> if category codes are not important.

New: 2020-07-14

<code>\tl_if_eq:nnTF</code>		<code>\tl_if_eq:nnTF {<token list₁>} {<token list₂>} {<true code>} {<false code>}</code>
		Tests if <i><token list₁></i> and <i><token list₂></i> contain the same list of tokens, both in respect of character codes and category codes. This conditional is not expandable: see <code>\tl_if_eq:NNTF</code> for an expandable version when token lists are stored in variables, or <code>\str_if_eq:nnTF</code> if category codes are not important.

<code>\tl_if_in:NnTF</code>		<code>\tl_if_in:NnTF <tl var> {<token list>} {<true code>} {<false code>}</code>
<code>\tl_if_in:cnTF</code>		Tests if the <i><token list></i> is found in the content of the <i><tl var></i> . The <i><token list></i> cannot contain the tokens <code>{</code> , <code>}</code> or <code>#</code> (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

<code>\tl_if_in:nnTF</code>		<code>\tl_if_in:nnTF {<token list₁>} {<token list₂>} {<true code>} {<false code>}</code>
<code>\tl_if_in:(Vn on no)TF</code>		Tests if <i><token list₂></i> is found inside <i><token list₁></i> . The <i><token list₂></i> cannot contain the tokens <code>{</code> , <code>}</code> or <code>#</code> (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

<code>\tl_if_novalue_p:n *</code>	<code>\tl_if_novalue_p:n {⟨token list⟩}</code>
<code>\tl_if_novalue:nTF *</code>	<code>\tl_if_novalue:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>

New: 2017-11-14

Tests if the $\langle token\ list \rangle$ is exactly equal to the special `\c_novalue_tl` marker. This function is intended to allow construction of flexible document interface structures in which missing optional arguments are detected.

<code>\tl_if_single_p:N *</code>	<code>\tl_if_single_p:N ⟨tl var⟩</code>
<code>\tl_if_single_p:c *</code>	<code>\tl_if_single:NTF ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}</code>
<code>\tl_if_single:NTF *</code>	
<code>\tl_if_single:cTF *</code>	

Updated: 2011-08-13

Tests if the content of the $\langle tl\ var \rangle$ consists of a single item, *i.e.* is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to `\tl_count:N`.

<code>\tl_if_single_p:n *</code>	<code>\tl_if_single_p:n {⟨token list⟩}</code>
<code>\tl_if_single:nTF *</code>	<code>\tl_if_single:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>

Updated: 2011-08-13

Tests if the $\langle token\ list \rangle$ has exactly one item, *i.e.* is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to `\tl_count:n`.

<code>\tl_if_single_token_p:n *</code>	<code>\tl_if_single_token_p:n {⟨token list⟩}</code>
<code>\tl_if_single_token:nTF *</code>	<code>\tl_if_single_token:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>

Tests if the token list consists of exactly one token, *i.e.* is either a single space character or a single “normal” token. Token groups $\{ \dots \}$ are not single tokens.

<code>\tl_case:Nn *</code>	<code>\tl_case:NnTF ⟨test token list variable⟩</code>
<code>\tl_case:cn *</code>	<code>{</code>
<code>\tl_case:NnTF *</code>	<code>⟨token list variable case₁⟩ {⟨code case₁⟩}</code>
<code>\tl_case:cnTF *</code>	<code>⟨token list variable case₂⟩ {⟨code case₂⟩}</code>
	<code>...</code>
	<code>⟨token list variable case_n⟩ {⟨code case_n⟩}</code>
	<code>}</code>
	<code>{⟨true code⟩}</code>
	<code>{⟨false code⟩}</code>

New: 2013-07-24

This function compares the $\langle test\ token\ list\ variable \rangle$ in turn with each of the $\langle token\ list\ variable\ cases \rangle$. If the two are equal (as described for `\tl_if_eq:NNTF`) then the associated $\langle code \rangle$ is left in the input stream and other cases are discarded. If any of the cases are matched, the $\langle true\ code \rangle$ is also inserted into the input stream (after the code for the appropriate case), while if none match then the $\langle false\ code \rangle$ is inserted. The function `\tl_case:Nn`, which does nothing if there is no match, is also available.

6 Mapping to token lists

All mappings are done at the current group level, *i.e.* any local assignments made by the $\langle function \rangle$ or $\langle code \rangle$ discussed below remain in effect after the loop.

<hr/> <code>\tl_map_function:NN</code> ☆ <code>\tl_map_function:cN</code> ☆ <hr/> Updated: 2012-06-29	<code>\tl_map_function:NN</code> $\langle tl\ var \rangle$ $\langle function \rangle$ Applies $\langle function \rangle$ to every $\langle item \rangle$ in the $\langle tl\ var \rangle$. The $\langle function \rangle$ receives one argument for each iteration. This may be a number of tokens if the $\langle item \rangle$ was stored within braces. Hence the $\langle function \rangle$ should anticipate receiving n-type arguments. See also <code>\tl_map_function:nN</code> .
<hr/> <code>\tl_map_function:nN</code> ☆ <hr/> Updated: 2012-06-29	<code>\tl_map_function:nN</code> $\{ \langle token\ list \rangle \}$ $\langle function \rangle$ Applies $\langle function \rangle$ to every $\langle item \rangle$ in the $\langle token\ list \rangle$, The $\langle function \rangle$ receives one argument for each iteration. This may be a number of tokens if the $\langle item \rangle$ was stored within braces. Hence the $\langle function \rangle$ should anticipate receiving n-type arguments. See also <code>\tl_map_function:NN</code> .
<hr/> <code>\tl_map_inline:Nn</code> <code>\tl_map_inline:cn</code> <hr/> Updated: 2012-06-29	<code>\tl_map_inline:Nn</code> $\langle tl\ var \rangle$ $\{ \langle inline\ function \rangle \}$ Applies the $\langle inline\ function \rangle$ to every $\langle item \rangle$ stored within the $\langle tl\ var \rangle$. The $\langle inline\ function \rangle$ should consist of code which receives the $\langle item \rangle$ as #1. See also <code>\tl_map_function:NN</code> .
<hr/> <code>\tl_map_inline:nn</code> <hr/> Updated: 2012-06-29	<code>\tl_map_inline:nn</code> $\{ \langle token\ list \rangle \}$ $\{ \langle inline\ function \rangle \}$ Applies the $\langle inline\ function \rangle$ to every $\langle item \rangle$ stored within the $\langle token\ list \rangle$. The $\langle inline\ function \rangle$ should consist of code which receives the $\langle item \rangle$ as #1. See also <code>\tl_map_function:nN</code> .
<hr/> <code>\tl_map_tokens:Nn</code> ☆ <code>\tl_map_tokens:cn</code> ☆ <code>\tl_map_tokens:nn</code> ☆ <hr/> New: 2019-09-02	<code>\tl_map_tokens:Nn</code> $\langle tl\ var \rangle$ $\{ \langle code \rangle \}$ <code>\tl_map_tokens:nn</code> $\langle tokens \rangle$ $\{ \langle code \rangle \}$ Analogue of <code>\tl_map_function:NN</code> which maps several tokens instead of a single function. The $\langle code \rangle$ receives each item in the $\langle tl\ var \rangle$ or $\langle tokens \rangle$ as two trailing brace groups. For instance, $\tl_map_tokens:Nn\ \l_my_tl\ \{ \prg_replicate:nn\ \{ 2 \} \}$ expands to twice each item in the $\langle sequence \rangle$: for each item in <code>\l_my_tl</code> the function <code>\prg_replicate:nn</code> receives 2 and $\langle item \rangle$ as its two arguments. The function <code>\tl_map_inline:Nn</code> is typically faster but is not expandable.
<hr/> <code>\tl_map_variable:NNn</code> <code>\tl_map_variable:cNn</code> <hr/> Updated: 2012-06-29	<code>\tl_map_variable:NNn</code> $\langle tl\ var \rangle$ $\langle variable \rangle$ $\{ \langle code \rangle \}$ Stores each $\langle item \rangle$ of the $\langle tl\ var \rangle$ in turn in the (token list) $\langle variable \rangle$ and applies the $\langle code \rangle$. The $\langle code \rangle$ will usually make use of the $\langle variable \rangle$, but this is not enforced. The assignments to the $\langle variable \rangle$ are local. Its value after the loop is the last $\langle item \rangle$ in the $\langle tl\ var \rangle$, or its original value if the $\langle tl\ var \rangle$ is blank. See also <code>\tl_map_inline:Nn</code> .
<hr/> <code>\tl_map_variable:nNn</code> <hr/> Updated: 2012-06-29	<code>\tl_map_variable:nNn</code> $\{ \langle token\ list \rangle \}$ $\langle variable \rangle$ $\{ \langle code \rangle \}$ Stores each $\langle item \rangle$ of the $\langle token\ list \rangle$ in turn in the (token list) $\langle variable \rangle$ and applies the $\langle code \rangle$. The $\langle code \rangle$ will usually make use of the $\langle variable \rangle$, but this is not enforced. The assignments to the $\langle variable \rangle$ are local. Its value after the loop is the last $\langle item \rangle$ in the $\langle tl\ var \rangle$, or its original value if the $\langle tl\ var \rangle$ is blank. See also <code>\tl_map_inline:nn</code> .

<hr/> <code>\tl_map_break:</code> ☆	<code>\tl_map_break:</code>
<hr/> Updated: 2012-06-29	Used to terminate a <code>\tl_map...</code> function before all entries in the <i>⟨token list variable⟩</i> have been processed. This normally takes place within a conditional statement, for example

```

\tl_map_inline:Nn \l_my_tl
{
  \str_if_eq:nnT { #1 } { bingo } { \tl_map_break: }
  % Do something useful
}

```

See also `\tl_map_break:n`. Use outside of a `\tl_map...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the *⟨tokens⟩* are inserted into the input stream. This depends on the design of the mapping function.

<hr/> <code>\tl_map_break:n</code> ☆	<code>\tl_map_break:n {⟨code⟩}</code>
<hr/> Updated: 2012-06-29	Used to terminate a <code>\tl_map...</code> function before all entries in the <i>⟨token list variable⟩</i> have been processed, inserting the <i>⟨code⟩</i> after the mapping has ended. This normally takes place within a conditional statement, for example

```

\tl_map_inline:Nn \l_my_tl
{
  \str_if_eq:nnT { #1 } { bingo }
  { \tl_map_break:n { <code> } }
  % Do something useful
}

```

Use outside of a `\tl_map...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the *⟨code⟩* is inserted into the input stream. This depends on the design of the mapping function.

7 Using token lists

<code>\tl_to_str:n</code>	★	<code>\tl_to_str:n {⟨token list⟩}</code>
<code>\tl_to_str:V</code>	★	

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, leaving the resulting character tokens in the input stream. A $\langle string \rangle$ is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This function requires only a single expansion. Its argument *must* be braced.

TeXhackers note: This is the ε -TeX primitive `\detokenize`. Converting a $\langle token\ list \rangle$ to a $\langle string \rangle$ yields a concatenation of the string representations of every token in the $\langle token\ list \rangle$. The string representation of a control sequence is

- an escape character, whose character code is given by the internal parameter `\escapechar`, absent if the `\escapechar` is negative or greater than the largest character code;
- the control sequence name, as defined by `\cs_to_str:N`;
- a space, unless the control sequence name is a single character whose category at the time of expansion of `\tl_to_str:n` is not “letter”.

The string representation of an explicit character token is that character, doubled in the case of (explicit) macro parameter characters (normally `#`). In particular, the string representation of a token list may depend on the category codes in effect when it is evaluated, and the value of the `\escapechar`: for instance `\tl_to_str:n {\a}` normally produces the three character “backslash”, “lower-case a”, “space”, but it may also produce a single “lower-case a” if the escape character is negative and `a` is currently not a letter.

<code>\tl_to_str:N</code>	★	<code>\tl_to_str:N ⟨tl var⟩</code>
<code>\tl_to_str:c</code>	★	

Converts the content of the $\langle tl\ var \rangle$ into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This $\langle string \rangle$ is then left in the input stream. For low-level details, see the notes given for `\tl_to_str:n`.

<code>\tl_use:N</code>	★	<code>\tl_use:N ⟨tl var⟩</code>
<code>\tl_use:c</code>	★	

Recovers the content of a $\langle tl\ var \rangle$ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a $\langle tl\ var \rangle$ directly without an accessor function.

8 Working with the content of token lists

<code>\tl_count:n</code>	★	<code>\tl_count:n {⟨tokens⟩}</code>
<code>\tl_count:(V o)</code>	★	

New: 2012-05-13

Counts the number of $\langle items \rangle$ in $\langle tokens \rangle$ and leaves this information in the input stream. Unbraced tokens count as one element as do each token group ($\{...\}$). This process ignores any unprotected spaces within $\langle tokens \rangle$. See also `\tl_count:N`. This function requires three expansions, giving an $\langle integer\ denotation \rangle$.

`\tl_count:N` ★
`\tl_count:c` ★

New: 2012-05-13

`\tl_count:N` $\langle tl\ var \rangle$
Counts the number of token groups in the $\langle tl\ var \rangle$ and leaves this information in the input stream. Unbraced tokens count as one element as do each token group $\{...\}$. This process ignores any unprotected spaces within the $\langle tl\ var \rangle$. See also `\tl_count:n`. This function requires three expansions, giving an *integer denotation*.

`\tl_count_tokens:n` ★

New: 2019-02-25

`\tl_count_tokens:n` $\{\langle tokens \rangle\}$
Counts the number of \TeX tokens in the $\langle tokens \rangle$ and leaves this information in the input stream. Every token, including spaces and braces, contributes one to the total; thus for instance, the token count of `a~{bc}` is 6.

`\tl_reverse:n` ★
`\tl_reverse:(V|o)` ★

Updated: 2012-01-08

`\tl_reverse:n` $\{\langle token\ list \rangle\}$
Reverses the order of the $\langle items \rangle$ in the $\langle token\ list \rangle$, so that $\langle item_1 \rangle \langle item_2 \rangle \langle item_3 \rangle \dots \langle item_n \rangle$ becomes $\langle item_n \rangle \dots \langle item_3 \rangle \langle item_2 \rangle \langle item_1 \rangle$. This process preserves unprotected space within the $\langle token\ list \rangle$. Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider `\tl_reverse_items:n`. See also `\tl_reverse:N`.

\TeX hackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an \mathbf{x} -type argument expansion.

`\tl_reverse:N`
`\tl_reverse:c`
`\tl_greverse:N`
`\tl_greverse:c`

Updated: 2012-01-08

`\tl_reverse:N` $\langle tl\ var \rangle$
Reverses the order of the $\langle items \rangle$ stored in $\langle tl\ var \rangle$, so that $\langle item_1 \rangle \langle item_2 \rangle \langle item_3 \rangle \dots \langle item_n \rangle$ becomes $\langle item_n \rangle \dots \langle item_3 \rangle \langle item_2 \rangle \langle item_1 \rangle$. This process preserves unprotected spaces within the $\langle token\ list\ variable \rangle$. Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. See also `\tl_reverse:n`, and, for improved performance, `\tl_reverse_items:n`.

`\tl_reverse_items:n` ★

New: 2012-01-08

`\tl_reverse_items:n` $\{\langle token\ list \rangle\}$
Reverses the order of the $\langle items \rangle$ stored in $\langle tl\ var \rangle$, so that $\{\langle item_1 \rangle\} \{\langle item_2 \rangle\} \{\langle item_3 \rangle\} \dots \{\langle item_n \rangle\}$ becomes $\{\langle item_n \rangle\} \dots \{\langle item_3 \rangle\} \{\langle item_2 \rangle\} \{\langle item_1 \rangle\}$. This process removes any unprotected space within the $\langle token\ list \rangle$. Braced token groups are copied without reversing the order of tokens, and keep the outer set of braces. Items which are initially not braced are copied with braces in the result. In cases where preserving spaces is important, consider the slower function `\tl_reverse:n`.

\TeX hackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an \mathbf{x} -type argument expansion.

`\tl_trim_spaces:n` ★
`\tl_trim_spaces:o` ★

New: 2011-07-09
Updated: 2012-06-25

`\tl_trim_spaces:n` $\{\langle token\ list \rangle\}$
Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the $\langle token\ list \rangle$ and leaves the result in the input stream.

\TeX hackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an \mathbf{x} -type argument expansion.

<code>\tl_trim_spaces_apply:nN</code> ★	<code>\tl_trim_spaces_apply:nN</code> $\{ \langle token\ list \rangle \}$ $\langle function \rangle$
<code>\tl_trim_spaces_apply:oN</code> ★	
New: 2018-04-12	Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the $\langle token\ list \rangle$ and passes the result to the $\langle function \rangle$ as an <i>n</i> -type argument.

<code>\tl_trim_spaces:N</code>	<code>\tl_trim_spaces:N</code> $\langle tl\ var \rangle$
<code>\tl_trim_spaces:c</code>	
<code>\tl_gtrim_spaces:N</code>	Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the content of the $\langle tl\ var \rangle$. Note that this therefore <i>resets</i> the content of the variable.
<code>\tl_gtrim_spaces:c</code>	
New: 2011-07-09	

<code>\tl_sort:Nn</code>	<code>\tl_sort:Nn</code> $\langle tl\ var \rangle$ $\{ \langle comparison\ code \rangle \}$
<code>\tl_sort:cn</code>	
<code>\tl_gsort:Nn</code>	Sorts the items in the $\langle tl\ var \rangle$ according to the $\langle comparison\ code \rangle$, and assigns the result to $\langle tl\ var \rangle$. The details of sorting comparison are described in Section 1.
<code>\tl_gsort:cn</code>	
New: 2017-02-06	

<code>\tl_sort:nN</code> ★	<code>\tl_sort:nN</code> $\{ \langle token\ list \rangle \}$ $\langle conditional \rangle$
New: 2017-02-06	Sorts the items in the $\langle token\ list \rangle$, using the $\langle conditional \rangle$ to compare items, and leaves the result in the input stream. The $\langle conditional \rangle$ should have signature <code>:nnTF</code> , and return true if the two items being compared should be left in the same order, and false if the items should be swapped. The details of sorting comparison are described in Section 1.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an *x*-type or *e*-type argument expansion.

9 The first token from a token list

Functions which deal with either only the very first item (balanced text or single normal token) in a token list, or the remaining tokens.

<hr/>	
<code>\tl_head:N</code>	★
<code>\tl_head:n</code>	★
<code>\tl_head:(V v f)</code>	★
<hr/>	
Updated: 2012-09-09	
<hr/>	

`\tl_head:n {⟨token list⟩}`

Leaves in the input stream the first *⟨item⟩* in the *⟨token list⟩*, discarding the rest of the *⟨token list⟩*. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example

`\tl_head:n { abc }`

and

`\tl_head:n { ~ abc }`

both leave `a` in the input stream. If the “head” is a brace group, rather than a single token, the braces are removed, and so

`\tl_head:n { ~ { ~ ab } c }`

yields `▯ab`. A blank *⟨token list⟩* (see `\tl_if_blank:nTF`) results in `\tl_head:n` leaving nothing in the input stream.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an *x*-type argument expansion.

<hr/>	
<code>\tl_head:w</code>	★
<hr/>	

`\tl_head:w ⟨token list⟩ { } \q_stop`

Leaves in the input stream the first *⟨item⟩* in the *⟨token list⟩*, discarding the rest of the *⟨token list⟩*. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank *⟨token list⟩* (which consists only of space characters) results in a low-level TeX error, which may be avoided by the inclusion of an empty group in the input (as shown), without the need for an explicit test. Alternatively, `\tl_if_blank:nF` may be used to avoid using the function with a “blank” argument. This function requires only a single expansion, and thus is suitable for use within an *o*-type expansion. In general, `\tl_head:n` should be preferred if the number of expansions is not critical.

<hr/>	
<code>\tl_tail:N</code>	★
<code>\tl_tail:n</code>	★
<code>\tl_tail:(V v f)</code>	★
<hr/>	

Updated: 2012-09-01

`\tl_tail:n {⟨token list⟩}`

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first *⟨item⟩* in the *⟨token list⟩*, and leaves the remaining tokens in the input stream. Thus for example

`\tl_tail:n { a ~ {bc} d }`

and

`\tl_tail:n { ~ a ~ {bc} d }`

both leave `▯{bc}d` in the input stream. A blank *⟨token list⟩* (see `\tl_if_blank:nTF`) results in `\tl_tail:n` leaving nothing in the input stream.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an *x*-type argument expansion.

```

\tl_if_head_eq_catcode_p:nN * \tl_if_head_eq_catcode_p:nN {\token list} \test token
\tl_if_head_eq_catcode_p:oN * \tl_if_head_eq_catcode:nNTF {\token list} \test token
\tl_if_head_eq_catcode:nNTF * {\true code} {\false code}
\tl_if_head_eq_catcode:oNTF *

```

Updated: 2012-07-09

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ has the same category code as the $\langle test token \rangle$. In the case where the $\langle token list \rangle$ is empty, the test is always **false**.

```

\tl_if_head_eq_charcode_p:nN * \tl_if_head_eq_charcode_p:nN {\token list} \test token
\tl_if_head_eq_charcode_p:fN * \tl_if_head_eq_charcode:nNTF {\token list} \test token
\tl_if_head_eq_charcode:nNTF * {\true code} {\false code}
\tl_if_head_eq_charcode:fNTF *

```

Updated: 2012-07-09

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ has the same character code as the $\langle test token \rangle$. In the case where the $\langle token list \rangle$ is empty, the test is always **false**.

```

\tl_if_head_eq_meaning_p:nN * \tl_if_head_eq_meaning_p:nN {\token list} \test token
\tl_if_head_eq_meaning:nNTF * \tl_if_head_eq_meaning:nNTF {\token list} \test token
\tl_if_head_eq_meaning:nNTF * {\true code} {\false code}

```

Updated: 2012-07-09

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ has the same meaning as the $\langle test token \rangle$. In the case where $\langle token list \rangle$ is empty, the test is always **false**.

```

\tl_if_head_is_group_p:n * \tl_if_head_is_group_p:n {\token list}
\tl_if_head_is_group:nTF * \tl_if_head_is_group:nTF {\token list} {\true code} {\false code}

```

New: 2012-07-08

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ is an explicit begin-group character (with category code 1 and any character code), in other words, if the $\langle token list \rangle$ starts with a brace group. In particular, the test is **false** if the $\langle token list \rangle$ starts with an implicit token such as `\c_group_begin_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

```

\tl_if_head_is_N_type_p:n * \tl_if_head_is_N_type_p:n {\token list}
\tl_if_head_is_N_type:nTF * \tl_if_head_is_N_type:nTF {\token list} {\true code} {\false code}

```

New: 2012-07-08

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ is a normal N-type argument. In other words, it is neither an explicit space character (explicit token with character code 32 and category code 10) nor an explicit begin-group character (with category code 1 and any character code). An empty argument yields **false**, as it does not have a “normal” first token. This function is useful to implement actions on token lists on a token by token basis.

```

\tl_if_head_is_space_p:n * \tl_if_head_is_space_p:n {\token list}
\tl_if_head_is_space:nTF * \tl_if_head_is_space:nTF {\token list} {\true code} {\false code}

```

Updated: 2012-07-08

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ is an explicit space character (explicit token with character code 12 and category code 10). In particular, the test is **false** if the $\langle token list \rangle$ starts with an implicit token such as `\c_space_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

10 Using a single item

<code>\tl_item:nn</code> *	<code>\tl_item:nn {(token list)} {(integer expression)}</code>
<code>\tl_item:Nn</code> *	Indexing items in the $\langle token list \rangle$ from 1 on the left, this function evaluates the $\langle integer expression \rangle$ and leaves the appropriate item from the $\langle token list \rangle$ in the input stream. If the $\langle integer expression \rangle$ is negative, indexing occurs from the right of the token list, starting at -1 for the right-most item. If the index is out of bounds, then the function expands to nothing.
<code>\tl_item:cn</code> *	

New: 2014-07-17

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ does not expand further when appearing in an `x`-type argument expansion.

<code>\tl_rand_item:N</code> *	<code>\tl_rand_item:N {tl var}</code>
<code>\tl_rand_item:c</code> *	<code>\tl_rand_item:n {(token list)}</code>
<code>\tl_rand_item:n</code> *	Selects a pseudo-random item of the $\langle token list \rangle$. If the $\langle token list \rangle$ is blank, the result is empty. This is not available in older versions of XeTeX.

New: 2016-12-06

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ does not expand further when appearing in an `x`-type argument expansion.

<code>\tl_range:Nnn</code> \star <code>\tl_range:nnn</code> \star	<code>\tl_range:Nnn</code> $\langle \text{tl var} \rangle$ $\{\langle \text{start index} \rangle\}$ $\{\langle \text{end index} \rangle\}$ <code>\tl_range:nnn</code> $\{\langle \text{token list} \rangle\}$ $\{\langle \text{start index} \rangle\}$ $\{\langle \text{end index} \rangle\}$
--	--

New: 2017-02-17
Updated: 2017-07-15

Leaves in the input stream the items from the $\langle \text{start index} \rangle$ to the $\langle \text{end index} \rangle$ inclusive. Spaces and braces are preserved between the items returned (but never at either end of the list). Here $\langle \text{start index} \rangle$ and $\langle \text{end index} \rangle$ should be $\langle \text{integer expressions} \rangle$. For describing in detail the functions' behavior, let m and n be the start and end index respectively. If either is 0, the result is empty. A positive index means 'start counting from the left end', and a negative index means 'from the right end'. Let l be the count of the token list.

The *actual start point* is determined as $M = m$ if $m > 0$ and as $M = l + m + 1$ if $m < 0$. Similarly the *actual end point* is $N = n$ if $n > 0$ and $N = l + n + 1$ if $n < 0$. If $M > N$, the result is empty. Otherwise it consists of all items from position M to position N inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions s for $s \leq 0$ or $s > l$.

Spaces in between items in the actual range are preserved. Spaces at either end of the token list will be removed anyway (think to the token list being passed to `\tl_trim_spaces:n` to begin with).

Thus, with $l = 7$ as in the examples below, all of the following are equivalent and result in the whole token list

```
\tl_range:nnn { abcd~{e{}}fg } { 1 } { 7 }
\tl_range:nnn { abcd~{e{}}fg } { 1 } { 12 }
\tl_range:nnn { abcd~{e{}}fg } { -7 } { 7 }
\tl_range:nnn { abcd~{e{}}fg } { -12 } { 7 }
```

Here are some more interesting examples. The calls

```
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { 2 } { 5 } }
\tl_range:nnn { abcd~{e{}}fg } { 2 } { -3 } }
\tl_range:nnn { abcd~{e{}}fg } { -6 } { 5 } }
\tl_range:nnn { abcd~{e{}}fg } { -6 } { -3 } }
```

are all equivalent and will print `bcd{e{}}` on the terminal; similarly

```
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { 2 } { 5 } }
\tl_range:nnn { abcd~{e{}}fg } { 2 } { -3 } }
\tl_range:nnn { abcd~{e{}}fg } { -6 } { 5 } }
\tl_range:nnn { abcd~{e{}}fg } { -6 } { -3 } }
```

are all equivalent and will print `bcd {e{}}` on the terminal (note the space in the middle). To the contrary,

```
\tl_range:nnn { abcd~{e{}}f } { 2 } { 4 }
```

will discard the space after 'd'.

If we want to get the items from, say, the third to the last in a token list $\langle \text{tl} \rangle$, the call is `\tl_range:nnn { <tl> } { 3 } { -1 }`. Similarly, for discarding the last item, we can do `\tl_range:nnn { <tl> } { 1 } { -2 }`.

For better performance, see `\tl_range_braced:nnn` and `\tl_range_unbraced:nnn`.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle \text{item} \rangle$ does not expand further when appearing in an `x`-type argument expansion.

11 Viewing token lists

`\tl_show:N`
`\tl_show:c`

Updated: 2015-08-01

`\tl_show:N` $\langle tl\ var \rangle$

Displays the content of the $\langle tl\ var \rangle$ on the terminal.

T_EXhackers note: This is similar to the T_EX primitive `\show`, wrapped to a fixed number of characters per line.

`\tl_show:n`

Updated: 2015-08-07

`\tl_show:n` $\{ \langle token\ list \rangle \}$

Displays the $\langle token\ list \rangle$ on the terminal.

T_EXhackers note: This is similar to the ϵ -T_EX primitive `\showtokens`, wrapped to a fixed number of characters per line.

`\tl_log:N`
`\tl_log:c`

New: 2014-08-22
Updated: 2015-08-01

`\tl_log:N` $\langle tl\ var \rangle$

Writes the content of the $\langle tl\ var \rangle$ in the log file. See also `\tl_show:N` which displays the result in the terminal.

`\tl_log:n`

New: 2014-08-22
Updated: 2015-08-07

`\tl_log:n` $\{ \langle token\ list \rangle \}$

Writes the $\langle token\ list \rangle$ in the log file. See also `\tl_show:n` which displays the result in the terminal.

12 Constant token lists

`\c_empty_tl`

Constant that is always empty.

`\c_novalue_tl`

New: 2017-11-14

A marker for the absence of an argument. This constant `tl` can safely be typeset (*cf.* `\q_nil`), with the result being `-NoValue-`. It is important to note that `\c_novalue_tl` is constructed such that it will *not* match the simple text input `-NoValue-`, *i.e.* that

`\tl_if_eq:NnTF \c_novalue_tl { -NoValue- }`

is logically **false**. The `\c_novalue_tl` marker is intended for use in creating document-level interfaces, where it serves as an indicator that an (optional) argument was omitted. In particular, it is distinct from a simple empty `tl`.

`\c_space_tl`

An explicit space character contained in a token list (compare this with `\c_space_token`). For use where an explicit space is required.

13 Scratch token lists

<hr/> <hr/>	
<code>\l_tmpa_tl</code>	
<code>\l_tmpb_tl</code>	

Scratch token lists for local assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

<hr/> <hr/>	
<code>\g_tmpa_tl</code>	
<code>\g_tmpb_tl</code>	

Scratch token lists for global assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Part VIII

The l3str package: Strings

TeX associates each character with a category code: as such, there is no concept of a “string” as commonly understood in many other programming languages. However, there are places where we wish to manipulate token lists while in some sense “ignoring” category codes: this is done by treating token lists as strings in a TeX sense.

A TeX string (and thus an expl3 string) is a series of characters which have category code 12 (“other”) with the exception of space characters which have category code 10 (“space”). Thus at a technical level, a TeX string is a token list with the appropriate category codes. In this documentation, these are simply referred to as strings.

String variables are simply specialised token lists, but by convention should be named with the suffix `...str`. Such variables should contain characters with category code 12 (other), except spaces, which have category code 10 (blank space). All the functions in this module which accept a token list argument first convert it to a string using `\tl_to_str:n` for internal processing, and do not treat a token list or the corresponding string representation differently.

As a string is a subset of the more general token list, it is sometimes unclear when one should be used over the other. Use a string variable for data that isn’t primarily intended for typesetting and for which a level of protection from unwanted expansion is suitable. This data type simplifies comparison of variables since there are no concerns about expansion of their contents.

The functions `\cs_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N` and `\token_to_str:N` (and variants) generate strings from the appropriate input: these are documented in `l3basics`, `l3tl` and `l3token`, respectively.

Most expandable functions in this module come in three flavours:

- `\str_...:N`, which expect a token list or string variable as their argument;
- `\str_...:n`, taking any token list (or string) as an argument;
- `\str_..._ignore_spaces:n`, which ignores any space encountered during the operation: these functions are typically faster than those which take care of escaping spaces appropriately.

1 Building strings

`\str_new:N``\str_new:c`

`New: 2015-09-18`

`\str_new:N <str var>`

Creates a new `<str var>` or raises an error if the name is already taken. The declaration is global. The `<str var>` is initially empty.

`\str_const:Nn``\str_const:(NV|Nx|cn|cV|cx)`

`New: 2015-09-18``Updated: 2018-07-28`

`\str_const:Nn <str var> {<token list>}`

Creates a new constant `<str var>` or raises an error if the name is already taken. The value of the `<str var>` is set globally to the `<token list>`, converted to a string.

<code>\str_clear:N</code>	<code>\str_clear:N <str var></code>
<code>\str_clear:c</code>	
<code>\str_gclear:N</code>	Clears the content of the $\langle str var \rangle$.
<code>\str_gclear:c</code>	
<hr/>	
New: 2015-09-18	

<code>\str_clear_new:N</code>	<code>\str_clear_new:N <str var></code>
<code>\str_clear_new:c</code>	
	Ensures that the $\langle str var \rangle$ exists globally by applying <code>\str_new:N</code> if necessary, then applies <code>\str_(g)clear:N</code> to leave the $\langle str var \rangle$ empty.
<hr/>	
New: 2015-09-18	

<code>\str_set_eq:NN</code>	<code>\str_set_eq:NN <str var₁> <str var₂></code>
<code>\str_set_eq:(cN Nc cc)</code>	
<code>\str_gset_eq:NN</code>	Sets the content of $\langle str var_1 \rangle$ equal to that of $\langle str var_2 \rangle$.
<code>\str_gset_eq:(cN Nc cc)</code>	
<hr/>	
New: 2015-09-18	

<code>\str_concat:NNN</code>	<code>\str_concat:NNN <str var₁> <str var₂> <str var₃></code>
<code>\str_concat:ccc</code>	
<code>\str_gconcat:NNN</code>	Concatenates the content of $\langle str var_2 \rangle$ and $\langle str var_3 \rangle$ together and saves the result in $\langle str var_1 \rangle$. The $\langle str var_2 \rangle$ is placed at the left side of the new string variable. The $\langle str var_2 \rangle$ and $\langle str var_3 \rangle$ must indeed be strings, as this function does not convert their contents to a string.
<code>\str_gconcat:ccc</code>	
<hr/>	
New: 2017-10-08	

2 Adding data to string variables

<code>\str_set:Nn</code>	<code>\str_set:Nn <str var> {<token list>}</code>
<code>\str_set:(NV Nx cn cV cx)</code>	
<code>\str_gset:Nn</code>	Converts the $\langle token list \rangle$ to a $\langle string \rangle$, and stores the result in $\langle str var \rangle$.
<code>\str_gset:(NV Nx cn cV cx)</code>	
<hr/>	
New: 2015-09-18	
Updated: 2018-07-28	

<code>\str_put_left:Nn</code>	<code>\str_put_left:Nn <str var> {<token list>}</code>
<code>\str_put_left:(NV Nx cn cV cx)</code>	
<code>\str_gput_left:Nn</code>	
<code>\str_gput_left:(NV Nx cn cV cx)</code>	
<hr/>	
New: 2015-09-18	
Updated: 2018-07-28	

Converts the $\langle token list \rangle$ to a $\langle string \rangle$, and prepends the result to $\langle str var \rangle$. The current contents of the $\langle str var \rangle$ are not automatically converted to a string.

<code>\str_put_right:Nn</code>	<code>\str_put_right:Nn <str var> {(token list)}</code>
<code>\str_put_right:(NV Nx cn cV cx)</code>	
<code>\str_gput_right:Nn</code>	
<code>\str_gput_right:(NV Nx cn cV cx)</code>	

New: 2015-09-18

Updated: 2018-07-28

Converts the $\langle token list \rangle$ to a $\langle string \rangle$, and appends the result to $\langle str var \rangle$. The current contents of the $\langle str var \rangle$ are not automatically converted to a string.

3 Modifying string variables

<code>\str_replace_once:Nnn</code>	<code>\str_replace_once:Nnn <str var> {(old)} {(new)}</code>
<code>\str_replace_once:cnn</code>	
<code>\str_greplace_once:Nnn</code>	
<code>\str_greplace_once:cnn</code>	

New: 2017-10-08

<code>\str_replace_all:Nnn</code>	<code>\str_replace_all:Nnn <str var> {(old)} {(new)}</code>
<code>\str_replace_all:cnn</code>	
<code>\str_greplace_all:Nnn</code>	
<code>\str_greplace_all:cnn</code>	

New: 2017-10-08

<code>\str_remove_once:Nn</code>	<code>\str_remove_once:Nn <str var> {(token list)}</code>
<code>\str_remove_once:cn</code>	
<code>\str_gremove_once:Nn</code>	
<code>\str_gremove_once:cn</code>	

New: 2017-10-08

<code>\str_remove_all:Nn</code>	<code>\str_remove_all:Nn <str var> {(token list)}</code>
<code>\str_remove_all:cn</code>	
<code>\str_gremove_all:Nn</code>	
<code>\str_gremove_all:cn</code>	

New: 2017-10-08

```
\str_set:Nn \l_tmpa_str {abbccd} \str_remove_all:Nn \l_tmpa_str
{bc}
```

results in `\l_tmpa_str` containing `abcd`.

4 String conditionals

<code>\str_if_exist_p:N</code>	★	<code>\str_if_exist_p:N</code> $\langle str\ var \rangle$
<code>\str_if_exist_p:c</code>	★	<code>\str_if_exist:N</code> $\langle str\ var \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\str_if_exist:N\underline{TF}</code>	★	Tests whether the $\langle str\ var \rangle$ is currently defined. This does not check that the $\langle str\ var \rangle$ really is a string.
<code>\str_if_exist:c\underline{TF}</code>	★	

New: 2015-09-18

<code>\str_if_empty_p:N</code>	★	<code>\str_if_empty_p:N</code> $\langle str\ var \rangle$
<code>\str_if_empty_p:c</code>	★	<code>\str_if_empty:N</code> $\langle str\ var \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\str_if_empty:N\underline{TF}</code>	★	Tests if the $\langle string\ variable \rangle$ is entirely empty (<i>i.e.</i> contains no characters at all).
<code>\str_if_empty:c\underline{TF}</code>	★	

New: 2015-09-18

<code>\str_if_eq_p:NN</code>	★	<code>\str_if_eq_p:NN</code> $\langle str\ var_1 \rangle$ $\langle str\ var_2 \rangle$
<code>\str_if_eq_p:(Nc cN cc)</code>	★	<code>\str_if_eq:N</code> $\langle str\ var_1 \rangle$ $\langle str\ var_2 \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\str_if_eq:NN\underline{TF}</code>	★	Compares the content of two $\langle str\ variables \rangle$ and is logically true if the two contain the same characters in the same order. See <code>\tl_if_eq:NN</code> to compare tokens (including their category codes) rather than characters.
<code>\str_if_eq:(Nc cN cc)\underline{TF}</code>	★	

New: 2015-09-18

<code>\str_if_eq_p:nn</code>	★	<code>\str_if_eq_p:nn</code> $\{\langle t_1 \rangle\}$ $\{\langle t_2 \rangle\}$
<code>\str_if_eq_p:(Vn on no nV VV vn nv ee)</code>	★	<code>\str_if_eq:nn</code> $\{\langle t_1 \rangle\}$ $\{\langle t_2 \rangle\}$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\str_if_eq:nn\underline{TF}</code>	★	
<code>\str_if_eq:(Vn on no nV VV vn nv ee)\underline{TF}</code>	★	

Updated: 2018-06-18

Compares the two $\langle token\ lists \rangle$ on a character by character basis (namely after converting them to strings), and is **true** if the two $\langle strings \rangle$ contain the same characters in the same order. Thus for example

`\str_if_eq_p:no { abc } { \tl_to_str:n { abc } }`

is logically **true**. See `\tl_if_eq:nn` to compare tokens (including their category codes) rather than characters.

<code>\str_if_in:Nn\underline{TF}</code>	<code>\str_if_in:N</code> $\langle str\ var \rangle$ $\{\langle token\ list \rangle\}$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\str_if_in:cn\underline{TF}</code>	Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$ and tests if that $\langle string \rangle$ is found in the content of the $\langle str\ var \rangle$.

New: 2017-10-08

<code>\str_if_in:nn\underline{TF}</code>	<code>\str_if_in:nn</code> $\langle t_1 \rangle$ $\{\langle t_2 \rangle\}$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
	Converts both $\langle token\ lists \rangle$ to $\langle strings \rangle$ and tests whether $\langle string_2 \rangle$ is found inside $\langle string_1 \rangle$.

New: 2017-10-08

<code>\str_case:nn</code>	★	<code>\str_case:nnTF {⟨test string⟩}</code>
<code>\str_case:(Vn on nV nv)</code>	★	{
<code>\str_case:nnTF</code>	★	{⟨string case ₁ ⟩} {⟨code case ₁ ⟩}
<code>\str_case:(Vn on nV nv)TF</code>	★	{⟨string case ₂ ⟩} {⟨code case ₂ ⟩}
		...
		{⟨string case _n ⟩} {⟨code case _n ⟩}
		}
		{⟨true code⟩}
		{⟨false code⟩}

New: 2013-07-24
Updated: 2015-02-28

Compares the *⟨test string⟩* in turn with each of the *⟨string cases⟩* (all token lists are converted to strings). If the two are equal (as described for `\str_if_eq:nnTF`) then the associated *⟨code⟩* is left in the input stream and other cases are discarded. If any of the cases are matched, the *⟨true code⟩* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *⟨false code⟩* is inserted. The function `\str_case:nn`, which does nothing if there is no match, is also available.

<code>\str_case_e:nn</code>	★	<code>\str_case_e:nnTF {⟨test string⟩}</code>
<code>\str_case_e:nnTF</code>	★	{
		{⟨string case ₁ ⟩} {⟨code case ₁ ⟩}
		{⟨string case ₂ ⟩} {⟨code case ₂ ⟩}
		...
		{⟨string case _n ⟩} {⟨code case _n ⟩}
		}
		{⟨true code⟩}
		{⟨false code⟩}

New: 2018-06-19

Compares the full expansion of the *⟨test string⟩* in turn with the full expansion of the *⟨string cases⟩* (all token lists are converted to strings). If the two full expansions are equal (as described for `\str_if_eq:nnTF`) then the associated *⟨code⟩* is left in the input stream and other cases are discarded. If any of the cases are matched, the *⟨true code⟩* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *⟨false code⟩* is inserted. The function `\str_case_e:nn`, which does nothing if there is no match, is also available. The *⟨test string⟩* is expanded in each comparison, and must always yield the same result: for example, random numbers must not be used within this string.

5 Mapping to strings

All mappings are done at the current group level, *i.e.* any local assignments made by the *⟨function⟩* or *⟨code⟩* discussed below remain in effect after the loop.

<code>\str_map_function:NN</code>	☆	<code>\str_map_function:NN ⟨str var⟩ ⟨function⟩</code>
<code>\str_map_function:cN</code>	☆	Applies <i>⟨function⟩</i> to every <i>⟨character⟩</i> in the <i>⟨str var⟩</i> including spaces. See also <code>\str_map_function:nN</code> .
<code>\str_map_function:nN</code>	☆	<code>\str_map_function:nN {⟨token list⟩} ⟨function⟩</code>
		Converts the <i>⟨token list⟩</i> to a <i>⟨string⟩</i> then applies <i>⟨function⟩</i> to every <i>⟨character⟩</i> in the <i>⟨string⟩</i> including spaces. See also <code>\str_map_function:NN</code> .

New: 2017-11-14

<hr/> <code>\str_map_inline:Nn</code> <code>\str_map_inline:cn</code> <hr/> New: 2017-11-14	<code>\str_map_inline:Nn <str var> {<inline function>}</code> Applies the <i><inline function></i> to every <i><character></i> in the <i><str var></i> including spaces. The <i><inline function></i> should consist of code which receives the <i><character></i> as #1. See also <code>\str_map_function:NN</code> .
<hr/> <code>\str_map_inline:nn</code> <hr/> New: 2017-11-14	<code>\str_map_inline:nn {<token list>} {<inline function>}</code> Converts the <i><token list></i> to a <i><string></i> then applies the <i><inline function></i> to every <i><character></i> in the <i><string></i> including spaces. The <i><inline function></i> should consist of code which receives the <i><character></i> as #1. See also <code>\str_map_function:NN</code> .
<hr/> <code>\str_map_variable:NNn</code> <code>\str_map_variable:cNn</code> <hr/> New: 2017-11-14	<code>\str_map_variable:NNn <str var> <variable> {<code>}</code> Stores each <i><character></i> of the <i><string></i> (including spaces) in turn in the (string or token list) <i><variable></i> and applies the <i><code></i> . The <i><code></i> will usually make use of the <i><variable></i> , but this is not enforced. The assignments to the <i><variable></i> are local. Its value after the loop is the last <i><character></i> in the <i><string></i> , or its original value if the <i><string></i> is empty. See also <code>\str_map_inline:Nn</code> .
<hr/> <code>\str_map_variable:nNn</code> <hr/> New: 2017-11-14	<code>\str_map_variable:nNn {<token list>} <variable> {<code>}</code> Converts the <i><token list></i> to a <i><string></i> then stores each <i><character></i> in the <i><string></i> (including spaces) in turn in the (string or token list) <i><variable></i> and applies the <i><code></i> . The <i><code></i> will usually make use of the <i><variable></i> , but this is not enforced. The assignments to the <i><variable></i> are local. Its value after the loop is the last <i><character></i> in the <i><string></i> , or its original value if the <i><string></i> is empty. See also <code>\str_map_inline:Nn</code> .
<hr/> <code>\str_map_break: ☆</code> <hr/> New: 2017-10-08	<code>\str_map_break:</code> Used to terminate a <code>\str_map...</code> function before all characters in the <i><string></i> have been processed. This normally takes place within a conditional statement, for example <pre> \str_map_inline:Nn \l_my_str { \str_if_eq:nnT { #1 } { bingo } { \str_map_break: } % Do something useful } </pre>

See also `\str_map_break:n`. Use outside of a `\str_map...` scenario leads to low level \TeX errors.

\TeX hackers note: When the mapping is broken, additional tokens may be inserted before continuing with the code that follows the loop. This depends on the design of the mapping function.

`\str_map_break:n` ☆

New: 2017-10-08

`\str_map_break:n {<code>}`

Used to terminate a `\str_map...` function before all characters in the *<string>* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\str_map_inline:Nn \l_my_str
{
  \str_if_eq:nnT { #1 } { bingo }
  { \str_map_break:n { <code> } }
  % Do something useful
}
```

Use outside of a `\str_map...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

6 Working with the content of strings

`\str_use:N` ★

`\str_use:c` ★

New: 2015-09-18

`\str_use:N <str var>`

Recovers the content of a *<str var>* and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a *<str>* directly without an accessor function.

<code>\str_count:N</code>	★	<code>\str_count:n {<token list>}</code>
<code>\str_count:c</code>	★	
<code>\str_count:n</code>	★	
<code>\str_count_ignore_spaces:n</code>	★	

New: 2015-09-18

Leaves in the input stream the number of characters in the string representation of *<token list>*, as an integer denotation. The functions differ in their treatment of spaces. In the case of `\str_count:N` and `\str_count:n`, all characters including spaces are counted. The `\str_count_ignore_spaces:n` function leaves the number of non-space characters in the input stream.

`\str_count_spaces:N` ★

`\str_count_spaces:c` ★

`\str_count_spaces:n` ★

New: 2015-09-18

`\str_count_spaces:n {<token list>}`

Leaves in the input stream the number of space characters in the string representation of *<token list>*, as an integer denotation. Of course, this function has no `_ignore_spaces` variant.

<code>\str_head:N</code>	★	<code>\str_head:n {⟨token list⟩}</code>
<code>\str_head:c</code>	★	
<code>\str_head:n</code>	★	
<code>\str_head_ignore_spaces:n</code>	★	

New: 2015-09-18

Converts the $\langle token\ list \rangle$ into a $\langle string \rangle$. The first character in the $\langle string \rangle$ is then left in the input stream, with category code “other”. The functions differ if the first character is a space: `\str_head:N` and `\str_head:n` return a space token with category code 10 (blank space), while the `\str_head_ignore_spaces:n` function ignores this space character and leaves the first non-space character in the input stream. If the $\langle string \rangle$ is empty (or only contains spaces in the case of the `_ignore_spaces` function), then nothing is left on the input stream.

<code>\str_tail:N</code>	★	<code>\str_tail:n {⟨token list⟩}</code>
<code>\str_tail:c</code>	★	
<code>\str_tail:n</code>	★	
<code>\str_tail_ignore_spaces:n</code>	★	

New: 2015-09-18

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, removes the first character, and leaves the remaining characters (if any) in the input stream, with category codes 12 and 10 (for spaces). The functions differ in the case where the first character is a space: `\str_tail:N` and `\str_tail:n` only trim that space, while `\str_tail_ignore_spaces:n` removes the first non-space character and any space before it. If the $\langle token\ list \rangle$ is empty (or blank in the case of the `_ignore_spaces` variant), then nothing is left on the input stream.

<code>\str_item:Nn</code>	★	<code>\str_item:nn {⟨token list⟩} {⟨integer expression⟩}</code>
<code>\str_item:nn</code>	★	
<code>\str_item_ignore_spaces:nn</code>	★	

New: 2015-09-18

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and leaves in the input stream the character in position $\langle integer\ expression \rangle$ of the $\langle string \rangle$, starting at 1 for the first (left-most) character. In the case of `\str_item:Nn` and `\str_item:nn`, all characters including spaces are taken into account. The `\str_item_ignore_spaces:nn` function skips spaces when counting characters. If the $\langle integer\ expression \rangle$ is negative, characters are counted from the end of the $\langle string \rangle$. Hence, -1 is the right-most character, *etc.*

<code>\str_range:Nnn</code>	<code>*</code>	<code>\str_range:nnn {⟨token list⟩} {⟨start index⟩} {⟨end index⟩}</code>
<code>\str_range:cnn</code>	<code>*</code>	
<code>\str_range:nnn</code>	<code>*</code>	
<code>\str_range_ignore_spaces:nnn</code>	<code>*</code>	

New: 2015-09-18

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and leaves in the input stream the characters from the $\langle start\ index \rangle$ to the $\langle end\ index \rangle$ inclusive. Spaces are preserved and counted as items (contrast this with `\tl_range:nnn` where spaces are not counted as items and are possibly discarded from the output).

Here $\langle start\ index \rangle$ and $\langle end\ index \rangle$ should be integer denotations. For describing in detail the functions' behavior, let m and n be the start and end index respectively. If either is 0, the result is empty. A positive index means 'start counting from the left end', a negative index means 'start counting from the right end'. Let l be the count of the token list.

The *actual start point* is determined as $M = m$ if $m > 0$ and as $M = l + m + 1$ if $m < 0$. Similarly the *actual end point* is $N = n$ if $n > 0$ and $N = l + n + 1$ if $n < 0$. If $M > N$, the result is empty. Otherwise it consists of all items from position M to position N inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions s for $s \leq 0$ or $s > l$. For instance,

```
\iow_term:x { \str_range:nnn { abcdef } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abcdef } { -4 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { -2 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { 0 } { -1 } }
```

prints bcde, cdef, ef, and an empty line to the terminal. The $\langle start\ index \rangle$ must always be smaller than or equal to the $\langle end\ index \rangle$: if this is not the case then no output is generated. Thus

```
\iow_term:x { \str_range:nnn { abcdef } { 5 } { 2 } }
\iow_term:x { \str_range:nnn { abcdef } { -1 } { -4 } }
```

both yield empty strings.

The behavior of `\str_range_ignore_spaces:nnn` is similar, but spaces are removed before starting the job. The input

```
\iow_term:x { \str_range:nnn { abcdefg } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abcdefg } { 2 } { -3 } }
\iow_term:x { \str_range:nnn { abcdefg } { -6 } { 5 } }
\iow_term:x { \str_range:nnn { abcdefg } { -6 } { -3 } }

\iow_term:x { \str_range:nnn { abc~efg } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abc~efg } { 2 } { -3 } }
\iow_term:x { \str_range:nnn { abc~efg } { -6 } { 5 } }
\iow_term:x { \str_range:nnn { abc~efg } { -6 } { -3 } }

\iow_term:x { \str_range_ignore_spaces:nnn { abcdefg } { 2 } { 5 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcdefg } { 2 } { -3 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcdefg } { -6 } { 5 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcdefg } { -6 } { -3 } }
```

```

\iow_term:x { \str_range_ignore_spaces:nnn { abcd~efg } { 2 } { 5 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcd~efg } { 2 } { -3 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcd~efg } { -6 } { 5 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcd~efg } { -6 } { -3 } }

```

will print four instances of `bcde`, four instances of `bc e` and eight instances of `bcde`.

7 String manipulation

```

\str_lowercase:n * \str_lowercase:n {<tokens>}
\str_lowercase:f * \str_uppercase:n {<tokens>}
\str_uppercase:n *
\str_uppercase:f *

```

New: 2019-11-26

Converts the input `<tokens>` to their string representation, as described for `\tl_to_str:n`, and then to the lower or upper case representation using a one-to-one mapping as described by the Unicode Consortium file `UnicodeData.txt`.

These functions are intended for case changing programmatic data in places where upper/lower case distinctions are meaningful. One example would be automatically generating a function name from user input where some case changing is needed. In this situation the input is programmatic, not textual, case does have meaning and a language-independent one-to-one mapping is appropriate. For example

```

\cs_new_protected:Npn \myfunc:nn #1#2
{
  \cs_set_protected:cpn
  {
    user
    \str_uppercase:f { \tl_head:n {#1} }
    \str_lowercase:f { \tl_tail:n {#1} }
  }
  { #2 }
}

```

would be used to generate a function with an auto-generated name consisting of the upper case equivalent of the supplied name followed by the lower case equivalent of the rest of the input.

These functions should *not* be used for

- Caseless comparisons: use `\str_foldcase:n` for this situation (case folding is distinct from lower casing).
- Case changing text for typesetting: see the `\text_lowercase:n(n)`, `\text_uppercase:n(n)` and `\text_titlecase:n(n)` functions which correctly deal with context-dependence and other factors appropriate to text case changing.

T_EXhackers note: As with all `expl3` functions, the input supported by `\str_foldcase:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with `pdfTEX` *only* the Latin alphabet characters A–Z are case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both `XYTEX` and `LuaTEX`.

<code>\str_foldcase:n</code> *	<code>\str_foldcase:n {<tokens>}</code>
--------------------------------	---

<code>\str_foldcase:V</code> *

New: 2019-11-26

Converts the input $\langle tokens \rangle$ to their string representation, as described for `\tl_to_str:n`, and then folds the case of the resulting $\langle string \rangle$ to remove case information. The result of this process is left in the input stream.

String folding is a process used for material such as identifiers rather than for “text”. The folding provided by `\str_foldcase:n` follows the mappings provided by the [Unicode Consortium](#), who [state](#):

Case folding is primarily used for caseless comparison of text, such as identifiers in a computer program, rather than actual text transformation. Case folding in Unicode is based on the lowercase mapping, but includes additional changes to the source text to help make it language-insensitive and consistent. As a result, case-folded text should be used solely for internal processing and generally should not be stored or displayed to the end user.

The folding approach implemented by `\str_foldcase:n` follows the “full” scheme defined by the Unicode Consortium (*e.g.* SSfolds to SS). As case-folding is a language-insensitive process, there is no special treatment of Turkic input (*i.e.* I always folds to i and not to ı).

TeXhackers note: As with all `expl3` functions, the input supported by `\str_foldcase:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with `pdfTeX` *only* the Latin alphabet characters A–Z are case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both `XYTeX` and `LuaTeX`, subject only to the fact that `XYTeX` in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with `\tl_to_str:n`.

8 Viewing strings

<code>\str_show:N</code>	<code>\str_show:N <str var></code>
--------------------------	--

<code>\str_show:c</code>

<code>\str_show:n</code>

Displays the content of the $\langle str var \rangle$ on the terminal.

New: 2015-09-18

<code>\str_log:N</code>	<code>\str_log:N <str var></code>
-------------------------	---

<code>\str_log:c</code>

<code>\str_log:n</code>

Writes the content of the $\langle str var \rangle$ in the log file.

New: 2019-02-15

9 Constant token lists

<code>\c_ampersand_str</code>	Constant strings, containing a single character token, with category code 12.
<code>\c_atsign_str</code>	
<code>\c_backslash_str</code>	
<code>\c_left_brace_str</code>	
<code>\c_right_brace_str</code>	
<code>\c_circumflex_str</code>	
<code>\c_colon_str</code>	
<code>\c_dollar_str</code>	
<code>\c_hash_str</code>	
<code>\c_percent_str</code>	
<code>\c_tilde_str</code>	
<code>\c_underscore_str</code>	

New: 2015-09-19

10 Scratch strings

<code>\l_tmpa_str</code>	Scratch strings for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_str</code>	

<code>\g_tmpa_str</code>	Scratch strings for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_str</code>	

Part IX

The `l3str-convert` package: string encoding conversions

1 Encoding and escaping schemes

Traditionally, string encodings only specify how strings of characters should be stored as bytes. However, the resulting lists of bytes are often to be used in contexts where only a restricted subset of bytes are permitted (*e.g.*, PDF string objects, URLs). Hence, storing a string of characters is done in two steps.

- The code points (“character codes”) are expressed as bytes following a given “encoding”. This can be UTF-16, ISO 8859-1, *etc.* See Table 1 for a list of supported encodings.⁵
- Bytes are translated to \TeX tokens through a given “escaping”. Those are defined for the most part by the pdf file format. See Table 2 for a list of escaping methods supported.⁵

2 Conversion functions

`\str_set_convert:Nnnn`
`\str_gset_convert:Nnnn`

`\str_set_convert:Nnnn <str var> {<string>} {<name 1>} {<name 2>}`

This function converts the $\langle string \rangle$ from the encoding given by $\langle name 1 \rangle$ to the encoding given by $\langle name 2 \rangle$, and stores the result in the $\langle str var \rangle$. Each $\langle name \rangle$ can have the form $\langle encoding \rangle$ or $\langle encoding \rangle / \langle escaping \rangle$, where the possible values of $\langle encoding \rangle$ and $\langle escaping \rangle$ are given in Tables 1 and 2, respectively. The default escaping is to input and output bytes directly. The special case of an empty $\langle name \rangle$ indicates the use of “native” strings, 8-bit for pdf \TeX , and Unicode strings for the other two engines.

For example,

```
\str_set_convert:Nnnn \l_foo_str { Hello! } { } { utf16/hex }
```

results in the variable `\l_foo_str` holding the string `FEFF00480065006C006C006F0021`. This is obtained by converting each character in the (native) string `Hello!` to the UTF-16 encoding, and expressing each byte as a pair of hexadecimal digits. Note the presence of a (big-endian) byte order mark “FEFF”, which can be avoided by specifying the encoding `utf16be/hex`.

An error is raised if the $\langle string \rangle$ is not valid according to the $\langle escaping 1 \rangle$ and $\langle encoding 1 \rangle$, or if it cannot be reencoded in the $\langle encoding 2 \rangle$ and $\langle escaping 2 \rangle$ (for instance, if a character does not exist in the $\langle encoding 2 \rangle$). Erroneous input is replaced by the Unicode replacement character “FFFD”, and characters which cannot be reencoded are replaced by either the replacement character “FFFD if it exists in the $\langle encoding 2 \rangle$, or an encoding-specific replacement character, or the question mark character.

⁵Encodings and escapings will be added as they are requested.

Table 1: Supported encodings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the encoding in this list.

<i>⟨Encoding⟩</i>	description
<code>utf8</code>	UTF-8
<code>utf16</code>	UTF-16, with byte-order mark
<code>utf16be</code>	UTF-16, big-endian
<code>utf16le</code>	UTF-16, little-endian
<code>utf32</code>	UTF-32, with byte-order mark
<code>utf32be</code>	UTF-32, big-endian
<code>utf32le</code>	UTF-32, little-endian
<code>iso88591, latin1</code>	ISO 8859-1
<code>iso88592, latin2</code>	ISO 8859-2
<code>iso88593, latin3</code>	ISO 8859-3
<code>iso88594, latin4</code>	ISO 8859-4
<code>iso88595</code>	ISO 8859-5
<code>iso88596</code>	ISO 8859-6
<code>iso88597</code>	ISO 8859-7
<code>iso88598</code>	ISO 8859-8
<code>iso88599, latin5</code>	ISO 8859-9
<code>iso885910, latin6</code>	ISO 8859-10
<code>iso885911</code>	ISO 8859-11
<code>iso885913, latin7</code>	ISO 8859-13
<code>iso885914, latin8</code>	ISO 8859-14
<code>iso885915, latin9</code>	ISO 8859-15
<code>iso885916, latin10</code>	ISO 8859-16
<code>clist</code>	comma-list of integers
<i>⟨empty⟩</i>	native (Unicode) string

Table 2: Supported escapings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the escaping in this list.

<i>⟨Escaping⟩</i>	description
<code>bytes</code> , or <code>empty</code>	arbitrary bytes
<code>hex</code> , <code>hexadecimal</code>	byte = two hexadecimal digits
<code>name</code>	see <code>\pdfescapename</code>
<code>string</code>	see <code>\pdfescapestring</code>
<code>url</code>	encoding used in URLs

<code>\str_set_convert:NnnnTF</code> <code>\str_gset_convert:NnnnTF</code>	<code>\str_set_convert:NnnnTF <str var> {<string>} {<name 1>} {<name 2>} {<true code>} {<false code>}</code>
---	--

As `\str_set_convert:Nnnn`, converts the $\langle string \rangle$ from the encoding given by $\langle name 1 \rangle$ to the encoding given by $\langle name 2 \rangle$, and assigns the result to $\langle str var \rangle$. Contrarily to `\str_set_convert:Nnnn`, the conditional variant does not raise errors in case the $\langle string \rangle$ is not valid according to the $\langle name 1 \rangle$ encoding, or cannot be expressed in the $\langle name 2 \rangle$ encoding. Instead, the $\langle false code \rangle$ is performed.

3 Conversion by expansion (for PDF contexts)

A small number of expandable functions are provided for use in PDF string/name contexts. These *assume UTF-8* and *no escaping* in the input.

<code>\str_convert_pdfname:n *</code>	<code>\str_convert_pdfname:n <string></code>
---------------------------------------	--

As `\str_set_convert:Nnnn`, converts the $\langle string \rangle$ on a byte-by-byte basis with non-ASCII codepoints escaped using hashes.

4 Creating 8-bit mappings

<code>\str_declare_eight_bit_encoding:nnn</code>	<code>\str_declare_eight_bit_encoding:nnn {<name>} {<mapping>} {<missing>}</code>
--	---

Declares the encoding $\langle name \rangle$ to map bytes to Unicode characters according to the $\langle mapping \rangle$, and map those bytes which are not mentioned in the $\langle mapping \rangle$ either to the replacement character (if they appear in $\langle missing \rangle$), or to themselves.

5 Possibilities, and things to do

Encoding/escaping-related tasks.

- In X_YTeX/LuaTeX, would it be better to use the `^^^...` approach to build a string from a given list of character codes? Namely, within a group, assign 0-9a-f and all characters we want to category “other”, then assign `^` the category superscript, and use `\scantokens`.
- Change `\str_set_convert:Nnnn` to expand its last two arguments.
- Describe the internal format in the code comments. Refuse code points in [“D800,”DFFF] in the internal representation?
- Add documentation about each encoding and escaping method, and add examples.
- The `hex` unescaping should raise an error for odd-token count strings.
- Decide what bytes should be escaped in the `url` escaping. Perhaps the characters `! ’ () * - . / 0 1 2 3 4 5 6 7 8 9 _` are safe, and all other characters should be escaped?
- Automate generation of 8-bit mapping files.

- Change the framework for 8-bit encodings: for decoding from 8-bit to Unicode, use 256 integer registers; for encoding, use a tree-box.
- More encodings (see Heiko's `stringenc`). CESU?
- More escapings: ASCII85, shell escapes, lua escapes, *etc.*?

Part X

The l3seq package

Sequences and stacks

L^AT_EX3 implements a “sequence” data type, which contain an ordered list of entries which may contain any *balanced text*. It is possible to map functions to sequences such that the function is applied to every item in the sequence.

Sequences are also used to implement stack functions in L^AT_EX3. This is achieved using a number of dedicated stack functions.

1 Creating and initialising sequences

<code>\seq_new:N</code>	<code>\seq_new:N <sequence></code>
<code>\seq_new:c</code>	

Creates a new *<sequence>* or raises an error if the name is already taken. The declaration is global. The *<sequence>* initially contains no items.

<code>\seq_clear:N</code>	<code>\seq_clear:N <sequence></code>
<code>\seq_clear:c</code>	
<code>\seq_gclear:N</code>	Clears all items from the <i><sequence></i> .
<code>\seq_gclear:c</code>	

<code>\seq_clear_new:N</code>	<code>\seq_clear_new:N <sequence></code>
<code>\seq_clear_new:c</code>	
<code>\seq_gclear_new:N</code>	Ensures that the <i><sequence></i> exists globally by applying <code>\seq_new:N</code> if necessary, then applies <code>\seq_(g)clear:N</code> to leave the <i><sequence></i> empty.
<code>\seq_gclear_new:c</code>	

<code>\seq_set_eq:NN</code>	<code>\seq_set_eq:NN <sequence₁> <sequence₂></code>
<code>\seq_set_eq:(cN Nc cc)</code>	
<code>\seq_gset_eq:NN</code>	Sets the content of <i><sequence₁></i> equal to that of <i><sequence₂></i> .
<code>\seq_gset_eq:(cN Nc cc)</code>	

<code>\seq_set_from_clist:NN</code>	<code>\seq_set_from_clist:NN <sequence> <comma-list></code>
<code>\seq_set_from_clist:(cN Nc cc)</code>	
<code>\seq_set_from_clist:Nn</code>	
<code>\seq_set_from_clist:cn</code>	
<code>\seq_gset_from_clist:NN</code>	
<code>\seq_gset_from_clist:(cN Nc cc)</code>	
<code>\seq_gset_from_clist:Nn</code>	
<code>\seq_gset_from_clist:cn</code>	

New: 2014-07-17

Converts the data in the *<comma list>* into a *<sequence>*: the original *<comma list>* is unchanged.

`\seq_const_from_clist:Nn`
`\seq_const_from_clist:cn`

New: 2017-11-28

`\seq_const_from_clist:Nn` $\langle seq\ var \rangle$ $\{\langle comma-list \rangle\}$

Creates a new constant $\langle seq\ var \rangle$ or raises an error if the name is already taken. The $\langle seq\ var \rangle$ is set globally to contain the items in the $\langle comma\ list \rangle$.

`\seq_set_split:Nnn`
`\seq_set_split:NnV`
`\seq_gset_split:Nnn`
`\seq_gset_split:NnV`

New: 2011-08-15
Updated: 2012-07-02

`\seq_set_split:Nnn` $\langle sequence \rangle$ $\{\langle delimiter \rangle\}$ $\{\langle token\ list \rangle\}$

Splits the $\langle token\ list \rangle$ into $\langle items \rangle$ separated by $\langle delimiter \rangle$, and assigns the result to the $\langle sequence \rangle$. Spaces on both sides of each $\langle item \rangle$ are ignored, then one set of outer braces is removed (if any); this space trimming behaviour is identical to that of `\l3clist` functions. Empty $\langle items \rangle$ are preserved by `\seq_set_split:Nnn`, and can be removed afterwards using `\seq_remove_all:Nn` $\langle sequence \rangle$ $\{\}$. The $\langle delimiter \rangle$ may not contain `{`, `}` or `#` (assuming \TeX 's normal category code régime). If the $\langle delimiter \rangle$ is empty, the $\langle token\ list \rangle$ is split into $\langle items \rangle$ as a $\langle token\ list \rangle$.

`\seq_concat:NNN`
`\seq_concat:ccc`
`\seq_gconcat:NNN`
`\seq_gconcat:ccc`

`\seq_concat:NNN` $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$ $\langle sequence_3 \rangle$

Concatenates the content of $\langle sequence_2 \rangle$ and $\langle sequence_3 \rangle$ together and saves the result in $\langle sequence_1 \rangle$. The items in $\langle sequence_2 \rangle$ are placed at the left side of the new sequence.

`\seq_if_exist_p:N *`
`\seq_if_exist_p:c *`
`\seq_if_exist:NTF *`
`\seq_if_exist:cTF *`

New: 2012-03-03

`\seq_if_exist_p:N` $\langle sequence \rangle$

`\seq_if_exist:NNTF` $\langle sequence \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests whether the $\langle sequence \rangle$ is currently defined. This does not check that the $\langle sequence \rangle$ really is a sequence variable.

2 Appending data to sequences

`\seq_put_left:Nn`
`\seq_put_left:(NV|Nv|No|Nx|cn|cV|cv|co|cx)`
`\seq_gput_left:Nn`
`\seq_gput_left:(NV|Nv|No|Nx|cn|cV|cv|co|cx)`

`\seq_put_left:Nn` $\langle sequence \rangle$ $\{\langle item \rangle\}$

Appends the $\langle item \rangle$ to the left of the $\langle sequence \rangle$.

`\seq_put_right:Nn`
`\seq_put_right:(NV|Nv|No|Nx|cn|cV|cv|co|cx)`
`\seq_gput_right:Nn`
`\seq_gput_right:(NV|Nv|No|Nx|cn|cV|cv|co|cx)`

`\seq_put_right:Nn` $\langle sequence \rangle$ $\{\langle item \rangle\}$

Appends the $\langle item \rangle$ to the right of the $\langle sequence \rangle$.

3 Recovering items from sequences

Items can be recovered from either the left or the right of sequences. For implementation reasons, the actions at the left of the sequence are faster than those acting on the right. These functions all assign the recovered material locally, *i.e.* setting the $\langle token\ list\ variable \rangle$ used with `\tl_set:Nn` and *never* `\tl_gset:Nn`.

<hr/> <code>\seq_get_left:NN</code> <code>\seq_get_left:cN</code> <hr/> Updated: 2012-05-14 <hr/>	<code>\seq_get_left:NN</code> $\langle sequence \rangle$ $\langle token list variable \rangle$ Stores the left-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token list variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker <code>\q_no_value</code> .
<hr/> <code>\seq_get_right:NN</code> <code>\seq_get_right:cN</code> <hr/> Updated: 2012-05-19 <hr/>	<code>\seq_get_right:NN</code> $\langle sequence \rangle$ $\langle token list variable \rangle$ Stores the right-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token list variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker <code>\q_no_value</code> .
<hr/> <code>\seq_pop_left:NN</code> <code>\seq_pop_left:cN</code> <hr/> Updated: 2012-05-14 <hr/>	<code>\seq_pop_left:NN</code> $\langle sequence \rangle$ $\langle token list variable \rangle$ Pops the left-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token list variable \rangle$. Both of the variables are assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker <code>\q_no_value</code> .
<hr/> <code>\seq_gpop_left:NN</code> <code>\seq_gpop_left:cN</code> <hr/> Updated: 2012-05-14 <hr/>	<code>\seq_gpop_left:NN</code> $\langle sequence \rangle$ $\langle token list variable \rangle$ Pops the left-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token list variable \rangle$. The $\langle sequence \rangle$ is modified globally, while the assignment of the $\langle token list variable \rangle$ is local. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker <code>\q_no_value</code> .
<hr/> <code>\seq_pop_right:NN</code> <code>\seq_pop_right:cN</code> <hr/> Updated: 2012-05-19 <hr/>	<code>\seq_pop_right:NN</code> $\langle sequence \rangle$ $\langle token list variable \rangle$ Pops the right-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token list variable \rangle$. Both of the variables are assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker <code>\q_no_value</code> .
<hr/> <code>\seq_gpop_right:NN</code> <code>\seq_gpop_right:cN</code> <hr/> Updated: 2012-05-19 <hr/>	<code>\seq_gpop_right:NN</code> $\langle sequence \rangle$ $\langle token list variable \rangle$ Pops the right-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token list variable \rangle$. The $\langle sequence \rangle$ is modified globally, while the assignment of the $\langle token list variable \rangle$ is local. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker <code>\q_no_value</code> .
<hr/> <code>\seq_item:Nn</code> ★ <code>\seq_item:cn</code> ★ <hr/> New: 2014-07-17 <hr/>	<code>\seq_item:Nn</code> $\langle sequence \rangle$ $\{ \langle integer expression \rangle \}$ Indexing items in the $\langle sequence \rangle$ from 1 at the top (left), this function evaluates the $\langle integer expression \rangle$ and leaves the appropriate item from the sequence in the input stream. If the $\langle integer expression \rangle$ is negative, indexing occurs from the bottom (right) of the sequence. If the $\langle integer expression \rangle$ is larger than the number of items in the $\langle sequence \rangle$ (as calculated by <code>\seq_count:N</code>) then the function expands to nothing.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ does not expand further when appearing in an x-type argument expansion.

`\seq_rand_item:N` ★
`\seq_rand_item:c` ★

New: 2016-12-06

`\seq_rand_item:N` $\langle seq\ var \rangle$

Selects a pseudo-random item of the $\langle sequence \rangle$. If the $\langle sequence \rangle$ is empty the result is empty. This is not available in older versions of Xe_{La}TeX.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ does not expand further when appearing in an x-type argument expansion.

4 Recovering values from sequences with branching

The functions in this section combine tests for non-empty sequences with recovery of an item from the sequence. They offer increased readability and performance over separate testing and recovery phases.

`\seq_get_left:NNTF`
`\seq_get_left:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_get_left:NNTF` $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

If the $\langle sequence \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle sequence \rangle$ is non-empty, stores the left-most item from the $\langle sequence \rangle$ in the $\langle token\ list\ variable \rangle$ without removing it from the $\langle sequence \rangle$, then leaves the $\langle true\ code \rangle$ in the input stream. The $\langle token\ list\ variable \rangle$ is assigned locally.

`\seq_get_right:NNTF`
`\seq_get_right:cNTF`

New: 2012-05-19

`\seq_get_right:NNTF` $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

If the $\langle sequence \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle sequence \rangle$ is non-empty, stores the right-most item from the $\langle sequence \rangle$ in the $\langle token\ list\ variable \rangle$ without removing it from the $\langle sequence \rangle$, then leaves the $\langle true\ code \rangle$ in the input stream. The $\langle token\ list\ variable \rangle$ is assigned locally.

`\seq_pop_left:NNTF`
`\seq_pop_left:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_pop_left:NNTF` $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

If the $\langle sequence \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle sequence \rangle$ is non-empty, pops the left-most item from the $\langle sequence \rangle$ in the $\langle token\ list\ variable \rangle$, *i.e.* removes the item from the $\langle sequence \rangle$, then leaves the $\langle true\ code \rangle$ in the input stream. Both the $\langle sequence \rangle$ and the $\langle token\ list\ variable \rangle$ are assigned locally.

`\seq_gpop_left:NNTF`
`\seq_gpop_left:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_gpop_left:NNTF` $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

If the $\langle sequence \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle sequence \rangle$ is non-empty, pops the left-most item from the $\langle sequence \rangle$ in the $\langle token\ list\ variable \rangle$, *i.e.* removes the item from the $\langle sequence \rangle$, then leaves the $\langle true\ code \rangle$ in the input stream. The $\langle sequence \rangle$ is modified globally, while the $\langle token\ list\ variable \rangle$ is assigned locally.

<code>\seq_pop_right:nnTF</code>	<code>\seq_pop_right:nnTF <sequence> <token list variable> {<true code>} {<false code>}</code>
<code>\seq_pop_right:cnTF</code>	
New: 2012-05-19	

If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<sequence>* is non-empty, pops the right-most item from the *<sequence>* in the *<token list variable>*, *i.e.* removes the item from the *<sequence>*, then leaves the *<true code>* in the input stream. Both the *<sequence>* and the *<token list variable>* are assigned locally.

<code>\seq_gpop_right:nnTF</code>	<code>\seq_gpop_right:nnTF <sequence> <token list variable> {<true code>} {<false code>}</code>
<code>\seq_gpop_right:cnTF</code>	
New: 2012-05-19	

If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<sequence>* is non-empty, pops the right-most item from the *<sequence>* in the *<token list variable>*, *i.e.* removes the item from the *<sequence>*, then leaves the *<true code>* in the input stream. The *<sequence>* is modified globally, while the *<token list variable>* is assigned locally.

5 Modifying sequences

While sequences are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update sequences, while retaining the order of the unaffected entries.

<code>\seq_remove_duplicates:n</code>	<code>\seq_remove_duplicates:n <sequence></code>
<code>\seq_remove_duplicates:c</code>	
<code>\seq_gremove_duplicates:n</code>	Removes duplicate items from the <i><sequence></i> , leaving the left most copy of each item in the <i><sequence></i> . The <i><item></i> comparison takes place on a token basis, as for <code>\tl_if_eq:nnTF</code> .
<code>\seq_gremove_duplicates:c</code>	

TeXhackers note: This function iterates through every item in the *<sequence>* and does a comparison with the *<items>* already checked. It is therefore relatively slow with large sequences.

<code>\seq_remove_all:nn</code>	<code>\seq_remove_all:nn <sequence> {<item>}</code>
<code>\seq_remove_all:cn</code>	
<code>\seq_gremove_all:nn</code>	Removes every occurrence of <i><item></i> from the <i><sequence></i> . The <i><item></i> comparison takes place on a token basis, as for <code>\tl_if_eq:nnTF</code> .
<code>\seq_gremove_all:cn</code>	

<code>\seq_reverse:n</code>	<code>\seq_reverse:n <sequence></code>
<code>\seq_reverse:c</code>	
<code>\seq_greverse:n</code>	Reverses the order of the items stored in the <i><sequence></i> .
<code>\seq_greverse:c</code>	

New: 2014-07-18

<code>\seq_sort:nn</code>	<code>\seq_sort:nn <sequence> {<comparison code>}</code>
<code>\seq_sort:cn</code>	
<code>\seq_gsort:nn</code>	Sorts the items in the <i><sequence></i> according to the <i><comparison code></i> , and assigns the result to <i><sequence></i> . The details of sorting comparison are described in Section 1.
<code>\seq_gsort:cn</code>	

New: 2017-02-06

```
\seq_shuffle:N
\seq_shuffle:c
\seq_gshuffle:N
\seq_gshuffle:c
```

New: 2018-04-29

```
\seq_shuffle:N <seq var>
```

Sets the $\langle seq\ var \rangle$ to the result of placing the items of the $\langle seq\ var \rangle$ in a random order. Each item is (roughly) as likely to end up in any given position.

T_EXhackers note: For sequences with more than 13 items or so, only a small proportion of all possible permutations can be reached, because the random seed `\sys_rand_seed` only has 28-bits. The use of `\toks` internally means that sequences with more than 32767 or 65535 items (depending on the engine) cannot be shuffled.

6 Sequence conditionals

```
\seq_if_empty_p:N *
\seq_if_empty_p:c *
\seq_if_empty:NTF *
\seq_if_empty:cTF *
```

```
\seq_if_empty_p:N <sequence>
```

```
\seq_if_empty:NTF <sequence> {\true code} {\false code}
```

Tests if the $\langle sequence \rangle$ is empty (containing no items).

```
\seq_if_in:NnTF
```

```
\seq_if_in:NnTF <sequence> {\item} {\true code} {\false code}
```

```
\seq_if_in:(NV|Nv|No|Nx|cn|cV|cv|co|cx)TF
```

Tests if the $\langle item \rangle$ is present in the $\langle sequence \rangle$.

7 Mapping to sequences

All mappings are done at the current group level, *i.e.* any local assignments made by the $\langle function \rangle$ or $\langle code \rangle$ discussed below remain in effect after the loop.

```
\seq_map_function:NN ☆
\seq_map_function:cN ☆
```

Updated: 2012-06-29

```
\seq_map_function:NN <sequence> <function>
```

Applies $\langle function \rangle$ to every $\langle item \rangle$ stored in the $\langle sequence \rangle$. The $\langle function \rangle$ will receive one argument for each iteration. The $\langle items \rangle$ are returned from left to right. To pass further arguments to the $\langle function \rangle$, see `\seq_map_tokens:Nn`. The function `\seq_map_inline:Nn` is faster than `\seq_map_function:NN` for sequences with more than about 10 items.

```
\seq_map_inline:Nn
\seq_map_inline:cn
```

Updated: 2012-06-29

```
\seq_map_inline:Nn <sequence> {\inline function}
```

Applies $\langle inline\ function \rangle$ to every $\langle item \rangle$ stored within the $\langle sequence \rangle$. The $\langle inline\ function \rangle$ should consist of code which will receive the $\langle item \rangle$ as #1. The $\langle items \rangle$ are returned from left to right.

`\seq_map_tokens:Nn` ☆

`\seq_map_tokens:cn` ☆

New: 2019-08-30

`\seq_map_tokens:Nn` $\langle sequence \rangle$ $\{ \langle code \rangle \}$

Analogue of `\seq_map_function:NN` which maps several tokens instead of a single function. The $\langle code \rangle$ receives each item in the $\langle sequence \rangle$ as two trailing brace groups. For instance,

`\seq_map_tokens:Nn \l_my_seq { \prg_replicate:nn { 2 } }`

expands to twice each item in the $\langle sequence \rangle$: for each item in `\l_my_seq` the function `\prg_replicate:nn` receives 2 and $\langle item \rangle$ as its two arguments. The function `\seq_map_inline:Nn` is typically faster but is not expandable.

`\seq_map_variable:NNn`

`\seq_map_variable:(Ncn|cNn|ccn)`

Updated: 2012-06-29

`\seq_map_variable:NNn` $\langle sequence \rangle$ $\langle variable \rangle$ $\{ \langle code \rangle \}$

Stores each $\langle item \rangle$ of the $\langle sequence \rangle$ in turn in the (token list) $\langle variable \rangle$ and applies the $\langle code \rangle$. The $\langle code \rangle$ will usually make use of the $\langle variable \rangle$, but this is not enforced. The assignments to the $\langle variable \rangle$ are local. Its value after the loop is the last $\langle item \rangle$ in the $\langle sequence \rangle$, or its original value if the $\langle sequence \rangle$ is empty. The $\langle items \rangle$ are returned from left to right.

`\seq_map_indexed_function:NN` ☆

`\seq_map_indexed_function:NN` $\langle seq var \rangle$ $\langle function \rangle$

New: 2018-05-03

Applies $\langle function \rangle$ to every entry in the $\langle sequence variable \rangle$. The $\langle function \rangle$ should have signature `:nn`. It receives two arguments for each iteration: the $\langle index \rangle$ (namely 1 for the first entry, then 2 and so on) and the $\langle item \rangle$.

`\seq_map_indexed_inline:Nn`

New: 2018-05-03

`\seq_map_indexed_inline:Nn` $\langle seq var \rangle$ $\{ \langle inline function \rangle \}$

Applies $\langle inline function \rangle$ to every entry in the $\langle sequence variable \rangle$. The $\langle inline function \rangle$ should consist of code which receives the $\langle index \rangle$ (namely 1 for the first entry, then 2 and so on) as #1 and the $\langle item \rangle$ as #2.

`\seq_map_break:` ☆

Updated: 2012-06-29

`\seq_map_break:`

Used to terminate a `\seq_map...` function before all entries in the $\langle sequence \rangle$ have been processed. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \seq_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\seq_map...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\seq_map_break:n` ☆

Updated: 2012-06-29

`\seq_map_break:n {<code>}`

Used to terminate a `\seq_map...` function before all entries in the $\langle sequence \rangle$ have been processed, inserting the $\langle code \rangle$ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \seq_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\seq_map...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the $\langle code \rangle$ is inserted into the input stream. This depends on the design of the mapping function.

`\seq_set_map:NNn`
`\seq_gset_map:NNn`

New: 2011-12-22
Updated: 2020-07-16

`\seq_set_map:NNn <sequence1> <sequence2> {<inline function>}`

Applies $\langle inline function \rangle$ to every $\langle item \rangle$ stored within the $\langle sequence_2 \rangle$. The $\langle inline function \rangle$ should consist of code which will receive the $\langle item \rangle$ as #1. The sequence resulting applying $\langle inline function \rangle$ to each $\langle item \rangle$ is assigned to $\langle sequence_1 \rangle$.

TeXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level TeX errors.

`\seq_set_map_x:Nn`
`\seq_gset_map_x:Nn`

New: 2020-07-16

`\seq_set_map_x:Nn` $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$ $\{\langle inline function \rangle\}$

Applies $\langle inline function \rangle$ to every $\langle item \rangle$ stored within the $\langle sequence_2 \rangle$. The $\langle inline function \rangle$ should consist of code which will receive the $\langle item \rangle$ as #1. The sequence resulting from x-expanding $\langle inline function \rangle$ applied to each $\langle item \rangle$ is assigned to $\langle sequence_1 \rangle$. As such, the code in $\langle inline function \rangle$ should be expandable.

T_EXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level T_EX errors.

`\seq_count:N` ★
`\seq_count:c` ★

New: 2012-07-13

`\seq_count:N` $\langle sequence \rangle$

Leaves the number of items in the $\langle sequence \rangle$ in the input stream as an $\langle integer denotation \rangle$. The total number of items in a $\langle sequence \rangle$ includes those which are empty and duplicates, *i.e.* every item in a $\langle sequence \rangle$ is unique.

8 Using the content of sequences directly

`\seq_use:Nnnn` ★
`\seq_use:cnnn` ★

New: 2013-05-26

`\seq_use:Nnnn` $\langle seq var \rangle$ $\{\langle separator between two \rangle\}$
 $\{\langle separator between more than two \rangle\}$ $\{\langle separator between final two \rangle\}$

Places the contents of the $\langle seq var \rangle$ in the input stream, with the appropriate $\langle separator \rangle$ between the items. Namely, if the sequence has more than two items, the $\langle separator between more than two \rangle$ is placed between each pair of items except the last, for which the $\langle separator between final two \rangle$ is used. If the sequence has exactly two items, then they are placed in the input stream separated by the $\langle separator between two \rangle$. If the sequence has a single item, it is placed in the input stream, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nnnn \l_tmpa_seq { ~and~ } { ,~ } { ,~and~ }
```

inserts “a, b, c, de, and f” in the input stream. The first separator argument is not used in this case because the sequence has more than 2 items.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle items \rangle$ do not expand further when appearing in an x-type argument expansion.

`\seq_use:Nn` ★
`\seq_use:cn` ★

New: 2013-05-26

`\seq_use:Nn` $\langle seq\ var \rangle$ $\{\langle separator \rangle\}$

Places the contents of the $\langle seq\ var \rangle$ in the input stream, with the $\langle separator \rangle$ between the items. If the sequence has a single item, it is placed in the input stream with no $\langle separator \rangle$, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nn \l_tmpa_seq { ~and~ }
```

inserts “a and b and c and de and f” in the input stream.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle items \rangle$ do not expand further when appearing in an `x`-type argument expansion.

9 Sequences as stacks

Sequences can be used as stacks, where data is pushed to and popped from the top of the sequence. (The left of a sequence is the top, for performance reasons.) The stack functions for sequences are not intended to be mixed with the general ordered data functions detailed in the previous section: a sequence should either be used as an ordered data type or as a stack, but not in both ways.

`\seq_get:NN`
`\seq_get:cn`

Updated: 2012-05-14

`\seq_get:NN` $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$

Reads the top item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token\ list\ variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ is set to the special marker `\q_no_value`.

`\seq_pop:NN`
`\seq_pop:cn`

Updated: 2012-05-14

`\seq_pop:NN` $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$

Pops the top item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$. Both of the variables are assigned locally. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ is set to the special marker `\q_no_value`.

`\seq_gpop:NN`
`\seq_gpop:cn`

Updated: 2012-05-14

`\seq_gpop:NN` $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$

Pops the top item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$. The $\langle sequence \rangle$ is modified globally, while the $\langle token\ list\ variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ is set to the special marker `\q_no_value`.

`\seq_get:NNTF`
`\seq_get:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_get:NNTF` $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

If the $\langle sequence \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle sequence \rangle$ is non-empty, stores the top item from a $\langle sequence \rangle$ in the $\langle token\ list\ variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token\ list\ variable \rangle$ is assigned locally.

`\seq_pop:NNTF`
`\seq_pop:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_pop:NNTF <sequence> <token list variable> {(true code)} {(false code)}`

If the `<sequence>` is empty, leaves the `<false code>` in the input stream. The value of the `<token list variable>` is not defined in this case and should not be relied upon. If the `<sequence>` is non-empty, pops the top item from the `<sequence>` in the `<token list variable>`, *i.e.* removes the item from the `<sequence>`. Both the `<sequence>` and the `<token list variable>` are assigned locally.

`\seq_gpop:NNTF`
`\seq_gpop:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_gpop:NNTF <sequence> <token list variable> {(true code)} {(false code)}`

If the `<sequence>` is empty, leaves the `<false code>` in the input stream. The value of the `<token list variable>` is not defined in this case and should not be relied upon. If the `<sequence>` is non-empty, pops the top item from the `<sequence>` in the `<token list variable>`, *i.e.* removes the item from the `<sequence>`. The `<sequence>` is modified globally, while the `<token list variable>` is assigned locally.

`\seq_push:Nn`
`\seq_push:(NV|Nv|No|Nx|cn|cV|cv|co|cx)`
`\seq_gpush:Nn`
`\seq_gpush:(NV|Nv|No|Nx|cn|cV|cv|co|cx)`

`\seq_push:Nn <sequence> {(item)}`

Adds the `{(item)}` to the top of the `<sequence>`.

10 Sequences as sets

Sequences can also be used as sets, such that all of their items are distinct. Usage of sequences as sets is not currently widespread, hence no specific set function is provided. Instead, it is explained here how common set operations can be performed by combining several functions described in earlier sections. When using sequences to implement sets, one should be careful not to rely on the order of items in the sequence representing the set.

Sets should not contain several occurrences of a given item. To make sure that a `<sequence variable>` only has distinct items, use `\seq_remove_duplicates:N <sequence variable>`. This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set `<seq var>` are straightforward. For instance, `\seq_count:N <seq var>` expands to the number of items, while `\seq_if_in:NnTF <seq var> {(item)}` tests if the `<item>` is in the set.

Adding an `<item>` to a set `<seq var>` can be done by appending it to the `<seq var>` if it is not already in the `<seq var>`:

```
\seq_if_in:NnF <seq var> {(item)}
{ \seq_put_right:Nn <seq var> {(item)} }
```

Removing an `<item>` from a set `<seq var>` can be done using `\seq_remove_all:Nn,`

```
\seq_remove_all:Nn <seq var> {(item)}
```

The intersection of two sets `<seq var1>` and `<seq var2>` can be stored into `<seq var3>` by collecting items of `<seq var1>` which are in `<seq var2>`.

```

\seq_clear:N <seq var3>
\seq_map_inline:Nn <seq var1>
{
\seq_if_in:NnT <seq var2> {#1}
{ \seq_put_right:Nn <seq var3> {#1} }
}

```

The code as written here only works if $\langle seq\ var_3 \rangle$ is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence $\backslash l_ \langle pkg \rangle_internal_seq$, then $\langle seq\ var_3 \rangle$ should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ through

```

\seq_concat:NNN <seq var3> <seq var1> <seq var2>
\seq_remove_duplicates:N <seq var3>

```

or by adding items to (a copy of) $\langle seq\ var_1 \rangle$ one by one

```

\seq_set_eq:NN <seq var3> <seq var1>
\seq_map_inline:Nn <seq var2>
{
\seq_if_in:NnF <seq var3> {#1}
{ \seq_put_right:Nn <seq var3> {#1} }
}

```

The second approach is faster than the first when the $\langle seq\ var_2 \rangle$ is short compared to $\langle seq\ var_1 \rangle$.

The difference of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ by removing items of the $\langle seq\ var_2 \rangle$ from (a copy of) the $\langle seq\ var_1 \rangle$ one by one.

```

\seq_set_eq:NN <seq var3> <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn <seq var3> {#1} }

```

The symmetric difference of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ by computing the difference between $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ and storing the result as $\backslash l_ \langle pkg \rangle_internal_seq$, then the difference between $\langle seq\ var_2 \rangle$ and $\langle seq\ var_1 \rangle$, and finally concatenating the two differences to get the symmetric differences.

```

\seq_set_eq:NN \l\_ \langle pkg \rangle\_internal\_seq <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn \l\_ \langle pkg \rangle\_internal\_seq {#1} }
\seq_set_eq:NN <seq var3> <seq var2>
\seq_map_inline:Nn <seq var1>
{ \seq_remove_all:Nn <seq var3> {#1} }
\seq_concat:NNN <seq var3> <seq var3> \l\_ \langle pkg \rangle\_internal\_seq

```

11 Constant and scratch sequences

$\backslash c_empty_seq$

Constant that is always empty.

New: 2012-07-02

`\l_tmpa_seq`
`\l_tmpb_seq`

New: 2012-04-26

Scratch sequences for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_seq`
`\g_tmpb_seq`

New: 2012-04-26

Scratch sequences for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

12 Viewing sequences

`\seq_show:N`
`\seq_show:c`

Updated: 2015-08-01

`\seq_show:N` $\langle sequence \rangle$
Displays the entries in the $\langle sequence \rangle$ in the terminal.

`\seq_log:N`
`\seq_log:c`

New: 2014-08-12
Updated: 2015-08-01

`\seq_log:N` $\langle sequence \rangle$
Writes the entries in the $\langle sequence \rangle$ in the log file.

Part XI

The l3int package

Integers

Calculation and comparison of integer values can be carried out using literal numbers, `int` registers, constants and integers stored in token list variables. The standard operators `+`, `-`, `/` and `*` and parentheses can be used within such expressions to carry arithmetic operations. This module carries out these functions on *integer expressions* (“`intexpr`”).

1 Integer expressions

`\int_eval:n *` `\int_eval:n {(integer expression)}`

Evaluates the *integer expression* and leaves the result in the input stream as an integer denotation: for positive results an explicit sequence of decimal digits not starting with 0, for negative results - followed by such a sequence, and 0 for zero. The *integer expression* should consist, after expansion, of +, -, *, /, (,) and of course integer operands. The result is calculated by applying standard mathematical rules with the following peculiarities:

- / denotes division rounded to the closest integer with ties rounded away from zero;
- there is an error and the overall expression evaluates to zero whenever the absolute value of any intermediate result exceeds $2^{31} - 1$, except in the case of scaling operations $a*b/c$, for which $a*b$ may be arbitrarily large;
- parentheses may not appear after unary + or -, namely placing +(or -(at the start of an expression or after +, -, *, / or (leads to an error.

Each integer operand can be either an integer variable (with no need for `\int_use:N`) or an integer denotation. For example both

```
\int_eval:n { 5 + 4 * 3 - ( 3 + 4 * 5 ) }
```

and

```
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { 5 }
\int_new:N \l_my_int
\int_set:Nn \l_my_int { 4 }
\int_eval:n { \l_my_tl + \l_my_int * 3 - ( 3 + 4 * 5 ) }
```

evaluate to -6 because `\l_my_tl` expands to the integer denotation 5. As the *integer expression* is fully expanded from left to right during evaluation, fully expandable and restricted-expandable functions can both be used, and `\exp_not:n` and its variants have no effect while `\exp_not:N` may incorrectly interrupt the expression.

T_EXhackers note: Exactly two expansions are needed to evaluate `\int_eval:n`. The result is *not* an *internal integer*, and therefore requires suitable termination if used in a T_EX-style integer assignment.

As all T_EX integers, integer operands can also be dimension or skip variables, converted to integers in `sp`, or octal numbers given as ' followed by digits other than 8 and 9, or hexadecimal numbers given as " followed by digits or upper case letters from A to F, or the character code of some character or one-character control sequence, given as 'char'.

<hr/> <code>\int_eval:w</code> ★ <hr/>	<code>\int_eval:w</code> $\langle integer\ expression \rangle$
New: 2018-03-30	Evaluates the $\langle integer\ expression \rangle$ as described for <code>\int_eval:n</code> . The end of the expression is the first token encountered that cannot form part of such an expression. If that token is <code>\scan_stop</code> : it is removed, otherwise not. Spaces do <i>not</i> terminate the expression. However, spaces terminate explicit integers, and this may terminate the expression: for instance, <code>\int_eval:w 1_+1_9</code> expands to 29 since the digit 9 is not part of the expression.
<hr/> <code>\int_sign:n</code> ★ <hr/>	<code>\int_sign:n</code> $\{\langle intexpr \rangle\}$
New: 2018-11-03	Evaluates the $\langle integer\ expression \rangle$ then leaves 1 or 0 or -1 in the input stream according to the sign of the result.
<hr/> <code>\int_abs:n</code> ★ <hr/>	<code>\int_abs:n</code> $\{\langle integer\ expression \rangle\}$
Updated: 2012-09-26	Evaluates the $\langle integer\ expression \rangle$ as described for <code>\int_eval:n</code> and leaves the absolute value of the result in the input stream as an $\langle integer\ denotation \rangle$ after two expansions.
<hr/> <code>\int_div_round:nn</code> ★ <hr/>	<code>\int_div_round:nn</code> $\{\langle intexpr_1 \rangle\} \{\langle intexpr_2 \rangle\}$
Updated: 2012-09-26	Evaluates the two $\langle integer\ expressions \rangle$ as described earlier, then divides the first value by the second, and rounds the result to the closest integer. Ties are rounded away from zero. Note that this is identical to using <code>/</code> directly in an $\langle integer\ expression \rangle$. The result is left in the input stream as an $\langle integer\ denotation \rangle$ after two expansions.
<hr/> <code>\int_div_truncate:nn</code> ★ <hr/>	<code>\int_div_truncate:nn</code> $\{\langle intexpr_1 \rangle\} \{\langle intexpr_2 \rangle\}$
Updated: 2012-02-09	Evaluates the two $\langle integer\ expressions \rangle$ as described earlier, then divides the first value by the second, and rounds the result towards zero. Note that division using <code>/</code> rounds to the closest integer instead. The result is left in the input stream as an $\langle integer\ denotation \rangle$ after two expansions.
<hr/> <code>\int_max:nn</code> ★ <hr/>	<code>\int_max:nn</code> $\{\langle intexpr_1 \rangle\} \{\langle intexpr_2 \rangle\}$
<code>\int_min:nn</code> ★ <hr/>	<code>\int_min:nn</code> $\{\langle intexpr_1 \rangle\} \{\langle intexpr_2 \rangle\}$
Updated: 2012-09-26	Evaluates the $\langle integer\ expressions \rangle$ as described for <code>\int_eval:n</code> and leaves either the larger or smaller value in the input stream as an $\langle integer\ denotation \rangle$ after two expansions.
<hr/> <code>\int_mod:nn</code> ★ <hr/>	<code>\int_mod:nn</code> $\{\langle intexpr_1 \rangle\} \{\langle intexpr_2 \rangle\}$
Updated: 2012-09-26	Evaluates the two $\langle integer\ expressions \rangle$ as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is obtained by subtracting <code>\int_div_truncate:nn</code> $\{\langle intexpr_1 \rangle\} \{\langle intexpr_2 \rangle\}$ times $\langle intexpr_2 \rangle$ from $\langle intexpr_1 \rangle$. Thus, the result has the same sign as $\langle intexpr_1 \rangle$ and its absolute value is strictly less than that of $\langle intexpr_2 \rangle$. The result is left in the input stream as an $\langle integer\ denotation \rangle$ after two expansions.

2 Creating and initialising integers

<hr/> <code>\int_new:N</code> <hr/>	<code>\int_new:N</code> $\langle integer \rangle$
<code>\int_new:c</code> <hr/>	Creates a new $\langle integer \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle integer \rangle$ is initially equal to 0.

<code>\int_const:Nn</code>	<code>\int_const:Nn <integer> {<integer expression>}</code>
<code>\int_const:cn</code>	Creates a new constant <i><integer></i> or raises an error if the name is already taken. The value of the <i><integer></i> is set globally to the <i><integer expression></i> .
Updated: 2011-10-22	

<code>\int_zero:N</code>	<code>\int_zero:N <integer></code>
<code>\int_zero:c</code>	
<code>\int_gzero:N</code>	Sets <i><integer></i> to 0.
<code>\int_gzero:c</code>	

<code>\int_zero_new:N</code>	<code>\int_zero_new:N <integer></code>
<code>\int_zero_new:c</code>	
<code>\int_gzero_new:N</code>	Ensures that the <i><integer></i> exists globally by applying <code>\int_new:N</code> if necessary, then applies <code>\int_(g)zero:N</code> to leave the <i><integer></i> set to zero.
<code>\int_gzero_new:c</code>	

New: 2011-12-13

<code>\int_set_eq:NN</code>	<code>\int_set_eq:NN <integer₁₂</code>
<code>\int_set_eq:(cN Nc cc)</code>	Sets the content of <i><integer_{1 equal to that of <i><integer_{2.}</i>}</i>
<code>\int_gset_eq:NN</code>	
<code>\int_gset_eq:(cN Nc cc)</code>	

<code>\int_if_exist_p:N *</code>	<code>\int_if_exist_p:N <int></code>
<code>\int_if_exist_p:c *</code>	<code>\int_if_exist:NTF <int> {<true code>} {<false code>}</code>
<code>\int_if_exist:NTF *</code>	
<code>\int_if_exist:cTF *</code>	Tests whether the <i><int></i> is currently defined. This does not check that the <i><int></i> really is an integer variable.

New: 2012-03-03

3 Setting and incrementing integers

<code>\int_add:Nn</code>	<code>\int_add:Nn <integer> {<integer expression>}</code>
<code>\int_add:cn</code>	Adds the result of the <i><integer expression></i> to the current content of the <i><integer></i> .
<code>\int_gadd:Nn</code>	
<code>\int_gadd:cn</code>	

Updated: 2011-10-22

<code>\int_decr:N</code>	<code>\int_decr:N <integer></code>
<code>\int_decr:c</code>	
<code>\int_gdecr:N</code>	Decreases the value stored in <i><integer></i> by 1.
<code>\int_gdecr:c</code>	

<code>\int_incr:N</code>	<code>\int_incr:N <integer></code>
<code>\int_incr:c</code>	
<code>\int_gincr:N</code>	Increases the value stored in <i><integer></i> by 1.
<code>\int_gincr:c</code>	

<code>\int_set:Nn</code>	<code>\int_set:Nn <integer> {<integer expression>}</code>
<code>\int_set:cn</code>	
<code>\int_gset:Nn</code>	Sets <i><integer></i> to the value of <i><integer expression></i> , which must evaluate to an integer (as described for <code>\int_eval:n</code>).
<code>\int_gset:cn</code>	

Updated: 2011-10-22

<code>\int_sub:Nn</code>	<code>\int_sub:Nn <integer> {<integer expression>}</code>
<code>\int_sub:cn</code>	
<code>\int_gsub:Nn</code>	Subtracts the result of the <i><integer expression></i> from the current content of the <i><integer></i> .
<code>\int_gsub:cn</code>	

Updated: 2011-10-22

4 Using integers

<code>\int_use:N</code> *	<code>\int_use:N <integer></code>
<code>\int_use:c</code> *	
	Recovers the content of an <i><integer></i> and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where an <i><integer></i> is required (such as in the first and third arguments of <code>\int_compare:nNnTF</code>).

Updated: 2011-10-22

TeXhackers note: `\int_use:N` is the TeX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

5 Integer expression conditionals

<code>\int_compare_p:nNn</code> *	<code>\int_compare_p:nNn {<intexpr₁>} <relation> {<intexpr₂>}</code>
<code>\int_compare:nNnTF</code> *	<code>\int_compare:nNnTF</code> <code>{<intexpr₁>} <relation> {<intexpr₂>}</code> <code>{<true code>} {<false code>}</code>

This function first evaluates each of the *<integer expressions>* as described for `\int_eval:n`. The two results are then compared using the *<relation>*:

Equal	=
Greater than	>
Less than	<

This function is less flexible than `\int_compare:nTF` but around 5 times faster.

```

\int_compare_p:n * \int_compare_p:n
\int_compare:nTF * {
    <intexpr1> <relation1>
    ...
    <intexprN> <relationN>
    <intexprN+1>
}
\int_compare:nTF
{
    <intexpr1> <relation1>
    ...
    <intexprN> <relationN>
    <intexprN+1>
}
{<true code>} {<false code>}

```

Updated: 2013-01-13

This function evaluates the *<integer expressions>* as described for `\int_eval:n` and compares consecutive result using the corresponding *<relation>*, namely it compares *<intexpr₁>* and *<intexpr₂>* using the *<relation₁>*, then *<intexpr₂>* and *<intexpr₃>* using the *<relation₂>*, until finally comparing *<intexpr_N>* and *<intexpr_{N+1}>* using the *<relation_N>*. The test yields **true** if all comparisons are **true**. Each *<integer expression>* is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is **false**, then no other *<integer expression>* is evaluated and no other comparison is performed. The *<relations>* can be any of the following:

Equal	= or ==
Greater than or equal to	>=
Greater than	>
Less than or equal to	<=
Less than	<
Not equal	!=

This function is more flexible than `\int_compare:nNnTF` but around 5 times slower.

<code>\int_case:nn</code> *	<code>\int_case:nnTF {⟨test integer expression⟩}</code>
<code>\int_case:nnTF</code> *	<code>{</code>
	<code>{⟨intexpr case₁⟩} {⟨code case₁⟩}</code>
	<code>{⟨intexpr case₂⟩} {⟨code case₂⟩}</code>
	<code>...</code>
	<code>{⟨intexpr case_n⟩} {⟨code case_n⟩}</code>
	<code>}</code>
	<code>{⟨true code⟩}</code>
	<code>{⟨false code⟩}</code>

New: 2013-07-24

This function evaluates the *⟨test integer expression⟩* and compares this in turn to each of the *⟨integer expression cases⟩*. If the two are equal then the associated *⟨code⟩* is left in the input stream and other cases are discarded. If any of the cases are matched, the *⟨true code⟩* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *⟨false code⟩* is inserted. The function `\int_case:nn`, which does nothing if there is no match, is also available. For example

```
\int_case:nnF
{ 2 * 5 }
{
  { 5 }      { Small }
  { 4 + 6 }   { Medium }
  { -2 * 10 } { Negative }
}
{ No idea! }
```

leaves “Medium” in the input stream.

<code>\int_if_even_p:n</code> *	<code>\int_if_odd_p:n {⟨integer expression⟩}</code>
<code>\int_if_even:nTF</code> *	<code>\int_if_odd:nTF {⟨integer expression⟩}</code>
<code>\int_if_odd_p:n</code> *	<code>{⟨true code⟩} {⟨false code⟩}</code>
<code>\int_if_odd:nTF</code> *	

This function first evaluates the *⟨integer expression⟩* as described for `\int_eval:n`. It then evaluates if this is odd or even, as appropriate.

6 Integer expression loops

<code>\int_do_until:nNnn</code> ☆	<code>\int_do_until:nNnn {⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Places the *⟨code⟩* in the input stream for T_EX to process, and then evaluates the relationship between the two *⟨integer expressions⟩* as described for `\int_compare:nNnTF`. If the test is **false** then the *⟨code⟩* is inserted into the input stream again and a loop occurs until the *⟨relation⟩* is **true**.

<code>\int_do_while:nNnn</code> ☆	<code>\int_do_while:nNnn {⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Places the *⟨code⟩* in the input stream for T_EX to process, and then evaluates the relationship between the two *⟨integer expressions⟩* as described for `\int_compare:nNnTF`. If the test is **true** then the *⟨code⟩* is inserted into the input stream again and a loop occurs until the *⟨relation⟩* is **false**.

<hr/> <code>\int_until_do:nNnn</code> ☆ <hr/>	<code>\int_until_do:nNnn {⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}</code>
	Evaluates the relationship between the two <i>⟨integer expressions⟩</i> as described for <code>\int_compare:nNnTF</code> , and then places the <i>⟨code⟩</i> in the input stream if the <i>⟨relation⟩</i> is false . After the <i>⟨code⟩</i> has been processed by T _E X the test is repeated, and a loop occurs until the test is true .
<hr/> <code>\int_while_do:nNnn</code> ☆ <hr/>	<code>\int_while_do:nNnn {⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}</code>
	Evaluates the relationship between the two <i>⟨integer expressions⟩</i> as described for <code>\int_compare:nNnTF</code> , and then places the <i>⟨code⟩</i> in the input stream if the <i>⟨relation⟩</i> is true . After the <i>⟨code⟩</i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .
<hr/> <code>\int_do_until:nn</code> ☆ <hr/>	<code>\int_do_until:nn {⟨integer relation⟩} {⟨code⟩}</code>
Updated: 2013-01-13	Places the <i>⟨code⟩</i> in the input stream for T _E X to process, and then evaluates the <i>⟨integer relation⟩</i> as described for <code>\int_compare:nTF</code> . If the test is false then the <i>⟨code⟩</i> is inserted into the input stream again and a loop occurs until the <i>⟨relation⟩</i> is true .
<hr/> <code>\int_do_while:nn</code> ☆ <hr/>	<code>\int_do_while:nn {⟨integer relation⟩} {⟨code⟩}</code>
Updated: 2013-01-13	Places the <i>⟨code⟩</i> in the input stream for T _E X to process, and then evaluates the <i>⟨integer relation⟩</i> as described for <code>\int_compare:nTF</code> . If the test is true then the <i>⟨code⟩</i> is inserted into the input stream again and a loop occurs until the <i>⟨relation⟩</i> is false .
<hr/> <code>\int_until_do:nn</code> ☆ <hr/>	<code>\int_until_do:nn {⟨integer relation⟩} {⟨code⟩}</code>
Updated: 2013-01-13	Evaluates the <i>⟨integer relation⟩</i> as described for <code>\int_compare:nTF</code> , and then places the <i>⟨code⟩</i> in the input stream if the <i>⟨relation⟩</i> is false . After the <i>⟨code⟩</i> has been processed by T _E X the test is repeated, and a loop occurs until the test is true .
<hr/> <code>\int_while_do:nn</code> ☆ <hr/>	<code>\int_while_do:nn {⟨integer relation⟩} {⟨code⟩}</code>
Updated: 2013-01-13	Evaluates the <i>⟨integer relation⟩</i> as described for <code>\int_compare:nTF</code> , and then places the <i>⟨code⟩</i> in the input stream if the <i>⟨relation⟩</i> is true . After the <i>⟨code⟩</i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .

7 Integer step functions

<code>\int_step_function:nN</code>	☆	<code>\int_step_function:nN {⟨final value⟩} ⟨function⟩</code>
<code>\int_step_function:nnN</code>	☆	<code>\int_step_function:nnN {⟨initial value⟩} {⟨final value⟩} ⟨function⟩</code>
<code>\int_step_function:nnnN</code>	☆	<code>\int_step_function:nnnN {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨function⟩</code>

New: 2012-06-04
Updated: 2018-04-22

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be integer expressions. The $\langle function \rangle$ is then placed in front of each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$). The $\langle step \rangle$ must be non-zero. If the $\langle step \rangle$ is positive, the loop stops when the $\langle value \rangle$ becomes larger than the $\langle final\ value \rangle$. If the $\langle step \rangle$ is negative, the loop stops when the $\langle value \rangle$ becomes smaller than the $\langle final\ value \rangle$. The $\langle function \rangle$ should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\int_step_function:nnnN { 1 } { 1 } { 5 } \my_func:n
```

would print

```
[I saw 1]   [I saw 2]   [I saw 3]   [I saw 4]   [I saw 5]
```

The functions `\int_step_function:nN` and `\int_step_function:nnN` both use a fixed $\langle step \rangle$ of 1, and in the case of `\int_step_function:nN` the $\langle initial\ value \rangle$ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

<code>\int_step_inline:nn</code>	<code>\int_step_inline:nn {⟨final value⟩} {⟨code⟩}</code>
<code>\int_step_inline:nnn</code>	<code>\int_step_inline:nnn {⟨initial value⟩} {⟨final value⟩} {⟨code⟩}</code>
<code>\int_step_inline:nnnn</code>	<code>\int_step_inline:nnnn {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} {⟨code⟩}</code>

New: 2012-06-04
Updated: 2018-04-22

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be integer expressions. Then for each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream with `#1` replaced by the current $\langle value \rangle$. Thus the $\langle code \rangle$ should define a function of one argument (`#1`).

The functions `\int_step_inline:nn` and `\int_step_inline:nnn` both use a fixed $\langle step \rangle$ of 1, and in the case of `\int_step_inline:nn` the $\langle initial\ value \rangle$ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

<code>\int_step_variable:nNn</code>	<code>\int_step_variable:nNn {⟨final value⟩} ⟨tl var⟩ {⟨code⟩}</code>
<code>\int_step_variable:nnNn</code>	<code>\int_step_variable:nnNn {⟨initial value⟩} {⟨final value⟩} ⟨tl var⟩ {⟨code⟩}</code>
<code>\int_step_variable:nnnNn</code>	<code>\int_step_variable:nnnNn {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨tl var⟩ {⟨code⟩}</code>

New: 2012-06-04
Updated: 2018-04-22

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be integer expressions. Then for each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream, with the $\langle tl\ var \rangle$ defined as the current $\langle value \rangle$. Thus the $\langle code \rangle$ should make use of the $\langle tl\ var \rangle$.

The functions `\int_step_variable:nNn` and `\int_step_variable:nnNn` both use a fixed $\langle step \rangle$ of 1, and in the case of `\int_step_variable:nNn` the $\langle initial\ value \rangle$ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

8 Formatting integers

Integers can be placed into the output stream with formatting. These conversions apply to any integer expressions.

<code>\int_to_arabic:n</code> *	<code>\int_to_arabic:n {⟨integer expression⟩}</code>
---------------------------------	--

Updated: 2011-10-22

Places the value of the $\langle integer\ expression \rangle$ in the input stream as digits, with category code 12 (other).

<code>\int_to_alph:n</code> *	<code>\int_to_alph:n {⟨integer expression⟩}</code>
<code>\int_to_Alph:n</code> *	

Updated: 2011-09-17

Evaluates the $\langle integer\ expression \rangle$ and converts the result into a series of letters, which are then left in the input stream. The conversion rule uses the 26 letters of the English alphabet, in order, adding letters when necessary to increase the total possible range of representable numbers. Thus

```
\int_to_alph:n { 1 }
```

places a in the input stream,

```
\int_to_alph:n { 26 }
```

is represented as z and

```
\int_to_alph:n { 27 }
```

is converted to aa. For conversions using other alphabets, use `\int_to_symbols:nnn` to define an alphabet-specific function. The basic `\int_to_alph:n` and `\int_to_Alph:n` functions should not be modified. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

<code>\int_to_symbols:nnn</code> *	<code>\int_to_symbols:nnn</code> <code>{⟨integer expression⟩} {⟨total symbols⟩}</code> <code>{⟨value to symbol mapping⟩}</code>
------------------------------------	---

Updated: 2011-09-17

This is the low-level function for conversion of an $\langle integer\ expression \rangle$ into a symbolic form (often letters). The $\langle total\ symbols \rangle$ available should be given as an integer expression. Values are actually converted to symbols according to the $\langle value\ to\ symbol\ mapping \rangle$. This should be given as $\langle total\ symbols \rangle$ pairs of entries, a number and the appropriate symbol. Thus the `\int_to_alph:n` function is defined as

```
\cs_new:Npn \int_to_alph:n #1
{
  \int_to_symbols:nnn {#1} { 26 }
  {
    { 1 } { a }
    { 2 } { b }
    ...
    { 26 } { z }
  }
}
```

<hr/>	
<code>\int_to_bin:n</code> ★	<code>\int_to_bin:n {⟨integer expression⟩}</code>
<hr/>	
New: 2014-02-11	Calculates the value of the <i>⟨integer expression⟩</i> and places the binary representation of the result in the input stream.
<hr/>	
<code>\int_to_hex:n</code> ★	<code>\int_to_hex:n {⟨integer expression⟩}</code>
<code>\int_to_Hex:n</code> ★	
<hr/>	
New: 2014-02-11	Calculates the value of the <i>⟨integer expression⟩</i> and places the hexadecimal (base 16) representation of the result in the input stream. Letters are used for digits beyond 9: lower case letters for <code>\int_to_hex:n</code> and upper case ones for <code>\int_to_Hex:n</code> . The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
<hr/>	
<code>\int_to_oct:n</code> ★	<code>\int_to_oct:n {⟨integer expression⟩}</code>
<hr/>	
New: 2014-02-11	Calculates the value of the <i>⟨integer expression⟩</i> and places the octal (base 8) representation of the result in the input stream. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
<hr/>	
<code>\int_to_base:nn</code> ★	<code>\int_to_base:nn {⟨integer expression⟩} {⟨base⟩}</code>
<code>\int_to_Base:nn</code> ★	
<hr/>	
Updated: 2014-02-11	Calculates the value of the <i>⟨integer expression⟩</i> and converts it into the appropriate representation in the <i>⟨base⟩</i> ; the later may be given as an integer expression. For bases greater than 10 the higher “digits” are represented by letters from the English alphabet: lower case letters for <code>\int_to_base:n</code> and upper case ones for <code>\int_to_Base:n</code> . The maximum <i>⟨base⟩</i> value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
<hr/>	
TeXhackers note: This is a generic version of <code>\int_to_bin:n</code> , <i>etc.</i>	
<hr/>	
<code>\int_to_roman:n</code> ☆	<code>\int_to_roman:n {⟨integer expression⟩}</code>
<code>\int_to_Roman:n</code> ☆	
<hr/>	
Updated: 2011-10-22	Places the value of the <i>⟨integer expression⟩</i> in the input stream as Roman numerals, either lower case (<code>\int_to_roman:n</code>) or upper case (<code>\int_to_Roman:n</code>). If the value is negative or zero, the output is empty. The Roman numerals are letters with category code 11 (letter). The letters used are <code>mdclxvi</code> , repeated as needed: the notation with bars (such as <code>v̄</code> for 5000) is <i>not</i> used. For instance <code>\int_to_roman:n { 8249 }</code> expands to <code>mmmmmmmmccxlix</code> .

9 Converting from other formats to integers

<hr/>	
<code>\int_from_alph:n</code> ★	<code>\int_from_alph:n {⟨letters⟩}</code>
<hr/>	
Updated: 2014-08-25	Converts the <i>⟨letters⟩</i> into the integer (base 10) representation and leaves this in the input stream. The <i>⟨letters⟩</i> are first converted to a string, with no expansion. Lower and upper case letters from the English alphabet may be used, with “a” equal to 1 through to “z” equal to 26. The function also accepts a leading sign, made of + and -. This is the inverse function of <code>\int_to_alph:n</code> and <code>\int_to_Alph:n</code> .

<hr/> <code>\int_from_bin:n</code> ★ <hr/>	<code>\int_from_bin:n {⟨binary number⟩}</code>
New: 2014-02-11 Updated: 2014-08-25	Converts the <i>⟨binary number⟩</i> into the integer (base 10) representation and leaves this in the input stream. The <i>⟨binary number⟩</i> is first converted to a string, with no expansion. The function accepts a leading sign, made of + and -, followed by binary digits. This is the inverse function of <code>\int_to_bin:n</code> .
<hr/> <code>\int_from_hex:n</code> ★ <hr/>	<code>\int_from_hex:n {⟨hexadecimal number⟩}</code>
New: 2014-02-11 Updated: 2014-08-25	Converts the <i>⟨hexadecimal number⟩</i> into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the <i>⟨hexadecimal number⟩</i> by upper or lower case letters. The <i>⟨hexadecimal number⟩</i> is first converted to a string, with no expansion. The function also accepts a leading sign, made of + and -. This is the inverse function of <code>\int_to_hex:n</code> and <code>\int_to_Hex:n</code> .
<hr/> <code>\int_from_oct:n</code> ★ <hr/>	<code>\int_from_oct:n {⟨octal number⟩}</code>
New: 2014-02-11 Updated: 2014-08-25	Converts the <i>⟨octal number⟩</i> into the integer (base 10) representation and leaves this in the input stream. The <i>⟨octal number⟩</i> is first converted to a string, with no expansion. The function accepts a leading sign, made of + and -, followed by octal digits. This is the inverse function of <code>\int_to_oct:n</code> .
<hr/> <code>\int_from_roman:n</code> ★ <hr/>	<code>\int_from_roman:n {⟨roman numeral⟩}</code>
Updated: 2014-08-25	Converts the <i>⟨roman numeral⟩</i> into the integer (base 10) representation and leaves this in the input stream. The <i>⟨roman numeral⟩</i> is first converted to a string, with no expansion. The <i>⟨roman numeral⟩</i> may be in upper or lower case; if the numeral contains characters besides <code>mdclxvi</code> or <code>MDCLXVI</code> then the resulting value is -1. This is the inverse function of <code>\int_to_roman:n</code> and <code>\int_to_Roman:n</code> .
<hr/> <code>\int_from_base:nn</code> ★ <hr/>	<code>\int_from_base:nn {⟨number⟩} {⟨base⟩}</code>
Updated: 2014-08-25	Converts the <i>⟨number⟩</i> expressed in <i>⟨base⟩</i> into the appropriate value in base 10. The <i>⟨number⟩</i> is first converted to a string, with no expansion. The <i>⟨number⟩</i> should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum <i>⟨base⟩</i> value is 36. This is the inverse function of <code>\int_to_base:nn</code> and <code>\int_to_Base:nn</code> .

10 Random integers

<hr/> <code>\int_rand:nn</code> ★ <hr/>	<code>\int_rand:nn {⟨integer₁⟩} {⟨integer₂⟩}</code>
New: 2016-12-06 Updated: 2018-04-27	Evaluates the two <i>⟨integer expressions⟩</i> and produces a pseudo-random number between the two (with bounds included). This is not available in older versions of X _Y TeX.
<hr/> <code>\int_rand:n</code> ★ <hr/>	<code>\int_rand:n {⟨integer⟩}</code>
New: 2018-05-05	Evaluates the <i>⟨integer expression⟩</i> then produces a pseudo-random number between 1 and the <i>⟨integer⟩</i> (included). This is not available in older versions of X _Y TeX.

11 Viewing integers

<hr/> <code>\int_show:N</code> <code>\int_show:c</code> <hr/>	<code>\int_show:N <integer></code> Displays the value of the <i><integer></i> on the terminal.
<hr/> <code>\int_show:n</code> <hr/> <div>New: 2011-11-22 Updated: 2015-08-07</div>	<code>\int_show:n {(integer expression)}</code> Displays the result of evaluating the <i><integer expression></i> on the terminal.
<hr/> <code>\int_log:N</code> <code>\int_log:c</code> <hr/> <div>New: 2014-08-22 Updated: 2015-08-03</div>	<code>\int_log:N <integer></code> Writes the value of the <i><integer></i> in the log file.
<hr/> <code>\int_log:n</code> <hr/> <div>New: 2014-08-22 Updated: 2015-08-07</div>	<code>\int_log:n {(integer expression)}</code> Writes the result of evaluating the <i><integer expression></i> in the log file.

12 Constant integers

<hr/> <code>\c_zero_int</code> <code>\c_one_int</code> <hr/> <div>New: 2018-05-07</div>	Integer values used with primitive tests and assignments: their self-terminating nature makes these more convenient and faster than literal numbers.
<hr/> <code>\c_max_int</code> <hr/>	The maximum value that can be stored as an integer.
<hr/> <code>\c_max_register_int</code> <hr/>	Maximum number of registers.
<hr/> <code>\c_max_char_int</code> <hr/>	Maximum character code completely supported by the engine.

13 Scratch integers

<hr/> <code>\l_tmpa_int</code> <code>\l_tmpb_int</code> <hr/>	Scratch integer for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <code>\g_tmpa_int</code> <code>\g_tmpb_int</code> <hr/>	Scratch integer for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

13.1 Direct number expansion

`\int_value:w` ★
 New: 2018-03-27

`\int_value:w` $\langle integer \rangle$
`\int_value:w` $\langle integer\ denotation \rangle$ $\langle optional\ space \rangle$

Expands the following tokens until an $\langle integer \rangle$ is formed, and leaves a normalized form (no leading sign except for negative numbers, no leading digit 0 except for zero) in the input stream as category code 12 (other) characters. The $\langle integer \rangle$ can consist of any number of signs (with intervening spaces) followed by

- an integer variable (in fact, any T_EX register except `\toks`) or
- explicit digits (or by ‘ $\langle octal\ digits \rangle$ ’ or “ $\langle hexadecimal\ digits \rangle$ ” or ‘ $\langle character \rangle$ ’).

In this last case expansion stops once a non-digit is found; if that is a space it is removed as in `f`-expansion, and so `\exp_stop_f`: may be employed as an end marker. Note that protected functions *are* expanded by this process.

This function requires exactly one expansion to produce a value, and so is suitable for use in cases where a number is required “directly”. In general, `\int_eval:n` is the preferred approach to generating numbers.

T_EXhackers note: This is the T_EX primitive `\number`.

14 Primitive conditionals

`\if_int_compare:w` ★

`\if_int_compare:w` $\langle integer_1 \rangle$ $\langle relation \rangle$ $\langle integer_2 \rangle$
 $\langle true\ code \rangle$
`\else:`
 $\langle false\ code \rangle$
`\fi:`

Compare two integers using $\langle relation \rangle$, which must be one of =, < or > with category code 12. The `\else:` branch is optional.

T_EXhackers note: These are both names for the T_EX primitive `\ifnum`.

`\if_case:w` ★
`\or:` ★

`\if_case:w` $\langle integer \rangle$ $\langle case_0 \rangle$
`\or:` $\langle case_1 \rangle$
`\or:` ...
`\else:` $\langle default \rangle$
`\fi:`

Selects a case to execute based on the value of the $\langle integer \rangle$. The first case ($\langle case_0 \rangle$) is executed if $\langle integer \rangle$ is 0, the second ($\langle case_1 \rangle$) if the $\langle integer \rangle$ is 1, *etc.* The $\langle integer \rangle$ may be a literal, a constant or an integer expression (*e.g.* using `\int_eval:n`).

T_EXhackers note: These are the T_EX primitives `\ifcase` and `\or`.

<code>\if_int_odd:w</code> ★	<code>\if_int_odd:w</code> $\langle tokens \rangle$ $\langle optional\ space \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle true\ code \rangle$ <code>\fi:</code>
------------------------------	---

Expands $\langle tokens \rangle$ until a non-numeric token or a space is found, and tests whether the resulting $\langle integer \rangle$ is odd. If so, $\langle true\ code \rangle$ is executed. The `\else:` branch is optional.

TeXhackers note: This is the TeX primitive `\ifodd`.

Part XII

The l3flag package: Expandable flags

Flags are the only data-type that can be modified in expansion-only contexts. This module is meant mostly for kernel use: in almost all cases, booleans or integers should be preferred to flags because they are very significantly faster.

A flag can hold any non-negative value, which we call its *height*. In expansion-only contexts, a flag can only be “raised”: this increases the *height* by 1. The *height* can also be queried expandably. However, decreasing it, or setting it to zero requires non-expandable assignments.

Flag variables are always local. They are referenced by a *flag name* such as `str_missing`. The *flag name* is used as part of `\use:c` constructions hence is expanded at point of use. It must expand to character tokens only, with no spaces.

A typical use case of flags would be to keep track of whether an exceptional condition has occurred during expandable processing, and produce a meaningful (non-expandable) message after the end of the expandable processing. This is exemplified by `l3str-convert`, which for performance reasons performs conversions of individual characters expandably and for readability reasons produces a single error message describing incorrect inputs that were encountered.

Flags should not be used without carefully considering the fact that raising a flag takes a time and memory proportional to its height. Flags should not be used unless unavoidable.

1 Setting up flags

<code>\flag_new:n</code>	<code>\flag_new:n {<flag name>}</code>
--------------------------	--

Creates a new flag with a name given by *flag name*, or raises an error if the name is already taken. The *flag name* may not contain spaces. The declaration is global, but flags are always local variables. The *flag* initially has zero height.

<code>\flag_clear:n</code>	<code>\flag_clear:n {<flag name>}</code>
----------------------------	--

The *flag*’s height is set to zero. The assignment is local.

<code>\flag_clear_new:n</code>	<code>\flag_clear_new:n {<flag name>}</code>
--------------------------------	--

Ensures that the *flag* exists globally by applying `\flag_new:n` if necessary, then applies `\flag_clear:n`, setting the height to zero locally.

<code>\flag_show:n</code>	<code>\flag_show:n {<flag name>}</code>
---------------------------	---

Displays the *flag*’s height in the terminal.

<code>\flag_log:n</code>	<code>\flag_log:n {<flag name>}</code>
--------------------------	--

Writes the *flag*’s height to the log file.

2 Expandable flag commands

<hr/> <code>\flag_if_exist:n *</code> <code>\flag_if_exist:n\underline{TF} *</code> <hr/>	<code>\flag_if_exist:n {⟨flag name⟩}</code> This function returns <code>true</code> if the <code>⟨flag name⟩</code> references a flag that has been defined previously, and <code>false</code> otherwise.
<hr/> <code>\flag_if_raised_p:n *</code> <code>\flag_if_raised:n\underline{TF} *</code> <hr/>	<code>\flag_if_raised:n {⟨flag name⟩}</code> This function returns <code>true</code> if the <code>⟨flag⟩</code> has non-zero height, and <code>false</code> if the <code>⟨flag⟩</code> has zero height.
<hr/> <code>\flag_height:n *</code> <hr/>	<code>\flag_height:n {⟨flag name⟩}</code> Expands to the height of the <code>⟨flag⟩</code> as an integer denotation.
<hr/> <code>\flag_raise:n *</code> <hr/>	<code>\flag_raise:n {⟨flag name⟩}</code> The <code>⟨flag⟩</code> 's height is increased by 1 locally.

Part XIII

The l3prg package

Control structures

Conditional processing in L^AT_EX3 is defined as something that performs a series of tests, possibly involving assignments and calling other functions that do not read further ahead in the input stream. After processing the input, a *state* is returned. The states returned are *⟨true⟩* and *⟨false⟩*.

L^AT_EX3 has two forms of conditional flow processing based on these states. The first form is predicate functions that turn the returned state into a boolean *⟨true⟩* or *⟨false⟩*. For example, the function `\cs_if_free_p:N` checks whether the control sequence given as its argument is free and then returns the boolean *⟨true⟩* or *⟨false⟩* values to be used in testing with `\if_predicate:w` or in functions to be described below. The second form is the kind of functions choosing a particular argument from the input stream based on the result of the testing as in `\cs_if_free:NTF` which also takes one argument (the *N*) and then executes either **true** or **false** depending on the result.

T_EXhackers note: The arguments are executed after exiting the underlying `\if... \fi` structure.

1 Defining a set of conditional functions

```
\prg_new_conditional:Npnn
\prg_set_conditional:Npnn
\prg_new_conditional:Nnn
\prg_set_conditional:Nnn
```

Updated: 2012-02-06

```
\prg_new_conditional:Npnn \⟨name⟩:⟨arg spec⟩ ⟨parameters⟩ {⟨conditions⟩} {⟨code⟩}
\prg_new_conditional:Nnn \⟨name⟩:⟨arg spec⟩ {⟨conditions⟩} {⟨code⟩}
```

These functions create a family of conditionals using the same *{⟨code⟩}* to perform the test created. Those conditionals are expandable if *⟨code⟩* is. The **new** versions check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the **set** versions do no check and perform assignments locally (*cf.* `\cs_set:Npn`). The conditionals created are dependent on the comma-separated list of *⟨conditions⟩*, which should be one or more of **p**, **T**, **F** and **TF**.

```
\prg_new_protected_conditional:Npnn \prg_new_protected_conditional:Npnn \⟨name⟩:⟨arg spec⟩ ⟨parameters⟩
\prg_set_protected_conditional:Npnn {⟨conditions⟩} {⟨code⟩}
\prg_new_protected_conditional:Nnn \prg_new_protected_conditional:Nnn \⟨name⟩:⟨arg spec⟩
\prg_set_protected_conditional:Nnn {⟨conditions⟩} {⟨code⟩}
```

Updated: 2012-02-06

These functions create a family of protected conditionals using the same *{⟨code⟩}* to perform the test created. The *⟨code⟩* does not need to be expandable. The **new** version check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the **set** version do not (*cf.* `\cs_set:Npn`). The conditionals created are depended on the comma-separated list of *⟨conditions⟩*, which should be one or more of **T**, **F** and **TF** (not **p**).

The conditionals are defined by `\prg_new_conditional:Npnn` and friends as:

- `\<name>_p:<arg spec>` — a predicate function which will supply either a logical `true` or logical `false`. This function is intended for use in cases where one or more logical tests are combined to lead to a final outcome. This function cannot be defined for `protected` conditionals.
- `\<name>:<arg spec>T` — a function with one more argument than the original `<arg spec>` demands. The `<true branch>` code in this additional argument will be left on the input stream only if the test is `true`.
- `\<name>:<arg spec>F` — a function with one more argument than the original `<arg spec>` demands. The `<false branch>` code in this additional argument will be left on the input stream only if the test is `false`.
- `\<name>:<arg spec>TF` — a function with two more argument than the original `<arg spec>` demands. The `<true branch>` code in the first additional argument will be left on the input stream if the test is `true`, while the `<false branch>` code in the second argument will be left on the input stream if the test is `false`.

The `<code>` of the test may use `<parameters>` as specified by the second argument to `\prg_set_conditional:Npnn`: this should match the `<argument specification>` but this is not enforced. The `Nnn` versions infer the number of arguments from the argument specification given (*cf.* `\cs_new:Nn`, *etc.*). Within the `<code>`, the functions `\prg_return_true:` and `\prg_return_false:` are used to indicate the logical outcomes of the test.

An example can easily clarify matters here:

```
\prg_set_conditional:Npnn \foo_if_bar:NN #1#2 { p , T , TF }
{
  \if_meaning:w \l_tmpa_tl #1
  \prg_return_true:
\else:
  \if_meaning:w \l_tmpa_tl #2
  \prg_return_true:
\else:
  \prg_return_false:
\fi:
\fi:
}
```

This defines the function `\foo_if_bar_p:NN`, `\foo_if_bar:NNTF` and `\foo_if_bar:NNT` but not `\foo_if_bar:NNF` (because `F` is missing from the `<conditions>` list). The return statements take care of resolving the remaining `\else:` and `\fi:` before returning the state. There must be a return statement for each branch; failing to do so will result in erroneous output if that branch is executed.

<code>\prg_new_eq_conditional:Nnn</code>	<code>\prg_new_eq_conditional:Nnn \<name1>:<arg spec1> \<name2>:<arg spec2></code>
<code>\prg_set_eq_conditional:Nnn</code>	<code>{<conditions>}</code>

These functions copy a family of conditionals. The `new` version checks for existing definitions (*cf.* `\cs_new_eq:NN`) whereas the `set` version does not (*cf.* `\cs_set_eq:NN`). The conditionals copied are depended on the comma-separated list of `<conditions>`, which should be one or more of `p`, `T`, `F` and `TF`.

<code>\prg_return_true:</code>	<code>*</code>	<code>\prg_return_true:</code>
<code>\prg_return_false:</code>	<code>*</code>	<code>\prg_return_false:</code>

These “return” functions define the logical state of a conditional statement. They appear within the code for a conditional function generated by `\prg_set_conditional:Npnn`, *etc.*, to indicate when a true or false branch should be taken. While they may appear multiple times each within the code of such conditionals, the execution of the conditional must result in the expansion of one of these two functions *exactly once*.

The return functions trigger what is internally an *f*-expansion process to complete the evaluation of the conditional. Therefore, after `\prg_return_true:` or `\prg_return_false:` there must be no non-expandable material in the input stream for the remainder of the expansion of the conditional code. This includes other instances of either of these functions.

<code>\prg_generate_conditional_variant:Nnn</code>	<code>\prg_generate_conditional_variant:Nnn \<name>:\<arg spec></code>
	<code>{\<variant argument specifiers>} {\<condition specifiers>}</code>

New: 2017-12-12

Defines argument-specifier variants of conditionals. This is equivalent to running `\cs_generate_variant:Nn \<conditional> {\<variant argument specifiers>}` on each *<conditional>* described by the *<condition specifiers>*. These base-form *<conditionals>* are obtained from the *<name>* and *<arg spec>* as described for `\prg_new_conditional:Npnn`, and they should be defined.

2 The boolean data type

This section describes a boolean data type which is closely connected to conditional processing as sometimes you want to execute some code depending on the value of a switch (*e.g.*, draft/final) and other times you perhaps want to use it as a predicate function in an `\if_predicate:w` test. The problem of the primitive `\if_false:` and `\if_true:` tokens is that it is not always safe to pass them around as they may interfere with scanning for termination of primitive conditional processing. Therefore, we employ two canonical booleans: `\c_true_bool` or `\c_false_bool`. Besides preventing problems as described above, it also allows us to implement a simple boolean parser supporting the logical operations And, Or, Not, *etc.* which can then be used on both the boolean type and predicate functions.

All conditional `\bool_` functions except assignments are expandable and expect the input to also be fully expandable (which generally means being constructed from predicate functions and booleans, possibly nested).

T_EXhackers note: The `bool` data type is not implemented using the `\iffalse/\iftrue` primitives, in contrast to `\newif`, *etc.*, in plain T_EX, L^AT_EX 2_ε and so on. Programmers should not base use of `bool` switches on any particular expectation of the implementation.

<code>\bool_new:N</code>	<code>\bool_new:N \<boolean></code>
<code>\bool_new:c</code>	

Creates a new *<boolean>* or raises an error if the name is already taken. The declaration is global. The *<boolean>* is initially **false**.

<hr/> \bool_const:Nn \bool_const:cn <hr/> New: 2017-11-28	\bool_const:Nn <boolean> {<boolexpr>} Creates a new constant <boolean> or raises an error if the name is already taken. The value of the <boolean> is set globally to the result of evaluating the <boolexpr>.
<hr/> \bool_set_false:N \bool_set_false:c \bool_gset_false:N \bool_gset_false:c <hr/>	\bool_set_false:N <boolean> Sets <boolean> logically false.
<hr/> \bool_set_true:N \bool_set_true:c \bool_gset_true:N \bool_gset_true:c <hr/>	\bool_set_true:N <boolean> Sets <boolean> logically true.
<hr/> \bool_set_eq:NN \bool_set_eq:(cN Nc cc) \bool_gset_eq:NN \bool_gset_eq:(cN Nc cc) <hr/>	\bool_set_eq:NN <boolean ₁ > <boolean ₂ > Sets <boolean ₁ > to the current value of <boolean ₂ >.
<hr/> \bool_set:Nn \bool_set:cn \bool_gset:Nn \bool_gset:cn <hr/> Updated: 2017-07-15	\bool_set:Nn <boolean> {<boolexpr>} Evaluates the <boolean expression> as described for \bool_if:nTF, and sets the <boolean> variable to the logical truth of this evaluation.
<hr/> \bool_if_p:N ★ \bool_if_p:c ★ \bool_if:nTF ★ \bool_if:cTF ★ <hr/> Updated: 2017-07-15	\bool_if_p:N <boolean> \bool_if:NTF <boolean> {<true code>} {<false code>} Tests the current truth of <boolean>, and continues expansion based on this result.
<hr/> \bool_show:N \bool_show:c <hr/> New: 2012-02-09 Updated: 2015-08-01	\bool_show:N <boolean> Displays the logical truth of the <boolean> on the terminal.
<hr/> \bool_show:n <hr/> New: 2012-02-09 Updated: 2017-07-15	\bool_show:n {<boolean expression>} Displays the logical truth of the <boolean expression> on the terminal.
<hr/> \bool_log:N \bool_log:c <hr/> New: 2014-08-22 Updated: 2015-08-03	\bool_log:N <boolean> Writes the logical truth of the <boolean> in the log file.

<code>\bool_log:n</code>	<code>\bool_log:n {⟨boolean expression⟩}</code>
--------------------------	---

New: 2014-08-22	Writes the logical truth of the $\langle boolean\ expression \rangle$ in the log file.
Updated: 2017-07-15	

<code>\bool_if_exist_p:N</code> *	<code>\bool_if_exist_p:N ⟨boolean⟩</code>
<code>\bool_if_exist_p:c</code> *	<code>\bool_if_exist:NTF ⟨boolean⟩ {⟨true code⟩} {⟨false code⟩}</code>
<code>\bool_if_exist:NTF</code> *	Tests whether the $\langle boolean \rangle$ is currently defined. This does not check that the $\langle boolean \rangle$ really is a boolean variable.
<code>\bool_if_exist:cTF</code> *	

New: 2012-03-03

<code>\l_tmpa_bool</code>	A scratch boolean for local assignment. It is never used by the kernel code, and so is safe for use with any L ^A T _E X3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_bool</code>	

<code>\g_tmpa_bool</code>	A scratch boolean for global assignment. It is never used by the kernel code, and so is safe for use with any L ^A T _E X3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_bool</code>	

3 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean $\langle true \rangle$ or $\langle false \rangle$ values, it seems only fitting that we also provide a parser for $\langle boolean\ expressions \rangle$.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean $\langle true \rangle$ or $\langle false \rangle$. It supports the logical operations And, Or and Not as the well-known infix operators `&&` and `||` and prefix `!` with their usual precedences (namely, `&&` binds more tightly than `||`). In addition to this, parentheses can be used to isolate sub-expressions. For example,

```
\int_compare_p:n { 1 = 1 } &&
(
  \int_compare_p:n { 2 = 3 } ||
  \int_compare_p:n { 4 <= 4 } ||
  \str_if_eq_p:nn { abc } { def }
) &&
! \int_compare_p:n { 2 = 4 }
```

is a valid boolean expression.

Contrarily to some other programming languages, the operators `&&` and `||` evaluate both operands in all cases, even when the first operand is enough to determine the result. This “eager” evaluation should be contrasted with the “lazy” evaluation of `\bool_lazy_...` functions.

T_EXhackers note: The eager evaluation of boolean expressions is unfortunately necessary in T_EX. Indeed, a lazy parser can get confused if `&&` or `||` or parentheses appear as (unbraced) arguments of some predicates. For instance, the innocuous-looking expression below would break (in a lazy parser) if `#1` were a closing parenthesis and `\l_tmpa_bool` were `true`.

```
( \l_tmpa_bool || \token_if_eq_meaning_p:NN X #1 )
```

Minimal (lazy) evaluation can be obtained using the conditionals `\bool_lazy_all:nTF`, `\bool_lazy_and:nnTF`, `\bool_lazy_any:nTF`, or `\bool_lazy_or:nnTF`, which only evaluate their boolean expression arguments when they are needed to determine the resulting truth value. For example, when evaluating the boolean expression

```
\bool_lazy_and_p:nn
{
  \bool_lazy_any_p:n
  {
    { \int_compare_p:n { 2 = 3 } }
    { \int_compare_p:n { 4 <= 4 } }
    { \int_compare_p:n { 1 = \error } } % skipped
  }
}
{ ! \int_compare_p:n { 2 = 4 } }
```

the line marked with `skipped` is not expanded because the result of `\bool_lazy_any_p:n` is known once the second boolean expression is found to be logically `true`. On the other hand, the last line is expanded because its logical value is needed to determine the result of `\bool_lazy_and_p:nn`.

<code>\bool_if_p:n *</code> <code>\bool_if:nTF *</code>	<code>\bool_if_p:n {<boolean expression>}</code> <code>\bool_if:nTF {<boolean expression>} {<true code>} {<false code>}</code>
--	---

Updated: 2017-07-15 Tests the current truth of *<boolean expression>*, and continues expansion based on this result. The *<boolean expression>* should consist of a series of predicates or boolean variables with the logical relationship between these defined using `&&` (“And”), `||` (“Or”), `!` (“Not”) and parentheses. The logical Not applies to the next predicate or group.

<code>\bool_lazy_all_p:n *</code> <code>\bool_lazy_all:nTF *</code>	<code>\bool_lazy_all_p:n { {<boolexpr₁>} {<boolexpr₂>} ... {<boolexpr_N>} }</code> <code>\bool_lazy_all:nTF { {<boolexpr₁>} {<boolexpr₂>} ... {<boolexpr_N>} } {<true code>} {<false code>}</code>
--	---

New: 2015-11-15
Updated: 2017-07-15 Implements the “And” operation on the *<boolean expressions>*, hence is `true` if all of them are `true` and `false` if any of them is `false`. Contrarily to the infix operator `&&`, only the *<boolean expressions>* which are needed to determine the result of `\bool_lazy_all:nTF` are evaluated. See also `\bool_lazy_and:nnTF` when there are only two *<boolean expressions>*.

<code>\bool_lazy_and_p:nn *</code> <code>\bool_lazy_and:nnTF *</code>	<code>\bool_lazy_and_p:nn {<boolexpr₁>} {<boolexpr₂>}</code> <code>\bool_lazy_and:nnTF {<boolexpr₁>} {<boolexpr₂>} {<true code>} {<false code>}</code>
--	---

New: 2015-11-15
Updated: 2017-07-15 Implements the “And” operation between two boolean expressions, hence is `true` if both are `true`. Contrarily to the infix operator `&&`, the *<boolexpr₂>* is only evaluated if it is needed to determine the result of `\bool_lazy_and:nnTF`. See also `\bool_lazy_all:nTF` when there are more than two *<boolean expressions>*.

<hr/> \bool_lazy_any_p:n ☆ \bool_lazy_any:nTF ☆ <hr/> New: 2015-11-15 Updated: 2017-07-15	\bool_lazy_any_p:n { {<boolean expr ₁ >} {<boolean expr ₂ >} ... {<boolean expr _N >} } \bool_lazy_any:nTF { {<boolean expr ₁ >} {<boolean expr ₂ >} ... {<boolean expr _N >} } {<true code>} {<false code>} <hr/> Implements the “Or” operation on the <i><boolean expressions></i> , hence is true if any of them is true and false if all of them are false . Contrarily to the infix operator <code> </code> , only the <i><boolean expressions></i> which are needed to determine the result of <code>\bool_lazy_any:nTF</code> are evaluated. See also <code>\bool_lazy_or:nnTF</code> when there are only two <i><boolean expressions></i> .
<hr/> \bool_lazy_or_p:nn ☆ \bool_lazy_or:nnTF ☆ <hr/> New: 2015-11-15 Updated: 2017-07-15	\bool_lazy_or_p:nn {<boolean expr ₁ >} {<boolean expr ₂ >} \bool_lazy_or:nnTF {<boolean expr ₁ >} {<boolean expr ₂ >} {<true code>} {<false code>} <hr/> Implements the “Or” operation between two boolean expressions, hence is true if either one is true . Contrarily to the infix operator <code> </code> , the <i><boolean expr₂></i> is only evaluated if it is needed to determine the result of <code>\bool_lazy_or:nnTF</code> . See also <code>\bool_lazy_any:nTF</code> when there are more than two <i><boolean expressions></i> .
<hr/> \bool_not_p:n ☆ <hr/> Updated: 2017-07-15	\bool_not_p:n {<boolean expression>} <hr/> Function version of <code>!(<boolean expression>)</code> within a boolean expression.
<hr/> \bool_xor_p:nn ☆ \bool_xor:nnTF ☆ <hr/> New: 2018-05-09	\bool_xor_p:nn {<boolean expr ₁ >} {<boolean expr ₂ >} \bool_xor:nnTF {<boolean expr ₁ >} {<boolean expr ₂ >} {<true code>} {<false code>} <hr/> Implements an “exclusive or” operation between two boolean expressions. There is no infix operation for this logical operation.

4 Logical loops

Loops using either boolean expressions or stored boolean values.

<hr/> \bool_do_until:Nn ☆ \bool_do_until:cn ☆ <hr/> Updated: 2017-07-15	\bool_do_until:Nn <boolean> {<code>} <hr/> Places the <i><code></i> in the input stream for T _E X to process, and then checks the logical value of the <i><boolean></i> . If it is false then the <i><code></i> is inserted into the input stream again and the process loops until the <i><boolean></i> is true .
<hr/> \bool_do_while:Nn ☆ \bool_do_while:cn ☆ <hr/> Updated: 2017-07-15	\bool_do_while:Nn <boolean> {<code>} <hr/> Places the <i><code></i> in the input stream for T _E X to process, and then checks the logical value of the <i><boolean></i> . If it is true then the <i><code></i> is inserted into the input stream again and the process loops until the <i><boolean></i> is false .
<hr/> \bool_until_do:Nn ☆ \bool_until_do:cn ☆ <hr/> Updated: 2017-07-15	\bool_until_do:Nn <boolean> {<code>} <hr/> This function firsts checks the logical value of the <i><boolean></i> . If it is false the <i><code></i> is placed in the input stream and expanded. After the completion of the <i><code></i> the truth of the <i><boolean></i> is re-evaluated. The process then loops until the <i><boolean></i> is true .
<hr/> \bool_while_do:Nn ☆ \bool_while_do:cn ☆ <hr/> Updated: 2017-07-15	\bool_while_do:Nn <boolean> {<code>} <hr/> This function firsts checks the logical value of the <i><boolean></i> . If it is true the <i><code></i> is placed in the input stream and expanded. After the completion of the <i><code></i> the truth of the <i><boolean></i> is re-evaluated. The process then loops until the <i><boolean></i> is false .

<hr/> <code>\bool_do_until:nn</code> ☆ <hr/>	<code>\bool_do_until:nn {<boolean expression>} {<code>}</code>
Updated: 2017-07-15 <hr/>	Places the <i><code></i> in the input stream for T _E X to process, and then checks the logical value of the <i><boolean expression></i> as described for <code>\bool_if:nTF</code> . If it is false then the <i><code></i> is inserted into the input stream again and the process loops until the <i><boolean expression></i> evaluates to true .
<hr/> <code>\bool_do_while:nn</code> ☆ <hr/>	<code>\bool_do_while:nn {<boolean expression>} {<code>}</code>
Updated: 2017-07-15 <hr/>	Places the <i><code></i> in the input stream for T _E X to process, and then checks the logical value of the <i><boolean expression></i> as described for <code>\bool_if:nTF</code> . If it is true then the <i><code></i> is inserted into the input stream again and the process loops until the <i><boolean expression></i> evaluates to false .
<hr/> <code>\bool_until_do:nn</code> ☆ <hr/>	<code>\bool_until_do:nn {<boolean expression>} {<code>}</code>
Updated: 2017-07-15 <hr/>	This function firsts checks the logical value of the <i><boolean expression></i> (as described for <code>\bool_if:nTF</code>). If it is false the <i><code></i> is placed in the input stream and expanded. After the completion of the <i><code></i> the truth of the <i><boolean expression></i> is re-evaluated. The process then loops until the <i><boolean expression></i> is true .
<hr/> <code>\bool_while_do:nn</code> ☆ <hr/>	<code>\bool_while_do:nn {<boolean expression>} {<code>}</code>
Updated: 2017-07-15 <hr/>	This function firsts checks the logical value of the <i><boolean expression></i> (as described for <code>\bool_if:nTF</code>). If it is true the <i><code></i> is placed in the input stream and expanded. After the completion of the <i><code></i> the truth of the <i><boolean expression></i> is re-evaluated. The process then loops until the <i><boolean expression></i> is false .

5 Producing multiple copies

<hr/> <code>\prg_replicate:nn</code> ★ <hr/>	<code>\prg_replicate:nn {<integer expression>} {<tokens>}</code>
Updated: 2011-07-04 <hr/>	Evaluates the <i><integer expression></i> (which should be zero or positive) and creates the resulting number of copies of the <i><tokens></i> . The function is both expandable and safe for nesting. It yields its result after two expansion steps.

6 Detecting T_EX's mode

<hr/> <code>\mode_if_horizontal_p:</code> ★ <code>\mode_if_horizontal:TF</code> ★ <hr/>	<code>\mode_if_horizontal_p:</code> <code>\mode_if_horizontal:TF {<true code>} {<false code>}</code>
	Detects if T _E X is currently in horizontal mode.
<hr/> <code>\mode_if_inner_p:</code> ★ <code>\mode_if_inner:TF</code> ★ <hr/>	<code>\mode_if_inner_p:</code> <code>\mode_if_inner:TF {<true code>} {<false code>}</code>
	Detects if T _E X is currently in inner mode.
<hr/> <code>\mode_if_math_p:</code> ★ <code>\mode_if_math:TF</code> ★ <hr/>	<code>\mode_if_math:TF {<true code>} {<false code>}</code>
Updated: 2011-09-05 <hr/>	Detects if T _E X is currently in maths mode.

<code>\mode_if_vertical_p: *</code>	<code>\mode_if_vertical_p:</code>
<code>\mode_if_vertical:TF *</code>	<code>\mode_if_vertical:TF {\langle true code \rangle} {\langle false code \rangle}</code>

Detects if T_EX is currently in vertical mode.

7 Primitive conditionals

<code>\if_predicate:w *</code>	<code>\if_predicate:w \langle predicate \rangle \langle true code \rangle \else: \langle false code \rangle \fi:</code>
--------------------------------	---

This function takes a predicate function and branches according to the result. (In practice this function would also accept a single boolean variable in place of the $\langle predicate \rangle$ but to make the coding clearer this should be done through `\if_bool:N`.)

<code>\if_bool:N *</code>	<code>\if_bool:N \langle boolean \rangle \langle true code \rangle \else: \langle false code \rangle \fi:</code>
---------------------------	--

This function takes a boolean variable and branches according to the result.

8 Nestable recursions and mappings

There are a number of places where recursion or mapping constructs are used in `expl3`. At a low-level, these typically require insertion of tokens at the end of the content to allow “clean up”. To support such mappings in a nestable form, the following functions are provided.

<code>\prg_break_point:Nn *</code>	<code>\prg_break_point:Nn \langle type \rangle_map_break: {\langle code \rangle}</code>
------------------------------------	---

New: 2018-03-26

Used to mark the end of a recursion or mapping: the functions `\langle type \rangle_map_break:` and `\langle type \rangle_map_break:n` use this to break out of the loop (see `\prg_map_break:Nn` for how to set these up). After the loop ends, the $\langle code \rangle$ is inserted into the input stream. This occurs even if the break functions are *not* applied: `\prg_break_point:Nn` is functionally-equivalent in these cases to `\use_ii:nn`.

<code>\prg_map_break:Nn *</code>	<code>\prg_map_break:Nn \langle type \rangle_map_break: {\langle user code \rangle}</code>
----------------------------------	--

New: 2018-03-26

`...`
`\prg_break_point:Nn \langle type \rangle_map_break: {\langle ending code \rangle}`

Breaks a recursion in mapping contexts, inserting in the input stream the $\langle user code \rangle$ after the $\langle ending code \rangle$ for the loop. The function breaks loops, inserting their $\langle ending code \rangle$, until reaching a loop with the same $\langle type \rangle$ as its first argument. This `\langle type \rangle_map_break:` argument must be defined; it is simply used as a recognizable marker for the $\langle type \rangle$.

For types with mappings defined in the kernel, `\langle type \rangle_map_break:` and `\langle type \rangle_map_break:n` are defined as `\prg_map_break:Nn \langle type \rangle_map_break: {}` and the same with `{}` omitted.

8.1 Simple mappings

In addition to the more complex mappings above, non-nestable mappings are used in a number of locations and support is provided for these.

<code>\prg_break_point:</code> *	This copy of <code>\prg_do_nothing:</code> is used to mark the end of a fast short-term recursion:
New: 2018-03-27	the function <code>\prg_break:n</code> uses this to break out of the loop.

<code>\prg_break:</code> *	<code>\prg_break:n {<code>} ... \prg_break_point:</code>
<code>\prg_break:n</code> *	Breaks a recursion which has no <i><ending code></i> and which is not a user-breakable mapping
New: 2018-03-27	(see for instance <code>\prop_get:Nn</code>), and inserts the <i><code></i> in the input stream.

9 Internal programming functions

<code>\group_align_safe_begin:</code> *	<code>\group_align_safe_begin:</code>
<code>\group_align_safe_end:</code> *	...
Updated: 2011-08-11	<code>\group_align_safe_end:</code>

These functions are used to enclose material in a T_EX alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the `&` token inside `\halign`. This is necessary to allow grabbing of tokens for testing purposes, as T_EX uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as `\peek_after:Nw` would result in a forbidden comparison of the internal `\endtemplate` token, yielding a fatal error. Each `\group_align_safe_begin:` must be matched by a `\group_align_safe_end:`, although this does not have to occur within the same function.

Part XIV

The l3sys package: System/runtime functions

1 The name of the job

`\c_sys_jobname_str`

New: 2015-09-19
Updated: 2019-10-27

Constant that gets the “job name” assigned when T_EX starts.

T_EXhackers note: This copies the contents of the primitive `\jobname`. For technical reasons, the string here is not of the same internal form as other, but may be manipulated using normal string functions.

2 Date and time

`\c_sys_minute_int`
`\c_sys_hour_int`
`\c_sys_day_int`
`\c_sys_month_int`
`\c_sys_year_int`

New: 2015-09-22

The date and time at which the current job was started: these are all reported as integers.

T_EXhackers note: Whilst the underlying primitives can be altered by the user, this interface to the time and date is intended to be the “real” values.

3 Engine

`\sys_if_engine luatex_p:` \star
`\sys_if_engine luatex:` *TF* \star
`\sys_if_engine pdftex_p:` \star
`\sys_if_engine pdftex:` *TF* \star
`\sys_if_engine ptex_p:` \star
`\sys_if_engine ptex:` *TF* \star
`\sys_if_engine uptex_p:` \star
`\sys_if_engine uptex:` *TF* \star
`\sys_if_engine xetex_p:` \star
`\sys_if_engine xetex:` *TF* \star

New: 2015-09-07

`\sys_if_engine pdftex:TF` $\{(true\ code)\} \{(false\ code)\}$

Conditionals which allow engine-specific code to be used. The names follow naturally from those of the engine binaries: note that the (u)ptex tests are for ε -pT_EX and ε -upT_EX as expl3 requires the ε -T_EX extensions. Each conditional is true for *exactly one* supported engine. In particular, `\sys_if_engine ptex_p:` is true for ε -pT_EX but false for ε -upT_EX.

`\c_sys_engine_str`

New: 2015-09-19

The current engine given as a lower case string: one of `luatex`, `pdftex`, `ptex`, `uptex` or `xetex`.

4 Output format

```
\sys_if_output_dvi_p: *
\sys_if_output_dvi:TF *
\sys_if_output_pdf_p: *
\sys_if_output_pdf:TF *
```

New: 2015-09-19

`\sys_if_output_dvi:TF` $\{\langle true\ code\rangle\}$ $\{\langle false\ code\rangle\}$

Conditionals which give the current output mode the \TeX run is operating in. This is always one of two outcomes, DVI mode or PDF mode. The two sets of conditionals are thus complementary and are both provided to allow the programmer to emphasise the most appropriate case.

```
\c_sys_output_str
```

New: 2015-09-19

The current output mode given as a lower case string: one of `dvi` or `pdf`.

5 Platform

```
\sys_if_platform_unix_p: *
\sys_if_platform_unix:TF *
\sys_if_platform_windows_p: *
\sys_if_platform_windows:TF *
```

New: 2018-07-27

Conditionals which allow platform-specific code to be used. The names follow the Lua `os.type()` function, *i.e.* all Unix-like systems are `unix` (including Linux and MacOS).

```
\c_sys_platform_str
```

New: 2018-07-27

The current platform given as a lower case string: one of `unix`, `windows` or `unknown`.

6 Random numbers

```
\sys_rand_seed: *
```

New: 2017-05-27

`\sys_rand_seed:`

Expands to the current value of the engine's random seed, a non-negative integer. In engines without random number support this expands to 0.

```
\sys_gset_rand_seed:n
```

New: 2017-05-27

`\sys_gset_rand_seed:n` $\{\langle intexpr\rangle\}$

Globally sets the seed for the engine's pseudo-random number generator to the $\langle integer\ expression\rangle$. This random seed affects all `\..._rand` functions (such as `\int_rand:nn` or `\clist_rand_item:n`) as well as other packages relying on the engine's random number generator. In engines without random number support this produces an error.

\TeX hackers note: While a 32-bit (signed) integer can be given as a seed, only the absolute value is used and any number beyond 2^{28} is divided by an appropriate power of 2. We recommend using an integer in $[0, 2^{28} - 1]$.

7 Access to the shell

<code>\sys_get_shell:nnN</code>	<code>\sys_get_shell:nnN {<shell command>} {<setup>} <tl var></code>
<code>\sys_get_shell:nnNTF</code>	<code>\sys_get_shell:nnNTF {<shell command>} {<setup>} <tl var> {<true code>} {<false code>}</code>

New: 2019-09-20

Defines `<tl>` to the text returned by the `<shell command>`. The `<shell command>` is converted to a string using `\tl_to_str:n`. Category codes may need to be set appropriately via the `<setup>` argument, which is run just before running the `<shell command>` (in a group). If shell escape is disabled, the `<tl var>` will be set to `\q_no_value` in the non-branching version. Note that quote characters (") *cannot* be used inside the `<shell command>`. The `\sys_get_shell:nnNTF` conditional returns `true` if the shell is available and no quote is detected, and `false` otherwise.

<code>\c_sys_shell_escape_int</code>

New: 2017-05-27

This variable exposes the internal triple of the shell escape status. The possible values are

- 0 Shell escape is disabled
- 1 Unrestricted shell escape is enabled
- 2 Restricted shell escape is enabled

<code>\sys_if_shell_p: *</code>	<code>\sys_if_shell_p:</code>
<code>\sys_if_shell:TF *</code>	<code>\sys_if_shell:TF {<true code>} {<false code>}</code>

New: 2017-05-27

Performs a check for whether shell escape is enabled. This returns true if either of restricted or unrestricted shell escape is enabled.

<code>\sys_if_shell_unrestricted_p: *</code>	<code>\sys_if_shell_unrestricted_p:</code>
<code>\sys_if_shell_unrestricted:TF *</code>	<code>\sys_if_shell_unrestricted:TF {<true code>} {<false code>}</code>

New: 2017-05-27

Performs a check for whether *unrestricted* shell escape is enabled.

<code>\sys_if_shell_restricted_p: *</code>	<code>\sys_if_shell_restricted_p:</code>
<code>\sys_if_shell_restricted:TF *</code>	<code>\sys_if_shell_restricted:TF {<true code>} {<false code>}</code>

New: 2017-05-27

Performs a check for whether *restricted* shell escape is enabled. This returns false if unrestricted shell escape is enabled. Unrestricted shell escape is not considered a superset of restricted shell escape in this case. To find whether any shell escape is enabled use `\sys_if_shell:`.

<code>\sys_shell_now:n</code>	<code>\sys_shell_now:n {<tokens>}</code>
<code>\sys_shell_now:x</code>	

New: 2017-05-27

Execute `<tokens>` through shell escape immediately.

<code>\sys_shell_shipout:n</code>	<code>\sys_shell_shipout:n {<tokens>}</code>
<code>\sys_shell_shipout:x</code>	

New: 2017-05-27

Execute `<tokens>` through shell escape at shipout.

7.1 Loading configuration data

<hr/> <code>\sys_load_backend:n</code> <hr/>	<code>\sys_load_backend:n {<backend>}</code>
<hr/> <small>New: 2019-09-12</small> <hr/>	Loads the additional configuration file needed for backend support. If the <i><backend></i> is empty, the standard backend for the engine in use will be loaded. This command may only be used once.
<hr/> <code>\c_sys_backend_str</code> <hr/>	Set to the name of the backend in use by <code>\sys_load_backend:n</code> when issued.
<hr/> <code>\sys_load_debug:</code> <code>\sys_load_deprecation:</code> <hr/>	<code>\sys_load_debug:</code> <code>\sys_load_deprecation:</code>
<hr/> <small>New: 2019-09-12</small> <hr/>	Load the additional configuration files for debugging support and rolling back deprecations, respectively.

7.2 Final settins

<hr/> <code>\sys_finalise:</code> <hr/>	<code>\sys_finalise:</code>
<hr/> <small>New: 2019-10-06</small> <hr/>	Finalises all system-dependent functionality: required before loading a backend.

Part XV

The `l3clist` package

Comma separated lists

Comma lists contain ordered data where items can be added to the left or right end of the list. This data type allows basic list manipulations such as adding/removing items, applying a function to every item, removing duplicate items, extracting a given item, using the comma list with specified separators, and so on. Sequences (defined in `l3seq`) are safer, faster, and provide more features, so they should often be preferred to comma lists. Comma lists are mostly useful when interfacing with L^AT_EX 2_ε or other code that expects or provides comma list data.

Several items can be added at once. To ease input of comma lists from data provided by a user outside an `\ExplSyntaxOn ... \ExplSyntaxOff` block, spaces are removed from both sides of each comma-delimited argument upon input. Blank arguments are ignored, to allow for trailing commas or repeated commas (which may otherwise arise when concatenating comma lists “by hand”). In addition, a set of braces is removed if the result of space-trimming is braced: this allows the storage of any item in a comma list. For instance,

```
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { ~a~ , ~{b}~ , c~\d }
\clist_put_right:Nn \l_my_clist { ~{e}~ , , {{f}} , }
```

results in `\l_my_clist` containing `a,b,c~\d,{e~},{{f}}` namely the five items `a`, `b`, `c~\d`, `e~` and `{f}`. Comma lists normally do not contain empty items so the following gives an empty comma list:

```
\clist_clear_new:N \l_my_clist
\clist_put_right:Nn \l_my_clist { , ~ , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
```

and it leaves `true` in the input stream. To include an “unsafe” item (empty, or one that contains a comma, or starts or ends with a space, or is a single brace group), surround it with braces.

Almost all operations on comma lists are noticeably slower than those on sequences so converting the data to sequences using `\seq_set_from_clist:Nn` (see `l3seq`) may be advisable if speed is important. The exception is that `\clist_if_in:NnTF` and `\clist_remove_duplicates:N` may be faster than their sequence analogues for large lists. However, these functions work slowly for “unsafe” items that must be braced, and may produce errors when their argument contains `{`, `}` or `#` (assuming the usual T_EX category codes apply). The sequence data type should thus certainly be preferred to comma lists to store such items.

1 Creating and initialising comma lists

<code>\clist_new:N</code>	<code>\clist_new:N <comma list></code>
<code>\clist_new:c</code>	

Creates a new *<comma list>* or raises an error if the name is already taken. The declaration is global. The *<comma list>* initially contains no items.

<hr/> <code>\clist_const:Nn</code> <code>\clist_const:(Nx cn cx)</code> <hr/> New: 2014-07-05	<code>\clist_const:Nn <clist var> {<comma list>}</code> Creates a new constant <code><clist var></code> or raises an error if the name is already taken. The value of the <code><clist var></code> is set globally to the <code><comma list></code> .
<hr/> <code>\clist_clear:N</code> <code>\clist_clear:c</code> <code>\clist_gclear:N</code> <code>\clist_gclear:c</code> <hr/>	<code>\clist_clear:N <comma list></code> Clears all items from the <code><comma list></code> .
<hr/> <code>\clist_clear_new:N</code> <code>\clist_clear_new:c</code> <code>\clist_gclear_new:N</code> <code>\clist_gclear_new:c</code> <hr/>	<code>\clist_clear_new:N <comma list></code> Ensures that the <code><comma list></code> exists globally by applying <code>\clist_new:N</code> if necessary, then applies <code>\clist_(g)clear:N</code> to leave the list empty.
<hr/> <code>\clist_set_eq:NN</code> <code>\clist_set_eq:(cN Nc cc)</code> <code>\clist_gset_eq:NN</code> <code>\clist_gset_eq:(cN Nc cc)</code> <hr/>	<code>\clist_set_eq:NN <comma list_{12 Sets the content of <code><comma list_{1 equal to that of <code><comma list_{2.}</code>}</code>}</code>
<hr/> <code>\clist_set_from_seq:NN</code> <code>\clist_set_from_seq:(cN Nc cc)</code> <code>\clist_gset_from_seq:NN</code> <code>\clist_gset_from_seq:(cN Nc cc)</code> <hr/> New: 2014-07-17	<code>\clist_set_from_seq:NN <comma list> <sequence></code> Converts the data in the <code><sequence></code> into a <code><comma list></code> : the original <code><sequence></code> is unchanged. Items which contain either spaces or commas are surrounded by braces.
<hr/> <code>\clist_concat:NNN</code> <code>\clist_concat:ccc</code> <code>\clist_gconcat:NNN</code> <code>\clist_gconcat:ccc</code> <hr/>	<code>\clist_concat:NNN <comma list_{123 Concatenates the content of <code><comma list_{2 and <code><comma list_{3 together and saves the result in <code><comma list_{1. The items in <code><comma list_{2 are placed at the left side of the new comma list.}</code>}</code>}</code>}</code>}</code>
<hr/> <code>\clist_if_exist_p:N *</code> <code>\clist_if_exist_p:c *</code> <code>\clist_if_exist:N\overline{TF} *</code> <code>\clist_if_exist:c\overline{TF} *</code> <hr/> New: 2012-03-03	<code>\clist_if_exist_p:N <comma list></code> <code>\clist_if_exist:NTF <comma list> {<true code>} {<false code>}</code> Tests whether the <code><comma list></code> is currently defined. This does not check that the <code><comma list></code> really is a comma list.

2 Adding data to comma lists

<code>\clist_set:Nn</code>	<code>\clist_set:Nn <comma list> {\<item_1>, ..., \<item_n>}</code>
<code>\clist_set:(NV No Nx cn cV co cx)</code>	
<code>\clist_gset:Nn</code>	
<code>\clist_gset:(NV No Nx cn cV co cx)</code>	

New: 2011-09-06

Sets $\langle comma list \rangle$ to contain the $\langle items \rangle$, removing any previous content from the variable. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To store some $\langle tokens \rangle$ as a single $\langle item \rangle$ even if the $\langle tokens \rangle$ contain commas or spaces, add a set of braces: `\clist_set:Nn <comma list> { {\<tokens>} }`.

<code>\clist_put_left:Nn</code>	<code>\clist_put_left:Nn <comma list> {\<item_1>, ..., \<item_n>}</code>
<code>\clist_put_left:(NV No Nx cn cV co cx)</code>	
<code>\clist_gput_left:Nn</code>	
<code>\clist_gput_left:(NV No Nx cn cV co cx)</code>	

Updated: 2011-09-05

Appends the $\langle items \rangle$ to the left of the $\langle comma list \rangle$. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some $\langle tokens \rangle$ as a single $\langle item \rangle$ even if the $\langle tokens \rangle$ contain commas or spaces, add a set of braces: `\clist_put_left:Nn <comma list> { {\<tokens>} }`.

<code>\clist_put_right:Nn</code>	<code>\clist_put_right:Nn <comma list> {\<item_1>, ..., \<item_n>}</code>
<code>\clist_put_right:(NV No Nx cn cV co cx)</code>	
<code>\clist_gput_right:Nn</code>	
<code>\clist_gput_right:(NV No Nx cn cV co cx)</code>	

Updated: 2011-09-05

Appends the $\langle items \rangle$ to the right of the $\langle comma list \rangle$. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some $\langle tokens \rangle$ as a single $\langle item \rangle$ even if the $\langle tokens \rangle$ contain commas or spaces, add a set of braces: `\clist_put_right:Nn <comma list> { {\<tokens>} }`.

3 Modifying comma lists

While comma lists are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update comma lists, while retaining the order of the unaffected entries.

<code>\clist_remove_duplicates:N</code>	<code>\clist_remove_duplicates:N <comma list></code>
<code>\clist_remove_duplicates:c</code>	
<code>\clist_gremove_duplicates:N</code>	
<code>\clist_gremove_duplicates:c</code>	

Removes duplicate items from the $\langle comma list \rangle$, leaving the left most copy of each item in the $\langle comma list \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

T_EXhackers note: This function iterates through every item in the $\langle comma list \rangle$ and does a comparison with the $\langle items \rangle$ already checked. It is therefore relatively slow with large comma lists. Furthermore, it may fail if any of the items in the $\langle comma list \rangle$ contains `{`, `}`, or `#` (assuming the usual T_EX category codes apply).

<code>\clist_remove_all:Nn</code>	<code>\clist_remove_all:Nn <comma list> {<item>}</code>
<code>\clist_remove_all:cn</code>	
<code>\clist_gremove_all:Nn</code>	
<code>\clist_gremove_all:cn</code>	

Updated: 2011-09-06

Removes every occurrence of $\langle item \rangle$ from the $\langle comma list \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

T_EXhackers note: The function may fail if the $\langle item \rangle$ contains `{`, `}`, or `#` (assuming the usual T_EX category codes apply).

<code>\clist_reverse:N</code>	<code>\clist_reverse:N <comma list></code>
<code>\clist_reverse:c</code>	
<code>\clist_greverse:N</code>	
<code>\clist_greverse:c</code>	

New: 2014-07-18

Reverses the order of items stored in the $\langle comma list \rangle$.

<code>\clist_reverse:n</code>	<code>\clist_reverse:n {<comma list>}</code>
-------------------------------	--

New: 2014-07-18

Leaves the items in the $\langle comma list \rangle$ in the input stream in reverse order. Contrarily to other what is done for other n-type $\langle comma list \rangle$ arguments, braces and spaces are preserved by this process.

T_EXhackers note: The result is returned within `\unexpanded`, which means that the comma list does not expand further when appearing in an x-type or e-type argument expansion.

<code>\clist_sort:Nn</code>	<code>\clist_sort:Nn <clist var> {<comparison code>}</code>
<code>\clist_sort:cn</code>	
<code>\clist_gsort:Nn</code>	
<code>\clist_gsort:cn</code>	

New: 2017-02-06

Sorts the items in the $\langle clist var \rangle$ according to the $\langle comparison code \rangle$, and assigns the result to $\langle clist var \rangle$. The details of sorting comparison are described in Section 1.

4 Comma list conditionals

<code>\clist_if_empty_p:N</code> *	<code>\clist_if_empty_p:N</code> $\langle comma list \rangle$
<code>\clist_if_empty_p:c</code> *	<code>\clist_if_empty:N</code> $\langle comma list \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\clist_if_empty:N</code> <u><i>TF</i></u> *	Tests if the $\langle comma list \rangle$ is empty (containing no items).
<code>\clist_if_empty:c</code> <u><i>TF</i></u> *	

<code>\clist_if_empty_p:n</code> *	<code>\clist_if_empty_p:n</code> $\{\langle comma list \rangle\}$
<code>\clist_if_empty:n</code> <u><i>TF</i></u> *	<code>\clist_if_empty:n</code> $\langle comma list \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

New: 2014-07-05

Tests if the $\langle comma list \rangle$ is empty (containing no items). The rules for space trimming are as for other n-type comma-list functions, hence the comma list $\{\sim, \sim, \sim\}$ (without outer braces) is empty, while $\{\sim, \{ \}, \}$ (without outer braces) contains one element, which happens to be empty: the comma-list is not empty.

<code>\clist_if_in:N</code> <u><i>nnTF</i></u>	<code>\clist_if_in:N</code> $\langle comma list \rangle$ $\langle item \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\clist_if_in:(NV No cn cV co)</code> <u><i>TF</i></u>	
<code>\clist_if_in:nn</code> <u><i>TF</i></u>	
<code>\clist_if_in:(nV no)</code> <u><i>TF</i></u>	

Updated: 2011-09-06

Tests if the $\langle item \rangle$ is present in the $\langle comma list \rangle$. In the case of an n-type $\langle comma list \rangle$, the usual rules of space trimming and brace stripping apply. Hence,

`\clist_if_in:nnTF { a , {b}~ , {b} , c } { b } {true} {false}`

yields true.

T_EXhackers note: The function may fail if the $\langle item \rangle$ contains $\{$, $\}$, or $\#$ (assuming the usual T_EX category codes apply).

5 Mapping to comma lists

The functions described in this section apply a specified function to each item of a comma list. All mappings are done at the current group level, *i.e.* any local assignments made by the $\langle function \rangle$ or $\langle code \rangle$ discussed below remain in effect after the loop.

When the comma list is given explicitly, as an n-type argument, spaces are trimmed around each item. If the result of trimming spaces is empty, the item is ignored. Otherwise, if the item is surrounded by braces, one set is removed, and the result is passed to the mapped function. Thus, if the comma list that is being mapped is $\{a_{_},_{_}\{b\}_{_},_{_},\{ \},_{_}\{c\},\}$ then the arguments passed to the mapped function are ‘a’, ‘{b}_{_}’, an empty argument, and ‘c’.

When the comma list is given as an N-type argument, spaces have already been trimmed on input, and items are simply stripped of one set of braces if any. This case is more efficient than using n-type comma lists.

<code>\clist_map_function:NN</code> ☆	<code>\clist_map_function:NN</code> $\langle comma list \rangle$ $\langle function \rangle$
<code>\clist_map_function:cN</code> ☆	Applies $\langle function \rangle$ to every $\langle item \rangle$ stored in the $\langle comma list \rangle$. The $\langle function \rangle$ receives one argument for each iteration. The $\langle items \rangle$ are returned from left to right. The function
<code>\clist_map_function:nN</code> ☆	<code>\clist_map_inline:Nn</code> is in general more efficient than <code>\clist_map_function:NN</code> .

Updated: 2012-06-29

```
\clist_map_inline:Nn
\clist_map_inline:cn
\clist_map_inline:nn
```

Updated: 2012-06-29

```
\clist_map_inline:Nn <comma list> {<inline function>}
```

Applies *<inline function>* to every *<item>* stored within the *<comma list>*. The *<inline function>* should consist of code which receives the *<item>* as #1. The *<items>* are returned from left to right.

```
\clist_map_variable:NNn
\clist_map_variable:cNn
\clist_map_variable:nNn
```

Updated: 2012-06-29

```
\clist_map_variable:NNn <comma list> <variable> {<code>}
```

Stores each *<item>* of the *<comma list>* in turn in the (token list) *<variable>* and applies the *<code>*. The *<code>* will usually make use of the *<variable>*, but this is not enforced. The assignments to the *<variable>* are local. Its value after the loop is the last *<item>* in the *<comma list>*, or its original value if there were no *<item>*. The *<items>* are returned from left to right.

```
\clist_map_break: ☆
```

Updated: 2012-06-29

```
\clist_map_break:
```

Used to terminate a `\clist_map_...` function before all entries in the *<comma list>* have been processed. This normally takes place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \clist_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\clist_map_...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\clist_map_break:n` ☆

Updated: 2012-06-29

`\clist_map_break:n` {<code>}

Used to terminate a `\clist_map_...` function before all entries in the <comma list> have been processed, inserting the <code> after the mapping has ended. This normally takes place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \clist_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\clist_map_...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the <code> is inserted into the input stream. This depends on the design of the mapping function.

`\clist_count:N` ★

`\clist_count:c` ★

`\clist_count:n` ★

New: 2012-07-13

`\clist_count:N` <comma list>

Leaves the number of items in the <comma list> in the input stream as an <integer denotation>. The total number of items in a <comma list> includes those which are duplicates, *i.e.* every item in a <comma list> is counted.

6 Using the content of comma lists directly

`\clist_use:Nnnn` ★

`\clist_use:cnnn` ★

New: 2013-05-26

`\clist_use:Nnnn` <clist var> {<separator between two>}

{<separator between more than two>} {<separator between final two>}

Places the contents of the <clist var> in the input stream, with the appropriate <separator> between the items. Namely, if the comma list has more than two items, the <separator between more than two> is placed between each pair of items except the last, for which the <separator between final two> is used. If the comma list has exactly two items, then they are placed in the input stream separated by the <separator between two>. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nnnn \l_tmpa_clist { ~and~ } { ,~ } { ,~and~ }
```

inserts “a, b, c, de, and f” in the input stream. The first separator argument is not used in this case because the comma list has more than 2 items.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the <items> do not expand further when appearing in an x-type argument expansion.

```
\clist_use:Nn ★
\clist_use:cn ★
```

New: 2013-05-26

```
\clist_use:Nn <clist var> {<separator>}
```

Places the contents of the $\langle\textit{clist var}\rangle$ in the input stream, with the $\langle\textit{separator}\rangle$ between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nn \l_tmpa_clist { ~and~ }
```

inserts “a and b and c and de and f” in the input stream.

TeXhackers note: The result is returned within the $\backslash\textit{unexpanded}$ primitive ($\backslash\textit{exp_not:n}$), which means that the $\langle\textit{items}\rangle$ do not expand further when appearing in an \textit{x} -type argument expansion.

7 Comma lists as stacks

Comma lists can be used as stacks, where data is pushed to and popped from the top of the comma list. (The left of a comma list is the top, for performance reasons.) The stack functions for comma lists are not intended to be mixed with the general ordered data functions detailed in the previous section: a comma list should either be used as an ordered data type or as a stack, but not in both ways.

```
\clist_get:NN
\clist_get:cn
\clist_get:NNTF
\clist_get:cNTF
```

New: 2012-05-14
Updated: 2019-02-16

```
\clist_get:NN <comma list> <token list variable>
```

Stores the left-most item from a $\langle\textit{comma list}\rangle$ in the $\langle\textit{token list variable}\rangle$ without removing it from the $\langle\textit{comma list}\rangle$. The $\langle\textit{token list variable}\rangle$ is assigned locally. In the non-branching version, if the $\langle\textit{comma list}\rangle$ is empty the $\langle\textit{token list variable}\rangle$ is set to the marker value $\backslash\textit{q_no_value}$.

```
\clist_pop:NN
\clist_pop:cn
```

Updated: 2011-09-06

```
\clist_pop:NN <comma list> <token list variable>
```

Pops the left-most item from a $\langle\textit{comma list}\rangle$ into the $\langle\textit{token list variable}\rangle$, *i.e.* removes the item from the comma list and stores it in the $\langle\textit{token list variable}\rangle$. Both of the variables are assigned locally.

```
\clist_gpop:NN
\clist_gpop:cn
```

```
\clist_gpop:NN <comma list> <token list variable>
```

Pops the left-most item from a $\langle\textit{comma list}\rangle$ into the $\langle\textit{token list variable}\rangle$, *i.e.* removes the item from the comma list and stores it in the $\langle\textit{token list variable}\rangle$. The $\langle\textit{comma list}\rangle$ is modified globally, while the assignment of the $\langle\textit{token list variable}\rangle$ is local.

```
\clist_pop:NNTF
\clist_pop:cNTF
```

New: 2012-05-14

```
\clist_pop:NNTF <comma list> <token list variable> {<true code>} {<false code>}
```

If the $\langle\textit{comma list}\rangle$ is empty, leaves the $\langle\textit{false code}\rangle$ in the input stream. The value of the $\langle\textit{token list variable}\rangle$ is not defined in this case and should not be relied upon. If the $\langle\textit{comma list}\rangle$ is non-empty, pops the top item from the $\langle\textit{comma list}\rangle$ in the $\langle\textit{token list variable}\rangle$, *i.e.* removes the item from the $\langle\textit{comma list}\rangle$. Both the $\langle\textit{comma list}\rangle$ and the $\langle\textit{token list variable}\rangle$ are assigned locally.

<hr/> <code>\clist_gpop:NNTF</code> <hr/>	<code>\clist_gpop:NNTF <comma list> <token list variable> {\true code} {\false code}</code>
<code>\clist_gpop:cNTF</code> <hr/>	
New: 2012-05-14	

If the *<comma list>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<comma list>* is non-empty, pops the top item from the *<comma list>* in the *<token list variable>*, *i.e.* removes the item from the *<comma list>*. The *<comma list>* is modified globally, while the *<token list variable>* is assigned locally.

<hr/> <code>\clist_push:Nn</code> <hr/>	<code>\clist_push:Nn <comma list> {\items}</code>
<code>\clist_push:(NV No Nx cn cV co cx)</code>	
<code>\clist_gpush:Nn</code>	
<code>\clist_gpush:(NV No Nx cn cV co cx)</code> <hr/>	

Adds the *{\items}* to the top of the *<comma list>*. Spaces are removed from both sides of each item as for any n-type comma list.

8 Using a single item

<hr/> <code>\clist_item:Nn *</code> <hr/>	<code>\clist_item:Nn <comma list> {\integer expression}</code>
<code>\clist_item:cn *</code>	
<code>\clist_item:nn *</code> <hr/>	
New: 2014-07-17	

Indexing items in the *<comma list>* from 1 at the top (left), this function evaluates the *<integer expression>* and leaves the appropriate item from the comma list in the input stream. If the *<integer expression>* is negative, indexing occurs from the bottom (right) of the comma list. When the *<integer expression>* is larger than the number of items in the *<comma list>* (as calculated by `\clist_count:N`) then the function expands to nothing.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<item>* does not expand further when appearing in an x-type argument expansion.

<hr/> <code>\clist_rand_item:N *</code> <hr/>	<code>\clist_rand_item:N <clist var></code>
<code>\clist_rand_item:c *</code>	<code>\clist_rand_item:n {\comma list}</code>
<code>\clist_rand_item:n *</code> <hr/>	
New: 2016-12-06	

Selects a pseudo-random item of the *<comma list>*. If the *<comma list>* has no item, the result is empty.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<item>* does not expand further when appearing in an x-type argument expansion.

9 Viewing comma lists

<hr/> <code>\clist_show:N</code> <hr/>	<code>\clist_show:N <comma list></code>
<code>\clist_show:c</code> <hr/>	
Updated: 2015-08-03	

Displays the entries in the *<comma list>* in the terminal.

<hr/> <code>\clist_show:n</code> <hr/>	<code>\clist_show:n {\tokens}</code>
Updated: 2013-08-03	Displays the entries in the comma list in the terminal.
<hr/>	
<code>\clist_log:N</code> <code>\clist_log:c</code> <hr/>	<code>\clist_log:N <comma list></code>
New: 2014-08-22 Updated: 2015-08-03	Writes the entries in the <i><comma list></i> in the log file. See also <code>\clist_show:N</code> which displays the result in the terminal.
<hr/>	
<code>\clist_log:n</code> <hr/>	<code>\clist_log:n {\tokens}</code>
New: 2014-08-22	Writes the entries in the comma list in the log file. See also <code>\clist_show:n</code> which displays the result in the terminal.

10 Constant and scratch comma lists

<hr/> <code>\c_empty_clist</code> <hr/>	Constant that is always empty.
New: 2012-07-02	
<hr/>	
<code>\l_tmpa_clist</code> <code>\l_tmpb_clist</code> <hr/>	Scratch comma lists for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
New: 2011-09-06	
<hr/>	
<code>\g_tmpa_clist</code> <code>\g_tmpb_clist</code> <hr/>	Scratch comma lists for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
New: 2011-09-06	

Part XVI

The l3token package

Token manipulation

This module deals with tokens. Now this is perhaps not the most precise description so let's try with a better description: When programming in T_EX, it is often desirable to know just what a certain token is: is it a control sequence or something else. Similarly one often needs to know if a control sequence is expandable or not, a macro or a primitive, how many arguments it takes etc. Another thing of great importance (especially when it comes to document commands) is looking ahead in the token stream to see if a certain character is present and maybe even remove it or disregard other tokens while scanning. This module provides functions for both and as such has two primary function categories: `\token_` for anything that deals with tokens and `\peek_` for looking ahead in the token stream.

Most functions we describe here can be used on control sequences, as those are tokens as well.

It is important to distinguish two aspects of a token: its “shape” (for lack of a better word), which affects the matching of delimited arguments and the comparison of token lists containing this token, and its “meaning”, which affects whether the token expands or what operation it performs. One can have tokens of different shapes with the same meaning, but not the converse.

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three names for the same internal operation of T_EX, namely the primitive testing the next two characters for equality of their character code. They have the same meaning hence behave identically in many situations. However, T_EX distinguishes them when searching for a delimited argument. Namely, the example function `\show_until_if:w` defined below takes everything until `\if:w` as an argument, despite the presence of other copies of `\if:w` under different names.

```
\cs_new:Npn \show_until_if:w #1 \if:w { \tl_show:n {#1} }
\show_until_if:w \tex_if:D \if_charcode:w \if:w
```

A list of all possible shapes and a list of all possible meanings are given in section 7.

1 Creating character tokens

```
\char_set_active_eq:NN
\char_set_active_eq:Nc
\char_gset_active_eq:NN
\char_gset_active_eq:Nc
```

Updated: 2015-11-12

```
\char_set_active_eq:NN <char> <function>
```

Sets the behaviour of the `<char>` in situations where it is active (category code 13) to be equivalent to that of the `<function>`. The category code of the `<char>` is *unchanged* by this process. The `<function>` may itself be an active character.

```
\char_set_active_eq:nN
\char_set_active_eq:nc
\char_gset_active_eq:nN
\char_gset_active_eq:nc
```

New: 2015-11-12

```
\char_set_active_eq:nN {<integer expression>} <function>
```

Sets the behaviour of the `<char>` which has character code as given by the `<integer expression>` in situations where it is active (category code 13) to be equivalent to that of the `<function>`. The category code of the `<char>` is *unchanged* by this process. The `<function>` may itself be an active character.

<hr/>	
<code>\char_generate:nn</code> ★	<code>\char_generate:nn {<charcode>} {<catcode>}</code>
New: 2015-09-09	Generates a character token of the given <i><charcode></i> and <i><catcode></i> (both of which may be integer expressions). The <i><catcode></i> may be one of
Updated: 2019-01-16	
<hr/>	

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 11 (letter)
- 12 (other)
- 13 (active)

and other values raise an error. The *<charcode>* may be any one valid for the engine in use. Active characters cannot be generated in older versions of X_YTeX.

T_EXhackers note: Exactly two expansions are needed to produce the character.

<hr/>	
<code>\char_lowercase:N</code> ★	<code>\char_lowercase:N <char></code>
<code>\char_uppercase:N</code> ★	
<code>\char_titlecase:N</code> ★	Converts the <i><char></i> to the equivalent case-changed character as detailed by the function name (see <code>\str_foldcase:n</code> and <code>\text_titlecase:n</code> for details of these terms).
<code>\char_foldcase:N</code> ★	The case mapping is carried out with no context-dependence (<i>cf.</i> <code>\text_uppercase:n</code> , <i>etc.</i>) The str versions always generate “other” (category code 12) characters, whilst the standard versions generate characters with the category code of the <i><char></i> (i.e. only the character code changes).
<code>\char_str_lowercase:N</code> ★	
<code>\char_str_uppercase:N</code> ★	
<code>\char_str_titlecase:N</code> ★	
<code>\char_str_foldcase:N</code> ★	
<hr/>	
New: 2020-01-09	
<hr/>	

<hr/>	
<code>\c_catcode_other_space_tl</code>	Token list containing one character with category code 12, (“other”), and character code 32 (space).
New: 2011-09-05	
<hr/>	

2 Manipulating and interrogating character tokens

<code>\char_set_catcode_escape:N</code>	<code>\char_set_catcode_letter:N</code> $\langle character \rangle$
<code>\char_set_catcode_group_begin:N</code>	
<code>\char_set_catcode_group_end:N</code>	
<code>\char_set_catcode_math_toggle:N</code>	
<code>\char_set_catcode_alignment:N</code>	
<code>\char_set_catcode_end_line:N</code>	
<code>\char_set_catcode_parameter:N</code>	
<code>\char_set_catcode_math_superscript:N</code>	
<code>\char_set_catcode_math_subscript:N</code>	
<code>\char_set_catcode_ignore:N</code>	
<code>\char_set_catcode_space:N</code>	
<code>\char_set_catcode_letter:N</code>	
<code>\char_set_catcode_other:N</code>	
<code>\char_set_catcode_active:N</code>	
<code>\char_set_catcode_comment:N</code>	
<code>\char_set_catcode_invalid:N</code>	

Updated: 2015-11-11

Sets the category code of the $\langle character \rangle$ to that indicated in the function name. Depending on the current category code of the $\langle token \rangle$ the escape token may also be needed:

`\char_set_catcode_other:N \%`

The assignment is local.

<code>\char_set_catcode_escape:n</code>	<code>\char_set_catcode_letter:n</code> $\{ \langle integer\ expression \rangle \}$
<code>\char_set_catcode_group_begin:n</code>	
<code>\char_set_catcode_group_end:n</code>	
<code>\char_set_catcode_math_toggle:n</code>	
<code>\char_set_catcode_alignment:n</code>	
<code>\char_set_catcode_end_line:n</code>	
<code>\char_set_catcode_parameter:n</code>	
<code>\char_set_catcode_math_superscript:n</code>	
<code>\char_set_catcode_math_subscript:n</code>	
<code>\char_set_catcode_ignore:n</code>	
<code>\char_set_catcode_space:n</code>	
<code>\char_set_catcode_letter:n</code>	
<code>\char_set_catcode_other:n</code>	
<code>\char_set_catcode_active:n</code>	
<code>\char_set_catcode_comment:n</code>	
<code>\char_set_catcode_invalid:n</code>	

Updated: 2015-11-11

Sets the category code of the $\langle character \rangle$ which has character code as given by the $\langle integer\ expression \rangle$. This version can be used to set up characters which cannot otherwise be given (*cf.* the N-type variants). The assignment is local.

<hr/> \char_set_catcode:nn <hr/>	\char_set_catcode:nn {<intexpr ₁ >} {<intexpr ₂ >}
Updated: 2015-11-11	These functions set the category code of the <character> which has character code as given by the <integer expression>. The first <integer expression> is the character code and the second is the category code to apply. The setting applies within the current T _E X group. In general, the symbolic functions \char_set_catcode_<type> should be preferred, but there are cases where these lower-level functions may be useful.
<hr/> \char_value_catcode:n ★ <hr/>	\char_value_catcode:n {<integer expression>}
	Expands to the current category code of the <character> with character code given by the <integer expression>.
<hr/> \char_show_value_catcode:n <hr/>	\char_show_value_catcode:n {<integer expression>}
	Displays the current category code of the <character> with character code given by the <integer expression> on the terminal.
<hr/> \char_set_lccode:nn <hr/>	\char_set_lccode:nn {<intexpr ₁ >} {<intexpr ₂ >}
Updated: 2015-08-06	Sets up the behaviour of the <character> when found inside \text_lowercase:n , such that <character ₁ > will be converted into <character ₂ >. The two <characters> may be specified using an <integer expression> for the character code concerned. This may include the T _E X ‘<character>’ method for converting a single character into its character code:
	<pre> \char_set_lccode:nn { ‘\A } { ‘a } % Standard behaviour \char_set_lccode:nn { ‘\A } { ‘\A + 32 } \char_set_lccode:nn { 50 } { 60 } </pre>
	The setting applies within the current T _E X group.
<hr/> \char_value_lccode:n ★ <hr/>	\char_value_lccode:n {<integer expression>}
	Expands to the current lower case code of the <character> with character code given by the <integer expression>.
<hr/> \char_show_value_lccode:n <hr/>	\char_show_value_lccode:n {<integer expression>}
	Displays the current lower case code of the <character> with character code given by the <integer expression> on the terminal.
<hr/> \char_set_uccode:nn <hr/>	\char_set_uccode:nn {<intexpr ₁ >} {<intexpr ₂ >}
Updated: 2015-08-06	Sets up the behaviour of the <character> when found inside \text_uppercase:n , such that <character ₁ > will be converted into <character ₂ >. The two <characters> may be specified using an <integer expression> for the character code concerned. This may include the T _E X ‘<character>’ method for converting a single character into its character code:
	<pre> \char_set_uccode:nn { ‘a } { ‘\A } % Standard behaviour \char_set_uccode:nn { ‘\A } { ‘\A - 32 } \char_set_uccode:nn { 60 } { 50 } </pre>
	The setting applies within the current T _E X group.

<hr/> <hr/> <code>\char_value_uccode:n</code> ★	<code>\char_value_uccode:n {⟨integer expression⟩}</code>
	Expands to the current upper case code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> .
<hr/> <hr/> <code>\char_show_value_uccode:n</code>	<code>\char_show_value_uccode:n {⟨integer expression⟩}</code>
	Displays the current upper case code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> on the terminal.
<hr/> <hr/> <code>\char_set_mathcode:nn</code>	<code>\char_set_mathcode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
Updated: 2015-08-06	This function sets up the math code of <i>⟨character⟩</i> . The <i>⟨character⟩</i> is specified as an <i>⟨integer expression⟩</i> which will be used as the character code of the relevant character. The setting applies within the current T _E X group.
<hr/> <hr/> <code>\char_value_mathcode:n</code> ★	<code>\char_value_mathcode:n {⟨integer expression⟩}</code>
	Expands to the current math code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> .
<hr/> <hr/> <code>\char_show_value_mathcode:n</code>	<code>\char_show_value_mathcode:n {⟨integer expression⟩}</code>
	Displays the current math code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> on the terminal.
<hr/> <hr/> <code>\char_set_sfcode:nn</code>	<code>\char_set_sfcode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
Updated: 2015-08-06	This function sets up the space factor for the <i>⟨character⟩</i> . The <i>⟨character⟩</i> is specified as an <i>⟨integer expression⟩</i> which will be used as the character code of the relevant character. The setting applies within the current T _E X group.
<hr/> <hr/> <code>\char_value_sfcode:n</code> ★	<code>\char_value_sfcode:n {⟨integer expression⟩}</code>
	Expands to the current space factor for the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> .
<hr/> <hr/> <code>\char_show_value_sfcode:n</code>	<code>\char_show_value_sfcode:n {⟨integer expression⟩}</code>
	Displays the current space factor for the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> on the terminal.
<hr/> <hr/> <code>\l_char_active_seq</code>	Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category <i>⟨active⟩</i> (catcode 13). Each entry in the sequence consists of a single escaped token, for example <code>\~</code> . Active tokens should be added to the sequence when they are defined for general document use.
New: 2012-01-23 Updated: 2015-11-11	
<hr/> <hr/> <code>\l_char_special_seq</code>	Used to track which tokens will require special handling when working with verbatim-like material at the document level as they are not of categories <i>⟨letter⟩</i> (catcode 11) or <i>⟨other⟩</i> (catcode 12). Each entry in the sequence consists of a single escaped token, for example <code>\</code> for the backslash or <code>\{</code> for an opening brace. Escaped tokens should be added to the sequence when they are defined for general document use.
New: 2012-01-23 Updated: 2015-11-11	

3 Generic tokens

```
\c_group_begin_token
\c_group_end_token
\c_math_toggle_token
\c_alignment_token
\c_parameter_token
\c_math_superscript_token
\c_math_subscript_token
\c_space_token
```

These are implicit tokens which have the category code described by their name. They are used internally for test purposes but are also available to the programmer for other uses.

```
\c_catcode_letter_token
\c_catcode_other_token
```

These are implicit tokens which have the category code described by their name. They are used internally for test purposes and should not be used other than for category code tests.

```
\c_catcode_active_tl
```

A token list containing an active token. This is used internally for test purposes and should not be used other than in appropriately-constructed category code tests.

4 Converting tokens

```
\token_to_meaning:N *
\token_to_meaning:c *
```

`\token_to_meaning:N` $\langle token \rangle$

Inserts the current meaning of the $\langle token \rangle$ into the input stream as a series of characters of category code 12 (other). This is the primitive \TeX description of the $\langle token \rangle$, thus for example both functions defined by `\cs_set_nopar:Npn` and token list variables defined using `\tl_new:N` are described as macros.

\TeX hackers note: This is the \TeX primitive `\meaning`. The $\langle token \rangle$ can thus be an explicit space tokens or an explicit begin-group or end-group character token (`{` or `}` when normal \TeX category codes apply) even though these are not valid N-type arguments.

```
\token_to_str:N *
\token_to_str:c *
```

`\token_to_str:N` $\langle token \rangle$

Converts the given $\langle token \rangle$ into a series of characters with category code 12 (other). If the $\langle token \rangle$ is a control sequence, this will start with the current escape character with category code 12 (the escape character is part of the $\langle token \rangle$). This function requires only a single expansion.

\TeX hackers note: `\token_to_str:N` is the \TeX primitive `\string` renamed. The $\langle token \rangle$ can thus be an explicit space tokens or an explicit begin-group or end-group character token (`{` or `}` when normal \TeX category codes apply) even though these are not valid N-type arguments.

5 Token conditionals

<code>\token_if_group_begin_p:N</code>	<code>*</code>	<code>\token_if_group_begin_p:N</code>	<code><token></code>
<code>\token_if_group_begin:NTF</code>	<code>*</code>	<code>\token_if_group_begin:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of a begin group token (`{` when normal `TEX` category codes are in force). Note that an explicit begin group token cannot be tested in this way, as it is not a valid N-type argument.

<code>\token_if_group_end_p:N</code>	<code>*</code>	<code>\token_if_group_end_p:N</code>	<code><token></code>
<code>\token_if_group_end:NTF</code>	<code>*</code>	<code>\token_if_group_end:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of an end group token (`}` when normal `TEX` category codes are in force). Note that an explicit end group token cannot be tested in this way, as it is not a valid N-type argument.

<code>\token_if_math_toggle_p:N</code>	<code>*</code>	<code>\token_if_math_toggle_p:N</code>	<code><token></code>
<code>\token_if_math_toggle:NTF</code>	<code>*</code>	<code>\token_if_math_toggle:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of a math shift token (`$` when normal `TEX` category codes are in force).

<code>\token_if_alignment_p:N</code>	<code>*</code>	<code>\token_if_alignment_p:N</code>	<code><token></code>
<code>\token_if_alignment:NTF</code>	<code>*</code>	<code>\token_if_alignment:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of an alignment token (`&` when normal `TEX` category codes are in force).

<code>\token_if_parameter_p:N</code>	<code>*</code>	<code>\token_if_parameter_p:N</code>	<code><token></code>
<code>\token_if_parameter:NTF</code>	<code>*</code>	<code>\token_if_parameter:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of a macro parameter token (`#` when normal `TEX` category codes are in force).

<code>\token_if_math_superscript_p:N</code>	<code>*</code>	<code>\token_if_math_superscript_p:N</code>	<code><token></code>
<code>\token_if_math_superscript:NTF</code>	<code>*</code>	<code>\token_if_math_superscript:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of a superscript token (`^` when normal `TEX` category codes are in force).

<code>\token_if_math_subscript_p:N</code>	<code>*</code>	<code>\token_if_math_subscript_p:N</code>	<code><token></code>
<code>\token_if_math_subscript:NTF</code>	<code>*</code>	<code>\token_if_math_subscript:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of a subscript token (`_` when normal `TEX` category codes are in force).

<code>\token_if_space_p:N</code>	<code>*</code>	<code>\token_if_space_p:N</code>	<code><token></code>
<code>\token_if_space:NTF</code>	<code>*</code>	<code>\token_if_space:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of a space token. Note that an explicit space token with character code 32 cannot be tested in this way, as it is not a valid N-type argument.

<code>\token_if_letter_p:N</code>	<code>\token_if_letter_p:N</code>	<code>\token</code>
<code>\token_if_letter:NTF</code>	<code>\token_if_letter:NTF</code>	<code>\token</code> <code>{\true code}</code> <code>{\false code}</code>

Tests if `\token` has the category code of a letter token.

<code>\token_if_other_p:N</code>	<code>\token_if_other_p:N</code>	<code>\token</code>
<code>\token_if_other:NTF</code>	<code>\token_if_other:NTF</code>	<code>\token</code> <code>{\true code}</code> <code>{\false code}</code>

Tests if `\token` has the category code of an “other” token.

<code>\token_if_active_p:N</code>	<code>\token_if_active_p:N</code>	<code>\token</code>
<code>\token_if_active:NTF</code>	<code>\token_if_active:NTF</code>	<code>\token</code> <code>{\true code}</code> <code>{\false code}</code>

Tests if `\token` has the category code of an active character.

<code>\token_if_eq_catcode_p:NN</code>	<code>\token_if_eq_catcode_p:NN</code>	<code>\token₁</code>	<code>\token₂</code>
<code>\token_if_eq_catcode:NNTF</code>	<code>\token_if_eq_catcode:NNTF</code>	<code>\token₁</code>	<code>\token₂</code> <code>{\true code}</code> <code>{\false code}</code>

Tests if the two `\tokens` have the same category code.

<code>\token_if_eq_charcode_p:NN</code>	<code>\token_if_eq_charcode_p:NN</code>	<code>\token₁</code>	<code>\token₂</code>
<code>\token_if_eq_charcode:NNTF</code>	<code>\token_if_eq_charcode:NNTF</code>	<code>\token₁</code>	<code>\token₂</code> <code>{\true code}</code> <code>{\false code}</code>

Tests if the two `\tokens` have the same character code.

<code>\token_if_eq_meaning_p:NN</code>	<code>\token_if_eq_meaning_p:NN</code>	<code>\token₁</code>	<code>\token₂</code>
<code>\token_if_eq_meaning:NNTF</code>	<code>\token_if_eq_meaning:NNTF</code>	<code>\token₁</code>	<code>\token₂</code> <code>{\true code}</code> <code>{\false code}</code>

Tests if the two `\tokens` have the same meaning when expanded.

<code>\token_if_macro_p:N</code>	<code>\token_if_macro_p:N</code>	<code>\token</code>
<code>\token_if_macro:NTF</code>	<code>\token_if_macro:NTF</code>	<code>\token</code> <code>{\true code}</code> <code>{\false code}</code>

Updated: 2011-05-23 Tests if the `\token` is a \TeX macro.

<code>\token_if_cs_p:N</code>	<code>\token_if_cs_p:N</code>	<code>\token</code>
<code>\token_if_cs:NTF</code>	<code>\token_if_cs:NTF</code>	<code>\token</code> <code>{\true code}</code> <code>{\false code}</code>

Tests if the `\token` is a control sequence.

<code>\token_if_expandable_p:N</code>	<code>\token_if_expandable_p:N</code>	<code>\token</code>
<code>\token_if_expandable:NTF</code>	<code>\token_if_expandable:NTF</code>	<code>\token</code> <code>{\true code}</code> <code>{\false code}</code>

Tests if the `\token` is expandable. This test returns `\false` for an undefined token.

<code>\token_if_long_macro_p:N</code>	<code>\token_if_long_macro_p:N</code>	<code>\token</code>
<code>\token_if_long_macro:NTF</code>	<code>\token_if_long_macro:NTF</code>	<code>\token</code> <code>{\true code}</code> <code>{\false code}</code>

Updated: 2012-01-20 Tests if the `\token` is a long macro.

<code>\token_if_protected_macro_p:N</code>	<code>\token_if_protected_macro_p:N</code>	<code>\token</code>
<code>\token_if_protected_macro:NTF</code>	<code>\token_if_protected_macro:NTF</code>	<code>\token</code> <code>{\true code}</code> <code>{\false code}</code>

Updated: 2012-01-20

Tests if the `\token` is a protected macro: for a macro which is both protected and long this returns `false`.

<code>\token_if_protected_long_macro_p:N</code>	<code>*</code>	<code>\token_if_protected_long_macro_p:N</code>	<code><token></code>
<code>\token_if_protected_long_macro:NTF</code>	<code>*</code>	<code>\token_if_protected_long_macro:NTF</code>	<code><token> {\true code} {\false code}</code>

Updated: 2012-01-20

Tests if the `<token>` is a protected long macro.

<code>\token_if_chardef_p:N</code>	<code>*</code>	<code>\token_if_chardef_p:N</code>	<code><token></code>
<code>\token_if_chardef:NTF</code>	<code>*</code>	<code>\token_if_chardef:NTF</code>	<code><token> {\true code} {\false code}</code>

Updated: 2012-01-20

Tests if the `<token>` is defined to be a chardef.

TeXhackers note: Booleans, boxes and small integer constants are implemented as `\chardefs`.

<code>\token_if_mathchardef_p:N</code>	<code>*</code>	<code>\token_if_mathchardef_p:N</code>	<code><token></code>
<code>\token_if_mathchardef:NTF</code>	<code>*</code>	<code>\token_if_mathchardef:NTF</code>	<code><token> {\true code} {\false code}</code>

Updated: 2012-01-20

Tests if the `<token>` is defined to be a mathchardef.

<code>\token_if_dim_register_p:N</code>	<code>*</code>	<code>\token_if_dim_register_p:N</code>	<code><token></code>
<code>\token_if_dim_register:NTF</code>	<code>*</code>	<code>\token_if_dim_register:NTF</code>	<code><token> {\true code} {\false code}</code>

Updated: 2012-01-20

Tests if the `<token>` is defined to be a dimension register.

<code>\token_if_int_register_p:N</code>	<code>*</code>	<code>\token_if_int_register_p:N</code>	<code><token></code>
<code>\token_if_int_register:NTF</code>	<code>*</code>	<code>\token_if_int_register:NTF</code>	<code><token> {\true code} {\false code}</code>

Updated: 2012-01-20

Tests if the `<token>` is defined to be a integer register.

TeXhackers note: Constant integers may be implemented as integer registers, `\chardefs`, or `\mathchardefs` depending on their value.

<code>\token_if_muskip_register_p:N</code>	<code>*</code>	<code>\token_if_muskip_register_p:N</code>	<code><token></code>
<code>\token_if_muskip_register:NTF</code>	<code>*</code>	<code>\token_if_muskip_register:NTF</code>	<code><token> {\true code} {\false code}</code>

New: 2012-02-15

Tests if the `<token>` is defined to be a muskip register.

<code>\token_if_skip_register_p:N</code>	<code>*</code>	<code>\token_if_skip_register_p:N</code>	<code><token></code>
<code>\token_if_skip_register:NTF</code>	<code>*</code>	<code>\token_if_skip_register:NTF</code>	<code><token> {\true code} {\false code}</code>

Updated: 2012-01-20

Tests if the `<token>` is defined to be a skip register.

<code>\token_if_toks_register_p:N</code>	<code>*</code>	<code>\token_if_toks_register_p:N</code>	$\langle token \rangle$
<code>\token_if_toks_register:NTF</code>	<code>*</code>	<code>\token_if_toks_register:NTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a toks register (not used by L^AT_EX3).

<code>\token_if_primitive_p:N</code>	<code>*</code>	<code>\token_if_primitive_p:N</code>	$\langle token \rangle$
<code>\token_if_primitive:NTF</code>	<code>*</code>	<code>\token_if_primitive:NTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Updated: 2011-05-23

Tests if the $\langle token \rangle$ is an engine primitive.

6 Peeking ahead at the next token

There is often a need to look ahead at the next token in the input stream while leaving it in place. This is handled using the “peek” functions. The generic `\peek_after:Nw` is provided along with a family of predefined tests for common cases. As peeking ahead does *not* skip spaces the predefined tests include both a space-respecting and space-skipping version.

<code>\peek_after:Nw</code>	<code>\peek_after:Nw</code>	$\langle function \rangle$	$\langle token \rangle$
-----------------------------	-----------------------------	----------------------------	-------------------------

Locally sets the test variable `\l_peek_token` equal to $\langle token \rangle$ (as an implicit token, *not* as a token list), and then expands the $\langle function \rangle$. The $\langle token \rangle$ remains in the input stream as the next item after the $\langle function \rangle$. The $\langle token \rangle$ here may be \sqcup , $\{$ or $\}$ (assuming normal T_EX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

<code>\peek_gafter:Nw</code>	<code>\peek_gafter:Nw</code>	$\langle function \rangle$	$\langle token \rangle$
------------------------------	------------------------------	----------------------------	-------------------------

Globally sets the test variable `\g_peek_token` equal to $\langle token \rangle$ (as an implicit token, *not* as a token list), and then expands the $\langle function \rangle$. The $\langle token \rangle$ remains in the input stream as the next item after the $\langle function \rangle$. The $\langle token \rangle$ here may be \sqcup , $\{$ or $\}$ (assuming normal T_EX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

<code>\l_peek_token</code>	Token set by <code>\peek_after:Nw</code> and available for testing as described above.
----------------------------	--

<code>\g_peek_token</code>	Token set by <code>\peek_gafter:Nw</code> and available for testing as described above.
----------------------------	---

<code>\peek_catcode:NTF</code>	<code>\peek_catcode:NTF</code>	$\langle test\ token \rangle$	$\{\langle true\ code \rangle\}$	$\{\langle false\ code \rangle\}$
--------------------------------	--------------------------------	-------------------------------	----------------------------------	-----------------------------------

Updated: 2012-12-20

Tests if the next $\langle token \rangle$ in the input stream has the same category code as the $\langle test\ token \rangle$ (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the $\langle token \rangle$ is left in the input stream after the $\langle true\ code \rangle$ or $\langle false\ code \rangle$ (as appropriate to the result of the test).

<code>\peek_catcode_ignore_spaces:NTF</code>	<code>\peek_catcode_ignore_spaces:NTF <test token> {(true code)} {(false code)}</code>
--	--

Updated: 2012-12-20

Tests if the next non-space *<token>* in the input stream has the same category code as the *<test token>* (as defined by the test `\token_if_eq_catcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* is left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

<code>\peek_catcode_remove:NTF</code>	<code>\peek_catcode_remove:NTF <test token> {(true code)} {(false code)}</code>
---------------------------------------	---

Updated: 2012-12-20

Tests if the next *<token>* in the input stream has the same category code as the *<test token>* (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the *<token>* is removed from the input stream if the test is true. The function then places either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

<code>\peek_catcode_remove_ignore_spaces:NTF</code>	<code>\peek_catcode_remove_ignore_spaces:NTF <test token> {(true code)} {(false code)}</code>
---	---

Updated: 2012-12-20

Tests if the next non-space *<token>* in the input stream has the same category code as the *<test token>* (as defined by the test `\token_if_eq_catcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* is removed from the input stream if the test is true. The function then places either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

<code>\peek_charcode:NTF</code>	<code>\peek_charcode:NTF <test token> {(true code)} {(false code)}</code>
---------------------------------	---

Updated: 2012-12-20

Tests if the next *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the *<token>* is left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

<code>\peek_charcode_ignore_spaces:NTF</code>	<code>\peek_charcode_ignore_spaces:NTF <test token> {(true code)} {(false code)}</code>
---	---

Updated: 2012-12-20

Tests if the next non-space *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test `\token_if_eq_charcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* is left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

<code>\peek_charcode_remove:NTF</code>	<code>\peek_charcode_remove:NTF <test token> {(true code)} {(false code)}</code>
--	--

Updated: 2012-12-20

Tests if the next *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the *<token>* is removed from the input stream if the test is true. The function then places either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

<u>\peek_charcode_remove_ignore_spaces:NTF</u>	\peek_charcode_remove_ignore_spaces:NTF <test token> {<true code>} {<false code>}
Updated: 2012-12-20	

Tests if the next non-space <token> in the input stream has the same character code as the <test token> (as defined by the test \token_if_eq_charcode:NNTF). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the <token> is removed from the input stream if the test is true. The function then places either the <true code> or <false code> in the input stream (as appropriate to the result of the test).

<u>\peek_meaning:NTF</u>	\peek_meaning:NTF <test token> {<true code>} {<false code>}
Updated: 2011-07-02	

Tests if the next <token> in the input stream has the same meaning as the <test token> (as defined by the test \token_if_eq_meaning:NNTF). Spaces are respected by the test and the <token> is left in the input stream after the <true code> or <false code> (as appropriate to the result of the test).

<u>\peek_meaning_ignore_spaces:NTF</u>	\peek_meaning_ignore_spaces:NTF <test token> {<true code>} {<false code>}
Updated: 2012-12-05	

Tests if the next non-space <token> in the input stream has the same meaning as the <test token> (as defined by the test \token_if_eq_meaning:NNTF). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the <token> is left in the input stream after the <true code> or <false code> (as appropriate to the result of the test).

<u>\peek_meaning_remove:NTF</u>	\peek_meaning_remove:NTF <test token> {<true code>} {<false code>}
Updated: 2011-07-02	

Tests if the next <token> in the input stream has the same meaning as the <test token> (as defined by the test \token_if_eq_meaning:NNTF). Spaces are respected by the test and the <token> is removed from the input stream if the test is true. The function then places either the <true code> or <false code> in the input stream (as appropriate to the result of the test).

<u>\peek_meaning_remove_ignore_spaces:NTF</u>	\peek_meaning_remove_ignore_spaces:NTF <test token> {<true code>} {<false code>}
Updated: 2012-12-05	

Tests if the next non-space <token> in the input stream has the same meaning as the <test token> (as defined by the test \token_if_eq_meaning:NNTF). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the <token> is removed from the input stream if the test is true. The function then places either the <true code> or <false code> in the input stream (as appropriate to the result of the test).

`\peek_N_type:TF`

Updated: 2012-12-20

`\peek_N_type:TF` `{⟨true code⟩}{⟨false code⟩}`

Tests if the next *⟨token⟩* in the input stream can be safely grabbed as an N-type argument. The test is *⟨false⟩* if the next *⟨token⟩* is either an explicit or implicit begin-group or end-group token (with any character code), or an explicit or implicit space character (with character code 32 and category code 10), or an outer token (never used in L^AT_EX3) and *⟨true⟩* in all other cases. Note that a *⟨true⟩* result ensures that the next *⟨token⟩* is a valid N-type argument. However, if the next *⟨token⟩* is for instance `\c_space_token`, the test takes the *⟨false⟩* branch, even though the next *⟨token⟩* is in fact a valid N-type argument. The *⟨token⟩* is left in the input stream after the *⟨true code⟩* or *⟨false code⟩* (as appropriate to the result of the test).

7 Description of all possible tokens

Let us end by reviewing every case that a given token can fall into. This section is quite technical and some details are only meant for completeness. We distinguish the meaning of the token, which controls the expansion of the token and its effect on T_EX's state, and its shape, which is used when comparing token lists such as for delimited arguments. Two tokens of the same shape must have the same meaning, but the converse does not hold.

A token has one of the following shapes.

- A control sequence, characterized by the sequence of characters that constitute its name: for instance, `\use:n` is a five-letter control sequence.
- An active character token, characterized by its character code (between 0 and 1114111 for LuaT_EX and X_ƎT_EX and less for other engines) and category code 13.
- A character token, characterized by its character code and category code (one of 1, 2, 3, 4, 6, 7, 8, 10, 11 or 12 whose meaning is described below).

There are also a few internal tokens. The following list may be incomplete in some engines.

- Expanding `\the\font` results in a token that looks identical to the command that was used to select the current font (such as `\tenrm`) but it differs from it in shape.
- A “frozen” `\relax`, which differs from the primitive in shape (but has the same meaning), is inserted when the closing `\fi` of a conditional is encountered before the conditional is evaluated.
- Expanding `\noexpand⟨token⟩` (when the *⟨token⟩* is expandable) results in an internal token, displayed (temporarily) as `\notexpanded:⟨token⟩`, whose shape coincides with the *⟨token⟩* and whose meaning differs from `\relax`.
- An `\outer endtemplate:` can be encountered when peeking ahead at the next token; this expands to another internal token, `end of alignment template`.
- Tricky programming might access a frozen `\endwrite`.
- Some frozen tokens can only be accessed in interactive sessions: `\cr`, `\right`, `\endgroup`, `\fi`, `\inaccessible`.

- In Lua \TeX , there is also the strange case of “bytes” $\sim\sim\sim\sim\sim\sim 1100xy$ where x, y are any two lowercase hexadecimal digits, so that the hexadecimal number ranges from $\text{\texttt{\textbackslash text{110000}}}=1114112\$$ to $\text{\texttt{\textbackslash$1100ff}} = 1114367$. These are used to output individual bytes to files, rather than UTF-8. For the purposes of token comparisons they behave like non-expandable primitive control sequences (*not characters*) whose $\text{\texttt{\textbackslash meaning}}$ is $\text{\texttt{the_character_}}$ followed by the given byte. If this byte is in the range $80\text{--}ff$ this gives an “invalid utf-8 sequence” error: applying $\text{\texttt{\textbackslash token_to_str:N}}$ or $\text{\texttt{\textbackslash token_to_meaning:N}}$ to these tokens is unsafe. Unfortunately, they don’t seem to be detectable safely by any means except perhaps Lua code.

The meaning of a (non-active) character token is fixed by its category code (and character code) and cannot be changed. We call these tokens *explicit* character tokens. Category codes that a character token can have are listed below by giving a sample output of the \TeX primitive $\text{\texttt{\textbackslash meaning}}$, together with their $\LaTeX 3$ names and most common example:

- 1 begin-group character ($\text{\texttt{group_begin}}$, often $\{$),
- 2 end-group character ($\text{\texttt{group_end}}$, often $\}$),
- 3 math shift character ($\text{\texttt{math_toggle}}$, often $\$$),
- 4 alignment tab character ($\text{\texttt{alignment}}$, often $\&$),
- 6 macro parameter character ($\text{\texttt{parameter}}$, often $\#$),
- 7 superscript character ($\text{\texttt{math_superscript}}$, often $\^$),
- 8 subscript character ($\text{\texttt{math_subscript}}$, often $_$),
- 10 blank space ($\text{\texttt{space}}$, often character code 32),
- 11 the letter ($\text{\texttt{letter}}$, such as A),
- 12 the character ($\text{\texttt{other}}$, such as 0).

Category code 13 (**active**) is discussed below. Input characters can also have several other category codes which do not lead to character tokens for later processing: 0 (**escape**), 5 (**end_line**), 9 (**ignore**), 14 (**comment**), and 15 (**invalid**).

The meaning of a control sequence or active character can be identical to that of any character token listed above (with any character code), and we call such tokens *implicit* character tokens. The meaning is otherwise in the following list:

- a macro, used in $\LaTeX 3$ for most functions and some variables ($\text{\texttt{tl}}$, $\text{\texttt{fp}}$, $\text{\texttt{seq}}$, ...),
- a primitive such as $\text{\texttt{\textbackslash def}}$ or $\text{\texttt{\textbackslash topmark}}$, used in $\LaTeX 3$ for some functions,
- a register such as $\text{\texttt{\textbackslash count123}}$, used in $\LaTeX 3$ for the implementation of some variables ($\text{\texttt{int}}$, $\text{\texttt{dim}}$, ...),
- a constant integer such as $\text{\texttt{\textbackslash char"56}}$ or $\text{\texttt{\textbackslash mathchar"121}}$,
- a font selection command,
- undefined.

Macros can be `\protected` or not, `\long` or not (the opposite of what L^AT_EX3 calls `nopar`), and `\outer` or not (unused in L^AT_EX3). Their `\meaning` takes the form

`⟨prefix⟩ macro:⟨argument⟩->⟨replacement⟩`

where `⟨prefix⟩` is among `\protected\long\outer`, `⟨argument⟩` describes parameters that the macro expects, such as `#1#2#3`, and `⟨replacement⟩` describes how the parameters are manipulated, such as `\int_eval:n{#2+#1*#3}`.

Now is perhaps a good time to mention some subtleties relating to tokens with category code 10 (space). Any input character with this category code (normally, space and tab characters) becomes a normal space, with character code 32 and category code 10.

When a macro takes an undelimited argument, explicit space characters (with character code 32 and category code 10) are ignored. If the following token is an explicit character token with category code 1 (begin-group) and an arbitrary character code, then T_EX scans ahead to obtain an equal number of explicit character tokens with category code 1 (begin-group) and 2 (end-group), and the resulting list of tokens (with outer braces removed) becomes the argument. Otherwise, a single token is taken as the argument for the macro: we call such single tokens “N-type”, as they are suitable to be used as an argument for a function with the signature `:N`.

When a macro takes a delimited argument T_EX scans ahead until finding the delimiter (outside any pairs of begin-group/end-group explicit characters), and the resulting list of tokens (with outer braces removed) becomes the argument. Note that explicit space characters at the start of the argument are *not* ignored in this case (and they prevent brace-stripping).

Part XVII

The l3prop package

Property lists

L^AT_EX3 implements a “property list” data type, which contain an unordered list of entries each of which consists of a $\langle key \rangle$ and an associated $\langle value \rangle$. The $\langle key \rangle$ and $\langle value \rangle$ may both be any $\langle balanced\ text \rangle$. It is possible to map functions to property lists such that the function is applied to every key–value pair within the list.

Each entry in a property list must have a unique $\langle key \rangle$: if an entry is added to a property list which already contains the $\langle key \rangle$ then the new entry overwrites the existing one. The $\langle keys \rangle$ are compared on a string basis, using the same method as `\str_if_eq:nn`.

Property lists are intended for storing key-based information for use within code. This is in contrast to key–value lists, which are a form of *input* parsed by the `keys` module.

1 Creating and initialising property lists

```
\prop_new:N
\prop_new:c
```

```
\prop_new:N <property list>
```

Creates a new $\langle property\ list \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle property\ list \rangle$ initially contains no entries.

```
\prop_clear:N
\prop_clear:c
\prop_gclear:N
\prop_gclear:c
```

```
\prop_clear:N <property list>
```

Clears all entries from the $\langle property\ list \rangle$.

```
\prop_clear_new:N
\prop_clear_new:c
\prop_gclear_new:N
\prop_gclear_new:c
```

```
\prop_clear_new:N <property list>
```

Ensures that the $\langle property\ list \rangle$ exists globally by applying `\prop_new:N` if necessary, then applies `\prop_(g)clear:N` to leave the list empty.

```
\prop_set_eq:NN
\prop_set_eq:(cN|Nc|cc)
\prop_gset_eq:NN
\prop_gset_eq:(cN|Nc|cc)
```

```
\prop_set_eq:NN <property list1> <property list2>
```

Sets the content of $\langle property\ list_1 \rangle$ equal to that of $\langle property\ list_2 \rangle$.

```
\prop_set_from_keyval:Nn
\prop_set_from_keyval:cn
\prop_gset_from_keyval:Nn
\prop_gset_from_keyval:cn
```

```
\prop_set_from_keyval:Nn <prop var>
```

```
{
  <key1> = <value1> ,
  <key2> = <value2> , ...
}
```

Sets $\langle prop\ var \rangle$ to contain key–value pairs given in the second argument. If duplicate keys appear only one of the values is kept.

New: 2017-11-28
Updated: 2019-08-25

```
\prop_const_from_keyval:Nn
\prop_const_from_keyval:cn
```

New: 2017-11-28
Updated: 2019-08-25

```
\prop_const_from_keyval:Nn <prop var>
{
  <key1> = <value1> ,
  <key2> = <value2> , ...
}
```

Creates a new constant *<prop var>* or raises an error if the name is already taken. The *<prop var>* is set globally to contain key–value pairs given in the second argument. If duplicate keys appear only one of the values is kept.

2 Adding entries to property lists

```
\prop_put:Nnn
\prop_put:(NnV|Nno|Nnx|NVn|NVV|NVx|Nvx|Non|Noo|Nxx|cnn|cnV|cno|cnx|cVn|cVV|cVx|cvx|con|coo|cxx)
\prop_gput:Nnn
\prop_gput:(NnV|Nno|Nnx|NVn|NVV|NVx|Nvx|Non|Noo|Nxx|cnn|cnV|cno|cnx|cVn|cVV|cVx|cvx|con|coo|cxx)
```

Updated: 2012-07-09

```
\prop_put:Nnn
  <property list>
  {<key>}
  {<value>}
```

Adds an entry to the *<property list>* which may be accessed using the *<key>* and which has *<value>*. Both the *<key>* and *<value>* may contain any *<balanced text>*. The *<key>* is stored after processing with `\tl_to_str:n`, meaning that category codes are ignored. If the *<key>* is already present in the *<property list>*, the existing entry is overwritten by the new *<value>*.

```
\prop_put_if_new:Nnn
\prop_put_if_new:cnn
\prop_gput_if_new:Nnn
\prop_gput_if_new:cnn
```

```
\prop_put_if_new:Nnn <property list> {<key>} {<value>}
```

If the *<key>* is present in the *<property list>* then no action is taken. If the *<key>* is not present in the *<property list>* then a new entry is added. Both the *<key>* and *<value>* may contain any *<balanced text>*. The *<key>* is stored after processing with `\tl_to_str:n`, meaning that category codes are ignored.

3 Recovering values from property lists

```
\prop_get:NnN
\prop_get:(NVN|NvN|NoN|cnN|cVN|cvN|coN)
```

Updated: 2011-08-28

```
\prop_get:NnN <property list> {<key>} <tl var>
```

Recovers the *<value>* stored with *<key>* from the *<property list>*, and places this in the *<token list variable>*. If the *<key>* is not found in the *<property list>* then the *<token list variable>* is set to the special marker `\q_no_value`. The *<token list variable>* is set within the current T_EX group. See also `\prop_get:NnNTF`.

```
\prop_pop:NnN
\prop_pop:(NoN|cnN|coN)
```

Updated: 2011-08-18

```
\prop_pop:NnN <property list> {<key>} <tl var>
```

Recovers the *<value>* stored with *<key>* from the *<property list>*, and places this in the *<token list variable>*. If the *<key>* is not found in the *<property list>* then the *<token list variable>* is set to the special marker `\q_no_value`. The *<key>* and *<value>* are then deleted from the property list. Both assignments are local. See also `\prop_pop:NnNTF`.

<code>\prop_gpop:NnN</code>
<code>\prop_gpop:(NoN cnN coN)</code>
Updated: 2011-08-18

`\prop_gpop:NnN` $\langle property list \rangle$ $\{\langle key \rangle\}$ $\langle tl var \rangle$

Recovers the $\langle value \rangle$ stored with $\langle key \rangle$ from the $\langle property list \rangle$, and places this in the $\langle token list variable \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ then the $\langle token list variable \rangle$ is set to the special marker `\q_no_value`. The $\langle key \rangle$ and $\langle value \rangle$ are then deleted from the property list. The $\langle property list \rangle$ is modified globally, while the assignment of the $\langle token list variable \rangle$ is local. See also `\prop_gpop:NnNTF`.

<code>\prop_item:Nn *</code>
<code>\prop_item:cn *</code>
New: 2014-07-17

`\prop_item:Nn` $\langle property list \rangle$ $\{\langle key \rangle\}$

Expands to the $\langle value \rangle$ corresponding to the $\langle key \rangle$ in the $\langle property list \rangle$. If the $\langle key \rangle$ is missing, this has an empty expansion.

TeXhackers note: This function is slower than the non-expandable analogue `\prop_get:NnN`. The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle value \rangle$ does not expand further when appearing in an **x**-type argument expansion.

<code>\prop_count:N *</code>
<code>\prop_count:c *</code>

`\prop_count:N` $\langle property list \rangle$

Leaves the number of key–value pairs in the $\langle property list \rangle$ in the input stream as an $\langle integer denotation \rangle$.

4 Modifying property lists

<code>\prop_remove:Nn</code>
<code>\prop_remove:(NV cn cV)</code>
<code>\prop_gremove:Nn</code>
<code>\prop_gremove:(NV cn cV)</code>
New: 2012-05-12

`\prop_remove:Nn` $\langle property list \rangle$ $\{\langle key \rangle\}$

Removes the entry listed under $\langle key \rangle$ from the $\langle property list \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ no change occurs, *i.e* there is no need to test for the existence of a key before deleting it.

5 Property list conditionals

<code>\prop_if_exist_p:N *</code>
<code>\prop_if_exist_p:c *</code>
<code>\prop_if_exist:NTF *</code>
<code>\prop_if_exist:cTF *</code>
New: 2012-03-03

`\prop_if_exist_p:N` $\langle property list \rangle$

`\prop_if_exist:NTF` $\langle property list \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests whether the $\langle property list \rangle$ is currently defined. This does not check that the $\langle property list \rangle$ really is a property list variable.

<code>\prop_if_empty_p:N *</code>
<code>\prop_if_empty_p:c *</code>
<code>\prop_if_empty:NTF *</code>
<code>\prop_if_empty:cTF *</code>

`\prop_if_empty_p:N` $\langle property list \rangle$

`\prop_if_empty:NTF` $\langle property list \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the $\langle property list \rangle$ is empty (containing no entries).

<code>\prop_if_in_p:Nn</code>	★	<code>\prop_if_in:NnTF</code>	$\langle property\ list \rangle$	$\{\langle key \rangle\}$	$\{\langle true\ code \rangle\}$	$\{\langle false\ code \rangle\}$
<code>\prop_if_in_p:(NV No cn cV co)</code>	★					
<code>\prop_if_in:NnTF</code>	★					
<code>\prop_if_in:(NV No cn cV co)TF</code>	★					

Updated: 2011-09-15

Tests if the $\langle key \rangle$ is present in the $\langle property\ list \rangle$, making the comparison using the method described by `\str_if_eq:NnTF`.

TeXhackers note: This function iterates through every key-value pair in the $\langle property\ list \rangle$ and is therefore slower than using the non-expandable `\prop_get:NnTF`.

6 Recovering values from property lists with branching

The functions in this section combine tests for the presence of a key in a property list with recovery of the associated valued. This makes them useful for cases where different cases follow dependent on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

<code>\prop_get:NnTF</code>	<code>\prop_get:NnTF</code>	$\langle property\ list \rangle$	$\{\langle key \rangle\}$	$\langle token\ list\ variable \rangle$
<code>\prop_get:(NVN NvN NoN cnN cVN cvN coN)TF</code>			$\{\langle true\ code \rangle\}$	$\{\langle false\ code \rangle\}$

Updated: 2012-05-19

If the $\langle key \rangle$ is not present in the $\langle property\ list \rangle$, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle key \rangle$ is present in the $\langle property\ list \rangle$, stores the corresponding $\langle value \rangle$ in the $\langle token\ list\ variable \rangle$ without removing it from the $\langle property\ list \rangle$, then leaves the $\langle true\ code \rangle$ in the input stream. The $\langle token\ list\ variable \rangle$ is assigned locally.

<code>\prop_pop:NnTF</code>	<code>\prop_pop:NnTF</code>	$\langle property\ list \rangle$	$\{\langle key \rangle\}$	$\langle token\ list\ variable \rangle$	$\{\langle true\ code \rangle\}$
<code>\prop_pop:cnTF</code>					$\{\langle false\ code \rangle\}$

New: 2011-08-18
Updated: 2012-05-19

If the $\langle key \rangle$ is not present in the $\langle property\ list \rangle$, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle key \rangle$ is present in the $\langle property\ list \rangle$, pops the corresponding $\langle value \rangle$ in the $\langle token\ list\ variable \rangle$, *i.e.* removes the item from the $\langle property\ list \rangle$. Both the $\langle property\ list \rangle$ and the $\langle token\ list\ variable \rangle$ are assigned locally.

<code>\prop_gpop:NnTF</code>	<code>\prop_gpop:NnTF</code>	$\langle property\ list \rangle$	$\{\langle key \rangle\}$	$\langle token\ list\ variable \rangle$	$\{\langle true\ code \rangle\}$
<code>\prop_gpop:cnTF</code>					$\{\langle false\ code \rangle\}$

New: 2011-08-18
Updated: 2012-05-19

If the $\langle key \rangle$ is not present in the $\langle property\ list \rangle$, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle key \rangle$ is present in the $\langle property\ list \rangle$, pops the corresponding $\langle value \rangle$ in the $\langle token\ list\ variable \rangle$, *i.e.* removes the item from the $\langle property\ list \rangle$. The $\langle property\ list \rangle$ is modified globally, while the $\langle token\ list\ variable \rangle$ is assigned locally.

7 Mapping to property lists

All mappings are done at the current group level, *i.e.* any local assignments made by the $\langle function \rangle$ or $\langle code \rangle$ discussed below remain in effect after the loop.

$\backslash prop_map_function:Nn$	☆
$\backslash prop_map_function:cN$	☆
Updated: 2013-01-08	

$\backslash prop_map_function:Nn$ $\langle property\ list \rangle$ $\langle function \rangle$

Applies $\langle function \rangle$ to every $\langle entry \rangle$ stored in the $\langle property\ list \rangle$. The $\langle function \rangle$ receives two arguments for each iteration: the $\langle key \rangle$ and associated $\langle value \rangle$. The order in which $\langle entries \rangle$ are returned is not defined and should not be relied upon. To pass further arguments to the $\langle function \rangle$, see $\backslash prop_map_tokens:Nn$.

$\backslash prop_map_inline:Nn$	
$\backslash prop_map_inline:cN$	
Updated: 2013-01-08	

$\backslash prop_map_inline:Nn$ $\langle property\ list \rangle$ $\{ \langle inline\ function \rangle \}$

Applies $\langle inline\ function \rangle$ to every $\langle entry \rangle$ stored within the $\langle property\ list \rangle$. The $\langle inline\ function \rangle$ should consist of code which receives the $\langle key \rangle$ as #1 and the $\langle value \rangle$ as #2. The order in which $\langle entries \rangle$ are returned is not defined and should not be relied upon.

$\backslash prop_map_tokens:Nn$	☆
$\backslash prop_map_tokens:cN$	☆

$\backslash prop_map_tokens:Nn$ $\langle property\ list \rangle$ $\{ \langle code \rangle \}$

Analogue of $\backslash prop_map_function:Nn$ which maps several tokens instead of a single function. The $\langle code \rangle$ receives each key–value pair in the $\langle property\ list \rangle$ as two trailing brace groups. For instance,

```
 $\backslash prop\_map\_tokens:Nn \backslash l\_my\_prop \{ \backslash str\_if\_eq:nnT \{ mykey \} \}$ 
```

expands to the value corresponding to `mykey`: for each pair in $\backslash l_my_prop$ the function $\backslash str_if_eq:nnT$ receives `mykey`, the $\langle key \rangle$ and the $\langle value \rangle$ as its three arguments. For that specific task, $\backslash prop_item:Nn$ is faster.

$\backslash prop_map_break:$	☆
Updated: 2012-06-29	

$\backslash prop_map_break:$

Used to terminate a $\backslash prop_map_...$ function before all entries in the $\langle property\ list \rangle$ have been processed. This normally takes place within a conditional statement, for example

```
 $\backslash prop\_map\_inline:Nn \backslash l\_my\_prop$ 
{
   $\backslash str\_if\_eq:nnTF \{ \#1 \} \{ bingo \}$ 
  {  $\backslash prop\_map\_break:$  }
  {
    % Do something useful
  }
}
```

Use outside of a $\backslash prop_map_...$ scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

<hr/> <code>\prop_map_break:n</code> ☆ <hr/>	<code>\prop_map_break:n {<code>}</code>
Updated: 2012-06-29 <hr/>	Used to terminate a <code>\prop_map_...</code> function before all entries in the <i><property list></i> have been processed, inserting the <i><code></i> after the mapping has ended. This normally takes place within a conditional statement, for example

```

\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \prop_map_break:n { <code> } }
  {
    % Do something useful
  }
}

```

Use outside of a `\prop_map_...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

8 Viewing property lists

<hr/> <code>\prop_show:N</code> <code>\prop_show:c</code> <hr/>	<code>\prop_show:N <property list></code>
Updated: 2015-08-01 <hr/>	Displays the entries in the <i><property list></i> in the terminal.

<hr/> <code>\prop_log:N</code> <code>\prop_log:c</code> <hr/>	<code>\prop_log:N <property list></code>
New: 2014-08-12 Updated: 2015-08-01 <hr/>	Writes the entries in the <i><property list></i> in the log file.

9 Scratch property lists

<hr/> <code>\l_tmpa_prop</code> <code>\l_tmpb_prop</code> <hr/>	Scratch property lists for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
New: 2012-06-23 <hr/>	

<hr/> <code>\g_tmpa_prop</code> <code>\g_tmpb_prop</code> <hr/>	Scratch property lists for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
New: 2012-06-23 <hr/>	

10 Constants

\c_empty_prop A permanently-empty property list used for internal comparisons.

Part XVIII

The l3msg package

Messages

Messages need to be passed to the user by modules, either when errors occur or to indicate how the code is proceeding. The `l3msg` module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by `l3msg` to create messages divides the process into two distinct parts. Named messages are created in the first part of the process; at this stage, no decision is made about the type of output that the message will produce. The second part of the process is actually producing a message. At this stage a choice of message *class* has to be made, for example `error`, `warning` or `info`.

By separating out the creation and use of messages, several benefits are available. First, the messages can be altered later without needing details of where they are used in the code. This makes it possible to alter the language used, the detail level and so on. Secondly, the output which results from a given message can be altered. This can be done on a message class, module or message name basis. In this way, message behaviour can be altered and messages can be entirely suppressed.

1 Creating new messages

All messages have to be created before they can be used. The text of messages is automatically wrapped to the length available in the console. As a result, formatting is only needed where it helps to show meaning. In particular, `\\` may be used to force a new line and `_` forces an explicit space. Additionally, `\{`, `\#`, `\}`, `\%` and `\~` can be used to produce the corresponding character.

Messages may be subdivided *by one level* using the `/` character. This is used within the message filtering system to allow for example the L^AT_EX kernel messages to belong to the module `LaTeX` while still being filterable at a more granular level. Thus for example

```
\msg_new:nnnn { mymodule } { submodule / message } ...
```

will allow to filter out specifically messages from the `submodule`.

```
\msg_new:nnnn
\msg_new:nnn
Updated: 2011-08-16
```

```
\msg_new:nnnn {<module>} {<message>} {<text>} {<more text>}
```

Creates a *<message>* for a given *<module>*. The message is defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (**#1** to **#4**) can be used: these will be supplied at the time the message is used. An error is raised if the *<message>* already exists.

```
\msg_set:nnnn
\msg_set:nnn
\msg_gset:nnnn
\msg_gset:nnn
```

```
\msg_set:nnnn {<module>} {<message>} {<text>} {<more text>}
```

Sets up the text for a *<message>* for a given *<module>*. The message is defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (**#1** to **#4**) can be used: these will be supplied at the time the message is used.

<code>\msg_if_exist_p:nn</code> *	<code>\msg_if_exist_p:nn {<module>} {<message>}</code>
<code>\msg_if_exist:nnTF</code> *	<code>\msg_if_exist:nnTF {<module>} {<message>} {<true code>} {<false code>}</code>
New: 2012-03-03	Tests whether the <i><message></i> for the <i><module></i> is currently defined.

2 Contextual information for messages

<code>\msg_line_context:</code> ☆	<code>\msg_line_context:</code>
	Prints the current line number when a message is given, and thus suitable for giving context to messages. The number itself is proceeded by the text on line.

<code>\msg_line_number:</code> *	<code>\msg_line_number:</code>
	Prints the current line number when a message is given.

<code>\msg_fatal_text:n</code> *	<code>\msg_fatal_text:n {<module>}</code>
	Produces the standard text
	Fatal Package <i><module></i> Error
	This function can be redefined to alter the language in which the message is given, using #1 as the name of the <i><module></i> to be included.

<code>\msg_critical_text:n</code> *	<code>\msg_critical_text:n {<module>}</code>
	Produces the standard text
	Critical Package <i><module></i> Error
	This function can be redefined to alter the language in which the message is given, using #1 as the name of the <i><module></i> to be included.

<code>\msg_error_text:n</code> *	<code>\msg_error_text:n {<module>}</code>
	Produces the standard text
	Package <i><module></i> Error
	This function can be redefined to alter the language in which the message is given, using #1 as the name of the <i><module></i> to be included.

<code>\msg_warning_text:n</code> *	<code>\msg_warning_text:n {<module>}</code>
	Produces the standard text
	Package <i><module></i> Warning
	This function can be redefined to alter the language in which the message is given, using #1 as the name of the <i><module></i> to be included. The <i><type></i> of <i><module></i> may be adjusted: Package is the standard outcome: see <code>\msg_module_type:n</code> .

<hr/> <code>\msg_info_text:n</code> ★ <hr/>	<code>\msg_info_text:n {⟨module⟩}</code> Produces the standard text: <div style="text-align: center;"><code>Package ⟨module⟩ Info</code></div> This function can be redefined to alter the language in which the message is given, using #1 as the name of the ⟨module⟩ to be included. The ⟨type⟩ of ⟨module⟩ may be adjusted: <code>Package</code> is the standard outcome: see <code>\msg_module_type:n</code> .
<hr/> <code>\msg_module_name:n</code> ★ <hr/> <div style="text-align: right;">New: 2018-10-10</div>	<code>\msg_module_name:n {⟨module⟩}</code> Expands to the public name of the ⟨module⟩ as defined by <code>\g_msg_module_name_prop</code> (or otherwise leaves the ⟨module⟩ unchanged).
<hr/> <code>\msg_module_type:n</code> ★ <hr/> <div style="text-align: right;">New: 2018-10-10</div>	<code>\msg_module_type:n {⟨module⟩}</code> Expands to the description which applies to the ⟨module⟩, for example a <code>Package</code> or <code>Class</code> . The information here is defined in <code>\g_msg_module_type_prop</code> , and will default to <code>Package</code> if an entry is not present.
<hr/> <code>\msg_see_documentation_text:n</code> ★ <hr/> <div style="text-align: right;">Updated: 2018-09-30</div>	<code>\msg_see_documentation_text:n {⟨module⟩}</code> Produces the standard text <div style="text-align: center;"><code>See the ⟨module⟩ documentation for further information.</code></div> This function can be redefined to alter the language in which the message is given, using #1 as the name of the ⟨module⟩ to be included. The name of the ⟨module⟩ may be altered by use of <code>\g_msg_module_documentation_prop</code>
<hr/> <code>\g_msg_module_name_prop</code> <hr/> <div style="text-align: right;">New: 2018-10-10</div>	Provides a mapping between the module name used for messages, and that for documentation. For example, <code>L^AT_EX3</code> core messages are stored in the reserved <code>L^AT_EX</code> tree, but are printed as <code>L^AT_EX3</code> .
<hr/> <code>\g_msg_module_type_prop</code> <hr/> <div style="text-align: right;">New: 2018-10-10</div>	Provides a mapping between the module name used for messages, and that type of module. For example, for <code>L^AT_EX3</code> core messages, an empty entry is set here meaning that they are not described using the standard <code>Package</code> text.

3 Issuing messages

Messages behave differently depending on the message class. In all cases, the message may be issued supplying 0 to 4 arguments. If the number of arguments supplied here does not match the number in the definition of the message, extra arguments are ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments are converted to strings before being added to the message text: the `x`-type variants should be used to expand material.

```

\msg_fatal:nnnnnn
\msg_fatal:nnxxxx
\msg_fatal:nnnnn
\msg_fatal:nnxxx
\msg_fatal:nnnn
\msg_fatal:nnxx
\msg_fatal:nnn
\msg_fatal:nnx
\msg_fatal:nn

```

Updated: 2012-08-11

```

\msg_fatal:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three}
{\arg four}

```

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. After issuing a fatal error the T_EX run halts. No PDF file will be produced in this case (DVI mode runs may produce a truncated DVI file).

```

\msg_critical:nnnnnn
\msg_critical:nnxxxx
\msg_critical:nnnnn
\msg_critical:nnxxx
\msg_critical:nnnn
\msg_critical:nnxx
\msg_critical:nnn
\msg_critical:nnx
\msg_critical:nn

```

Updated: 2012-08-11

```

\msg_critical:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three}
{\arg four}

```

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. After issuing a critical error, T_EX stops reading the current input file. This may halt the T_EX run (if the current file is the main file) or may abort reading a sub-file.

T_EXhackers note: The T_EX `\endinput` primitive is used to exit the file. In particular, the rest of the current line remains in the input stream.

```

\msg_error:nnnnnn
\msg_error:nnxxxx
\msg_error:nnnnn
\msg_error:nnxxx
\msg_error:nnnn
\msg_error:nnxx
\msg_error:nnn
\msg_error:nnx
\msg_error:nn

```

Updated: 2012-08-11

```

\msg_error:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three}
{\arg four}

```

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The error interrupts processing and issues the text at the terminal. After user input, the run continues.

```

\msg_warning:nnnnnn
\msg_warning:nnxxxx
\msg_warning:nnnnn
\msg_warning:nnxxx
\msg_warning:nnnn
\msg_warning:nnxx
\msg_warning:nnn
\msg_warning:nnx
\msg_warning:nn

```

Updated: 2012-08-11

```

\msg_warning:nnxxxx {\module} {\message} {\arg one} {\arg two} {\arg three}
{\arg four}

```

Issues $\langle module \rangle$ warning $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The warning text is added to the log file and the terminal, but the T_EX run is not interrupted.

<hr/> <pre> \msg_info:nnnnnn \msg_info:nnxxxx \msg_info:nnnnn \msg_info:nnxxx \msg_info:nnnn \msg_info:nnxx \msg_info:nnn \msg_info:nnx \msg_info:nn </pre> <hr/> <p>Updated: 2012-08-11</p> <hr/>	<pre> \msg_info:nnnnnn {\module}} {\message}} {\arg one}} {\arg two}} {\arg three}} {\arg four}} </pre> <p>Issues <i>⟨module⟩</i> information <i>⟨message⟩</i>, passing <i>⟨arg one⟩</i> to <i>⟨arg four⟩</i> to the text-creating functions. The information text is added to the log file.</p>
<hr/> <pre> \msg_log:nnnnnn \msg_log:nnxxxx \msg_log:nnnnn \msg_log:nnxxx \msg_log:nnnn \msg_log:nnxx \msg_log:nnn \msg_log:nnx \msg_log:nn </pre> <hr/> <p>Updated: 2012-08-11</p> <hr/>	<pre> \msg_log:nnnnnn {\module}} {\message}} {\arg one}} {\arg two}} {\arg three}} {\arg four}} </pre> <p>Issues <i>⟨module⟩</i> information <i>⟨message⟩</i>, passing <i>⟨arg one⟩</i> to <i>⟨arg four⟩</i> to the text-creating functions. The information text is added to the log file: the output is briefer than <code>\msg_info:nnnnnn</code>.</p>
<hr/> <pre> \msg_term:nnnnnn \msg_term:nnxxxx \msg_term:nnnnn \msg_term:nnxxx \msg_term:nnnn \msg_term:nnxx \msg_term:nnn \msg_term:nnx \msg_term:nn </pre> <hr/> <p>New: 2020-07-16</p> <hr/>	<pre> \msg_term:nnnnnn {\module}} {\message}} {\arg one}} {\arg two}} {\arg three}} {\arg four}} </pre> <p>Issues <i>⟨module⟩</i> information <i>⟨message⟩</i>, passing <i>⟨arg one⟩</i> to <i>⟨arg four⟩</i> to the text-creating functions. The information text is printed on the terminal (and added to the log file): the output is similar to that of <code>\msg_log:nnnnnn</code>.</p>
<hr/> <pre> \msg_none:nnnnnn \msg_none:nnxxxx \msg_none:nnnnn \msg_none:nnxxx \msg_none:nnnn \msg_none:nnxx \msg_none:nnn \msg_none:nnx \msg_none:nn </pre> <hr/> <p>Updated: 2012-08-11</p> <hr/>	<pre> \msg_none:nnnnnn {\module}} {\message}} {\arg one}} {\arg two}} {\arg three}} {\arg four}} </pre> <p>Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).</p>

3.1 Expandable error messages

In very rare cases it may be necessary to produce errors in an expansion-only context. The functions in this section should only be used if there is no alternative approach using `\msg_error:nnnnnn` or other non-expandable commands from the previous section.

Despite having a similar interface as non-expandable messages, expandable errors must be handled internally very differently from normal error messages, as none of the tools to print to the terminal or the log file are expandable. As a result, short-hands such as `\{` or `\|` do not work, and messages must be very short (with default settings, they are truncated after approximately 50 characters). It is advisable to ensure that the message is understandable even when truncated, by putting the most important information up front. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

<code>\msg_expandable_error:nnnnnn</code>	*	<code>\msg_expandable_error:nnnnnn {<module>} {<message>} {<arg one>} {<arg</code>
<code>\msg_expandable_error:nnffff</code>	*	<code>two)} {<arg three>} {<arg four>}</code>
<code>\msg_expandable_error:nnnnn</code>	*	
<code>\msg_expandable_error:nnfff</code>	*	
<code>\msg_expandable_error:nnnn</code>	*	
<code>\msg_expandable_error:nnff</code>	*	
<code>\msg_expandable_error:nnn</code>	*	
<code>\msg_expandable_error:nnf</code>	*	
<code>\msg_expandable_error:nn</code>	*	

New: 2015-08-06

Updated: 2019-02-28

Issues an “Undefined error” message from T_EX itself using the undefined control sequence `\::error` then prints “! *<module>*: ”*<error message>*”, which should be short. With default settings, anything beyond approximately 60 characters long (or bytes in some engines) is cropped. A leading space might be removed as well.

4 Redirecting messages

Each message has a “name”, which can be used to alter the behaviour of the message when it is given. Thus we might have

```
\msg_new:nnnn { module } { my-message } { Some-text } { Some-more-text }
```

to define a message, with

```
\msg_error:nn { module } { my-message }
```

when it is used. With no filtering, this raises an error. However, we could alter the behaviour with

```
\msg_redirect_class:nn { error } { warning }
```

to turn all errors into warnings, or with

```
\msg_redirect_module:nnn { module } { error } { warning }
```

to alter only messages from that module, or even

```
\msg_redirect_name:nnn { module } { my-message } { warning }
```

to target just one message. Redirection applies first to individual messages, then to messages from one module and finally to messages of one class. Thus it is possible to select out an individual message for special treatment even if the entire class is already redirected.

Multiple redirections are possible. Redirections can be cancelled by providing an empty argument for the target class. Redirection to a missing class raises an error immediately. Infinite loops are prevented by eliminating the redirection starting from the target of the redirection that caused the loop to appear. Namely, if redirections are requested as $A \rightarrow B$, $B \rightarrow C$ and $C \rightarrow A$ in this order, then the $A \rightarrow B$ redirection is cancelled.

`\msg_redirect_class:nn`

Updated: 2012-04-27

`\msg_redirect_class:nn {<class one>} {<class two>}`

Changes the behaviour of messages of *<class one>* so that they are processed using the code for those of *<class two>*.

`\msg_redirect_module:nnn`

Updated: 2012-04-27

`\msg_redirect_module:nnn {<module>} {<class one>} {<class two>}`

Redirects message of *<class one>* for *<module>* to act as though they were from *<class two>*. Messages of *<class one>* from sources other than *<module>* are not affected by this redirection. This function can be used to make some messages “silent” by default. For example, all of the **warning** messages of *<module>* could be turned off with:

`\msg_redirect_module:nnn { module } { warning } { none }`

`\msg_redirect_name:nnn`

Updated: 2012-04-27

`\msg_redirect_name:nnn {<module>} {<message>} {<class>}`

Redirects a specific *<message>* from a specific *<module>* to act as a member of *<class>* of messages. No further redirection is performed. This function can be used to make a selected message “silent” without changing global parameters:

`\msg_redirect_name:nnn { module } { annoying-message } { none }`

Part XIX

The l3file package

File and I/O operations

This module provides functions for working with external files. Some of these functions apply to an entire file, and have prefix `\file_...`, while others are used to work with files on a line by line basis and have prefix `\ior_...` (reading) or `\iow_...` (writing).

It is important to remember that when reading external files \TeX attempts to locate them using both the operating system path and entries in the \TeX file database (most \TeX systems use such a database). Thus the “current path” for \TeX is somewhat broader than that for other programs.

For functions which expect a $\langle file\ name \rangle$ argument, this argument may contain both literal items and expandable content, which should on full expansion be the desired file name. Active characters (as declared in `\l_char_active_seq`) are *not* expanded, allowing the direct use of these in file names. Quote tokens (") are not permitted in file names as they are reserved for internal use by some \TeX primitives.

Spaces are trimmed at the beginning and end of the file name: this reflects the fact that some file systems do not allow or interact unpredictably with spaces in these positions. When no extension is given, this will trim spaces from the start of the name only.

1 Input–output stream management

As \TeX engines have a limited number of input and output streams, direct use of the streams by the programmer is not supported in \LaTeX 3. Instead, an internal pool of streams is maintained, and these are allocated and deallocated as needed by other modules. As a result, the programmer should close streams when they are no longer needed, to release them for other processes.

Note that I/O operations are global: streams should all be declared with global names and treated accordingly.

`\ior_new:N`
`\ior_new:c`
`\iow_new:N`
`\iow_new:c`

New: 2011-09-26
Updated: 2011-12-27

`\ior_new:N` $\langle stream \rangle$
`\iow_new:N` $\langle stream \rangle$

Globally reserves the name of the $\langle stream \rangle$, either for reading or for writing as appropriate. The $\langle stream \rangle$ is not opened until the appropriate `\..._open:Nn` function is used. Attempting to use a $\langle stream \rangle$ which has not been opened is an error, and the $\langle stream \rangle$ will behave as the corresponding `\c_term_....`

`\ior_open:Nn`
`\ior_open:cn`

Updated: 2012-02-10

`\ior_open:Nn` $\langle stream \rangle$ $\{ \langle file\ name \rangle \}$

Opens $\langle file\ name \rangle$ for reading using $\langle stream \rangle$ as the control sequence for file access. If the $\langle stream \rangle$ was already open it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately and will remain allocated to $\langle file\ name \rangle$ until a `\ior_close:N` instruction is given or the \TeX run ends. If the file is not found, an error is raised.

<hr/> <code>\ior_open:NnTF</code> <hr/>	<code>\ior_open:NnTF <stream> {<file name>} {<true code>} {<false code>}</code>
<code>\ior_open:cnTF</code> <hr/>	
<hr/> New: 2013-01-12 <hr/>	Opens <i><file name></i> for reading using <i><stream></i> as the control sequence for file access. If the <i><stream></i> was already open it is closed before the new operation begins. The <i><stream></i> is available for access immediately and will remain allocated to <i><file name></i> until a <code>\ior_close:N</code> instruction is given or the TeX run ends. The <i><true code></i> is then inserted into the input stream. If the file is not found, no error is raised and the <i><false code></i> is inserted into the input stream.
<hr/>	
<code>\iow_open:Nn</code> <hr/>	<code>\iow_open:Nn <stream> {<file name>}</code>
<code>\iow_open:cn</code> <hr/>	
<hr/> Updated: 2012-02-09 <hr/>	Opens <i><file name></i> for writing using <i><stream></i> as the control sequence for file access. If the <i><stream></i> was already open it is closed before the new operation begins. The <i><stream></i> is available for access immediately and will remain allocated to <i><file name></i> until a <code>\iow_close:N</code> instruction is given or the TeX run ends. Opening a file for writing clears any existing content in the file (<i>i.e.</i> writing is <i>not</i> additive).
<hr/>	
<code>\ior_close:N</code> <hr/>	<code>\ior_close:N <stream></code>
<code>\ior_close:c</code> <hr/>	<code>\iow_close:N <stream></code>
<code>\iow_close:N</code> <hr/>	Closes the <i><stream></i> . Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.
<code>\iow_close:c</code> <hr/>	
<hr/> Updated: 2012-07-31 <hr/>	
<hr/>	
<code>\ior_show_list:</code> <hr/>	<code>\ior_show_list:</code>
<code>\ior_log_list:</code> <hr/>	<code>\ior_log_list:</code>
<code>\iow_show_list:</code> <hr/>	<code>\iow_show_list:</code>
<code>\iow_log_list:</code> <hr/>	<code>\iow_log_list:</code>
<hr/> New: 2017-06-27 <hr/>	Display (to the terminal or log file) a list of the file names associated with each open (read or write) stream. This is intended for tracking down problems.

1.1 Reading from files

Reading from files and reading from the terminal are separate processes in `expl3`. The functions `\ior_get:NN` and `\ior_str_get:NN`, and their branching equivalents, are designed to work with *files*.

<code>\ior_get:NN</code>
<code>\ior_get:NNTF</code>
New: 2012-06-24
Updated: 2019-03-23

`\ior_get:NN` $\langle stream \rangle$ $\langle token list variable \rangle$
`\ior_get:NNTF` $\langle stream \rangle$ $\langle token list variable \rangle$ $\langle true code \rangle$ $\langle false code \rangle$

Function that reads one or more lines (until an equal number of left and right braces are found) from the file input $\langle stream \rangle$ and stores the result locally in the $\langle token list \rangle$ variable. The material read from the $\langle stream \rangle$ is tokenized by T_EX according to the category codes and `\endlinechar` in force when the function is used. Assuming normal settings, any lines which do not end in a comment character % have the line ending converted to a space, so for example input

a b c

results in a token list `a_b_c_`. Any blank line is converted to the token `\par`. Therefore, blank lines can be skipped by using a test such as

```
\ior_get:NN \l_my_stream \l_tmpa_tl
\tl_set:Nn \l_tmpb_tl { \par }
\tl_if_eq:NMF \l_tmpa_tl \l_tmpb_tl
...
```

Also notice that if multiple lines are read to match braces then the resulting token list can contain `\par` tokens. In the non-branching version, where the $\langle stream \rangle$ is not open the $\langle tl var \rangle$ is set to `\q_no_value`.

T_EXhackers note: This protected macro is a wrapper around the T_EX primitive `\read`. Regardless of settings, T_EX replaces trailing space and tab characters (character codes 32 and 9) in each line by an end-of-line character (character code `\endlinechar`, omitted if `\endlinechar` is negative or too large) before turning characters into tokens according to current category codes. With default settings, spaces appearing at the beginning of lines are also ignored.

<code>\ior_str_get:NN</code>
<code>\ior_str_get:NNTF</code>
New: 2016-12-04
Updated: 2019-03-23

`\ior_str_get:NN` $\langle stream \rangle$ $\langle token list variable \rangle$
`\ior_str_get:NNTF` $\langle stream \rangle$ $\langle token list variable \rangle$ $\langle true code \rangle$ $\langle false code \rangle$

Function that reads one line from the file input $\langle stream \rangle$ and stores the result locally in the $\langle token list \rangle$ variable. The material is read from the $\langle stream \rangle$ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). Multiple whitespace characters are retained by this process. It always only reads one line and any blank lines in the input result in the $\langle token list variable \rangle$ being empty. Unlike `\ior_get:NN`, line ends do not receive any special treatment. Thus input

a b c

results in a token list `a b c` with the letters a, b, and c having category code 12. In the non-branching version, where the $\langle stream \rangle$ is not open the $\langle tl var \rangle$ is set to `\q_no_value`.

T_EXhackers note: This protected macro is a wrapper around the ε -T_EX primitive `\readline`. Regardless of settings, T_EX removes trailing space and tab characters (character codes 32 and 9). However, the end-line character normally added by this primitive is not included in the result of `\ior_str_get:NN`.

All mappings are done at the current group level, *i.e.* any local assignments made by the $\langle function \rangle$ or $\langle code \rangle$ discussed below remain in effect after the loop.

<hr/> <code>\ior_map_inline:Nn</code> <hr/>	<code>\ior_map_inline:Nn <stream> {<inline function>}</code>
<hr/> New: 2012-02-11 <hr/>	Applies the <i><inline function></i> to each set of <i><lines></i> obtained by calling <code>\ior_get:NN</code> until reaching the end of the file. \TeX ignores any trailing new-line marker from the file it reads. The <i><inline function></i> should consist of code which receives the <i><line></i> as <i>#1</i> .
<hr/> <code>\ior_str_map_inline:Nn</code> <hr/>	<code>\ior_str_map_inline:Nn <stream> {<inline function>}</code>
<hr/> New: 2012-02-11 <hr/>	Applies the <i><inline function></i> to every <i><line></i> in the <i><stream></i> . The material is read from the <i><stream></i> as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The <i><inline function></i> should consist of code which receives the <i><line></i> as <i>#1</i> . Note that \TeX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. \TeX also ignores any trailing new-line marker from the file it reads.
<hr/> <code>\ior_map_variable:NNn</code> <hr/>	<code>\ior_map_variable:NNn <stream> <tl var> {<code>}</code>
<hr/> New: 2019-01-13 <hr/>	For each set of <i><lines></i> obtained by calling <code>\ior_get:NN</code> until reaching the end of the file, stores the <i><lines></i> in the <i><tl var></i> then applies the <i><code></i> . The <i><code></i> will usually make use of the <i><variable></i> , but this is not enforced. The assignments to the <i><variable></i> are local. Its value after the loop is the last set of <i><lines></i> , or its original value if the <i><stream></i> is empty. \TeX ignores any trailing new-line marker from the file it reads. This function is typically faster than <code>\ior_map_inline:Nn</code> .
<hr/> <code>\ior_str_map_variable:NNn</code> <hr/>	<code>\ior_str_map_variable:NNn <stream> <variable> {<code>}</code>
<hr/> New: 2019-01-13 <hr/>	For each <i><line></i> in the <i><stream></i> , stores the <i><line></i> in the <i><variable></i> then applies the <i><code></i> . The material is read from the <i><stream></i> as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The <i><code></i> will usually make use of the <i><variable></i> , but this is not enforced. The assignments to the <i><variable></i> are local. Its value after the loop is the last <i><line></i> , or its original value if the <i><stream></i> is empty. Note that \TeX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. \TeX also ignores any trailing new-line marker from the file it reads. This function is typically faster than <code>\ior_str_map_inline:Nn</code> .
<hr/> <code>\ior_map_break:</code> <hr/>	<code>\ior_map_break:</code>
<hr/> New: 2012-06-29 <hr/>	Used to terminate a <code>\ior_map_...</code> function before all lines from the <i><stream></i> have been processed. This normally takes place within a conditional statement, for example <pre> \ior_map_inline:Nn \l_my_ior { \str_if_eq:nnTF { #1 } { bingo } { \ior_map_break: } { % Do something useful } }</pre>

Use outside of a `\ior_map_...` scenario leads to low level \TeX errors.

\TeX hackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\ior_map_break:n`

`New: 2012-06-29`

`\ior_map_break:n {<code>}`

Used to terminate a `\ior_map_...` function before all lines in the `<stream>` have been processed, inserting the `<code>` after the mapping has ended. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \ior_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\ior_map_...` scenario leads to low level `TeX` errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the `<code>` is inserted into the input stream. This depends on the design of the mapping function.

`\ior_if_eof_p:N *``\ior_if_eof:NTF *`

`Updated: 2012-02-10`

`\ior_if_eof_p:N <stream>``\ior_if_eof:NTF <stream> {<true code>} {<false code>}`

Tests if the end of a file `<stream>` has been reached during a reading operation. The test also returns a `true` value if the `<stream>` is not open.

1.2 Writing to files

`\iow_now:Nn``\iow_now:(Nx|cn|cx)`

`Updated: 2012-06-05`

`\iow_now:Nn <stream> {<tokens>}`

This functions writes `<tokens>` to the specified `<stream>` immediately (*i.e.* the write operation is called on expansion of `\iow_now:Nn`).

`\iow_log:n``\iow_log:x``\iow_log:n {<tokens>}`

This function writes the given `<tokens>` to the log (transcript) file immediately: it is a dedicated version of `\iow_now:Nn`.

`\iow_term:n``\iow_term:x``\iow_term:n {<tokens>}`

This function writes the given `<tokens>` to the terminal file immediately: it is a dedicated version of `\iow_now:Nn`.

<hr/> <code>\iow_shipout:Nn</code> <hr/>	<code>\iow_shipout:Nn <stream> {<tokens>}</code>
<code>\iow_shipout:(Nx cn cx)</code> <hr/>	This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ when the current page is finalised (<i>i.e.</i> at shipout). The x -type variants expand the $\langle tokens \rangle$ at the point where the function is used but <i>not</i> when the resulting tokens are written to the $\langle stream \rangle$ (<i>cf.</i> <code>\iow_shipout_x:Nn</code>).

T_EXhackers note: When using expl3 with a format other than L^AT_EX, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

<hr/> <code>\iow_shipout_x:Nn</code> <hr/>	<code>\iow_shipout_x:Nn <stream> {<tokens>}</code>
<code>\iow_shipout_x:(Nx cn cx)</code> <hr/>	This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ when the current page is finalised (<i>i.e.</i> at shipout). The $\langle tokens \rangle$ are expanded at the time of writing in addition to any expansion when the function is used. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).

Updated: 2012-09-08

T_EXhackers note: This is a wrapper around the T_EX primitive `\write`. When using expl3 with a format other than L^AT_EX, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

<hr/> <code>\iow_char:N *</code> <hr/>	<code>\iow_char:N \<char></code>
	Inserts $\langle char \rangle$ into the output stream. Useful when trying to write difficult characters such as %, {, }, <i>etc.</i> in messages, for example:

`\iow_now:Nx \g_my_iow { \iow_char:N \{ text \iow_char:N \} }`

The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

<hr/> <code>\iow_newline: *</code> <hr/>	<code>\iow_newline:</code>
	Function to add a new line within the $\langle tokens \rangle$ written to a file. The function has no effect if writing is taking place without expansion (<i>e.g.</i> in the second argument of <code>\iow_now:Nn</code>).

T_EXhackers note: When using expl3 with a format other than L^AT_EX, the character inserted by `\iow_newline:` is not recognized by T_EX, which may lead to the insertion of additional unwanted line-breaks. This issue only affects `\iow_shipout:Nn`, `\iow_shipout_x:Nn` and direct uses of primitive operations.

1.3 Wrapping lines in output

`\iow_wrap:nnnN`
`\iow_wrap:nxnN`

New: 2012-06-28
Updated: 2017-12-04

`\iow_wrap:nnnN` $\langle text \rangle$ $\langle run-on\ text \rangle$ $\langle set\ up \rangle$ $\langle function \rangle$

This function wraps the $\langle text \rangle$ to a fixed number of characters per line. At the start of each line which is wrapped, the $\langle run-on\ text \rangle$ is inserted. The line character count targeted is the value of `\l_iow_line_count_int` minus the number of characters in the $\langle run-on\ text \rangle$ for all lines except the first, for which the target number of characters is simply `\l_iow_line_count_int` since there is no run-on text. The $\langle text \rangle$ and $\langle run-on\ text \rangle$ are exhaustively expanded by the function, with the following substitutions:

- `\` or `\iow_newline`: may be used to force a new line,
- `_` may be used to represent a forced space (for example after a control sequence),
- `\#`, `\%`, `\{`, `\}`, `\~` may be used to represent the corresponding character,
- `\iow_allow_break`: may be used to allow a line-break without inserting a space (this is experimental),
- `\iow_indent:n` may be used to indent a part of the $\langle text \rangle$ (not the $\langle run-on\ text \rangle$).

Additional functions may be added to the wrapping by using the $\langle set\ up \rangle$, which is executed before the wrapping takes place: this may include overriding the substitutions listed.

Any expandable material in the $\langle text \rangle$ which is not to be expanded on wrapping should be converted to a string using `\token_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N`, *etc.*

The result of the wrapping operation is passed as a braced argument to the $\langle function \rangle$, which is typically a wrapper around a write operation. The output of `\iow_wrap:nnnN` (*i.e.* the argument passed to the $\langle function \rangle$) consists of characters of category “other” (category code 12), with the exception of spaces which have category “space” (category code 10). This means that the output does *not* expand further when written to a file.

T_EXhackers note: Internally, `\iow_wrap:nnnN` carries out an x-type expansion on the $\langle text \rangle$ to expand it. This is done in such a way that `\exp_not:N` or `\exp_not:n` *could* be used to prevent expansion of material. However, this is less conceptually clear than conversion to a string, which is therefore the supported method for handling expandable material in the $\langle text \rangle$.

`\iow_indent:n`

New: 2011-09-21

`\iow_indent:n` $\langle text \rangle$

In the first argument of `\iow_wrap:nnnN` (for instance in messages), indents $\langle text \rangle$ by four spaces. This function does not cause a line break, and only affects lines which start within the scope of the $\langle text \rangle$. In case the indented $\langle text \rangle$ should appear on separate lines from the surrounding text, use `\` to force line breaks.

`\l_iow_line_count_int`

New: 2012-06-24

The maximum number of characters in a line to be written by the `\iow_wrap:nnnN` function. This value depends on the T_EX system in use: the standard value is 78, which is typically correct for unmodified T_EXlive and MiK_TE_X systems.

1.4 Constant input–output streams, and variables

<code>\g_tmpa_iow</code> <code>\g_tmpb_iow</code>	Scratch input stream for global use. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> New: 2017-12-11 <hr/>	

<code>\c_log_iow</code> <code>\c_term_iow</code>	Constant output streams for writing to the log and to the terminal (plus the log), respectively.
<hr/>	

<code>\g_tmpa_iow</code> <code>\g_tmpb_iow</code>	Scratch output stream for global use. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> New: 2017-12-11 <hr/>	

1.5 Primitive conditionals

<code>\if_eof:w</code> ★	<code>\if_eof:w <stream></code> <code><true code></code> <code>\else:</code> <code><false code></code> <code>\fi:</code> Tests if the <code><stream></code> returns “end of file”, which is true for non-existent files. The <code>\else:</code> branch is optional.
<hr/>	

T_EXhackers note: This is the T_EX primitive `\ifeof`.

2 File operation functions

<code>\g_file_curr_dir_str</code> <code>\g_file_curr_name_str</code> <code>\g_file_curr_ext_str</code>	Contain the directory, name and extension of the current file. The directory is empty if the file was loaded without an explicit path (<i>i.e.</i> if it is in the T _E X search path), and does <i>not</i> end in / other than the case that it is exactly equal to the root directory. The <code><name></code> and <code><ext></code> parts together make up the file name, thus the <code><name></code> part may be thought of as the “job name” for the current file. Note that T _E X does not provide information on the <code><ext></code> part for the main (top level) file and that this file always has an empty <code><dir></code> component. Also, the <code><name></code> here will be equal to <code>\c_sys_jobname_str</code> , which may be different from the real file name (if set using <code>--jobname</code> , for example).
<hr/> New: 2017-06-21 <hr/>	

<hr/> <code>\l_file_search_path_seq</code> <hr/> New: 2017-06-18 <hr/>	<p>Each entry is the path to a directory which should be searched when seeking a file. Each path can be relative or absolute, and should not include the trailing slash. The entries are not expanded when used so may contain active characters but should not feature any variable content. Spaces need not be quoted.</p> <p>T_EXhackers note: When working as a package in L^AT_EX 2_ε, <code>expl3</code> will automatically append the current <code>\input@path</code> to the set of values from <code>\l_file_search_path_seq</code>.</p>
<hr/> <code>\file_if_exist:nTF</code> <hr/> Updated: 2012-02-10 <hr/>	<code>\file_if_exist:nTF {<file name>} {<true code>} {<false code>}</code> <p>Searches for <code><file name></code> using the current T_EX search path and the additional paths controlled by <code>\l_file_search_path_seq</code>.</p>
<hr/> <code>\file_get:nnN</code> <code>\file_get:nnNTF</code> <hr/> New: 2019-01-16 Updated: 2019-02-16 <hr/>	<code>\file_get:nnN {<filename>} {<setup>} <tl></code> <code>\file_get:nnNTF {<filename>} {<setup>} <tl> {<true code>} {<false code>}</code> <p>Defines <code><tl></code> to the contents of <code><filename></code>. Category codes may need to be set appropriately via the <code><setup></code> argument. The non-branching version sets the <code><tl></code> to <code>\q_no_value</code> if the file is not found. The branching version runs the <code><true code></code> after the assignment to <code><tl></code> if the file is found, and <code><false code></code> otherwise.</p>
<hr/> <code>\file_get_full_name:nN</code> <code>\file_get_full_name:VN</code> <code>\file_get_full_name:nNTF</code> <code>\file_get_full_name:VNTF</code> <hr/> Updated: 2019-02-16 <hr/>	<code>\file_get_full_name:nN {<file name>} <tl></code> <code>\file_get_full_name:nNTF {<file name>} <tl> {<true code>} {<false code>}</code> <p>Searches for <code><file name></code> in the path as detailed for <code>\file_if_exist:nTF</code>, and if found sets the <code><tl var></code> the fully-qualified name of the file, <i>i.e.</i> the path and file name. This includes an extension <code>.tex</code> when the given <code><file name></code> has no extension but the file found has that extension. In the non-branching version, the <code><tl var></code> will be set to <code>\q_no_value</code> in the case that the file does not exist.</p>
<hr/> <code>\file_full_name:n</code> ☆ <code>\file_full_name:V</code> ☆ <hr/> New: 2019-09-03 <hr/>	<code>\file_full_name:n {<file name>}</code> <p>Searches for <code><file name></code> in the path as detailed for <code>\file_if_exist:nTF</code>, and if found leaves the fully-qualified name of the file, <i>i.e.</i> the path and file name, in the input stream. This includes an extension <code>.tex</code> when the given <code><file name></code> has no extension but the file found has that extension. If the file is not found on the path, the expansion is empty.</p>

<code>\file_parse_full_name:nNNN</code>
<code>\file_parse_full_name:VNNN</code>
New: 2017-06-23
Updated: 2020-06-24

`\file_parse_full_name:nNNN {<full name>} <dir> <name> <ext>`

Parses the `<full name>` and splits it into three parts, each of which is returned by setting the appropriate local string variable:

- The `<dir>`: everything up to the last / (path separator) in the `<file path>`. As with system PATH variables and related functions, the `<dir>` does *not* include the trailing / unless it points to the root directory. If there is no path (only a file name), `<dir>` is empty.
- The `<name>`: everything after the last / up to the last ., where both of those characters are optional. The `<name>` may contain multiple . characters. It is empty if `<full name>` consists only of a directory name.
- The `<ext>`: everything after the last . (including the dot). The `<ext>` is empty if there is no . after the last /.

Before parsing, the `<full name>` is expanded until only non-expandable tokens remain, except that active characters are also not expanded. Quotes (") are invalid in file names and are discarded from the input.

<code>\file_parse_full_name:n *</code>
New: 2020-06-24

`\file_parse_full_name:n {<full name>}`

Parses the `<full name>` as described for `\file_parse_full_name:nNNN`, and leaves `<dir>`, `<name>`, and `<ext>` in the input stream, each inside a pair of braces.

<code>\file_parse_full_name_apply:nN *</code>	<code>\file_parse_full_name_apply:nN {<full name>} <function></code>
New: 2020-06-24	

Parses the `<full name>` as described for `\file_parse_full_name:nNNN`, and passes `<dir>`, `<name>`, and `<ext>` as arguments to `<function>`, as an n-type argument each, in this order.

<code>\file_hex_dump:n ☆</code>
<code>\file_hex_dump:nnn ☆</code>
New: 2019-11-19

`\file_hex_dump:n {<file name>}`
`\file_hex_dump:nnn {<file name>} {<start index>} {<end index>}`

Searches for `<file name>` using the current T_EX search path and the additional paths controlled by `\l_file_search_path_seq`. It then expands to leave the hexadecimal dump of the file content in the input stream. The file is read as bytes, which means that in contrast to most T_EX behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty. The `{<start index>}` and `{<end index>}` values work as described for `\str_range:nnn`.

<code>\file_get_hex_dump:nN</code>
<code>\file_get_hex_dump:nNTF</code>
<code>\file_get_hex_dump:nnnN</code>
<code>\file_get_hex_dump:nnnNTF</code>
New: 2019-11-19

`\file_get_hex_dump:nN {<file name>} <tl var>`
`\file_get_hex_dump:nnnN {<file name>} {<start index>} {<end index>} <tl var>`

Sets the `<tl var>` to the result of applying `\file_hex_dump:n/\file_hex_dump:nnn` to the `<file>`. If the file is not found, the `<tl var>` will be set to `\q_no_value`.

<hr/> <code>\file_md5ive_hash:n</code> ☆ <hr/> <div>New: 2019-09-03</div> <hr/>	<code>\file_md5ive_hash:n</code> $\{\langle file\ name\rangle\}$ <p>Searches for $\langle file\ name\rangle$ using the current \TeX search path and the additional paths controlled by <code>\l_file_search_path_seq</code>. It then expands to leave the MD5 sum generated from the contents of the file in the input stream. The file is read as bytes, which means that in contrast to most \TeX behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty.</p>
<hr/> <code>\file_get_md5ive_hash:nN</code> <code>\file_get_md5ive_hash:nNTF</code> <hr/> <div>New: 2017-07-11 Updated: 2019-02-16</div> <hr/>	<code>\file_get_md5ive_hash:nN</code> $\{\langle file\ name\rangle\}$ $\langle tl\ var\rangle$ <p>Sets the $\langle tl\ var\rangle$ to the result of applying <code>\file_md5ive_hash:n</code> to the $\langle file\rangle$. If the file is not found, the $\langle tl\ var\rangle$ will be set to <code>\q_no_value</code>.</p>
<hr/> <code>\file_size:n</code> ☆ <hr/> <div>New: 2019-09-03</div> <hr/>	<code>\file_size:n</code> $\{\langle file\ name\rangle\}$ <p>Searches for $\langle file\ name\rangle$ using the current \TeX search path and the additional paths controlled by <code>\l_file_search_path_seq</code>. It then expands to leave the size of the file in bytes in the input stream. When the file is not found, the result of expansion is empty.</p>
<hr/> <code>\file_get_size:nN</code> <code>\file_get_size:nNTF</code> <hr/> <div>New: 2017-07-09 Updated: 2019-02-16</div> <hr/>	<code>\file_get_size:nN</code> $\{\langle file\ name\rangle\}$ $\langle tl\ var\rangle$ <p>Sets the $\langle tl\ var\rangle$ to the result of applying <code>\file_size:n</code> to the $\langle file\rangle$. If the file is not found, the $\langle tl\ var\rangle$ will be set to <code>\q_no_value</code>. This is not available in older versions of \XeTeX.</p>
<hr/> <code>\file_timestamp:n</code> ☆ <hr/> <div>New: 2019-09-03</div> <hr/>	<code>\file_timestamp:n</code> $\{\langle file\ name\rangle\}$ <p>Searches for $\langle file\ name\rangle$ using the current \TeX search path and the additional paths controlled by <code>\l_file_search_path_seq</code>. It then expands to leave the modification timestamp of the file in the input stream. The timestamp is of the form $D:\langle year\rangle\langle month\rangle\langle day\rangle\langle hour\rangle\langle minute\rangle\langle second\rangle\langle offset\rangle$, where the latter may be Z (UTC) or $\langle plus-minus\rangle\langle hours\rangle'\langle minutes\rangle'$. When the file is not found, the result of expansion is empty. This is not available in older versions of \XeTeX.</p>
<hr/> <code>\file_get_timestamp:nN</code> <code>\file_get_timestamp:nNTF</code> <hr/> <div>New: 2017-07-09 Updated: 2019-02-16</div> <hr/>	<code>\file_get_timestamp:nN</code> $\{\langle file\ name\rangle\}$ $\langle tl\ var\rangle$ <p>Sets the $\langle tl\ var\rangle$ to the result of applying <code>\file_timestamp:n</code> to the $\langle file\rangle$. If the file is not found, the $\langle tl\ var\rangle$ will be set to <code>\q_no_value</code>. This is not available in older versions of \XeTeX.</p>

<code>\file_compare_timestamp_p:nNn</code>	★	<code>\file_compare_timestamp:nNn {<file-1>} <comparator> {<file-2>} {<true code>} {<false code>}</code>
<code>\file_compare_timestamp:nNnTF</code>	★	

New: 2019-05-13

Updated: 2019-09-20

Compares the file stamps on the two *<files>* as indicated by the *<comparator>*, and inserts either the *<true code>* or *<false case>* as required. A file which is not found is treated as older than any file which is found. This allows for example the construct

```
\file_compare_timestamp:nNnT { source-file } > { derived-file }
{
  % Code to regenerate derived file
}
```

to work when the derived file is entirely absent. The timestamp of two absent files is regarded as different. This is not available in older versions of X_YTeX.

<code>\file_input:n</code>	<code>\file_input:n {<file name>}</code>
----------------------------	--

Updated: 2017-06-26

Searches for *<file name>* in the path as detailed for `\file_if_exist:nTF`, and if found reads in the file as additional L^AT_EX source. All files read are recorded for information and the file name stack is updated by this function. An error is raised if the file is not found.

<code>\file_if_exist_input:n</code>	<code>\file_if_exist_input:n {<file name>}</code>
<code>\file_if_exist_input:nF</code>	<code>\file_if_exist_input:nF {<file name>} {<false code>}</code>

New: 2014-07-02

Searches for *<file name>* using the current T_EX search path and the additional paths controlled by `\file_path_include:n`. If found then reads in the file as additional L^AT_EX source as described for `\file_input:n`, otherwise inserts the *<false code>*. Note that these functions do not raise an error if the file is not found, in contrast to `\file_input:n`.

<code>\file_input_stop:</code>	<code>\file_input_stop:</code>
--------------------------------	--------------------------------

New: 2017-07-07

Ends the reading of a file started by `\file_input:n` or similar before the end of the file is reached. Where the file reading is being terminated due to an error, `\msg_critical:nn(nn)` should be preferred.

T_EXhackers note: This function must be used on a line on its own: T_EX reads files line-by-line and so any additional tokens in the “current” line will still be read.

This is also true if the function is hidden inside another function (which will be the normal case), i.e., all tokens on the same line in the source file are still processed. Putting it on a line by itself in the definition doesn’t help as it is the line where it is used that counts!

<code>\file_show_list:</code>	<code>\file_show_list:</code>
<code>\file_log_list:</code>	<code>\file_log_list:</code>

These functions list all files loaded by L^AT_EX 2_ε commands that populate `\@filelist` or by `\file_input:n`. While `\file_show_list:` displays the list in the terminal, `\file_log_list:` outputs it to the log file only.

Part XX

The l3skip package

Dimensions and skips

L^AT_EX3 provides two general length variables: `dim` and `skip`. Lengths stored as `dim` variables have a fixed length, whereas `skip` lengths have a rubber (stretch/shrink) component. In addition, the `muskip` type is available for use in math mode: this is a special form of `skip` where the lengths involved are determined by the current math font (in μ). There are common features in the creation and setting of length variables, but for clarity the functions are grouped by variable type.

1 Creating and initialising `dim` variables

<code>\dim_new:N</code>
<code>\dim_new:c</code>

`\dim_new:N` $\langle dimension \rangle$

Creates a new $\langle dimension \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle dimension \rangle$ is initially equal to 0pt.

<code>\dim_const:Nn</code>
<code>\dim_const:cn</code>

`\dim_const:Nn` $\langle dimension \rangle$ $\{ \langle dimension expression \rangle \}$

Creates a new constant $\langle dimension \rangle$ or raises an error if the name is already taken. The value of the $\langle dimension \rangle$ is set globally to the $\langle dimension expression \rangle$.

New: 2012-03-05

<code>\dim_zero:N</code>
<code>\dim_zero:c</code>
<code>\dim_gzero:N</code>
<code>\dim_gzero:c</code>

`\dim_zero:N` $\langle dimension \rangle$

Sets $\langle dimension \rangle$ to 0pt.

<code>\dim_zero_new:N</code>
<code>\dim_zero_new:c</code>
<code>\dim_gzero_new:N</code>
<code>\dim_gzero_new:c</code>

`\dim_zero_new:N` $\langle dimension \rangle$

Ensures that the $\langle dimension \rangle$ exists globally by applying `\dim_new:N` if necessary, then applies `\dim_(g)zero:N` to leave the $\langle dimension \rangle$ set to zero.

New: 2012-01-07

<code>\dim_if_exist_p:N</code> \star
<code>\dim_if_exist_p:c</code> \star
<code>\dim_if_exist:N\overline{TF}</code> \star
<code>\dim_if_exist:c\overline{TF}</code> \star

`\dim_if_exist_p:N` $\langle dimension \rangle$

`\dim_if_exist:N \overline{TF}` $\langle dimension \rangle$ $\{ \langle true code \rangle \} \{ \langle false code \rangle \}$

Tests whether the $\langle dimension \rangle$ is currently defined. This does not check that the $\langle dimension \rangle$ really is a dimension variable.

New: 2012-03-03

2 Setting dim variables

<code>\dim_add:Nn</code>	<code>\dim_add:Nn <dimension> {<dimension expression>}</code>
<code>\dim_add:cn</code>	
<code>\dim_gadd:Nn</code>	Adds the result of the $\langle dimension expression \rangle$ to the current content of the $\langle dimension \rangle$.
<code>\dim_gadd:cn</code>	

Updated: 2011-10-22

<code>\dim_set:Nn</code>	<code>\dim_set:Nn <dimension> {<dimension expression>}</code>
<code>\dim_set:cn</code>	
<code>\dim_gset:Nn</code>	Sets $\langle dimension \rangle$ to the value of $\langle dimension expression \rangle$, which must evaluate to a length with units.
<code>\dim_gset:cn</code>	

Updated: 2011-10-22

<code>\dim_set_eq:NN</code>	<code>\dim_set_eq:NN <dimension₁₂</code>
<code>\dim_set_eq:(cN Nc cc)</code>	Sets the content of $\langle dimension_1 \rangle$ equal to that of $\langle dimension_2 \rangle$.
<code>\dim_gset_eq:NN</code>	
<code>\dim_gset_eq:(cN Nc cc)</code>	

<code>\dim_sub:Nn</code>	<code>\dim_sub:Nn <dimension> {<dimension expression>}</code>
<code>\dim_sub:cn</code>	
<code>\dim_gsub:Nn</code>	Subtracts the result of the $\langle dimension expression \rangle$ from the current content of the $\langle dimension \rangle$.
<code>\dim_gsub:cn</code>	

Updated: 2011-10-22

3 Utilities for dimension calculations

<code>\dim_abs:n</code>	<code>\dim_abs:n {<dimexpr>}</code>
Updated: 2012-09-26	Converts the $\langle dimexpr \rangle$ to its absolute value, leaving the result in the input stream as a $\langle dimension denotation \rangle$.

<code>\dim_max:nn</code>	<code>\dim_max:nn {<dimexpr₁>} {<dimexpr₂>}</code>
<code>\dim_min:nn</code>	<code>\dim_min:nn {<dimexpr₁>} {<dimexpr₂>}</code>
New: 2012-09-09	Evaluates the two $\langle dimension expressions \rangle$ and leaves either the maximum or minimum value in the input stream as appropriate, as a $\langle dimension denotation \rangle$.
Updated: 2012-09-26	

`\dim_ratio:nn` ☆

Updated: 2011-10-22

`\dim_ratio:nn {⟨dimexpr1⟩} {⟨dimexpr2⟩}`

Parses the two *⟨dimension expressions⟩* and converts the ratio of the two to a form suitable for use inside a *⟨dimension expression⟩*. This ratio is then left in the input stream, allowing syntax such as

```
\dim_set:Nn \l_my_dim
{ 10 pt * \dim_ratio:nn { 5 pt } { 10 pt } }
```

The output of `\dim_ratio:nn` on full expansion is a ratio expression between two integers, with all distances converted to scaled points. Thus

```
\tl_set:Nx \l_my_tl { \dim_ratio:nn { 5 pt } { 10 pt } }
\tl_show:N \l_my_tl
```

displays 327680/655360 on the terminal.

4 Dimension expression conditionals

`\dim_compare_p:nNn` ★

`\dim_compare:nNnTF` ★

`\dim_compare_p:nNn {⟨dimexpr1⟩} <relation> {⟨dimexpr2⟩}`

`\dim_compare:nNnTF`

```
{⟨dimexpr1⟩} <relation> {⟨dimexpr2⟩}
{⟨true code⟩} {⟨false code⟩}
```

This function first evaluates each of the *⟨dimension expressions⟩* as described for `\dim_eval:n`. The two results are then compared using the *⟨relation⟩*:

Equal	=
Greater than	>
Less than	<

This function is less flexible than `\dim_compare:nTF` but around 5 times faster.

```

\dim_compare_p:n * \dim_compare_p:n
\dim_compare:nTF * {
    <dimexpr1> <relation1>
    ...
    <dimexprN> <relationN>
    <dimexprN+1>
}
\dim_compare:nTF
{
    <dimexpr1> <relation1>
    ...
    <dimexprN> <relationN>
    <dimexprN+1>
}
{{true code}} {{false code}}

```

Updated: 2013-01-13

This function evaluates the *<dimension expressions>* as described for `\dim_eval:n` and compares consecutive result using the corresponding *<relation>*, namely it compares *<dimexpr₁>* and *<dimexpr₂>* using the *<relation₁>*, then *<dimexpr₂>* and *<dimexpr₃>* using the *<relation₂>*, until finally comparing *<dimexpr_N>* and *<dimexpr_{N+1}>* using the *<relation_N>*. The test yields `true` if all comparisons are `true`. Each *<dimension expression>* is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other *<dimension expression>* is evaluated and no other comparison is performed. The *<relations>* can be any of the following:

Equal	= or ==
Greater than or equal to	>=
Greater than	>
Less than or equal to	<=
Less than	<
Not equal	!=

This function is more flexible than `\dim_compare:nNnTF` but around 5 times slower.

<code>\dim_case:nn</code> ☆	<code>\dim_case:nnTF {⟨test dimension expression⟩}</code>
<code>\dim_case:nnTF</code> ☆	<code>{</code>
New: 2013-07-24	<code>{⟨dimexpr case₁⟩} {⟨code case₁⟩}</code>
	<code>{⟨dimexpr case₂⟩} {⟨code case₂⟩}</code>
	<code>...</code>
	<code>{⟨dimexpr case_n⟩} {⟨code case_n⟩}</code>
	<code>}</code>
	<code>{⟨true code⟩}</code>
	<code>{⟨false code⟩}</code>

This function evaluates the *⟨test dimension expression⟩* and compares this in turn to each of the *⟨dimension expression cases⟩*. If the two are equal then the associated *⟨code⟩* is left in the input stream and other cases are discarded. If any of the cases are matched, the *⟨true code⟩* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *⟨false code⟩* is inserted. The function `\dim_case:nn`, which does nothing if there is no match, is also available. For example

```

\dim_set:Nn \l_tmpa_dim { 5 pt }
\dim_case:nnF
{ 2 \l_tmpa_dim }
{
  { 5 pt }      { Small }
  { 4 pt + 6 pt } { Medium }
  { - 10 pt }   { Negative }
}
{ No idea! }

```

leaves “Medium” in the input stream.

5 Dimension expression loops

<code>\dim_do_until:nNnn</code> ☆	<code>\dim_do_until:nNnn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Places the *⟨code⟩* in the input stream for T_EX to process, and then evaluates the relationship between the two *⟨dimension expressions⟩* as described for `\dim_compare:nNnTF`. If the test is **false** then the *⟨code⟩* is inserted into the input stream again and a loop occurs until the *⟨relation⟩* is **true**.

<code>\dim_do_while:nNnn</code> ☆	<code>\dim_do_while:nNnn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Places the *⟨code⟩* in the input stream for T_EX to process, and then evaluates the relationship between the two *⟨dimension expressions⟩* as described for `\dim_compare:nNnTF`. If the test is **true** then the *⟨code⟩* is inserted into the input stream again and a loop occurs until the *⟨relation⟩* is **false**.

<code>\dim_until_do:nNnn</code> ☆	<code>\dim_until_do:nNnn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Evaluates the relationship between the two *⟨dimension expressions⟩* as described for `\dim_compare:nNnTF`, and then places the *⟨code⟩* in the input stream if the *⟨relation⟩* is **false**. After the *⟨code⟩* has been processed by T_EX the test is repeated, and a loop occurs until the test is **true**.

<hr/> <code>\dim_while_do:nNnn</code> ☆ <hr/>	<code>\dim_while_do:nNnn {<dimexpr₁>} <relation> {<dimexpr₂>} {<code>}</code>
	Evaluates the relationship between the two <i><dimension expressions></i> as described for <code>\dim_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .
<hr/> <code>\dim_do_until:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_do_until:nn {<dimension relation>} {<code>}</code>
	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> . If the test is false then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is true .
<hr/> <code>\dim_do_while:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_do_while:nn {<dimension relation>} {<code>}</code>
	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> . If the test is true then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is false .
<hr/> <code>\dim_until_do:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_until_do:nn {<dimension relation>} {<code>}</code>
	Evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is true .
<hr/> <code>\dim_while_do:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_while_do:nn {<dimension relation>} {<code>}</code>
	Evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .

6 Dimension step functions

<hr/> <code>\dim_step_function:nnnN</code> ☆ <hr/> <div>New: 2018-02-18</div>	<code>\dim_step_function:nnnN {<initial value>} {<step>} {<final value>} <function></code>
	This function first evaluates the <i><initial value></i> , <i><step></i> and <i><final value></i> , all of which should be dimension expressions. The <i><function></i> is then placed in front of each <i><value></i> from the <i><initial value></i> to the <i><final value></i> in turn (using <i><step></i> between each <i><value></i>). The <i><step></i> must be non-zero. If the <i><step></i> is positive, the loop stops when the <i><value></i> becomes larger than the <i><final value></i> . If the <i><step></i> is negative, the loop stops when the <i><value></i> becomes smaller than the <i><final value></i> . The <i><function></i> should absorb one argument.
<hr/> <code>\dim_step_inline:nnnn</code> <hr/> <div>New: 2018-02-18</div>	<code>\dim_step_inline:nnnn {<initial value>} {<step>} {<final value>} {<code>}</code>
	This function first evaluates the <i><initial value></i> , <i><step></i> and <i><final value></i> , all of which should be dimension expressions. Then for each <i><value></i> from the <i><initial value></i> to the <i><final value></i> in turn (using <i><step></i> between each <i><value></i>), the <i><code></i> is inserted into the input stream with #1 replaced by the current <i><value></i> . Thus the <i><code></i> should define a function of one argument (#1).

`\dim_step_variable:nnnNn`
 New: 2018-02-18

`\dim_step_variable:nnnNn`
`{\langle initial value \rangle}{\langle step \rangle}{\langle final value \rangle}{\langle tl var \rangle}{\langle code \rangle}`

This function first evaluates the $\langle initial value \rangle$, $\langle step \rangle$ and $\langle final value \rangle$, all of which should be dimension expressions. Then for each $\langle value \rangle$ from the $\langle initial value \rangle$ to the $\langle final value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream, with the $\langle tl var \rangle$ defined as the current $\langle value \rangle$. Thus the $\langle code \rangle$ should make use of the $\langle tl var \rangle$.

7 Using dim expressions and variables

`\dim_eval:n` ★
 Updated: 2011-10-22

`\dim_eval:n` $\{\langle dimension expression \rangle\}$

Evaluates the $\langle dimension expression \rangle$, expanding any dimensions and token list variables within the $\langle expression \rangle$ to their content (without requiring `\dim_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle dimension denotation \rangle$ after two expansions. This is expressed in points (**pt**), and requires suitable termination if used in a TeX-style assignment as it is *not* an $\langle internal dimension \rangle$.

`\dim_sign:n` ★
 New: 2018-11-03

`\dim_sign:n` $\{\langle dimexpr \rangle\}$

Evaluates the $\langle dimexpr \rangle$ then leaves 1 or 0 or -1 in the input stream according to the sign of the result.

`\dim_use:N` ★
`\dim_use:c` ★

`\dim_use:N` $\langle dimension \rangle$

Recovers the content of a $\langle dimension \rangle$ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle dimension \rangle$ is required (such as in the argument of `\dim_eval:n`).

TeXhackers note: `\dim_use:N` is the TeX primitive `\the`: this is one of several L^ATeX3 names for this primitive.

`\dim_to_decimal:n` ★
 New: 2014-07-15

`\dim_to_decimal:n` $\{\langle dimexpr \rangle\}$

Evaluates the $\langle dimension expression \rangle$, and leaves the result, expressed in points (**pt**) in the input stream, with *no units*. The result is rounded by TeX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

`\dim_to_decimal:n { 1bp }`

leaves 1.00374 in the input stream, *i.e.* the magnitude of one “big point” when converted to (TeX) points.

<hr/> <code>\dim_to_decimal_in_bp:n</code> ★ <hr/>	<code>\dim_to_decimal_in_bp:n {⟨dimexpr⟩}</code>
<hr/> New: 2014-07-15 <hr/>	Evaluates the <i>⟨dimension expression⟩</i> , and leaves the result, expressed in big points (bp) in the input stream, with <i>no units</i> . The result is rounded by T _E X to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_bp:n { 1pt }
```

leaves 0.99628 in the input stream, *i.e.* the magnitude of one (T_EX) point when converted to big points.

<hr/> <code>\dim_to_decimal_in_sp:n</code> ★ <hr/>	<code>\dim_to_decimal_in_sp:n {⟨dimexpr⟩}</code>
<hr/> New: 2015-05-18 <hr/>	Evaluates the <i>⟨dimension expression⟩</i> , and leaves the result, expressed in scaled points (sp) in the input stream, with <i>no units</i> . The result is necessarily an integer.

<hr/> <code>\dim_to_decimal_in_unit:nn</code> ★ <hr/>	<code>\dim_to_decimal_in_unit:nn {⟨dimexpr₁⟩} {⟨dimexpr₂⟩}</code>
<hr/> New: 2014-07-15 <hr/>	

Evaluates the *⟨dimension expressions⟩*, and leaves the value of *⟨dimexpr₁⟩*, expressed in a unit given by *⟨dimexpr₂⟩*, in the input stream. The result is a decimal number, rounded by T_EX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_unit:nn { 1bp } { 1mm }
```

leaves 0.35277 in the input stream, *i.e.* the magnitude of one big point when converted to millimetres.

Note that this function is not optimised for any particular output and as such may give different results to `\dim_to_decimal_in_bp:n` or `\dim_to_decimal_in_sp:n`. In particular, the latter is able to take a wider range of input values as it is not limited by the ability to calculate a ratio using ε -T_EX primitives, which is required internally by `\dim_to_decimal_in_unit:nn`.

<hr/> <code>\dim_to_fp:n</code> ★ <hr/>	<code>\dim_to_fp:n {⟨dimexpr⟩}</code>
<hr/> New: 2012-05-08 <hr/>	Expands to an internal floating point number equal to the value of the <i>⟨dimexpr⟩</i> in pt. Since dimension expressions are evaluated much faster than their floating point equivalent, <code>\dim_to_fp:n</code> can be used to speed up parts of a computation where a low precision and a smaller range are acceptable.

8 Viewing dim variables

<hr/> <code>\dim_show:N</code> <hr/>	<code>\dim_show:N ⟨dimension⟩</code>
<code>\dim_show:c</code>	Displays the value of the <i>⟨dimension⟩</i> on the terminal.

<hr/> <code>\dim_show:n</code> <hr/>	<code>\dim_show:n {⟨<i>dimension expression</i>⟩}</code>
New: 2011-11-22 Updated: 2015-08-07	Displays the result of evaluating the $\langle dimension\ expression \rangle$ on the terminal.
<hr/> <code>\dim_log:N</code> <code>\dim_log:c</code> <hr/>	<code>\dim_log:N ⟨<i>dimension</i>⟩</code>
New: 2014-08-22 Updated: 2015-08-03	Writes the value of the $\langle dimension \rangle$ in the log file.
<hr/> <code>\dim_log:n</code> <hr/>	<code>\dim_log:n {⟨<i>dimension expression</i>⟩}</code>
New: 2014-08-22 Updated: 2015-08-07	Writes the result of evaluating the $\langle dimension\ expression \rangle$ in the log file.

9 Constant dimensions

<hr/> <code>\c_max_dim</code> <hr/>	The maximum value that can be stored as a dimension. This can also be used as a component of a skip.
<hr/> <code>\c_zero_dim</code> <hr/>	A zero length as a dimension. This can also be used as a component of a skip.

10 Scratch dimensions

<hr/> <code>\l_tmpa_dim</code> <code>\l_tmppb_dim</code> <hr/>	Scratch dimension for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <code>\g_tmpa_dim</code> <code>\g_tmppb_dim</code> <hr/>	Scratch dimension for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

11 Creating and initialising skip variables

<hr/> <code>\skip_new:N</code> <code>\skip_new:c</code> <hr/>	<code>\skip_new:N ⟨<i>skip</i>⟩</code>
	Creates a new $\langle skip \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle skip \rangle$ is initially equal to 0 pt.

<code>\skip_const:Nn</code>	<code>\skip_const:Nn <skip> {<skip expression>}</code>
<code>\skip_const:cn</code>	Creates a new constant <i><skip></i> or raises an error if the name is already taken. The value of the <i><skip></i> is set globally to the <i><skip expression></i> .
New: 2012-03-05	

<code>\skip_zero:N</code>	<code>\skip_zero:N <skip></code>
<code>\skip_zero:c</code>	Sets <i><skip></i> to 0 pt.
<code>\skip_gzero:N</code>	
<code>\skip_gzero:c</code>	

<code>\skip_zero_new:N</code>	<code>\skip_zero_new:N</code> $\langle skip \rangle$
<code>\skip_zero_new:c</code>	Ensures that the $\langle skip \rangle$ exists globally by applying <code>\skip_new:N</code> if necessary, then applies <code>\skip_(g)zero:N</code> to leave the $\langle skip \rangle$ set to zero.
<code>\skip_gzero_new:N</code>	
<code>\skip_gzero_new:c</code>	
<hr/>	
New: 2012-01-07	

<code>\skip_if_exist_p:N *</code>	<code>\skip_if_exist_p:N <skip></code>
<code>\skip_if_exist_p:c *</code>	<code>\skip_if_exist:NTF <skip> {<true code>} {<false code>}</code>
<code>\skip_if_exist:NTF *</code>	Tests whether the <code><skip></code> is currently defined. This does not check that the <code><skip></code> really is a skip variable.
<code>\skip_if_exist:cTF *</code>	

New: 2012-03-03

12 Setting skip variables

<code>\skip_add:Nn</code>	<code>\skip_add:Nn <skip> {<skip expression>}</code>
<code>\skip_add:cn</code>	Adds the result of the <i><skip expression></i> to the current content of the <i><skip></i> .
<code>\skip_gadd:Nn</code>	
<code>\skip_gadd:cn</code>	

Updated: 2011-10-22

<code>\skip_set:Nn</code>	<code>\skip_set:Nn <skip> {<skip expression>}</code>
<code>\skip_set:cn</code>	Sets <code><skip></code> to the value of <code><skip expression></code> , which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm).
<code>\skip_gset:Nn</code>	
<code>\skip_gset:cn</code>	
Updated: 2011-10-22	

<code>\skip_set_eq:NN</code>	<code>\skip_set_eq:NN <skip₁₂</code>
<code>\skip_set_eq:(cN Nc cc)</code>	Sets the content of <i><skip_{1 equal to that of <i><skip_{2.}</i>}</i>
<code>\skip_gset_eq:NN</code>	
<code>\skip_gset_eq:(cN Nc cc)</code>	

<code>\skip_sub:Nn</code>	<code>\skip_sub:Nn <skip> {<skip expression>}</code>
<code>\skip_sub:cn</code>	Subtracts the result of the <i><skip expression></i> from the current content of the <i><skip></i> .
<code>\skip_gsub:Nn</code>	
<code>\skip_gsub:cn</code>	

Updated: 2011-10-22

13 Skip expression conditionals

<code>\skip_if_eq_p:nn</code> *	<code>\skip_if_eq_p:nn {\langle skipexpr_1 \rangle} {\langle skipexpr_2 \rangle}</code>
<code>\skip_if_eq:nnTF</code> *	<code>\skip_if_eq:nnTF</code> <code>{\langle skipexpr_1 \rangle} {\langle skipexpr_2 \rangle}</code> <code>{\langle true code \rangle} {\langle false code \rangle}</code>

This function first evaluates each of the $\langle skip\ expressions \rangle$ as described for `\skip_eval:n`. The two results are then compared for exact equality, *i.e.* both the fixed and rubber components must be the same for the test to be true.

<code>\skip_if_finite_p:n</code> *	<code>\skip_if_finite_p:n {\langle skipexpr \rangle}</code>
<code>\skip_if_finite:nTF</code> *	<code>\skip_if_finite:nTF {\langle skipexpr \rangle} {\langle true code \rangle} {\langle false code \rangle}</code>

New: 2012-03-05

Evaluates the $\langle skip\ expression \rangle$ as described for `\skip_eval:n`, and then tests if all of its components are finite.

14 Using skip expressions and variables

<code>\skip_eval:n</code> *	<code>\skip_eval:n {\langle skip expression \rangle}</code>
-----------------------------	---

Updated: 2011-10-22

Evaluates the $\langle skip\ expression \rangle$, expanding any skips and token list variables within the $\langle expression \rangle$ to their content (without requiring `\skip_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle glue\ denotation \rangle$ after two expansions. This is expressed in points (`pt`), and requires suitable termination if used in a T_EX-style assignment as it is *not* an $\langle internal\ glue \rangle$.

<code>\skip_use:N</code> *	<code>\skip_use:N \langle skip \rangle</code>
<code>\skip_use:c</code> *	

Recovers the content of a $\langle skip \rangle$ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle dimension \rangle$ or $\langle skip \rangle$ is required (such as in the argument of `\skip_eval:n`).

T_EXhackers note: `\skip_use:N` is the T_EX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

15 Viewing skip variables

<code>\skip_show:N</code>	<code>\skip_show:N \langle skip \rangle</code>
<code>\skip_show:c</code>	

Updated: 2015-08-03

Displays the value of the $\langle skip \rangle$ on the terminal.

<code>\skip_show:n</code>	<code>\skip_show:n {\langle skip expression \rangle}</code>
---------------------------	---

New: 2011-11-22
Updated: 2015-08-07

Displays the result of evaluating the $\langle skip\ expression \rangle$ on the terminal.

<code>\skip_log:N</code>	<code>\skip_log:N <skip></code>
<code>\skip_log:c</code>	Writes the value of the $\langle skip \rangle$ in the log file.
New: 2014-08-22	
Updated: 2015-08-03	

<code>\skip_log:n</code>	<code>\skip_log:n {\langle skip expression \rangle}</code>
	Writes the result of evaluating the $\langle skip expression \rangle$ in the log file.
New: 2014-08-22	
Updated: 2015-08-07	

16 Constant skips

<code>\c_max_skip</code>	The maximum value that can be stored as a skip (equal to <code>\c_max_dim</code> in length), with no stretch nor shrink component.
Updated: 2012-11-02	

<code>\c_zero_skip</code>	A zero length as a skip, with no stretch nor shrink component.
Updated: 2012-11-01	

17 Scratch skips

<code>\l_tmpa_skip</code> <code>\l_tmpb_skip</code>	Scratch skip for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
--	--

<code>\g_tmpa_skip</code> <code>\g_tmpb_skip</code>	Scratch skip for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
--	---

18 Inserting skips into the output

<code>\skip_horizontal:N</code>	<code>\skip_horizontal:N <skip></code>
<code>\skip_horizontal:c</code>	<code>\skip_horizontal:n {\langle skipexpr \rangle}</code>
<code>\skip_horizontal:n</code>	Inserts a horizontal $\langle skip \rangle$ into the current list. The argument can also be a $\langle dim \rangle$.
Updated: 2011-10-22	
T_EXhackers note: <code>\skip_horizontal:N</code> is the T _E X primitive <code>\hskip</code> renamed.	

<hr/>	
<code>\skip_vertical:N</code>	<code>\skip_vertical:N <skip></code>
<code>\skip_vertical:c</code>	<code>\skip_vertical:n {<skipexpr>}</code>
<code>\skip_vertical:n</code>	Inserts a vertical <code><skip></code> into the current list. The argument can also be a <code><dim></code> .
<hr/> Updated: 2011-10-22 <hr/>	

T_EXhackers note: `\skip_vertical:N` is the T_EX primitive `\vskip` renamed.

19 Creating and initialising muskip variables

<hr/>	
<code>\muskip_new:N</code>	<code>\muskip_new:N <muskip></code>
<code>\muskip_new:c</code>	Creates a new <code><muskip></code> or raises an error if the name is already taken. The declaration is global. The <code><muskip></code> is initially equal to 0 mu.
<hr/>	

<hr/>	
<code>\muskip_const:Nn</code>	<code>\muskip_const:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_const:cn</code>	Creates a new constant <code><muskip></code> or raises an error if the name is already taken. The value of the <code><muskip></code> is set globally to the <code><muskip expression></code> .
<hr/> New: 2012-03-05 <hr/>	

<hr/>	
<code>\muskip_zero:N</code>	<code>\skip_zero:N</code> $\langle muskip \rangle$
<code>\muskip_zero:c</code>	Sets $\langle muskip \rangle$ to 0 mu.
<code>\muskip_gzero:N</code>	
<code>\muskip_gzero:c</code>	
<hr/>	

<code>\muskip_zero_new:N</code>	<code>\muskip_zero_new:N</code> $\langle muskip \rangle$
<code>\muskip_zero_new:c</code>	Ensures that the $\langle muskip \rangle$ exists globally by applying <code>\muskip_new:N</code> if necessary, then applies <code>\muskip_(g)zero:N</code> to leave the $\langle muskip \rangle$ set to zero.
<code>\muskip_gzero_new:N</code>	
<code>\muskip_gzero_new:c</code>	
<hr/>	
New: 2012-01-07	

<hr/>	
<code>\muskip_if_exist_p:N *</code>	<code>\muskip_if_exist_p:N <muskip></code>
<code>\muskip_if_exist_p:c *</code>	<code>\muskip_if_exist:NTF <muskip> {<true code>} {<false code>}</code>
<code>\muskip_if_exist:NTF *</code>	Tests whether the <code><muskip></code> is currently defined. This does not check that the <code><muskip></code> really is a muskip variable.
<code>\muskip_if_exist:cTF *</code>	
<hr/>	
New: 2012-03-03	

20 Setting muskip variables

<code>\muskip_add:Nn</code>	<code>\muskip_add:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_add:cn</code>	Adds the result of the <i><muskip expression></i> to the current content of the <i><muskip></i> .
<code>\muskip_gadd:Nn</code>	
<code>\muskip_gadd:cn</code>	
<hr/>	
Updated: 2011-10-22	

<code>\muskip_set:Nn</code>	<code>\muskip_set:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_set:cn</code>	Sets <i><muskip></i> to the value of <i><muskip expression></i> , which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu).
<code>\muskip_gset:Nn</code>	
<code>\muskip_gset:cn</code>	
Updated: 2011-10-22	

<code>\muskip_set_eq:NN</code>	<code>\muskip_set_eq:NN <muskip₁> <muskip₂></code>
<code>\muskip_set_eq:(cN Nc cc)</code>	Sets the content of <i><muskip₁></i> equal to that of <i><muskip₂></i> .
<code>\muskip_gset_eq:NN</code>	
<code>\muskip_gset_eq:(cN Nc cc)</code>	

<code>\muskip_sub:Nn</code>	<code>\muskip_sub:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_sub:cn</code>	Subtracts the result of the <i><muskip expression></i> from the current content of the <i><muskip></i> .
<code>\muskip_gsub:Nn</code>	
<code>\muskip_gsub:cn</code>	
Updated: 2011-10-22	

21 Using muskip expressions and variables

<code>\muskip_eval:n *</code>	<code>\muskip_eval:n {<muskip expression>}</code>
Updated: 2011-10-22	Evaluates the <i><muskip expression></i> , expanding any skips and token list variables within the <i><expression></i> to their content (without requiring <code>\muskip_use:N/\tl_use:N</code>) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a <i><mu glue denotation></i> after two expansions. This is expressed in mu , and requires suitable termination if used in a T _E X-style assignment as it is <i>not</i> an <i><internal mu glue></i> .

<code>\muskip_use:N *</code>	<code>\muskip_use:N <muskip></code>
<code>\muskip_use:c *</code>	Recovers the content of a <i><skip></i> and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a <i><dimension></i> is required (such as in the argument of <code>\muskip_eval:n</code>).

T_EXhackers note: `\muskip_use:N` is the T_EX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

22 Viewing muskip variables

<code>\muskip_show:N</code>	<code>\muskip_show:N <muskip></code>
<code>\muskip_show:c</code>	Displays the value of the <i><muskip></i> on the terminal.
Updated: 2015-08-03	

<hr/> <code>\muskip_show:n</code> <hr/>	<code>\muskip_show:n {⟨<i>muskip expression</i>⟩}</code>
New: 2011-11-22 Updated: 2015-08-07	Displays the result of evaluating the $\langle muskip expression \rangle$ on the terminal.
<hr/> <code>\muskip_log:N</code> <code>\muskip_log:c</code> <hr/>	<code>\muskip_log:N ⟨<i>muskip</i>⟩</code>
New: 2014-08-22 Updated: 2015-08-03	Writes the value of the $\langle muskip \rangle$ in the log file.
<hr/> <code>\muskip_log:n</code> <hr/>	<code>\muskip_log:n {⟨<i>muskip expression</i>⟩}</code>
New: 2014-08-22 Updated: 2015-08-07	Writes the result of evaluating the $\langle muskip expression \rangle$ in the log file.

23 Constant muskips

<hr/> <code>\c_max_muskip</code> <hr/>	The maximum value that can be stored as a muskip, with no stretch nor shrink component.
<hr/> <code>\c_zero_muskip</code> <hr/>	A zero length as a muskip, with no stretch nor shrink component.

24 Scratch muskips

<hr/> <code>\l_tmpa_muskip</code> <code>\l_tmpb_muskip</code> <hr/>	Scratch muskip for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <code>\g_tmpa_muskip</code> <code>\g_tmpb_muskip</code> <hr/>	Scratch muskip for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

25 Primitive conditional

<hr/> <code>\if_dim:w ★</code> <hr/>	<code>\if_dim:w ⟨<i>dimen</i>₁⟩ ⟨<i>relation</i>⟩ ⟨<i>dimen</i>₂⟩</code> <code> ⟨<i>true code</i>⟩</code> <code> \else:</code> <code> ⟨<i>false</i>⟩</code> <code> \fi:</code>
	Compare two dimensions. The $\langle relation \rangle$ is one of $<$, $=$ or $>$ with category code 12.

T_EXhackers note: This is the T_EX primitive `\ifdim`.

Part XXI

The l3keys package

Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. The system normally results in input of the form

```
\MyModuleSetup{
  key-one = value one,
  key-two = value two
}
```

or

```
\MyModuleMacro[
  key-one = value one,
  key-two = value two
]{argument}
```

for the user.

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimising the number of functions and arguments required. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { mymodule }
{
  key-one .code:n = code including parameter #1,
  key-two .tl_set:N = \l_mymodule_store_tl
}
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { mymodule }
{
  key-one = value one,
  key-two = value two
}
```

At a document level, `\keys_set:nn` is used within a document function, for example

```
\DeclareDocumentCommand \MyModuleSetup { m }
{ \keys_set:nn { mymodule } { #1 } }
\DeclareDocumentCommand \MyModuleMacro { o m }
{
  \group_begin:
    \keys_set:nn { mymodule } { #1 }
    % Main code for \MyModuleMacro
  \group_end:
}
```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`. As discussed in section 2, it is suggested that the character `/` is reserved for sub-division of keys into logical groups. Functions and variables are *not* expanded when creating key names, and so

```
\tl_set:Nn \l_mymodule_tmp_tl { key }
\keys_define:nn { mymodule }
{
  \l_mymodule_tmp_tl .code:n = code
}
```

creates a key called `\l_mymodule_tmp_tl`, and not one called `key`.

1 Creating keys

`\keys_define:nn`

Updated: 2017-11-14

`\keys_define:nn {<module>} {<keyval list>}`

Parses the *<keyval list>* and defines the keys listed there for *<module>*. The *<module>* name is treated as a string. In practice the *<module>* should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The *<keyval list>* should consist of one or more key names along with an associated key *property*. The properties of a key determine how it acts. The individual properties are described in the following text; a typical use of `\keys_define:nn` might read

```
\keys_define:nn { mymodule }
{
  keyname .code:n = Some-code~using~#1,
  keyname .value_required:n = true
}
```

where the properties of the key begin from the `.` after the key name.

The various properties available take either no arguments at all, or require one or more arguments. This is indicated in the name of the property using an argument specification. In the following discussion, each property is illustrated attached to an arbitrary *<key>*, which when used may be supplied with a *<value>*. All key *definitions* are local.

Key properties are applied in the reading order and so the ordering is significant. Key properties which define “actions”, such as `.code:n`, `.tl_set:N`, *etc.*, override one another. Some other properties are mutually exclusive, notably `.value_required:n` and `.value_forbidden:n`, and so they replace one another. However, properties covering non-exclusive behaviours may be given in any order. Thus for example the following definitions are equivalent.

```
\keys_define:nn { mymodule }
{
  keyname .code:n          = Some-code~using~#1,
  keyname .value_required:n = true
}
\keys_define:nn { mymodule }
{
```

```

    keyname .value_required:n = true,
    keyname .code:n           = Some~code~using~#1
}

```

Note that with the exception of the special `.undefine:` property, all key properties define the key within the current \TeX scope.

```

.bool_set:N
.bool_set:c
.bool_gset:N
.bool_gset:c

```

Updated: 2013-07-08

$\langle key \rangle$ `.bool_set:N` = $\langle boolean \rangle$

Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to $\langle value \rangle$ (which must be either `true` or `false`). If the variable does not exist, it will be created globally at the point that the key is set up.

```

.bool_set_inverse:N
.bool_set_inverse:c
.bool_gset_inverse:N
.bool_gset_inverse:c

```

New: 2011-08-28
Updated: 2013-07-08

$\langle key \rangle$ `.bool_set_inverse:N` = $\langle boolean \rangle$

Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to the logical inverse of $\langle value \rangle$ (which must be either `true` or `false`). If the $\langle boolean \rangle$ does not exist, it will be created globally at the point that the key is set up.

`.choice:`

$\langle key \rangle$ `.choice:`

Sets $\langle key \rangle$ to act as a choice key. Each valid choice for $\langle key \rangle$ must then be created, as discussed in section 3.

```

.choices:nn
.choices:(Vn|on|xn)

```

New: 2011-08-21
Updated: 2013-07-10

$\langle key \rangle$ `.choices:nn` = $\{\langle choices \rangle\} \{\langle code \rangle\}$

Sets $\langle key \rangle$ to act as a choice key, and defines a series $\langle choices \rangle$ which are implemented using the $\langle code \rangle$. Inside $\langle code \rangle$, `\l_keys_choice_tl` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of $\langle choices \rangle$ (indexed from 1). Choices are discussed in detail in section 3.

```

.clist_set:N
.clist_set:c
.clist_gset:N
.clist_gset:c

```

New: 2011-09-11

$\langle key \rangle$ `.clist_set:N` = $\langle comma list variable \rangle$

Defines $\langle key \rangle$ to set $\langle comma list variable \rangle$ to $\langle value \rangle$. Spaces around commas and empty items will be stripped. If the variable does not exist, it is created globally at the point that the key is set up.

```

.code:n

```

Updated: 2013-07-10

$\langle key \rangle$ `.code:n` = $\{\langle code \rangle\}$

Stores the $\langle code \rangle$ for execution when $\langle key \rangle$ is used. The $\langle code \rangle$ can include one parameter (`#1`), which will be the $\langle value \rangle$ given for the $\langle key \rangle$.

```

.cs_set:Np
.cs_set:cp
.cs_set_protected:Np
.cs_set_protected:cp
.cs_gset:Np
.cs_gset:cp
.cs_gset_protected:Np
.cs_gset_protected:cp

```

New: 2020-01-11

$\langle key \rangle$ `.cs_set:Np` = $\langle control sequence \rangle \langle arg. spec. \rangle$

Defines $\langle key \rangle$ to set $\langle control sequence \rangle$ to have $\langle arg. spec. \rangle$ and replacement text $\langle value \rangle$.

```
.default:n
.default:(V|o|x)
Updated: 2013-07-09
```

$\langle key \rangle$.default:n = { $\langle default \rangle$ }

Creates a $\langle default \rangle$ value for $\langle key \rangle$, which is used if no value is given. This will be used if only the key name is given, but not if a blank $\langle value \rangle$ is given:

```
\keys_define:nn { mymodule }
{
  key .code:n      = Hello~#1,
  key .default:n = World
}
\keys_set:nn { mymodule }
{
  key = Fred, % Prints 'Hello Fred'
  key,      % Prints 'Hello World'
  key = ,    % Prints 'Hello '
}
```

The default does not affect keys where values are required or forbidden. Thus a required value cannot be supplied by a default value, and giving a default value for a key which cannot take a value does not trigger an error.

```
.dim_set:N
.dim_set:c
.dim_gset:N
.dim_gset:c
Updated: 2020-01-17
```

$\langle key \rangle$.dim_set:N = $\langle dimension \rangle$

Defines $\langle key \rangle$ to set $\langle dimension \rangle$ to $\langle value \rangle$ (which must a dimension expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

```
.fp_set:N
.fp_set:c
.fp_gset:N
.fp_gset:c
Updated: 2020-01-17
```

$\langle key \rangle$.fp_set:N = $\langle floating point \rangle$

Defines $\langle key \rangle$ to set $\langle floating point \rangle$ to $\langle value \rangle$ (which must a floating point expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

```
.groups:n
New: 2013-07-14
```

$\langle key \rangle$.groups:n = { $\langle groups \rangle$ }

Defines $\langle key \rangle$ as belonging to the $\langle groups \rangle$ declared. Groups provide a “secondary axis” for selectively setting keys, and are described in Section 6.

```
.inherit:n
New: 2016-11-22
```

$\langle key \rangle$.inherit:n = { $\langle parents \rangle$ }

Specifies that the $\langle key \rangle$ path should inherit the keys listed as $\langle parents \rangle$. For example, after setting

```
\keys_define:nn { foo } { test .code:n = \tl_show:n {#1} }
\keys_define:nn { } { bar .inherit:n = foo }
```

setting

```
\keys_set:nn { bar } { test = a }
```

will be equivalent to

```
\keys_set:nn { foo } { test = a }
```

<hr/> <code>.initial:n</code> <hr/> <code>.initial:(V o x)</code> <hr/> Updated: 2013-07-09 <hr/>	$\langle key \rangle$ <code>.initial:n = {\langle value \rangle}</code> Initialises the $\langle key \rangle$ with the $\langle value \rangle$, equivalent to $\backslash keys_set:nn \{ \langle module \rangle \} \{ \langle key \rangle = \langle value \rangle \}$
<hr/> <code>.int_set:N</code> <hr/> <code>.int_set:c</code> <hr/> <code>.int_gset:N</code> <hr/> <code>.int_gset:c</code> <hr/> Updated: 2020-01-17 <hr/>	$\langle key \rangle$ <code>.int_set:N = \langle integer \rangle</code> Defines $\langle key \rangle$ to set $\langle integer \rangle$ to $\langle value \rangle$ (which must be an integer expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.
<hr/> <code>.meta:n</code> <hr/> Updated: 2013-07-10 <hr/>	$\langle key \rangle$ <code>.meta:n = {\langle keyval list \rangle}</code> Makes $\langle key \rangle$ a meta-key, which will set $\langle keyval list \rangle$ in one go. The $\langle keyval list \rangle$ can refer as #1 to the value given at the time the $\langle key \rangle$ is used (or, if no value is given, the $\langle key \rangle$'s default value).
<hr/> <code>.meta:nn</code> <hr/> New: 2013-07-10 <hr/>	$\langle key \rangle$ <code>.meta:nn = {\langle path \rangle} {\langle keyval list \rangle}</code> Makes $\langle key \rangle$ a meta-key, which will set $\langle keyval list \rangle$ in one go using the $\langle path \rangle$ in place of the current one. The $\langle keyval list \rangle$ can refer as #1 to the value given at the time the $\langle key \rangle$ is used (or, if no value is given, the $\langle key \rangle$'s default value).
<hr/> <code>.multichoice:</code> <hr/> New: 2011-08-21 <hr/>	$\langle key \rangle$ <code>.multichoice:</code> Sets $\langle key \rangle$ to act as a multiple choice key. Each valid choice for $\langle key \rangle$ must then be created, as discussed in section 3.
<hr/> <code>.multichoices:nn</code> <hr/> <code>.multichoices:(Vn on xn)</code> <hr/> New: 2011-08-21 Updated: 2013-07-10 <hr/>	$\langle key \rangle$ <code>.multichoices:nn {\langle choices \rangle} {\langle code \rangle}</code> Sets $\langle key \rangle$ to act as a multiple choice key, and defines a series $\langle choices \rangle$ which are implemented using the $\langle code \rangle$. Inside $\langle code \rangle$, <code>\l_keys_choice_tl</code> will be the name of the choice made, and <code>\l_keys_choice_int</code> will be the position of the choice in the list of $\langle choices \rangle$ (indexed from 1). Choices are discussed in detail in section 3.
<hr/> <code>.muskip_set:N</code> <hr/> <code>.muskip_set:c</code> <hr/> <code>.muskip_gset:N</code> <hr/> <code>.muskip_gset:c</code> <hr/> New: 2019-05-05 Updated: 2020-01-17 <hr/>	$\langle key \rangle$ <code>.muskip_set:N = \langle muskip \rangle</code> Defines $\langle key \rangle$ to set $\langle muskip \rangle$ to $\langle value \rangle$ (which must be a muskip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.
<hr/> <code>.prop_put:N</code> <hr/> <code>.prop_put:c</code> <hr/> <code>.prop_gput:N</code> <hr/> <code>.prop_gput:c</code> <hr/> New: 2019-01-31 <hr/>	$\langle key \rangle$ <code>.prop_put:N = \langle property list \rangle</code> Defines $\langle key \rangle$ to put the $\langle value \rangle$ onto the $\langle property list \rangle$ stored under the $\langle key \rangle$. If the variable does not exist, it is created globally at the point that the key is set up.

<code>.skip_set:N</code>	$\langle key \rangle$ <code>.skip_set:N = $\langle skip \rangle$</code>
--------------------------	--

<code>.skip_set:c</code>	Defines $\langle key \rangle$ to set $\langle skip \rangle$ to $\langle value \rangle$ (which must be a skip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.
<code>.skip_gset:N</code>	
<code>.skip_gset:c</code>	

Updated: 2020-01-17

<code>.tl_set:N</code>	$\langle key \rangle$ <code>.tl_set:N = $\langle token list variable \rangle$</code>
------------------------	---

<code>.tl_set:c</code>	Defines $\langle key \rangle$ to set $\langle token list variable \rangle$ to $\langle value \rangle$. If the variable does not exist, it is created globally at the point that the key is set up.
<code>.tl_gset:N</code>	
<code>.tl_gset:c</code>	

<code>.tl_set_x:N</code>	$\langle key \rangle$ <code>.tl_set_x:N = $\langle token list variable \rangle$</code>
--------------------------	---

<code>.tl_set_x:c</code>	Defines $\langle key \rangle$ to set $\langle token list variable \rangle$ to $\langle value \rangle$, which will be subjected to an x-type expansion (<i>i.e.</i> using <code>\tl_set:Nx</code>). If the variable does not exist, it is created globally at the point that the key is set up.
<code>.tl_gset_x:N</code>	
<code>.tl_gset_x:c</code>	

<code>.undefine:</code>	$\langle key \rangle$ <code>.undefine:</code>
-------------------------	---

New: 2015-07-14

Removes the definition of the $\langle key \rangle$ within the current scope.

<code>.value_forbidden:n</code>	$\langle key \rangle$ <code>.value_forbidden:n = true false</code>
---------------------------------	--

New: 2015-07-14

Specifies that $\langle key \rangle$ cannot receive a $\langle value \rangle$ when used. If a $\langle value \rangle$ is given then an error will be issued. Setting the property `false` cancels the restriction.

<code>.value_required:n</code>	$\langle key \rangle$ <code>.value_required:n = true false</code>
--------------------------------	---

New: 2015-07-14

Specifies that $\langle key \rangle$ must receive a $\langle value \rangle$ when used. If a $\langle value \rangle$ is not given then an error will be issued. Setting the property `false` cancels the restriction.

2 Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several sub-groups for a given module. This can be achieved either by adding a sub-division to the module name:

```
\keys_define:nn { mymodule / subgroup }
  { key .code:n = code }
```

or to the key name:

```
\keys_define:nn { mymodule }
  { subgroup / key .code:n = code }
```

As illustrated, the best choice of token for sub-dividing keys in this way is `/`. This is because of the method that is used to represent keys internally. Both of the above code fragments set the same key, which has full name `mymodule/subgroup/key`.

As illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

3 Choice and multiple choice keys

The `l3keys` system supports two types of choice key, in which a series of pre-defined input values are linked to varying implementations. Choice keys are usually created so that the various values are mutually-exclusive: only one can apply at any one time. “Multiple” choice keys are also supported: these allow a selection of values to be chosen at the same time.

Mutually-exclusive choices are created by setting the `.choice:` property:

```
\keys_define:nn { mymodule }
  { key .choice: }
```

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependant only on the name of the choice or the position of the choice in the list of all possibilities. Here, the keys can share the same code, and can be rapidly created using the `.choices:nn` property.

```
\keys_define:nn { mymodule }
{
  key .choices:nn =
    { choice-a, choice-b, choice-c }
    {
      You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
      which~is~in~position~\int_use:N \l_keys_choice_int \c_space_tl
      in~the~list.
    }
}
```

The index `\l_keys_choice_int` in the list of choices starts at 1.

`\l_keys_choice_int`
`\l_keys_choice_tl`

Inside the code block for a choice generated using `.choices:nn`, the variables `\l_keys_choice_tl` and `\l_keys_choice_int` are available to indicate the name of the current choice, and its position in the comma list. The position is indexed from 1. Note that, as with standard key code generated using `.code:n`, the value passed to the key (i.e. the choice name) is also available as `#1`.

On the other hand, it is sometimes useful to create choices which use entirely different code from one another. This can be achieved by setting the `.choice:` property of a key, then manually defining sub-keys.

```
\keys_define:nn { mymodule }
{
  key .choice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
}
```

It is possible to mix the two methods, but manually-created choices should *not* use `\l_keys_choice_tl` or `\l_keys_choice_int`. These variables do not have defined

behaviour when used outside of code created using `.choices:nn` (*i.e.* anything might happen).

It is possible to allow choice keys to take values which have not previously been defined by adding code for the special `unknown` choice. The general behavior of the `unknown` key is described in Section 5. A typical example in the case of a choice would be to issue a custom error message:

```
\keys_define:nn { mymodule }
{
  key .choice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
  key / unknown .code:n =
    \msg_error:nnxxx { mymodule } { unknown-choice }
    { key } % Name of choice key
    { choice-a , choice-b , choice-c } % Valid choices
    { \exp_not:n {#1} } % Invalid choice given
  %
  %
}
```

Multiple choices are created in a very similar manner to mutually-exclusive choices, using the properties `.multichoice:` and `.multichoices:nn`. As with mutually exclusive choices, multiple choices are define as sub-keys. Thus both

```
\keys_define:nn { mymodule }
{
  key .multichoices:nn =
    { choice-a, choice-b, choice-c }
    {
      You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
      which~is~in~position~
      \int_use:N \l_keys_choice_int \c_space_tl
      in~the~list.
    }
}
```

and

```
\keys_define:nn { mymodule }
{
  key .multichoice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
}
```

are valid.

When a multiple choice key is set

```
\keys_set:nn { mymodule }
{
  key = { a , b , c } % 'key' defined as a multiple choice
}
```

each choice is applied in turn, equivalent to a `clist` mapping or to applying each value individually:

```
\keys_set:nn { mymodule }
{
  key = a ,
  key = b ,
  key = c ,
}
```

Thus each separate choice will have passed to it the `\l_keys_choice_tl` and `\l_keys_choice_int` in exactly the same way as described for `.choices:nn`.

4 Setting keys

```
\keys_set:nn
\keys_set:(nV|nv|no)
```

Updated: 2017-11-14

```
\keys_set:nn {<module>} {<keyval list>}
```

Parses the `<keyval list>`, and sets those keys which are defined for `<module>`. The behaviour on finding an unknown key can be set by defining a special `unknown` key: this is illustrated later.

```
\l_keys_key_str
\l_keys_path_str
\l_keys_value_tl
```

Updated: 2020-02-08

For each key processed, information of the full *path* of the key, the *name* of the key and the *value* of the key is available within three token list variables. These may be used within the code of the key.

The *value* is everything after the `=`, which may be empty if no value was given. This is stored in `\l_keys_value_tl`, and is not processed in any way by `\keys_set:nn`.

The *path* of the key is a “full” description of the key, and is unique for each key. It consists of the module and full key name, thus for example

```
\keys_set:nn { mymodule } { key-a = some-value }
```

has path `mymodule/key-a` while

```
\keys_set:nn { mymodule } { subset / key-a = some-value }
```

has path `mymodule/subset/key-a`. This information is stored in `\l_keys_path_str`.

The *name* of the key is the part of the path after the last `/`, and thus is not unique. In the preceding examples, both keys have name `key-a` despite having different paths. This information is stored in `\l_keys_key_str`.

5 Handling of unknown keys

If a key has not previously been defined (is unknown), `\keys_set:nn` looks for a special `unknown` key for the same module, and if this is not defined raises an error indicating that

the key name was unknown. This mechanism can be used for example to issue custom error texts.

```
\keys_define:nn { mymodule }
{
  unknown .code:n =
    You~tried~to~set~key~'\l_keys_key_str'~to~'#1'.
}
```

<code>\keys_set_known:nn</code>	<code>\keys_set_known:nn {<module>} {<keyval list>}</code>
<code>\keys_set_known:(nV nv no)</code>	<code>\keys_set_known:nnN {<module>} {<keyval list>} <tl></code>
<code>\keys_set_known:nnN</code>	<code>\keys_set_known:nnnN {<module>} {<keyval list>} {<root>} <tl></code>
<code>\keys_set_known:(nVN nvN noN)</code>	
<code>\keys_set_known:nnnN</code>	
<code>\keys_set_known:(nVnN nvN nonN)</code>	

New: 2011-08-23

Updated: 2019-01-29

These functions set keys which are known for the `<module>`, and simply ignore other keys. The `\keys_set_known:nn` function parses the `<keyval list>`, and sets those keys which are defined for `<module>`. Any keys which are unknown are not processed further by the parser. In addition, `\keys_set_known:nnN` stores the key–value pairs in the `<tl>` in comma-separated form (*i.e.* an edited version of the `<keyval list>`). When a `<root>` is given (`\keys_set_known:nnnN`), the key–value entries are returned relative to this point in the key tree. When it is absent, only the key name and value are provided. The correct list is returned by nested calls.

6 Selective key setting

In some cases it may be useful to be able to select only some keys for setting, even though these keys have the same path. For example, with a set of keys defined using

```
\keys_define:nn { mymodule }
{
  key-one .code:n = { \my_func:n {#1} } ,
  key-two .tl_set:N = \l_my_a_tl ,
  key-three .tl_set:N = \l_my_b_tl ,
  key-four .fp_set:N = \l_my_a_fp ,
}
```

the use of `\keys_set:nn` attempts to set all four keys. However, in some contexts it may only be sensible to set some keys, or to control the order of setting. To do this, keys may be assigned to *groups*: arbitrary sets which are independent of the key tree. Thus modifying the example to read

```
\keys_define:nn { mymodule }
{
  key-one .code:n = { \my_func:n {#1} } ,
  key-one .groups:n = { first } ,
  key-two .tl_set:N = \l_my_a_tl ,
}
```

```

key-two    .groups:n = { first }           ,
key-three  .tl_set:N = \l_my_b_tl          ,
key-three  .groups:n = { second }          ,
key-four   .fp_set:N = \l_my_a_fp          ,
}

```

assigns `key-one` and `key-two` to group `first`, `key-three` to group `second`, while `key-four` is not assigned to a group.

Selective key setting may be achieved either by selecting one or more groups to be made “active”, or by marking one or more groups to be ignored in key setting.

<code>\keys_set_filter:nnn</code>	<code>\keys_set_filter:nnn {<module>} {<groups>} {<keyval list>}</code>
<code>\keys_set_filter:(nnV nnv nno)</code>	<code>\keys_set_filter:nnnN {<module>} {<groups>} {<keyval list>} <tl></code>
<code>\keys_set_filter:nnnN</code>	<code>\keys_set_filter:nnnnN {<module>} {<groups>} {<keyval list>} <root></code>
<code>\keys_set_filter:(nnVN nnvN nnoN)</code>	<code><tl></code>
<code>\keys_set_filter:nnnnN</code>	
<code>\keys_set_filter:(nnVnN nnvnN nnonN)</code>	

New: 2013-07-14

Updated: 2019-01-29

Activates key filtering in an “opt-out” sense: keys assigned to any of the `<groups>` specified are ignored. The `<groups>` are given as a comma-separated list. Unknown keys are not assigned to any group and are thus always set. The key–value pairs for each key which is filtered out are stored in the `<tl>` in a comma-separated form (*i.e.* an edited version of the `<keyval list>`). The `\keys_set_filter:nnn` version skips this stage.

Use of `\keys_set_filter:nnnN` can be nested, with the correct residual `<keyval list>` returned at each stage. In the version which takes a `<root>` argument, the key list is returned relative to that point in the key tree. In the cases without a `<root>` argument, only the key names and values are returned.

<code>\keys_set_groups:nnn</code>	<code>\keys_set_groups:nnn {<module>} {<groups>} {<keyval list>}</code>
<code>\keys_set_groups:(nnV nnv nno)</code>	

New: 2013-07-14

Updated: 2017-05-27

Activates key filtering in an “opt-in” sense: only keys assigned to one or more of the `<groups>` specified are set. The `<groups>` are given as a comma-separated list. Unknown keys are not assigned to any group and are thus never set.

7 Utility functions for keys

<code>\keys_if_exist_p:nn *</code>	<code>\keys_if_exist_p:nn {<module>} {<key>}</code>
<code>\keys_if_exist:nnTF *</code>	<code>\keys_if_exist:nnTF {<module>} {<key>} {<true code>} {<false code>}</code>

Updated: 2017-11-14

Tests if the `<key>` exists for `<module>`, *i.e.* if any code has been defined for `<key>`.

<code>\keys_if_choice_exist_p:nnn</code>	<code>*</code>	<code>\keys_if_choice_exist_p:nnn {<module>} {<key>} {<choice>}</code>
<code>\keys_if_choice_exist:nnnTF</code>	<code>*</code>	<code>\keys_if_choice_exist:nnnTF {<module>} {<key>} {<choice>} {<true code>}</code>
		<code>{<false code>}</code>

New: 2011-08-21

Updated: 2017-11-14

Tests if the $\langle choice \rangle$ is defined for the $\langle key \rangle$ within the $\langle module \rangle$, *i.e.* if any code has been defined for $\langle key \rangle / \langle choice \rangle$. The test is **false** if the $\langle key \rangle$ itself is not defined.

<code>\keys_show:nn</code>	<code>\keys_show:nn {<module>} {<key>}</code>
----------------------------	---

Updated: 2015-08-09

Displays in the terminal the information associated to the $\langle key \rangle$ for a $\langle module \rangle$, including the function which is used to actually implement it.

<code>\keys_log:nn</code>	<code>\keys_log:nn {<module>} {<key>}</code>
---------------------------	--

New: 2014-08-22

Updated: 2015-08-09

Writes in the log file the information associated to the $\langle key \rangle$ for a $\langle module \rangle$. See also `\keys_show:nn` which displays the result in the terminal.

8 Low-level interface for parsing key–val lists

To re-cap from earlier, a key–value list is input of the form

```
KeyOne = ValueOne ,
KeyTwo = ValueTwo ,
KeyThree
```

where each key–value pair is separated by a comma from the rest of the list, and each key–value pair does not necessarily contain an equals sign or a value! Processing this type of input correctly requires a number of careful steps, to correctly account for braces, spaces and the category codes of separators.

While the functions described earlier are used as a high-level interface for processing such input, in special circumstances you may wish to use a lower-level approach. The low-level parsing system converts a $\langle key\text{--}value\text{ list} \rangle$ into $\langle keys \rangle$ and associated $\langle values \rangle$. After the parsing phase is completed, the resulting keys and values (or keys alone) are available for further processing. This processing is not carried out by the low-level parser itself, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (it receives two arguments), and a second function is required for keys given without any value (it is called with a single argument).

The parser does not double `#` tokens or expand any input. Active tokens `=` and `,` appearing at the outer level of braces are converted to category “other” (12) so that the parser does not “miss” any due to category code changes. Spaces are removed from the ends of the keys and values. Keys and values which are given in braces have exactly one set removed (after space trimming), thus

```
key = {value here},
```

and

```
key = value here,
```

are treated identically.

`\keyval_parse:NNn` ★

Updated: 2020-02-20

`\keyval_parse:NNn` $\langle function_1 \rangle$ $\langle function_2 \rangle$ $\{ \langle key-value list \rangle \}$

Parses the $\langle key-value list \rangle$ into a series of $\langle keys \rangle$ and associated $\langle values \rangle$, or keys alone (if no $\langle value \rangle$ was given). $\langle function_1 \rangle$ should take one argument, while $\langle function_2 \rangle$ should absorb two arguments. After `\keyval_parse:NNn` has parsed the $\langle key-value list \rangle$, $\langle function_1 \rangle$ is used to process keys given with no value and $\langle function_2 \rangle$ is used to process keys given with a value. The order of the $\langle keys \rangle$ in the $\langle key-value list \rangle$ is preserved. Thus

```
\keyval_parse:NNn \function:n \function:nn
{ key1 = value1 , key2 = value2, key3 = , key4 }
```

is converted into an input stream

```
\function:nn { key1 } { value1 }
\function:nn { key2 } { value2 }
\function:nn { key3 } { }
\function:n { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the $\langle key \rangle$ and $\langle value \rangle$, then one *outer* set of braces is removed from the $\langle key \rangle$ and $\langle value \rangle$ as part of the processing.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the converted input stream does not expand further when appearing in an **x**-type or **e**-type argument expansion.

Part XXII

The l3intarray package: fast global integer arrays

1 l3intarray documentation

For applications requiring heavy use of integers, this module provides arrays which can be accessed in constant time (contrast l3seq, where access time is linear). These arrays have several important features

- The size of the array is fixed and must be given at point of initialisation
- The absolute value of each entry has maximum $2^{30} - 1$ (*i.e.* one power lower than the usual `\c_max_int` ceiling of $2^{31} - 1$)

The use of `intarray` data is therefore recommended for cases where the need for fast access is of paramount importance.

`\intarray_new:Nn`
`\intarray_new:cn`

New: 2018-03-29

`\intarray_new:Nn` $\langle\text{intarray var}\rangle$ $\{\langle\text{size}\rangle\}$

Evaluates the integer expression $\langle\text{size}\rangle$ and allocates an $\langle\text{integer array variable}\rangle$ with that number of (zero) entries. The variable name should start with `\g_` because assignments are always global.

`\intarray_count:N *`
`\intarray_count:c *`

New: 2018-03-29

`\intarray_count:N` $\langle\text{intarray var}\rangle$

Expands to the number of entries in the $\langle\text{integer array variable}\rangle$. Contrarily to `\seq-count:N` this is performed in constant time.

`\intarray_gset:Nnn`
`\intarray_gset:cnn`

New: 2018-03-29

`\intarray_gset:Nnn` $\langle\text{intarray var}\rangle$ $\{\langle\text{position}\rangle\}$ $\{\langle\text{value}\rangle\}$

Stores the result of evaluating the integer expression $\langle\text{value}\rangle$ into the $\langle\text{integer array variable}\rangle$ at the (integer expression) $\langle\text{position}\rangle$. If the $\langle\text{position}\rangle$ is not between 1 and the `\intarray_count:N`, or the $\langle\text{value}\rangle$'s absolute value is bigger than $2^{30} - 1$, an error occurs. Assignments are always global.

`\intarray_const_from_clist:Nn`
`\intarray_const_from_clist:cn`

New: 2018-05-04

`\intarray_const_from_clist:Nn` $\langle\text{intarray var}\rangle$ $\langle\text{intexpr clist}\rangle$

Creates a new constant $\langle\text{integer array variable}\rangle$ or raises an error if the name is already taken. The $\langle\text{integer array variable}\rangle$ is set (globally) to contain as its items the results of evaluating each $\langle\text{integer expression}\rangle$ in the $\langle\text{comma list}\rangle$.

`\intarray_gzero:N`
`\intarray_gzero:c`

New: 2018-05-04

`\intarray_gzero:N` $\langle\text{intarray var}\rangle$

Sets all entries of the $\langle\text{integer array variable}\rangle$ to zero. Assignments are always global.

<hr/>	
<code>\intarray_item:Nn</code> *	<code>\intarray_item:Nn <intarray var> {<position>}</code>
<code>\intarray_item:cn</code> *	Expands to the integer entry stored at the (integer expression) <i><position></i> in the <i><integer array variable></i> . If the <i><position></i> is not between 1 and the <code>\intarray_count:N</code> , an error occurs.
<hr/>	
New: 2018-03-29	
<hr/>	
<code>\intarray_rand_item:N</code> *	<code>\intarray_rand_item:N <intarray var></code>
<code>\intarray_rand_item:c</code> *	Selects a pseudo-random item of the <i><integer array></i> . If the <i><integer array></i> is empty, produce an error.
<hr/>	
New: 2018-05-05	
<hr/>	
<code>\intarray_show:N</code>	<code>\intarray_show:N <intarray var></code>
<code>\intarray_show:c</code>	<code>\intarray_log:N <intarray var></code>
<code>\intarray_log:N</code>	Displays the items in the <i><integer array variable></i> in the terminal or writes them in the log file.
<code>\intarray_log:c</code>	
<hr/>	
New: 2018-05-04	
<hr/>	

1.1 Implementation notes

It is a wrapper around the `\fontdimen` primitive, used to store arrays of integers (with a restricted range: absolute value at most $2^{30} - 1$). In contrast to `l3seq` sequences the access to individual entries is done in constant time rather than linear time, but only integers can be stored. More precisely, the primitive `\fontdimen` stores dimensions but the `l3intarray` package transparently converts these from/to integers. Assignments are always global.

While LuaTeX’s memory is extensible, other engines can “only” deal with a bit less than 4×10^6 entries in all `\fontdimen` arrays combined (with default TeXLive settings).

Part XXIII

The l3fp package: Floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. Floating point expressions support the following operations with their usual precedence.

- Basic arithmetic: addition $x + y$, subtraction $x - y$, multiplication $x * y$, division x / y , square root \sqrt{x} , and parentheses.
 - Comparison operators: $x < y$, $x \leq y$, $x > y$, $x \neq y$ etc.
 - Boolean logic: sign $\text{sign } x$, negation $!x$, conjunction $x \&\& y$, disjunction $x || y$, ternary operator $x ? y : z$.
 - Exponentials: $\exp x$, $\ln x$, x^y , $\log b x$.
 - Integer factorial: $\text{fact } x$.
 - Trigonometry: $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\sec x$, $\csc x$ expecting their arguments in radians, and $\text{sind } x$, $\text{cosd } x$, $\text{tand } x$, $\text{cotd } x$, $\text{secd } x$, $\text{cscd } x$ expecting their arguments in degrees.
 - Inverse trigonometric functions: $\text{asin } x$, $\text{acos } x$, $\text{atan } x$, $\text{acot } x$, $\text{asec } x$, $\text{acsc } x$ giving a result in radians, and $\text{asind } x$, $\text{acosd } x$, $\text{atand } x$, $\text{acotd } x$, $\text{asecd } x$, $\text{acscd } x$ giving a result in degrees.
- (not yet) Hyperbolic functions and their inverse functions: $\sinh x$, $\cosh x$, $\tanh x$, $\coth x$, $\text{sech } x$, $\text{csch } x$, and $\text{asinh } x$, $\text{acosh } x$, $\text{atanh } x$, $\text{acoth } x$, $\text{asech } x$, $\text{acsch } x$.
- Extrema: $\max(x_1, x_2, \dots)$, $\min(x_1, x_2, \dots)$, $\text{abs}(x)$.
 - Rounding functions, controlled by two optional values, n (number of places, 0 by default) and t (behavior on a tie, NaN by default):
 - $\text{trunc}(x, n)$ rounds towards zero,
 - $\text{floor}(x, n)$ rounds towards $-\infty$,
 - $\text{ceil}(x, n)$ rounds towards $+\infty$,
 - $\text{round}(x, n, t)$ rounds to the closest value, with ties rounded to an even value by default, towards zero if $t = 0$, towards $+\infty$ if $t > 0$ and towards $-\infty$ if $t < 0$.

And (not yet) modulo, and “quantize”.

- Random numbers: $\text{rand}()$, $\text{randint}(m, n)$.
- Constants: pi , deg (one degree in radians).
- Dimensions, automatically expressed in points, e.g., pc is 12.

- Automatic conversion (no need for `\langle type \rangle_use:N`) of integer, dimension, and skip variables to floating point numbers, expressing dimensions in points and ignoring the stretch and shrink components of skips.
- Tuples: (x_1, \dots, x_n) that can be stored in variables, added together, multiplied or divided by a floating point number, and nested.

Floating point numbers can be given either explicitly (in a form such as $1.234\text{e-}34$, or $-.0001$), or as a stored floating point variable, which is automatically replaced by its current value. A “floating point” is a floating point number or a tuple thereof. See section 9.1 for a description of what a floating point is, section 9.2 for details about how an expression is parsed, and section 9.3 to know what the various operations do. Some operations may raise exceptions (error messages), described in section 7.

An example of use could be the following.

```
\LaTeX{} can now compute: $ \frac{\sin (3.5)}{2} + 2\cdot 10^{-3}
= \ExplSyntaxOn \fp_to_decimal:n {\sin(3.5)/2 + 2e-3} $.
```

The operation `round` can be used to limit the result’s precision. Adding `+0` avoids the possibly undesirable output `-0`, replacing it by `+0`. However, the `l3fp` module is mostly meant as an underlying tool for higher-level commands. For example, one could provide a function to typeset nicely the result of floating point computations.

```
\documentclass{article}
\usepackage{xparse, siunitx}
\ExplSyntaxOn
\NewDocumentCommand { \calcnun } { m }
{ \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\begin{document}
\calcnun { 2 pi * sin ( 2.3 ^ 5 ) }
\end{document}
```

See the documentation of `siunitx` for various options of `\num`.

1 Creating and initialising floating point variables

<hr/> <code>\fp_new:N</code> <hr/> <code>\fp_new:c</code> <hr/> <small>Updated: 2012-05-08</small>	<code>\fp_new:N <fp var></code> Creates a new <code><fp var></code> or raises an error if the name is already taken. The declaration is global. The <code><fp var></code> is initially <code>+0</code> .
<hr/> <code>\fp_const:Nn</code> <hr/> <code>\fp_const:cn</code> <hr/> <small>Updated: 2012-05-08</small>	<code>\fp_const:Nn <fp var> {<floating point expression>}</code> Creates a new constant <code><fp var></code> or raises an error if the name is already taken. The <code><fp var></code> is set globally equal to the result of evaluating the <code><floating point expression></code> .
<hr/> <code>\fp_zero:N</code> <hr/> <code>\fp_zero:c</code> <hr/> <code>\fp_gzero:N</code> <hr/> <code>\fp_gzero:c</code> <hr/> <small>Updated: 2012-05-08</small>	<code>\fp_zero:N <fp var></code> Sets the <code><fp var></code> to <code>+0</code> .

```

\fp_zero_new:N
\fp_zero_new:c
\fp_gzero_new:N
\fp_gzero_new:c

```

Updated: 2012-05-08

```
\fp_zero_new:N <fp var>
```

Ensures that the $\langle fp\ var \rangle$ exists globally by applying $\backslash fp_new:N$ if necessary, then applies $\backslash fp_(\mathit{g})zero:N$ to leave the $\langle fp\ var \rangle$ set to +0.

2 Setting floating point variables

```

\fp_set:Nn
\fp_set:cn
\fp_gset:Nn
\fp_gset:cn

```

Updated: 2012-05-08

```
\fp_set:Nn <fp var> {<floating point expression>}
```

Sets $\langle fp\ var \rangle$ equal to the result of computing the $\langle floating\ point\ expression \rangle$.

```

\fp_set_eq:Nn
\fp_set_eq:(cN|Nc|cc)
\fp_gset_eq:Nn
\fp_gset_eq:(cN|Nc|cc)

```

Updated: 2012-05-08

```
\fp_set_eq:Nn <fp var1> <fp var2>
```

Sets the floating point variable $\langle fp\ var_1 \rangle$ equal to the current value of $\langle fp\ var_2 \rangle$.

```

\fp_add:Nn
\fp_add:cn
\fp_gadd:Nn
\fp_gadd:cn

```

Updated: 2012-05-08

```
\fp_add:Nn <fp var> {<floating point expression>}
```

Adds the result of computing the $\langle floating\ point\ expression \rangle$ to the $\langle fp\ var \rangle$. This also applies if $\langle fp\ var \rangle$ and $\langle floating\ point\ expression \rangle$ evaluate to tuples of the same size.

```

\fp_sub:Nn
\fp_sub:cn
\fp_gsub:Nn
\fp_gsub:cn

```

Updated: 2012-05-08

```
\fp_sub:Nn <fp var> {<floating point expression>}
```

Subtracts the result of computing the $\langle floating\ point\ expression \rangle$ from the $\langle fp\ var \rangle$. This also applies if $\langle fp\ var \rangle$ and $\langle floating\ point\ expression \rangle$ evaluate to tuples of the same size.

3 Using floating points

```

\fp_eval:n  ★

```

New: 2012-05-08
Updated: 2012-07-08

```
\fp_eval:n {<floating point expression>}
```

Evaluates the $\langle floating\ point\ expression \rangle$ and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and NaN trigger an “invalid operation” exception. For a tuple, each item is converted using $\backslash fp_eval:n$ and they are combined as $(\langle fp_1 \rangle, \sqcup \langle fp_2 \rangle, \sqcup \dots \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items. This function is identical to $\backslash fp_to_decimal:n$.

<code>\fp_sign:n</code> *	<code>\fp_sign:n {<fpexpr>}</code>
---------------------------	--

New: 2018-11-03

Evaluates the $\langle fpexpr \rangle$ and leaves its sign in the input stream using `\fp_eval:n {sign(<result>)}`: +1 for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ± 0 for ± 0 . If the operand is a tuple or is NaN, then “invalid operation” occurs and the result is 0.

<code>\fp_to_decimal:N</code> *	<code>\fp_to_decimal:N <fp var></code>
<code>\fp_to_decimal:c</code> *	<code>\fp_to_decimal:n {<floating point expression>}</code>
<code>\fp_to_decimal:n</code> *	

New: 2012-05-08

Updated: 2012-07-08

Evaluates the $\langle floating point expression \rangle$ and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and NaN trigger an “invalid operation” exception. For a tuple, each item is converted using `\fp_to_decimal:n` and they are combined as $\langle fp_1 \rangle, \sqcup \langle fp_2 \rangle, \sqcup \dots \langle fp_n \rangle$ if $n > 1$ and $\langle fp_1 \rangle,)$ or $()$ for fewer items.

<code>\fp_to_dim:N</code> *	<code>\fp_to_dim:N <fp var></code>
<code>\fp_to_dim:c</code> *	<code>\fp_to_dim:n {<floating point expression>}</code>
<code>\fp_to_dim:n</code> *	

Updated: 2016-03-22

Evaluates the $\langle floating point expression \rangle$ and expresses the result as a dimension (in pt) suitable for use in dimension expressions. The output is identical to `\fp_to_decimal:n`, with an additional trailing pt (both letter tokens). In particular, the result may be outside the range $[-2^{14} + 2^{-17}, 2^{14} - 2^{-17}]$ of valid T_EX dimensions, leading to overflow errors if used as a dimension. Tuples, as well as the values $\pm\infty$ and NaN, trigger an “invalid operation” exception.

<code>\fp_to_int:N</code> *	<code>\fp_to_int:N <fp var></code>
<code>\fp_to_int:c</code> *	<code>\fp_to_int:n {<floating point expression>}</code>
<code>\fp_to_int:n</code> *	

Updated: 2012-07-08

Evaluates the $\langle floating point expression \rangle$, and rounds the result to the closest integer, rounding exact ties to an even integer. The result may be outside the range $[-2^{31} + 1, 2^{31} - 1]$ of valid T_EX integers, leading to overflow errors if used in an integer expression. Tuples, as well as the values $\pm\infty$ and NaN, trigger an “invalid operation” exception.

<code>\fp_to_scientific:N</code> *	<code>\fp_to_scientific:N <fp var></code>
<code>\fp_to_scientific:c</code> *	<code>\fp_to_scientific:n {<floating point expression>}</code>
<code>\fp_to_scientific:n</code> *	

New: 2012-05-08

Updated: 2016-03-22

Evaluates the $\langle floating point expression \rangle$ and expresses the result in scientific notation:

$\langle optional - \rangle \langle digit \rangle . \langle 15 digits \rangle e \langle optional sign \rangle \langle exponent \rangle$

The leading $\langle digit \rangle$ is non-zero except in the case of ± 0 . The values $\pm\infty$ and NaN trigger an “invalid operation” exception. Normal category codes apply: thus the `e` is category code 11 (a letter). For a tuple, each item is converted using `\fp_to_scientific:n` and they are combined as $\langle fp_1 \rangle, \sqcup \langle fp_2 \rangle, \sqcup \dots \langle fp_n \rangle$ if $n > 1$ and $\langle fp_1 \rangle,)$ or $()$ for fewer items.

<code>\fp_to_tl:N</code>	★	<code>\fp_to_tl:N <fp var></code>
<code>\fp_to_tl:c</code>	★	<code>\fp_to_tl:n {\floating point expression}</code>
<code>\fp_to_tl:n</code>	★	Evaluates the <i><floating point expression></i> and expresses the result in (almost) the shortest possible form. Numbers in the ranges $(0, 10^{-3})$ and $[10^{16}, \infty)$ are expressed in scientific notation with trailing zeros trimmed and no decimal separator when there is a single significant digit (this differs from <code>\fp_to_scientific:n</code>). Numbers in the range $[10^{-3}, 10^{16})$ are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see <code>\fp_to_decimal:n</code>). Negative numbers start with <code>-</code> . The special values ± 0 , $\pm\infty$ and NaN are rendered as <code>0</code> , <code>-0</code> , <code>inf</code> , <code>-inf</code> , and <code>nan</code> respectively. Normal category codes apply and thus <code>inf</code> or <code>nan</code> , if produced, are made up of letters. For a tuple, each item is converted using <code>\fp_to_tl:n</code> and they are combined as $\langle\langle fp_1 \rangle, \sqcup \langle fp_2 \rangle, \sqcup \dots \langle fp_n \rangle\rangle$ if $n > 1$ and $\langle\langle fp_1 \rangle, \rangle$ or $\langle \rangle$ for fewer items.

Updated: 2016-03-22

<code>\fp_use:N</code>	★	<code>\fp_use:N <fp var></code>
<code>\fp_use:c</code>	★	Inserts the value of the <i><fp var></i> into the input stream as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values $\pm\infty$ and NaN trigger an “invalid operation” exception. For a tuple, each item is converted using <code>\fp_to_decimal:n</code> and they are combined as $\langle\langle fp_1 \rangle, \sqcup \langle fp_2 \rangle, \sqcup \dots \langle fp_n \rangle\rangle$ if $n > 1$ and $\langle\langle fp_1 \rangle, \rangle$ or $\langle \rangle$ for fewer items. This function is identical to <code>\fp_to_decimal:N</code> .

Updated: 2012-07-08

4 Floating point conditionals

<code>\fp_if_exist_p:N</code>	★	<code>\fp_if_exist_p:N <fp var></code>
<code>\fp_if_exist_p:c</code>	★	<code>\fp_if_exist:N\TF <fp var> {\true code} {\false code}</code>
<code>\fp_if_exist:N\TF</code>	★	Tests whether the <i><fp var></i> is currently defined. This does not check that the <i><fp var></i> really is a floating point variable.
<code>\fp_if_exist:c\TF</code>	★	

Updated: 2012-05-08

<code>\fp_compare_p:nNn</code>	★	<code>\fp_compare_p:nNn {\fpexpr₁} <relation> {\fpexpr₂}</code>
<code>\fp_compare:nNn\TF</code>	★	<code>\fp_compare:nNn\TF {\fpexpr₁} <relation> {\fpexpr₂} {\true code} {\false code}</code>

Updated: 2012-05-08

Compares the *<fpexpr₁>* and the *<fpexpr₂>*, and returns `true` if the *<relation>* is obeyed. Two floating points x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or $x?y$ (“not ordered”). The last case occurs exactly if one or both operands is NaN or is a tuple, unless they are equal tuples. Note that a NaN is distinct from any value, even another NaN, hence $x = x$ is not true for a NaN. To test if a value is NaN, compare it to an arbitrary number with the “not ordered” relation.

```

\fp_compare:nNn\TF { <value> } ? { 0 }
{ } % <value> is nan
{ } % <value> is not nan

```

Tuples are equal if they have the same number of items and items compare equal (in particular there must be no NaN). At present any other comparison with tuples yields ? (not ordered). This is experimental.

This function is less flexible than `\fp_compare:n\TF` but slightly faster. It is provided for consistency with `\int_compare:nNn\TF` and `\dim_compare:nNn\TF`.

<code>\fp_compare_p:n</code> ☆	<code>\fp_compare_p:n</code>
<code>\fp_compare:nTF</code> ☆	{
Updated: 2013-12-14	$\langle fpexpr_1 \rangle$ $\langle relation_1 \rangle$
	...
	$\langle fpexpr_N \rangle$ $\langle relation_N \rangle$
	$\langle fpexpr_{N+1} \rangle$
	}
	<code>\fp_compare:nTF</code>
	{
	$\langle fpexpr_1 \rangle$ $\langle relation_1 \rangle$
	...
	$\langle fpexpr_N \rangle$ $\langle relation_N \rangle$
	$\langle fpexpr_{N+1} \rangle$
	}
	{ $\langle true\ code \rangle$ } { $\langle false\ code \rangle$ }

Evaluates the $\langle floating\ point\ expressions \rangle$ as described for `\fp_eval:n` and compares consecutive result using the corresponding $\langle relation \rangle$, namely it compares $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$ using the $\langle relation_1 \rangle$, then $\langle fpexpr_2 \rangle$ and $\langle fpexpr_3 \rangle$ using the $\langle relation_2 \rangle$, until finally comparing $\langle fpexpr_N \rangle$ and $\langle fpexpr_{N+1} \rangle$ using the $\langle relation_N \rangle$. The test yields **true** if all comparisons are **true**. Each $\langle floating\ point\ expression \rangle$ is evaluated only once. Contrarily to `\int_compare:nTF`, all $\langle floating\ point\ expressions \rangle$ are computed, even if one comparison is **false**. Two floating points x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or $x?y$ (“not ordered”). The last case occurs exactly if one or both operands is NaN or is a tuple, unless they are equal tuples. Each $\langle relation \rangle$ can be any (non-empty) combination of $<$, $=$, $>$, and $?$, plus an optional leading $!$ (which negates the $\langle relation \rangle$), with the restriction that the $\langle relation \rangle$ may not start with $?$, as this symbol has a different meaning (in combination with $:$) within floating point expressions. The comparison $x \langle relation \rangle y$ is then **true** if the $\langle relation \rangle$ does not start with $!$ and the actual relation ($<$, $=$, $>$, or $?$) between x and y appears within the $\langle relation \rangle$, or on the contrary if the $\langle relation \rangle$ starts with $!$ and the relation between x and y does not appear within the $\langle relation \rangle$. Common choices of $\langle relation \rangle$ include $>=$ (greater or equal), $!=$ (not equal), $!?$ or $<=>$ (comparable).

This function is more flexible than `\fp_compare:nNnTF` and only slightly slower.

5 Floating point expression loops

<code>\fp_do_until:nNnn</code> ☆	<code>\fp_do_until:nNnn {$\langle fpexpr_1 \rangle$} $\langle relation \rangle$ {$\langle fpexpr_2 \rangle$} {$\langle code \rangle$}</code>
New: 2012-08-16	Places the $\langle code \rangle$ in the input stream for \TeX to process, and then evaluates the relationship between the two $\langle floating\ point\ expressions \rangle$ as described for <code>\fp_compare:nNnTF</code> . If the test is false then the $\langle code \rangle$ is inserted into the input stream again and a loop occurs until the $\langle relation \rangle$ is true .
<code>\fp_do_while:nNnn</code> ☆	<code>\fp_do_while:nNnn {$\langle fpexpr_1 \rangle$} $\langle relation \rangle$ {$\langle fpexpr_2 \rangle$} {$\langle code \rangle$}</code>
New: 2012-08-16	Places the $\langle code \rangle$ in the input stream for \TeX to process, and then evaluates the relationship between the two $\langle floating\ point\ expressions \rangle$ as described for <code>\fp_compare:nNnTF</code> . If the test is true then the $\langle code \rangle$ is inserted into the input stream again and a loop occurs until the $\langle relation \rangle$ is false .

<hr/>	
<code>\fp_until_do:nNnn</code> ☆	<code>\fp_until_do:nNnn {<fpexpr1>} <relation> {<fpexpr2>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false. After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is true .
<hr/>	
<code>\fp_while_do:nNnn</code> ☆	<code>\fp_while_do:nNnn {<fpexpr1>} <relation> {<fpexpr2>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true. After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .
<hr/>	
<code>\fp_do_until:nn</code> ☆	<code>\fp_do_until:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16 Updated: 2013-12-14	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> . If the test is false then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is true .
<hr/>	
<code>\fp_do_while:nn</code> ☆	<code>\fp_do_while:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16 Updated: 2013-12-14	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> . If the test is true then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is false .
<hr/>	
<code>\fp_until_do:nn</code> ☆	<code>\fp_until_do:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16 Updated: 2013-12-14	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false. After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is true .
<hr/>	
<code>\fp_while_do:nn</code> ☆	<code>\fp_while_do:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16 Updated: 2013-12-14	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true. After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .

`\fp_step_function:nnnN` ☆
`\fp_step_function:nnnc` ☆

New: 2016-11-21
Updated: 2016-12-06

`\fp_step_function:nnnN` { $\langle initial\ value \rangle$ } { $\langle step \rangle$ } { $\langle final\ value \rangle$ } $\langle function \rangle$

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, each of which should be a floating point expression evaluating to a floating point number, not a tuple. The $\langle function \rangle$ is then placed in front of each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$). The $\langle step \rangle$ must be non-zero. If the $\langle step \rangle$ is positive, the loop stops when the $\langle value \rangle$ becomes larger than the $\langle final\ value \rangle$. If the $\langle step \rangle$ is negative, the loop stops when the $\langle value \rangle$ becomes smaller than the $\langle final\ value \rangle$. The $\langle function \rangle$ should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\fp_step_function:nnnN { 1.0 } { 0.1 } { 1.5 } \my_func:n
```

would print

```
[I saw 1.0]   [I saw 1.1]   [I saw 1.2]   [I saw 1.3]   [I saw 1.4]   [I saw 1.5]
```

TpXhackers note: Due to rounding, it may happen that adding the $\langle step \rangle$ to the $\langle value \rangle$ does not change the $\langle value \rangle$; such cases give an error, as they would otherwise lead to an infinite loop.

`\fp_step_inline:nnnn`

New: 2016-11-21
Updated: 2016-12-06

`\fp_step_inline:nnnn` { $\langle initial\ value \rangle$ } { $\langle step \rangle$ } { $\langle final\ value \rangle$ } { $\langle code \rangle$ }

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream with `#1` replaced by the current $\langle value \rangle$. Thus the $\langle code \rangle$ should define a function of one argument (`#1`).

`\fp_step_variable:nnnNn`

New: 2017-04-12

`\fp_step_variable:nnnNn`
{ $\langle initial\ value \rangle$ } { $\langle step \rangle$ } { $\langle final\ value \rangle$ } $\langle tl\ var \rangle$ { $\langle code \rangle$ }

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream, with the $\langle tl\ var \rangle$ defined as the current $\langle value \rangle$. Thus the $\langle code \rangle$ should make use of the $\langle tl\ var \rangle$.

6 Some useful constants, and scratch variables

`\c_zero_fp`
`\c_minus_zero_fp`

New: 2012-05-08

Zero, with either sign.

`\c_one_fp`

New: 2012-05-08

One as an fp: useful for comparisons in some places.

<hr/> <code>\c_inf_fp</code> <code>\c_minus_inf_fp</code> <hr/> New: 2012-05-08 <hr/>	<p>Infinity, with either sign. These can be input directly in a floating point expression as <code>inf</code> and <code>-inf</code>.</p>
<hr/> <code>\c_e_fp</code> <hr/> Updated: 2012-05-08 <hr/>	<p>The value of the base of the natural logarithm, $e = \exp(1)$.</p>
<hr/> <code>\c_pi_fp</code> <hr/> Updated: 2013-11-17 <hr/>	<p>The value of π. This can be input directly in a floating point expression as <code>pi</code>.</p>
<hr/> <code>\c_one_degree_fp</code> <hr/> New: 2012-05-08 Updated: 2013-11-17 <hr/>	<p>The value of 1° in radians. Multiply an angle given in degrees by this value to obtain a result in radians. Note that trigonometric functions expecting an argument in radians or in degrees are both available. Within floating point expressions, this can be accessed as <code>deg</code>.</p>
<hr/> <code>\l_tmpa_fp</code> <code>\l_tmpb_fp</code> <hr/>	<p>Scratch floating points for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.</p>
<hr/> <code>\g_tmpa_fp</code> <code>\g_tmpb_fp</code> <hr/>	<p>Scratch floating points for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.</p>

7 Floating point exceptions

The functions defined in this section are experimental, and their functionality may be altered or removed altogether.

“Exceptions” may occur when performing some floating point operations, such as $0 / 0$, or $10 ** 1e9999$. The relevant IEEE standard defines 5 types of exceptions, of which we implement 4.

- *Overflow* occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in $\pm\infty$.
- *Underflow* occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in ± 0 .
- *Invalid operation* occurs for operations with no defined outcome, for instance $0/0$ or $\sin(\infty)$, and results in a NaN. It also occurs for conversion functions whose target type does not have the appropriate infinite or NaN value (e.g., `\fp_to_dim:n`).
- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating functions at poles, e.g., $\ln(0)$ or $\cot(0)$. This results in $\pm\infty$.

(*not yet*) *Inexact* occurs whenever the result of a computation is not exact, in other words, almost always. At the moment, this exception is entirely ignored in L^AT_EX3.

To each exception we associate a “flag”: `fp_overflow`, `fp_underflow`, `fp_invalid_operation` and `fp_division_by_zero`. The state of these flags can be tested and modified with commands from `l3flag`

By default, the “invalid operation” exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions raise the corresponding flag but do not trigger an error. The behaviour when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and raise the flag, or only raise the flag, or do nothing at all.

<code>\fp_trap:nn</code>	<code>\fp_trap:nn {<exception>} {<trap type>}</code>
New: 2012-07-19 Updated: 2017-02-13	All occurrences of the <code><exception></code> (<code>overflow</code> , <code>underflow</code> , <code>invalid_operation</code> or <code>division_by_zero</code>) within the current group are treated as <code><trap type></code> , which can be <ul style="list-style-type: none"> • none: the <code><exception></code> will be entirely ignored, and leave no trace; • flag: the <code><exception></code> will turn the corresponding flag on when it occurs; • error: additionally, the <code><exception></code> will halt the T_EX run and display some information about the current operation in the terminal.

This function is experimental, and may be altered or removed.

`flag_fp_overflow`
`flag_fp_underflow`
`flag_fp_invalid_operation`
`flag_fp_division_by_zero`

Flags denoting the occurrence of various floating-point exceptions.

8 Viewing floating points

<code>\fp_show:N</code>	<code>\fp_show:N <fp var></code>
<code>\fp_show:c</code>	<code>\fp_show:n {<floating point expression>}</code>
<code>\fp_show:n</code>	Evaluates the <code><floating point expression></code> and displays the result in the terminal.

New: 2012-05-08
Updated: 2015-08-07

<code>\fp_log:N</code>	<code>\fp_log:N <fp var></code>
<code>\fp_log:c</code>	<code>\fp_log:n {<floating point expression>}</code>
<code>\fp_log:n</code>	Evaluates the <code><floating point expression></code> and writes the result in the log file.

New: 2014-08-22
Updated: 2015-08-07

9 Floating point expressions

9.1 Input of floating point numbers

We support four types of floating point numbers:

- $\pm m \cdot 10^n$, a floating point number, with integer $1 \leq m \leq 10^{16}$, and $-10000 \leq n \leq 10000$;
- ± 0 , zero, with a given sign;
- $\pm \infty$, infinity, with a given sign;
- NaN, is “not a number”, and can be either quiet or signalling (*not yet*: this distinction is currently unsupported);

Normal floating point numbers are stored in base 10, with up to 16 significant figures.

On input, a normal floating point number consists of:

- $\langle sign \rangle$: a possibly empty string of + and - characters;
- $\langle significand \rangle$: a non-empty string of digits together with zero or one dot;
- $\langle exponent \rangle$ optionally: the character e or E, followed by a possibly empty string of + and - tokens, and a non-empty string of digits.

The sign of the resulting number is + if $\langle sign \rangle$ contains an even number of -, and - otherwise, hence, an empty $\langle sign \rangle$ denotes a non-negative input. The stored significand is obtained from $\langle significand \rangle$ by omitting the decimal separator and leading zeros, and rounding to 16 significant digits, filling with trailing zeros if necessary. In particular, the value stored is exact if the input $\langle significand \rangle$ has at most 16 digits. The stored $\langle exponent \rangle$ is obtained by combining the input $\langle exponent \rangle$ (0 if absent) with a shift depending on the position of the significand and the number of leading zeros.

A special case arises if the resulting $\langle exponent \rangle$ is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by $\pm \infty$), or an underflow (resulting in ± 0).

The result is thus ± 0 if and only if $\langle significand \rangle$ contains no non-zero digit (*i.e.*, consists only in characters 0, and an optional period), or if there is an underflow. Note that a single dot is currently a valid floating point number, equal to +0, but that is not guaranteed to remain true.

The $\langle significand \rangle$ must be non-empty, so e1 and e-1 are not valid floating point numbers. Note that the latter could be mistaken with the difference of “e” and 1. To avoid confusions, the base of natural logarithms cannot be input as e and should be input as exp(1) or \c_e_fp (which is faster).

Special numbers are input as follows:

- inf represents $+\infty$, and can be preceded by any $\langle sign \rangle$, yielding $\pm \infty$ as appropriate.
- nan represents a (quiet) non-number. It can be preceded by any sign, but that sign is ignored.
- Any unrecognizable string triggers an error, and produces a NaN.
- Note that commands such as \infy, \pi, or \sin *do not* work in floating point expressions. They may silently be interpreted as completely unexpected numbers, because integer constants (allowed in expressions) are commonly stored as mathematical characters.

9.2 Precedence of operators

We list here all the operations supported in floating point expressions, in order of decreasing precedence: operations listed earlier bind more tightly than operations listed below them.

- Function calls (`sin`, `ln`, *etc*).
- Binary `**` and `^` (right associative).
- Unary `+`, `-`, `!`.
- Implicit multiplication by juxtaposition (`2pi`) when neither factor is in parentheses.
- Binary `*` and `/`, implicit multiplication by juxtaposition with parentheses (for instance `3(4+5)`).
- Binary `+` and `-`.
- Comparisons `>=`, `!=`, `<?`, *etc*.
- Logical `and`, denoted by `&&`.
- Logical `or`, denoted by `||`.
- Ternary operator `?:` (right associative).
- Comma (to build tuples).

The precedence of operations can be overridden using parentheses. In particular, the precedence of juxtaposition implies that

$$\begin{aligned}1/2\text{pi} &= 1/(2\pi), \\1/2\text{pi}(\text{pi} + \text{pi}) &= (2\pi)^{-1}(\pi + \pi) \simeq 1, \\ \text{sin}2\text{pi} &= \sin(2)\pi \neq 0, \\ 2^2\text{max}(3, 5) &= 2^2 \max(3, 5) = 20, \\ 1\text{in}/1\text{cm} &= (1\text{in})/(1\text{cm}) = 2.54.\end{aligned}$$

Functions are called on the value of their argument, contrarily to `TeX` macros.

9.3 Operations

We now present the various operations allowed in floating point expressions, from the lowest precedence to the highest. When used as a truth value, a floating point expression is `false` if it is ± 0 , and `true` otherwise, including when it is `NaN` or a tuple such as `(0, 0)`. Tuples are only supported to some extent by operations that work with truth values (`?:`, `||`, `&&`, `!`), by comparisons (`!<=>?`), and by `+`, `-`, `*`, `/`. Unless otherwise specified, providing a tuple as an argument of any other operation yields the “invalid operation” exception and a `NaN` result.

```
?: \fp_eval:n { <operand1> ? <operand2> : <operand3> }
```

The ternary operator `?:` results in $\langle operand_2 \rangle$ if $\langle operand_1 \rangle$ is true (not ± 0), and $\langle operand_3 \rangle$ if $\langle operand_1 \rangle$ is false (± 0). All three $\langle operands \rangle$ are evaluated in all cases; they may be tuples. The operator is right associative, hence

```
\fp_eval:n
{
  1 + 3 > 4 ? 1 :
  2 + 4 > 5 ? 2 :
  3 + 5 > 6 ? 3 : 4
}
```

first tests whether $1 + 3 > 4$; since this isn't true, the branch following `:` is taken, and $2 + 4 > 5$ is compared; since this is true, the branch before `:` is taken, and everything else is (evaluated then) ignored. That allows testing for various cases in a concise manner, with the drawback that all computations are made in all cases.

```
|| \fp_eval:n { <operand1> || <operand2> }
```

If $\langle operand_1 \rangle$ is true (not ± 0), use that value, otherwise the value of $\langle operand_2 \rangle$. Both $\langle operands \rangle$ are evaluated in all cases; they may be tuples. In $\langle operand_1 \rangle || \langle operand_2 \rangle || \dots || \langle operand_n \rangle$, the first true (nonzero) $\langle operand \rangle$ is used and if all are zero the last one (± 0) is used.

```
&& \fp_eval:n { <operand1> && <operand2> }
```

If $\langle operand_1 \rangle$ is false (equal to ± 0), use that value, otherwise the value of $\langle operand_2 \rangle$. Both $\langle operands \rangle$ are evaluated in all cases; they may be tuples. In $\langle operand_1 \rangle \&\& \langle operand_2 \rangle \&\& \dots \&\& \langle operand_n \rangle$, the first false (± 0) $\langle operand \rangle$ is used and if none is zero the last one is used.

```
< \fp_eval:n
= {
>   <operand1> <relation1>
?   ...
    <operand_N> <relation_N>
    <operand_{N+1}>
}
```

Updated: 2013-12-14

Each $\langle relation \rangle$ consists of a non-empty string of `<`, `=`, `>`, and `?`, optionally preceded by `!`, and may not start with `?`. This evaluates to $+1$ if all comparisons $\langle operand_i \rangle \langle relation_i \rangle \langle operand_{i+1} \rangle$ are true, and $+0$ otherwise. All $\langle operands \rangle$ are evaluated (once) in all cases. See `\fp_compare:nTF` for details.

```
+ \fp_eval:n { <operand1> + <operand2> }
- \fp_eval:n { <operand1> - <operand2> }
```

Computes the sum or the difference of its two $\langle operands \rangle$. The “invalid operation” exception occurs for $\infty - \infty$. “Underflow” and “overflow” occur when appropriate. These operations supports the itemwise addition or subtraction of two tuples, but if they have a different number of items the “invalid operation” exception occurs and the result is NaN.

```

* \fp_eval:n { <operand1> * <operand2> }
/ \fp_eval:n { <operand1> / <operand2> }

```

Computes the product or the ratio of its two $\langle \text{operands} \rangle$. The “invalid operation” exception occurs for ∞/∞ , $0/0$, or $0 * \infty$. “Division by zero” occurs when dividing a finite non-zero number by ± 0 . “Underflow” and “overflow” occur when appropriate. When $\langle \text{operand}_1 \rangle$ is a tuple and $\langle \text{operand}_2 \rangle$ is a floating point number, each item of $\langle \text{operand}_1 \rangle$ is multiplied or divided by $\langle \text{operand}_2 \rangle$. Multiplication also supports the case where $\langle \text{operand}_1 \rangle$ is a floating point number and $\langle \text{operand}_2 \rangle$ a tuple. Other combinations yield an “invalid operation” exception and a NaN result.

```

+ \fp_eval:n { + <operand> }
- \fp_eval:n { - <operand> }
! \fp_eval:n { ! <operand> }

```

The unary $+$ does nothing, the unary $-$ changes the sign of the $\langle \text{operand} \rangle$ (for a tuple, of all its components), and $!$ $\langle \text{operand} \rangle$ evaluates to 1 if $\langle \text{operand} \rangle$ is false (is ± 0) and 0 otherwise (this is the `not` boolean function). Those operations never raise exceptions.

```

** \fp_eval:n { <operand1> ** <operand2> }
^ \fp_eval:n { <operand1> ^ <operand2> }

```

Raises $\langle \text{operand}_1 \rangle$ to the power $\langle \text{operand}_2 \rangle$. This operation is right associative, hence 2^{2^3} equals $2^{2^3} = 256$. If $\langle \text{operand}_1 \rangle$ is negative or -0 then: the result’s sign is $+$ if the $\langle \text{operand}_2 \rangle$ is infinite and $(-1)^p$ if the $\langle \text{operand}_2 \rangle$ is $p/5^q$ with p, q integers; the result is $+0$ if $\text{abs}(\langle \text{operand}_1 \rangle)^{\langle \text{operand}_2 \rangle}$ evaluates to zero; in other cases the “invalid operation” exception occurs because the sign cannot be determined. “Division by zero” occurs when raising ± 0 to a finite strictly negative power. “Underflow” and “overflow” occur when appropriate. If either operand is a tuple, “invalid operation” occurs.

```

abs \fp_eval:n { abs( <fpexpr> ) }

```

Computes the absolute value of the $\langle \text{fpexpr} \rangle$. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases. See also `\fp_abs:n`.

```

exp \fp_eval:n { exp( <fpexpr> ) }

```

Computes the exponential of the $\langle \text{fpexpr} \rangle$. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

```

fact \fp_eval:n { fact( <fpexpr> ) }

```

Computes the factorial of the $\langle \text{fpexpr} \rangle$. If the $\langle \text{fpexpr} \rangle$ is an integer between -0 and 3248 included, the result is finite and correctly rounded. Larger positive integers give $+\infty$ with “overflow”, while $\text{fact}(+\infty) = +\infty$ and $\text{fact}(\text{nan}) = \text{nan}$ with no exception. All other inputs give NaN with the “invalid operation” exception.

```

ln \fp_eval:n { ln( <fpexpr> ) }

```

Computes the natural logarithm of the $\langle \text{fpexpr} \rangle$. Negative numbers have no (real) logarithm, hence the “invalid operation” is raised in that case, including for $\ln(-0)$. “Division by zero” occurs when evaluating $\ln(+0) = -\infty$. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

<hr/> logb <hr/>	★	<code>\fp_eval:n { logb(<fpexpr>) }</code>	
<hr/> New: 2018-11-03 <hr/>			Determines the exponent of the $\langle fpexpr \rangle$, namely the floor of the base-10 logarithm of its absolute value. “Division by zero” occurs when evaluating $\text{logb}(\pm 0) = -\infty$. Other special values are $\text{logb}(\pm\infty) = +\infty$ and $\text{logb}(\text{NaN}) = \text{NaN}$. If the operand is a tuple or is NaN, then “invalid operation” occurs and the result is NaN.
<hr/> max <hr/>		<code>\fp_eval:n { max(<fpexpr₁> , <fpexpr₂> , ...) }</code>	
<hr/> min <hr/>		<code>\fp_eval:n { min(<fpexpr₁> , <fpexpr₂> , ...) }</code>	
			Evaluates each $\langle fpexpr \rangle$ and computes the largest (smallest) of those. If any of the $\langle fpexpr \rangle$ is a NaN or tuple, the result is NaN. If any operand is a tuple, “invalid operation” occurs; these operations do not raise exceptions in other cases.
<hr/> round <hr/>		<code>\fp_eval:n { round (<fpexpr>) }</code>	
trunc		<code>\fp_eval:n { round (<fpexpr₁> , <fpexpr₂>) }</code>	
ceil		<code>\fp_eval:n { round (<fpexpr₁> , <fpexpr₂> , <fpexpr₃>) }</code>	
floor			
<hr/> New: 2013-12-14 <hr/> Updated: 2015-08-08 <hr/>			Only round accepts a third argument. Evaluates $\langle fpexpr_1 \rangle = x$ and $\langle fpexpr_2 \rangle = n$ and $\langle fpexpr_3 \rangle = t$ then rounds x to n places. If n is an integer, this rounds x to a multiple of 10^{-n} ; if $n = +\infty$, this always yields x ; if $n = -\infty$, this yields one of ± 0 , $\pm\infty$, or NaN; if $n = \text{NaN}$, this yields NaN; if n is neither $\pm\infty$ nor an integer, then an “invalid operation” exception is raised. When $\langle fpexpr_2 \rangle$ is omitted, $n = 0$, <i>i.e.</i> , $\langle fpexpr_1 \rangle$ is rounded to an integer. The rounding direction depends on the function. <ul style="list-style-type: none"> • round yields the multiple of 10^{-n} closest to x, with ties (x half-way between two such multiples) rounded as follows. If t is nan (or not given) the even multiple is chosen (“ties to even”), if $t = \pm 0$ the multiple closest to 0 is chosen (“ties to zero”), if t is positive/negative the multiple closest to $\infty/-\infty$ is chosen (“ties towards positive/negative infinity”). • floor yields the largest multiple of 10^{-n} smaller or equal to x (“round towards negative infinity”); • ceil yields the smallest multiple of 10^{-n} greater or equal to x (“round towards positive infinity”); • trunc yields a multiple of 10^{-n} with the same sign as x and with the largest absolute value less than that of x (“round towards zero”). <p>“Overflow” occurs if x is finite and the result is infinite (this can only happen if $\langle fpexpr_2 \rangle < -9984$). If any operand is a tuple, “invalid operation” occurs.</p>
<hr/> sign <hr/>		<code>\fp_eval:n { sign(<fpexpr>) }</code>	
			Evaluates the $\langle fpexpr \rangle$ and determines its sign: +1 for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ± 0 for ± 0 , and NaN for NaN. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases.

<code>sin</code>	<code>\fp_eval:n { sin(<fpexpr>) }</code>
<code>cos</code>	<code>\fp_eval:n { cos(<fpexpr>) }</code>
<code>tan</code>	<code>\fp_eval:n { tan(<fpexpr>) }</code>
<code>cot</code>	<code>\fp_eval:n { cot(<fpexpr>) }</code>
<code>csc</code>	<code>\fp_eval:n { csc(<fpexpr>) }</code>
<code>sec</code>	<code>\fp_eval:n { sec(<fpexpr>) }</code>

Updated: 2013-11-17

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fpexpr \rangle$ given in radians. For arguments given in degrees, see `sind`, `cosd`, *etc.* Note that since π is irrational, `sin(8pi)` is not quite zero, while its analogue `sind(8 × 180)` is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

<code>sind</code>	<code>\fp_eval:n { sind(<fpexpr>) }</code>
<code>cosd</code>	<code>\fp_eval:n { cosd(<fpexpr>) }</code>
<code>tand</code>	<code>\fp_eval:n { tand(<fpexpr>) }</code>
<code>cotd</code>	<code>\fp_eval:n { cotd(<fpexpr>) }</code>
<code>cscd</code>	<code>\fp_eval:n { cscd(<fpexpr>) }</code>
<code>secd</code>	<code>\fp_eval:n { secd(<fpexpr>) }</code>

New: 2013-11-02

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fpexpr \rangle$ given in degrees. For arguments given in radians, see `sin`, `cos`, *etc.* Note that since π is irrational, `sin(8pi)` is not quite zero, while its analogue `sind(8 × 180)` is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

<code>asin</code>	<code>\fp_eval:n { asin(<fpexpr>) }</code>
<code>acos</code>	<code>\fp_eval:n { acos(<fpexpr>) }</code>
<code>acsc</code>	<code>\fp_eval:n { acsc(<fpexpr>) }</code>
<code>asec</code>	<code>\fp_eval:n { asec(<fpexpr>) }</code>

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the $\langle fpexpr \rangle$ and returns the result in radians, in the range $[-\pi/2, \pi/2]$ for `asin` and `acsc` and $[0, \pi]$ for `acos` and `asec`. For a result in degrees, use `asind`, *etc.* If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

<code>asind</code>	<code>\fp_eval:n { asind(<fpexpr>) }</code>
<code>acosd</code>	<code>\fp_eval:n { acosd(<fpexpr>) }</code>
<code>acscd</code>	<code>\fp_eval:n { acscd(<fpexpr>) }</code>
<code>asecd</code>	<code>\fp_eval:n { asecd(<fpexpr>) }</code>

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the $\langle fpexpr \rangle$ and returns the result in degrees, in the range $[-90, 90]$ for `asin` and `acsc` and $[0, 180]$ for `acos` and `asec`. For a result in radians, use `asin`, *etc.* If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

atan	<code>\fp_eval:n { atan(<fpexpr>) }</code>
acot	<code>\fp_eval:n { atan(<fpexpr₁> , <fpexpr₂>) }</code>
<hr/>	
New: 2013-11-02	<code>\fp_eval:n { acot(<fpexpr>) }</code>
	<code>\fp_eval:n { acot(<fpexpr₁> , <fpexpr₂>) }</code>

Those functions yield an angle in radians: **atand** and **acotd** are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the $\langle fpexpr \rangle$: arctangent takes values in the range $[-\pi/2, \pi/2]$, and arccotangent in the range $[0, \pi]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fpexpr_2 \rangle, \langle fpexpr_1 \rangle)$: this is the arctangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by π depending on the signs of $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpexpr_1 \rangle, \langle fpexpr_2 \rangle)$, equal to the arccotangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by π . Both two-argument functions take values in the wider range $[-\pi, \pi]$. The ratio $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0 , or when both yield $\pm \infty$, the resulting angle is one of $\{\pm\pi/4, \pm 3\pi/4\}$ depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

atand	<code>\fp_eval:n { atand(<fpexpr>) }</code>
acotd	<code>\fp_eval:n { atand(<fpexpr₁> , <fpexpr₂>) }</code>
<hr/>	
New: 2013-11-02	<code>\fp_eval:n { acotd(<fpexpr>) }</code>
	<code>\fp_eval:n { acotd(<fpexpr₁> , <fpexpr₂>) }</code>

Those functions yield an angle in degrees: **atand** and **acotd** are their analogs in radians. The one-argument versions compute the arctangent or arccotangent of the $\langle fpexpr \rangle$: arctangent takes values in the range $[-90, 90]$, and arccotangent in the range $[0, 180]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fpexpr_2 \rangle, \langle fpexpr_1 \rangle)$: this is the arctangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by 180 depending on the signs of $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpexpr_1 \rangle, \langle fpexpr_2 \rangle)$, equal to the arccotangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by 180. Both two-argument functions take values in the wider range $[-180, 180]$. The ratio $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0 , or when both yield $\pm \infty$, the resulting angle is one of $\{\pm 45, \pm 135\}$ depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

sqrt	<code>\fp_eval:n { sqrt(<fpexpr>) }</code>
-------------	--

New: 2013-12-14 Computes the square root of the $\langle fpexpr \rangle$. The “invalid operation” is raised when the $\langle fpexpr \rangle$ is negative or is a tuple; no other exception can occur. Special values yield $\sqrt{-0} = -0$, $\sqrt{+0} = +0$, $\sqrt{+\infty} = +\infty$ and $\sqrt{\text{NaN}} = \text{NaN}$.

<hr/> rand <hr/>	<code>\fp_eval:n { rand() }</code>
<hr/> New: 2016-12-05 <hr/>	Produces a pseudo-random floating-point number (multiple of 10^{-16}) between 0 included and 1 excluded. This is not available in older versions of $\text{X}\text{_}\text{T}\text{E}\text{X}$. The random seed can be queried using <code>\sys_rand_seed:</code> and set using <code>\sys_gset_rand_seed:n</code> .
	<p>TEXhackers note: This is based on pseudo-random numbers provided by the engine’s primitive <code>\pdfuniformdeviate</code> in $\text{pdf}\text{_}\text{T}\text{E}\text{X}$, $\text{p}\text{_}\text{T}\text{E}\text{X}$, $\text{up}\text{_}\text{T}\text{E}\text{X}$ and <code>\uniformdeviate</code> in $\text{Lua}\text{_}\text{T}\text{E}\text{X}$ and $\text{X}\text{_}\text{T}\text{E}\text{X}$. The underlying code is based on Metapost, which follows an additive scheme recommended in Section 3.6 of “The Art of Computer Programming, Volume 2”.</p> <p>While we are more careful than <code>\uniformdeviate</code> to preserve uniformity of the underlying stream of 28-bit pseudo-random integers, these pseudo-random numbers should of course not be relied upon for serious numerical computations nor cryptography.</p>
<hr/> randint <hr/>	<code>\fp_eval:n { randint(<fpexpr>) }</code>
<hr/> New: 2016-12-05 <hr/>	<code>\fp_eval:n { randint(<fpexpr₁> , <fpexpr₂>) }</code>
	Produces a pseudo-random integer between 1 and $\langle fpexpr \rangle$ or between $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$ inclusive. The bounds must be integers in the range $(-10^{16}, 10^{16})$ and the first must be smaller or equal to the second. See rand for important comments on how these pseudo-random numbers are generated.
<hr/> inf nan <hr/>	The special values $+\infty$, $-\infty$, and NaN are represented as inf , -inf and nan (see <code>\c_minus_inf_fp</code> , <code>\c_minus_inf_fp</code> and <code>\c_nan_fp</code>).
<hr/> pi <hr/>	The value of π (see <code>\c_pi_fp</code>).
<hr/> deg <hr/>	The value of 1° in radians (see <code>\c_one_degree_fp</code>).

`em` Those units of measurement are equal to their values in `pt`, namely

`ex`
`in` $1\text{in} = 72.27\text{pt}$
`pt` $1\text{pt} = 1\text{pt}$
`pc` $1\text{pc} = 12\text{pt}$
`cm`
`mm` $1\text{cm} = \frac{1}{2.54}\text{in} = 28.45275590551181\text{pt}$
`dd`
`cc` $1\text{mm} = \frac{1}{25.4}\text{in} = 2.845275590551181\text{pt}$
`nd`
`nc` $1\text{dd} = 0.376065\text{mm} = 1.07000856496063\text{pt}$
`bp` $1\text{cc} = 12\text{dd} = 12.84010277952756\text{pt}$
`sp` $1\text{nd} = 0.375\text{mm} = 1.066978346456693\text{pt}$

 $1\text{nc} = 12\text{nd} = 12.80374015748031\text{pt}$

 $1\text{bp} = \frac{1}{72}\text{in} = 1.00375\text{pt}$
 $1\text{sp} = 2^{-16}\text{pt} = 1.52587890625e - 5\text{pt}.$

The values of the (font-dependent) units `em` and `ex` are gathered from \TeX when the surrounding floating point expression is evaluated.

`true` Other names for 1 and +0.
`false`

<code>\fp_abs:n</code> \star <div style="font-size: small;"> New: 2012-05-14 Updated: 2012-07-08 </div>	<code>\fp_abs:n</code> $\{\langle\textit{floating point expression}\rangle\}$ Evaluates the $\langle\textit{floating point expression}\rangle$ as described for <code>\fp_eval:n</code> and leaves the absolute value of the result in the input stream. If the argument is $\pm\infty$, <code>NaN</code> or a tuple, “invalid operation” occurs. Within floating point expressions, <code>abs()</code> can be used; it accepts $\pm\infty$ and <code>NaN</code> as arguments.
---	--

<code>\fp_max:nn</code> \star <code>\fp_min:nn</code> \star <div style="font-size: small;"> New: 2012-09-26 </div>	<code>\fp_max:nn</code> $\{\langle\textit{fp expression 1}\rangle\} \{\langle\textit{fp expression 2}\rangle\}$ Evaluates the $\langle\textit{floating point expressions}\rangle$ as described for <code>\fp_eval:n</code> and leaves the resulting larger (<code>max</code>) or smaller (<code>min</code>) value in the input stream. If the argument is a tuple, “invalid operation” occurs, but no other case raises exceptions. Within floating point expressions, <code>max()</code> and <code>min()</code> can be used.
--	--

10 Disclaimer and roadmap

The package may break down if the escape character is among `0123456789_+`, or if it receives a \TeX primitive conditional affected by `\exp_not:N`.

The following need to be done. I’ll try to time-order the items.

- Function to count items in a tuple (and to determine if something is a tuple).
- Decide what exponent range to consider.

- Support signalling `nan`.
- Modulo and remainder, and rounding function `quantize` (and its friends analogous to `trunc`, `ceil`, `floor`).
- `\fp_format:nn {<fpepr>} {<format>}`, but what should `<format>` be? More general pretty printing?
- Add `and`, `or`, `xor`? Perhaps under the names `all`, `any`, and `xor`?
- Add `log(x,b)` for logarithm of x in base b .
- `hypot` (Euclidean length). Cartesian-to-polar transform.
- Hyperbolic functions `cosh`, `sinh`, `tanh`.
- Inverse hyperbolics.
- Base conversion, input such as `0xAB.CDEF`.
- Factorial (not with `!`), gamma function.
- Improve coefficients of the `sin` and `tan` series.
- Treat upper and lower case letters identically in identifiers, and ignore underscores.
- Add an `array(1,2,3)` and `i=complex(0,1)`.
- Provide an experimental `map` function? Perhaps easier to implement if it is a single character, `@sin(1,2)`?
- Provide an `isnan` function analogue of `\fp_if_nan:nTF`?
- Support keyword arguments?

`Pgfmath` also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs, and tests to add.

- Check that functions are monotonic when they should.
- Add exceptions to `?:`, `!<=>?`, `&&`, `||`, and `!`.
- Logarithms of numbers very close to 1 are inaccurate.
- When rounding towards $-\infty$, `\dim_to_fp:n {Opt}` should return -0 , not $+0$.
- The result of $(\pm 0) + (\pm 0)$, of $x + (-x)$, and of $(-x) + x$ should depend on the rounding mode.
- `0e9999999999` gives a $\text{T}_{\text{E}}\text{X}$ “number too large” error.
- Subnormals are not implemented.

Possible optimizations/improvements.

- Document that `l3trial/l3fp-types` introduces tools for adding new types.
- In subsection 9.1, write a grammar.

- It would be nice if the `parse` auxiliaries for each operation were set up in the corresponding module, rather than centralizing in `l3fp-parse`.
- Some functions should get an `_o` ending to indicate that they expand after their result.
- More care should be given to distinguish expandable/restricted expandable (auxiliary and internal) functions.
- The code for the `ternary` set of functions is ugly.
- There are many `~` missing in the doc to avoid bad line-breaks.
- The algorithm for computing the logarithm of the significand could be made to use a 5 terms Taylor series instead of 10 terms by taking $c = 2000/([200x]+1) \in [10, 95]$ instead of $c \in [1, 10]$. Also, it would then be possible to simplify the computation of t . However, we would then have to hard-code the logarithms of 44 small integers instead of 9.
- Improve notations in the explanations of the division algorithm (`l3fp-basics`).
- Understand and document `_fp_basics_pack_weird_low:NNNNw` and `_fp_basics_pack_weird_high:NNNNNNNNw` better. Move the other `basics_pack` auxiliaries to `l3fp-aux` under a better name.
- Find out if underflow can really occur for trigonometric functions, and redoc as appropriate.
- Add bibliography. Some of Kahan’s articles, some previous T_EX fp packages, the international standards,...
- Also take into account the “inexact” exception?
- Support multi-character prefix operators (*e.g.*, `@/` or whatever)?

Part XXIV

The l3farray package: fast global floating point arrays

1 l3farray documentation

For applications requiring heavy use of floating points, this module provides arrays which can be accessed in constant time (contrast l3seq, where access time is linear). The interface is very close to that of l3intarray. The size of the array is fixed and must be given at point of initialisation

<code>\farray_new:Nn</code>	<code>\farray_new:Nn <farray var> {<size>}</code>
-----------------------------	---

New: 2018-05-05

Evaluates the integer expression *<size>* and allocates an *<floating point array variable>* with that number of (zero) entries. The variable name should start with `\g_` because assignments are always global.

<code>\farray_count:N</code> ★	<code>\farray_count:N <farray var></code>
--------------------------------	---

New: 2018-05-05

Expands to the number of entries in the *<floating point array variable>*. This is performed in constant time.

<code>\farray_gset:Nnn</code>	<code>\farray_gset:Nnn <farray var> {<position>} {<value>}</code>
-------------------------------	---

New: 2018-05-05

Stores the result of evaluating the floating point expression *<value>* into the *<floating point array variable>* at the (integer expression) *<position>*. If the *<position>* is not between 1 and the `\farray_count:N`, an error occurs. Assignments are always global.

<code>\farray_gzero:N</code>	<code>\farray_gzero:N <farray var></code>
------------------------------	---

New: 2018-05-05

Sets all entries of the *<floating point array variable>* to +0. Assignments are always global.

<code>\farray_item:Nn</code> ★	<code>\farray_item:Nn <farray var> {<position>}</code>
--------------------------------	--

`\farray_item_to_tl:Nn` ★

New: 2018-05-05

Applies `\fp_use:N` or `\fp_to_tl:N` (respectively) to the floating point entry stored at the (integer expression) *<position>* in the *<floating point array variable>*. If the *<position>* is not between 1 and the `\farray_count:N`, an error occurs.

Part XXV

The l3cctab package

Category code tables

A category code table enables rapid switching of all category codes in one operation. For LuaTeX, this is possible over the entire Unicode range. For other engines, only the 8-bit range (0–255) is covered by such tables.

1 Creating and initialising category code tables

<hr/> <code>\cctab_new:N</code> <code>\cctab_new:c</code> <hr/> <small>Updated: 2020-07-02</small> <hr/>	<code>\cctab_new:N</code> \langle <i>category code table</i> \rangle Creates a new \langle <i>category code table</i> \rangle variable or raises an error if the name is already taken. The declaration is global. The \langle <i>category code table</i> \rangle is initialised with the codes as used by <code>iniTeX</code> .
<hr/> <code>\cctab_const:Nn</code> <code>\cctab_const:cn</code> <hr/> <small>Updated: 2020-07-07</small> <hr/>	<code>\cctab_const:Nn</code> \langle <i>category code table</i> \rangle $\{ \langle$ <i>category code set up</i> $\rangle \}$ Creates a new \langle <i>category code table</i> \rangle , applies (in a group) the \langle <i>category code set up</i> \rangle on top of <code>iniTeX</code> settings, then saves them globally as a constant table. The \langle <i>category code set up</i> \rangle can include a call to <code>\cctab_select:N</code> .
<hr/> <code>\cctab_gset:Nn</code> <code>\cctab_gset:cn</code> <hr/> <small>Updated: 2020-07-07</small> <hr/>	<code>\cctab_gset:Nn</code> \langle <i>category code table</i> \rangle $\{ \langle$ <i>category code set up</i> $\rangle \}$ Starting from the <code>iniTeX</code> category codes, applies (in a group) the \langle <i>category code set up</i> \rangle , then saves them globally in the \langle <i>category code table</i> \rangle . The \langle <i>category code set up</i> \rangle can include a call to <code>\cctab_select:N</code> .

2 Using category code tables

<hr/> <code>\cctab_begin:N</code> <code>\cctab_begin:c</code> <hr/> <small>Updated: 2020-07-02</small> <hr/>	<code>\cctab_begin:N</code> \langle <i>category code table</i> \rangle Switches locally the category codes in force to those stored in the \langle <i>category code table</i> \rangle . The prevailing codes before the function is called are added to a stack, for use with <code>\cctab_end:.</code> This function does not start a TeX group.
<hr/> <code>\cctab_end:</code> <hr/> <small>Updated: 2020-07-02</small> <hr/>	<code>\cctab_end:</code> Ends the scope of a \langle <i>category code table</i> \rangle started using <code>\cctab_begin:N</code> , returning the codes to those in force before the matching <code>\cctab_begin:N</code> was used. This must be used within the same TeX group (and at the same TeX group level) as the matching <code>\cctab_begin:N</code> .
<hr/> <code>\cctab_select:N</code> <hr/> <small>New: 2020-05-19 Updated: 2020-07-02</small> <hr/>	<code>\cctab_select:N</code> \langle <i>category code table</i> \rangle Selects the \langle <i>category code table</i> \rangle for the scope of the current group. This is in particular useful in the \langle <i>setup</i> \rangle arguments of <code>\tl_set_rescan:Nnn</code> , <code>\tl_rescan:nn</code> , <code>\cctab_const:Nn</code> , and <code>\cctab_gset:Nn</code> .

3 Category code table conditionals

<hr/> <code>\cctab_if_exist_p:N</code> *	<code>\cctab_if_exist_p:N</code> \langle <i>category code table</i> \rangle
<code>\cctab_if_exist_p:c</code> *	<code>\cctab_if_exist:NTF</code> \langle <i>category code table</i> \rangle $\{\langle$ <i>true code</i> $\rangle\}$ $\{\langle$ <i>false code</i> $\rangle\}$
<code>\cctab_if_exist:NTF</code> *	Tests whether the \langle <i>category code table</i> \rangle is currently defined. This does not check that the \langle <i>category code table</i> \rangle really is a category code table.
<code>\cctab_if_exist:cTF</code> *	

4 Constant category code tables

<hr/> <code>\c_code_cctab</code>	Category code table for the <code>expl3</code> code environment; this does <i>not</i> include <code>@</code> , which is retained as an “other” character.
Updated: 2020-07-10	

<hr/> <code>\c_document_cctab</code>	Category code table for a standard L ^A T _E X document, as set by the L ^A T _E X kernel. In particular, the upper-half of the 8-bit range will be set to “active” with pdfT _E X <i>only</i> . No <code>babel</code> shorthands will be activated.
Updated: 2020-07-08	

<hr/> <code>\c_initex_cctab</code>	Category code table as set up by iniT _E X.
Updated: 2020-07-02	

<hr/> <code>\c_other_cctab</code>	Category code table where all characters have category code 12 (other).
Updated: 2020-07-02	

<hr/> <code>\c_str_cctab</code>	Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space).
Updated: 2020-07-02	

Part XXVI

The l3sort package

Sorting functions

1 Controlling sorting

L^AT_EX3 comes with a facility to sort list variables (sequences, token lists, or comma-lists) according to some user-defined comparison. For instance,

```
\clist_set:Nn \l_foo_clist { 3 , 01 , -2 , 5 , +1 }
\clist_sort:Nn \l_foo_clist
{
  \int_compare:nNnTF { #1 } > { #2 }
  { \sort_return_swapped: }
  { \sort_return_same: }
}
```

results in `\l_foo_clist` holding the values `{ -2 , 01 , +1 , 3 , 5 }` sorted in non-decreasing order.

The code defining the comparison should call `\sort_return_swapped:` if the two items given as `#1` and `#2` are not in the correct order, and otherwise it should call `\sort_return_same:` to indicate that the order of this pair of items should not be changed.

For instance, a *comparison code* consisting only of `\sort_return_same:` with no test yields a trivial sort: the final order is identical to the original order. Conversely, using a *comparison code* consisting only of `\sort_return_swapped:` reverses the list (in a fairly inefficient way).

T_EXhackers note: The current implementation is limited to sorting approximately 20000 items (40000 in LuaT_EX), depending on what other packages are loaded.

Internally, the code from l3sort stores items in `\toks` registers allocated locally. Thus, the *comparison code* should not call `\newtoks` or other commands that allocate new `\toks` registers. On the other hand, altering the value of a previously allocated `\toks` register is not a problem.

```
\sort_return_same:
\sort_return_swapped:
```

New: 2017-02-06

```
\seq_sort:Nn <seq var>
{ ... \sort_return_same: or \sort_return_swapped: ... }
```

Indicates whether to keep the order or swap the order of two items that are compared in the sorting code. Only one of the `\sort_return_...` functions should be used by the code, according to the results of some tests on the items `#1` and `#2` to be compared.

Part XXVII

The l3tl-analysis package: Analysing token lists

1 l3tl-analysis documentation

This module mostly provides internal functions for use in the l3regex module. However, it provides as a side-effect a user debugging function, very similar to the \ShowTokens macro from the ted package.

```
\tl_analysis_show:N
\tl_analysis_show:n
```

New: 2018-04-09

```
\tl_analysis_show:n {<token list>}
```

Displays to the terminal the detailed decomposition of the *<token list>* into tokens, showing the category code of each character token, the meaning of control sequences and active characters, and the value of registers.

```
\tl_analysis_map_inline:nn
\tl_analysis_map_inline:Nn
```

New: 2018-04-09

```
\tl_analysis_map_inline:nn {<token list>} {<inline function>}
```

Applies the *<inline function>* to each individual *<token>* in the *<token list>*. The *<inline function>* receives three arguments:

- *<tokens>*, which both o-expand and x-expand to the *<token>*. The detailed form of *<token>* may change in later releases.
- *<char code>*, a decimal representation of the character code of the token, -1 if it is a control sequence (with *<catcode>* 0).
- *<catcode>*, a capital hexadecimal digit which denotes the category code of the *<token>* (0: control sequence, 1: begin-group, 2: end-group, 3: math shift, 4: alignment tab, 6: parameter, 7: superscript, 8: subscript, A: space, B: letter, C:other, D:active).

As all other mappings the mapping is done at the current group level, *i.e.* any local assignments made by the *<inline function>* remain in effect after the loop.

Part XXVIII

The `l3regex` package: Regular expressions in `TEX`

The `l3regex` package provides regular expression testing, extraction of submatches, splitting, and replacement, all acting on token lists. The syntax of regular expressions is mostly a subset of the PCRE syntax (and very close to POSIX), with some additions due to the fact that `TEX` manipulates tokens rather than characters. For performance reasons, only a limited set of features are implemented. Notably, back-references are not supported.

Let us give a few examples. After

```
\tl_set:Nn \l_my_tl { That~cat. }
\regex_replace_once:nnN { at } { is } \l_my_tl
```

the token list variable `\l_my_tl` holds the text “`This cat.`”, where the first occurrence of “`at`” was replaced by “`is`”. A more complicated example is a pattern to emphasize each word and add a comma after it:

```
\regex_replace_all:nnN { \w+ } { \c{emph}\cB\{ \0 \cE\} , } \l_my_tl
```

The `\w` sequence represents any “word” character, and `+` indicates that the `\w` sequence should be repeated as many times as possible (at least once), hence matching a word in the input token list. In the replacement text, `\0` denotes the full match (here, a word). The command `\emph` is inserted using `\c{emph}`, and its argument `\0` is put between braces `\cB\{` and `\cE\}`.

If a regular expression is to be used several times, it can be compiled once, and stored in a regex variable using `\regex_const:Nn`. For example,

```
\regex_const:Nn \c_foo_regex { \c{begin} \cB. (\c[~BE].*) \cE. }
```

stores in `\c_foo_regex` a regular expression which matches the starting marker for an environment: `\begin`, followed by a begin-group token (`\cB.`), then any number of tokens which are neither begin-group nor end-group character tokens (`\c[~BE].*`), ending with an end-group token (`\cE.`). As explained in the next section, the parentheses “capture” the result of `\c[~BE].*`, giving us access to the name of the environment when doing replacements.

1 Syntax of regular expressions

We start with a few examples, and encourage the reader to apply `\regex_show:n` to these regular expressions.

- `Cat` matches the word “Cat” capitalized in this way, but also matches the beginning of the word “Cattle”: use `\bCat\b` to match a complete word only.
- `[abc]` matches one letter among “a”, “b”, “c”; the pattern `(a|b|c)` matches the same three possible letters (but see the discussion of submatches below).
- `[A-Za-z]*` matches any number (due to the quantifier `*`) of Latin letters (not accented).

- `\c{[A-Za-z]*}` matches a control sequence made of Latin letters.
- `_[^_]*_` matches an underscore, any number of characters other than underscore, and another underscore; it is equivalent to `_.*?_` where `.` matches arbitrary characters and the lazy quantifier `*?` means to match as few characters as possible, thus avoiding matching underscores.
- `[\+|-]?[d+]` matches an explicit integer with at most one sign.
- `[\+|-_]*[d+_]*` matches an explicit integer with any number of `+` and `-` signs, with spaces allowed except within the mantissa, and surrounded by spaces.
- `[\+|-_]*(\d+|\d*\.\d+)__*` matches an explicit integer or decimal number; using `[.,]` instead of `\.` would allow the comma as a decimal marker.
- `[\+|-_]*(\d+|\d*\.\d+)__*((?i)pt|in|[cem]m|ex|[bs]p|[dn]d|[pcn]c)__*` matches an explicit dimension with any unit that T_EX knows, where `(?i)` means to treat lowercase and uppercase letters identically.
- `[\+|-_]*((?i)nan|inf|(\d+|\d*\.\d+)__*(e[\+|-_]*/\d+)__*)` matches an explicit floating point number or the special values `nan` and `inf` (with signs and spaces allowed).
- `[\+|-_]*(\d+|\dC.)__*` matches an explicit integer or control sequence (without checking whether it is an integer variable).
- `\G.*?\K` at the beginning of a regular expression matches and discards (due to `\K`) everything between the end of the previous match (`\G`) and what is matched by the rest of the regular expression; this is useful in `\regex_replace_all:nnN` when the goal is to extract matches or submatches in a finer way than with `\regex_extract_all:nnN`.

While it is impossible for a regular expression to match only integer expressions, `[\+|-\(\)*\d+\)\(\[\+|-*/ \(\)*\d+\)\(\)*` matches among other things all valid integer expressions (made only with explicit integers). One should follow it with further testing.

Most characters match exactly themselves, with an arbitrary category code. Some characters are special and must be escaped with a backslash (*e.g.*, `*` matches a star character). Some escape sequences of the form backslash-letter also have a special meaning (for instance `\d` matches any digit). As a rule,

- every alphanumeric character (`A-Z`, `a-z`, `0-9`) matches exactly itself, and should not be escaped, because `\A`, `\B`, ... have special meanings;
- non-alphanumeric printable ascii characters can (and should) always be escaped: many of them have special meanings (*e.g.*, use `\(`, `\)`, `\?`, `\.`);
- spaces should always be escaped (even in character classes);
- any other character may be escaped or not, without any effect: both versions match exactly that character.

Note that these rules play nicely with the fact that many non-alphanumeric characters are difficult to input into T_EX under normal category codes. For instance, `\abc%` matches the characters `\abc%` (with arbitrary category codes), but does not match the control sequence `\abc` followed by a percent character. Matching control sequences can be done using the `\c{<regex>}` syntax (see below).

Any special character which appears at a place where its special behaviour cannot apply matches itself instead (for instance, a quantifier appearing at the beginning of a string), after raising a warning.

Characters.

`\x{hh...}` Character with hex code `hh...`

`\xhh` Character with hex code `hh`.

`\a` Alarm (hex 07).

`\e` Escape (hex 1B).

`\f` Form-feed (hex 0C).

`\n` New line (hex 0A).

`\r` Carriage return (hex 0D).

`\t` Horizontal tab (hex 09).

Character types.

`.` A single period matches any token.

`\d` Any decimal digit.

`\h` Any horizontal space character, equivalent to `[\ \^^I]`: space and tab.

`\s` Any space character, equivalent to `[\ \^^I\^^J\^^L\^^M]`.

`\v` Any vertical space character, equivalent to `[\^^J\^^K\^^L\^^M]`. Note that `\^^K` is a vertical space, but not a space, for compatibility with Perl.

`\w` Any word character, *i.e.*, alphanumerics and underscore, equivalent to the explicit class `[A-Za-z0-9_]`.

`\D` Any token not matched by `\d`.

`\H` Any token not matched by `\h`.

`\N` Any token other than the `\n` character (hex 0A).

`\S` Any token not matched by `\s`.

`\V` Any token not matched by `\v`.

`\W` Any token not matched by `\w`.

Of those, `.`, `\D`, `\H`, `\N`, `\S`, `\V`, and `\W` match arbitrary control sequences.

Character classes match exactly one token in the subject.

`[...]` Positive character class. Matches any of the specified tokens.

[**^...**] Negative character class. Matches any token other than the specified characters.

x-y Within a character class, this denotes a range (can be used with escaped characters).

[:**<name>**:] Within a character class (one more set of brackets), this denotes the POSIX character class **<name>**, which can be **alnum**, **alpha**, **ascii**, **blank**, **cntrl**, **digit**, **graph**, **lower**, **print**, **punct**, **space**, **upper**, **word**, or **xdigit**.

[:**~<name>**:] Negative POSIX character class.

For instance, [**a-oq-z\cC.**] matches any lowercase latin letter except **p**, as well as control sequences (see below for a description of **\c**).

Quantifiers (repetition).

? 0 or 1, greedy.

?? 0 or 1, lazy.

***** 0 or more, greedy.

***?** 0 or more, lazy.

+ 1 or more, greedy.

+? 1 or more, lazy.

{n} Exactly *n*.

{n,} *n* or more, greedy.

{n,}? *n* or more, lazy.

{n,m} At least *n*, no more than *m*, greedy.

{n,m}? At least *n*, no more than *m*, lazy.

Anchors and simple assertions.

\b Word boundary: either the previous token is matched by **\w** and the next by **\W**, or the opposite. For this purpose, the ends of the token list are considered as **\W**.

\B Not a word boundary: between two **\w** tokens or two **\W** tokens (including the boundary).

^ or **\A** Start of the subject token list.

\$, **\Z** or **\z** End of the subject token list.

\G Start of the current match. This is only different from **^** in the case of multiple matches: for instance **\regex_count:nnN { \G a } { aaba } \l_tmpa_int** yields 2, but replacing **\G** by **^** would result in **\l_tmpa_int** holding the value 1.

Alternation and capturing groups.

A|B|C Either one of **A**, **B**, or **C**.

(...) Capturing group.

(?:...) Non-capturing group.

(?<|...)) Non-capturing group which resets the group number for capturing groups in each alternative. The following group is numbered with the first unused group number.

The `\c` escape sequence allows to test the category code of tokens, and match control sequences. Each character category is represented by a single uppercase letter:

- C for control sequences;
- B for begin-group tokens;
- E for end-group tokens;
- M for math shift;
- T for alignment tab tokens;
- P for macro parameter tokens;
- U for superscript tokens (up);
- D for subscript tokens (down);
- S for spaces;
- L for letters;
- O for others; and
- A for active characters.

The `\c` escape sequence is used as follows.

`\c{<regex>}` A control sequence whose *cname* matches the *<regex>*, anchored at the beginning and end, so that `\c{begin}` matches exactly `\begin`, and nothing else.

`\cX` Applies to the next object, which can be a character, character property, class, or group, and forces this object to only match tokens with category **X** (any of CBEMTPUDSLOA). For instance, `\cL[A-Z\d]` matches uppercase letters and digits of category code letter, `\cC.` matches any control sequence, and `\cO(abc)` matches `abc` where each character has category other.

`\c[XYZ]` Applies to the next object, and forces it to only match tokens with category **X**, **Y**, or **Z** (each being any of CBEMTPUDSLOA). For instance, `\c[LSO](..)` matches two tokens of category letter, space, or other.

`\c[^XYZ]` Applies to the next object and prevents it from matching any token with category **X**, **Y**, or **Z** (each being any of CBEMTPUDSLOA). For instance, `\c[^O]\d` matches digits which have any category different from other.

The category code tests can be used inside classes; for instance, `[\cO\d \c[LO][A-F]]` matches what \TeX considers as hexadecimal digits, namely digits with category other, or uppercase letters from **A** to **F** with category either letter or other. Within a group affected by a category code test, the outer test can be overridden by a nested test: for instance, `\cL(ab\cO*cd)` matches `ab*cd` where all characters are of category letter, except `*` which has category other.

The `\u` escape sequence allows to insert the contents of a token list directly into a regular expression or a replacement, avoiding the need to escape special characters.

Namely, `\u{<tl var name>}` matches the exact contents of the token list `<tl var>`. Within a `\c{...}` control sequence matching, the `\u` escape sequence only expands its argument once, in effect performing `\tl_to_str:v`. Quantifiers are not supported directly: use a group.

The option `(?i)` makes the match case insensitive (identifying A–Z with a–z; no Unicode support yet). This applies until the end of the group in which it appears, and can be reverted using `(?-i)`. For instance, in `(?i)(a(?-i)b|c)d`, the letters `a` and `d` are affected by the `i` option. Characters within ranges and classes are affected individually: `(?i)[Y-\]` is equivalent to `[YZ\[\]yz]`, and `(?i)[^aeiou]` matches any character which is not a vowel. Neither character properties, nor `\c{...}` nor `\u{...}` are affected by the `i` option.

In character classes, only `[`, `^`, `-`, `]`, `\` and spaces are special, and should be escaped. Other non-alphanumeric characters can still be escaped without harm. Any escape sequence which matches a single character (`\d`, `\D`, *etc.*) is supported in character classes. If the first character is `^`, then the meaning of the character class is inverted; `^` appearing anywhere else in the range is not special. If the first character (possibly following a leading `^`) is `]` then it does not need to be escaped since ending the range there would make it empty. Ranges of characters can be expressed using `-`, for instance, `[\D 0–5]` and `[^6–9]` are equivalent.

Capturing groups are a means of extracting information about the match. Parenthesized groups are labelled in the order of their opening parenthesis, starting at 1. The contents of those groups corresponding to the “best” match (leftmost longest) can be extracted and stored in a sequence of token lists using for instance `\regex_extract_once:nnTF`.

The `\K` escape sequence resets the beginning of the match to the current position in the token list. This only affects what is reported as the full match. For instance,

```
\regex_extract_all:nnN { a \K . } { a123aaxyz } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{1}` and `{a}`: the true matches are `{a1}` and `{aa}`, but they are trimmed by the use of `\K`. The `\K` command does not affect capturing groups: for instance,

```
\regex_extract_once:nnN { (. \K c)+ \d } { acbc3 } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{c3}` and `{bc}`: the true match is `{acbc3}`, with first submatch `{bc}`, but `\K` resets the beginning of the match to the last position where it appears.

2 Syntax of the replacement text

Most of the features described in regular expressions do not make sense within the replacement text. Backslash introduces various special constructions, described further below:

- `\0` is the whole match;
- `\1` is the submatch that was matched by the first (capturing) group `(...)`; similarly for `\2`, ..., `\9` and `\g{<number>}`;
- `_` inserts a space (spaces are ignored when not escaped);

- `\a, \e, \f, \n, \r, \t, \xhh, \x{hhh}` correspond to single characters as in regular expressions;
- `\c{<cs name>}` inserts a control sequence;
- `\c{<category>}<character>` (see below);
- `\u{<tl var name>}` inserts the contents of the `<tl var>` (see below).

Characters other than backslash and space are simply inserted in the result (but since the replacement text is first converted to a string, one should also escape characters that are special for T_EX, for instance use `\#`). Non-alphanumeric characters can always be safely escaped with a backslash.

For instance,

```
\tl_set:Nn \l_my_tl { Hello,~world! }
\regex_replace_all:nnN { ([er]?l|o) . } { (\0--\1) } \l_my_tl
```

results in `\l_my_tl` holding `H(e1l--e1)(o,--o) w(or--o)(ld--l)!`

The submatches are numbered according to the order in which the opening parenthesis of capturing groups appear in the regular expression to match. The n -th submatch is empty if there are fewer than n capturing groups or for capturing groups that appear in alternatives that were not used for the match. In case a capturing group matches several times during a match (due to quantifiers) only the last match is used in the replacement text. Submatches always keep the same category codes as in the original token list.

The characters inserted by the replacement have category code 12 (other) by default, with the exception of space characters. Spaces inserted through `_` have category code 10, while spaces inserted through `\x20` or `\x{20}` have category code 12. The escape sequence `\c` allows to insert characters with arbitrary category codes, as well as control sequences.

`\cX(...)` Produces the characters “...” with category `X`, which must be one of `CBEMTPUDSLOA` as in regular expressions. Parentheses are optional for a single character (which can be an escape sequence). When nested, the innermost category code applies, for instance `\cL(Hello\cS\ world)!` gives this text with standard category codes.

`\c{<text>}` Produces the control sequence with csname `<text>`. The `<text>` may contain references to the submatches `\0`, `\1`, and so on, as in the example for `\u` below.

The escape sequence `\u{<tl var name>}` allows to insert the contents of the token list with name `<tl var name>` directly into the replacement, giving an easier control of category codes. When nested in `\c{...}` and `\u{...}` constructions, the `\u` and `\c` escape sequences perform `\tl_to_str:v`, namely extract the value of the control sequence and turn it into a string. Matches can also be used within the arguments of `\c` and `\u`. For instance,

```
\tl_set:Nn \l_my_one_tl { first }
\tl_set:Nn \l_my_two_tl { \emph{second} }
\tl_set:Nn \l_my_tl { one , two , one , one }
\regex_replace_all:nnN { [,]+ } { \u{1_my_\0_tl} } \l_my_tl
```

results in `\l_my_tl` holding `first,\emph{second},first,first`.

3 Pre-compiling regular expressions

If a regular expression is to be used several times, it is better to compile it once rather than doing it each time the regular expression is used. The compiled regular expression is stored in a variable. All of the `l3regex` module's functions can be given their regular expression argument either as an explicit string or as a compiled regular expression.

`\regex_new:N`

New: 2017-05-26

`\regex_new:N` $\langle regex\ var \rangle$

Creates a new $\langle regex\ var \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle regex\ var \rangle$ is initially such that it never matches.

`\regex_set:Nn`
`\regex_gset:Nn`
`\regex_const:Nn`

New: 2017-05-26

`\regex_set:Nn` $\langle regex\ var \rangle$ $\{ \langle regex \rangle \}$

Stores a compiled version of the $\langle regular\ expression \rangle$ in the $\langle regex\ var \rangle$. For instance, this function can be used as

```
\regex_new:N \l_my_regex
\regex_set:Nn \l_my_regex { my\ (simple\ )? reg(ex|ular\ expression) }
```

The assignment is local for `\regex_set:Nn` and global for `\regex_gset:Nn`. Use `\regex_const:Nn` for compiled expressions which never change.

`\regex_show:n`
`\regex_show:N`

New: 2017-05-26

`\regex_show:n` $\{ \langle regex \rangle \}$

Shows how `l3regex` interprets the $\langle regex \rangle$. For instance, `\regex_show:n { \A X|Y }` shows

```
+--branch
  anchor at start (\A)
  char code 88
+--branch
  char code 89
```

indicating that the anchor `\A` only applies to the first branch: the second branch is not anchored to the beginning of the match.

4 Matching

All regular expression functions are available in both `:n` and `:N` variants. The former require a “standard” regular expression, while the later require a compiled expression as generated by `\regex_(g)set:Nn`.

`\regex_match:nnTF`
`\regex_match:NnTF`

New: 2017-05-26

`\regex_match:nnTF` $\{ \langle regex \rangle \}$ $\{ \langle token\ list \rangle \}$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$

Tests whether the $\langle regular\ expression \rangle$ matches any part of the $\langle token\ list \rangle$. For instance,

```
\regex_match:nnTF { b [cde]* } { abecdxcx } { TRUE } { FALSE }
\regex_match:nnTF { [b-dq-w] } { example } { TRUE } { FALSE }
```

leaves TRUE then FALSE in the input stream.

```
\regex_count:nnN
\regex_count:NnN
```

New: 2017-05-26

```
\regex_count:nnN {<regex>} {<token list>} {<int var>}
```

Sets *<int var>* within the current T_EX group level equal to the number of times *<regular expression>* appears in *<token list>*. The search starts by finding the left-most longest match, respecting greedy and lazy (non-greedy) operators. Then the search starts again from the character following the last character of the previous match, until reaching the end of the token list. Infinite loops are prevented in the case where the regular expression can match an empty token list: then we count one match between each pair of characters. For instance,

```
\int_new:N \l_foo_int
\regex_count:nnN { (b+|c) } { abbababcb } \l_foo_int
```

results in `\l_foo_int` taking the value 5.

5 Submatch extraction

```
\regex_extract_once:nnN
\regex_extract_once:nnNTF
\regex_extract_once:NnN
\regex_extract_once:NnNTF
```

New: 2017-05-26

```
\regex_extract_once:nnN {<regex>} {<token list>} {<seq var>}
\regex_extract_once:nnNTF {<regex>} {<token list>} {<seq var>} {<true code>} {<false code>}
```

Finds the first match of the *<regular expression>* in the *<token list>*. If it exists, the match is stored as the first item of the *<seq var>*, and further items are the contents of capturing groups, in the order of their opening parenthesis. The *<seq var>* is assigned locally. If there is no match, the *<seq var>* is cleared. The testing versions insert the *<true code>* into the input stream if a match was found, and the *<false code>* otherwise.

For instance, assume that you type

```
\regex_extract_once:nnNTF { \A(La)?TeX(!*)\Z } { LaTeX!!! } \l_foo_seq
{ true } { false }
```

Then the regular expression (anchored at the start with `\A` and at the end with `\Z`) must match the whole token list. The first capturing group, `(La)?`, matches `La`, and the second capturing group, `(!*)`, matches `!!!`. Thus, `\l_foo_seq` contains as a result the items `{LaTeX!!!}`, `{La}`, and `{!!!}`, and the `true` branch is left in the input stream. Note that the n -th item of `\l_foo_seq`, as obtained using `\seq_item:Nn`, correspond to the submatch numbered $(n - 1)$ in functions such as `\regex_replace_once:nnN`.

```
\regex_extract_all:nnN
\regex_extract_all:nnNTF
\regex_extract_all:NnN
\regex_extract_all:NnNTF
```

New: 2017-05-26

```
\regex_extract_all:nnN {<regex>} {<token list>} {<seq var>}
\regex_extract_all:nnNTF {<regex>} {<token list>} {<seq var>} {<true code>} {<false code>}
```

Finds all matches of the *<regular expression>* in the *<token list>*, and stores all the submatch information in a single sequence (concatenating the results of multiple `\regex_extract_once:nnN` calls). The *<seq var>* is assigned locally. If there is no match, the *<seq var>* is cleared. The testing versions insert the *<true code>* into the input stream if a match was found, and the *<false code>* otherwise. For instance, assume that you type

```
\regex_extract_all:nnNTF { \w+ } { Hello,~world! } \l_foo_seq
{ true } { false }
```

Then the regular expression matches twice, the resulting sequence contains the two items `{Hello}` and `{world}`, and the `true` branch is left in the input stream.

```
\regex_split:nnN
\regex_split:nnNTF
\regex_split:NnN
\regex_split:NnNTF
```

New: 2017-05-26

```
\regex_split:nnN {<regular expression>} {<token list>} <seq var>
\regex_split:nnNTF {<regular expression>} {<token list>} <seq var> {<true code>}
{<false code>}
```

Splits the *<token list>* into a sequence of parts, delimited by matches of the *<regular expression>*. If the *<regular expression>* has capturing groups, then the token lists that they match are stored as items of the sequence as well. The assignment to *<seq var>* is local. If no match is found the resulting *<seq var>* has the *<token list>* as its sole item. If the *<regular expression>* matches the empty token list, then the *<token list>* is split into single tokens. The testing versions insert the *<true code>* into the input stream if a match was found, and the *<false code>* otherwise. For example, after

```
\seq_new:N \l_path_seq
\regex_split:nnNTF { / } { the/path/for/this/file.tex } \l_path_seq
{ true } { false }
```

the sequence `\l_path_seq` contains the items `{the}`, `{path}`, `{for}`, `{this}`, and `{file.tex}`, and the `true` branch is left in the input stream.

6 Replacement

```
\regex_replace_once:nnN
\regex_replace_once:nnNTF
\regex_replace_once:NnN
\regex_replace_once:NnNTF
```

New: 2017-05-26

```
\regex_replace_once:nnN {<regular expression>} {<replacement>} <tl var>
\regex_replace_once:nnNTF {<regular expression>} {<replacement>} <tl var> {<true
code>} {<false code>}
```

Searches for the *<regular expression>* in the *<token list>* and replaces the first match with the *<replacement>*. The result is assigned locally to *<tl var>*. In the *<replacement>*, `\0` represents the full match, `\1` represent the contents of the first capturing group, `\2` of the second, *etc.*

```
\regex_replace_all:nnN
\regex_replace_all:nnNTF
\regex_replace_all:NnN
\regex_replace_all:NnNTF
```

New: 2017-05-26

```
\regex_replace_all:nnN {<regular expression>} {<replacement>} <tl var>
\regex_replace_all:nnNTF {<regular expression>} {<replacement>} <tl var> {<true
code>} {<false code>}
```

Replaces all occurrences of the *<regular expression>* in the *<token list>* by the *<replacement>*, where `\0` represents the full match, `\1` represent the contents of the first capturing group, `\2` of the second, *etc.* Every match is treated independently, and matches cannot overlap. The result is assigned locally to *<tl var>*.

7 Constants and variables

```
\l_tmpa_regex
\l_tmpb_regex
```

New: 2017-12-11

Scratch regex for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

```
\g_tmpa_regex
\g_tmpb_regex
```

New: 2017-12-11

Scratch regex for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

8 Bugs, misfeatures, future work, and other possibilities

The following need to be done now.

- Rewrite the documentation in a more ordered way, perhaps add a BNF?

Additional error-checking to come.

- Clean up the use of messages.
- Cleaner error reporting in the replacement phase.
- Add tracing information.
- Detect attempts to use back-references and other non-implemented syntax.
- Test for the maximum register `\c_max_register_int`.
- Find out whether the fact that `\W` and friends match the end-marker leads to bugs. Possibly update `__regex_item_reverse:n`.
- The empty `cs` should be matched by `\c{}`, not by `\c{csname.?endcsname\s?}`.

Code improvements to come.

- Shift arrays so that the useful information starts at position 1.
- Only build `\c{...}` once.
- Use arrays for the left and right state stacks when compiling a regex.
- Should `__regex_action_free_group:n` only be used for greedy `{n,}` quantifier? (I think not.)
- Quantifiers for `\u` and assertions.
- When matching, keep track of an explicit stack of `current_state` and `current_submatches`.
- If possible, when a state is reused by the same thread, kill other subthreads.
- Use an array rather than `\l__regex_balance_tl` to build the function `__regex_replacement_balance_one_match:n`.
- Reduce the number of epsilon-transitions in alternatives.
- Optimize simple strings: use less states (`abcade` should give two states, for `abc` and `ade`). [Does that really make sense?]
- Optimize groups with no alternative.
- Optimize states with a single `__regex_action_free:n`.
- Optimize the use of `__regex_action_success:` by inserting it in state 2 directly instead of having an extra transition.
- Optimize the use of `\int_step_...` functions.

- Groups don't capture within regexes for csnames; optimize and document.
- Better “show” for anchors, properties, and catcode tests.
- Does \K really need a new state for itself?
- When compiling, use a boolean `in_cs` and less magic numbers.
- Instead of checking whether the character is special or alphanumeric using its character code, check if it is special in regexes with `\cs_if_exist` tests.

The following features are likely to be implemented at some point in the future.

- General look-ahead/behind assertions.
- Regex matching on external files.
- Conditional subpatterns with look ahead/behind: “if what follows is [...], then [...]”.
- `(*..)` and `(?..)` sequences to set some options.
- UTF-8 mode for pdfTeX.
- Newline conventions are not done. In particular, we should have an option for `.` not to match newlines. Also, `\A` should differ from `^`, and `\Z`, `\z` and `$` should differ.
- Unicode properties: `\p{..}` and `\P{..}`; `\X` which should match any “extended” Unicode sequence. This requires to manipulate a lot of data, probably using tree-boxes.
- Provide a syntax such as `\ur{1_my_regex}` to use an already-compiled regex in a more complicated regex. This makes regexes more easily composable.
- Allowing `\u{1_my_t1}` in more places, for instance as the number of repetitions in a quantifier.

The following features of PCRE or Perl may or may not be implemented.

- Callout with `(?C...)` or other syntax: some internal code changes make that possible, and it can be useful for instance in the replacement code to stop a regex replacement when some marker has been found; this raises the question of a potential `\regex_break`: and then of playing well with `\t1_map_break`: called from within the code in a regex. It also raises the question of nested calls to the regex machinery, which is a problem since `\fontdimen` are global.
- Conditional subpatterns (other than with a look-ahead or look-behind condition): this is non-regular, isn't it?
- Named subpatterns: TeX programmers have lived so far without any need for named macro parameters.

The following features of PCRE or Perl will definitely not be implemented.

- Back-references: non-regular feature, this requires backtracking, which is prohibitively slow.

- Recursion: this is a non-regular feature.
- Atomic grouping, possessive quantifiers: those tools, mostly meant to fix catastrophic backtracking, are unnecessary in a non-backtracking algorithm, and difficult to implement.
- Subroutine calls: this syntactic sugar is difficult to include in a non-backtracking algorithm, in particular because the corresponding group should be treated as atomic.
- Backtracking control verbs: intrinsically tied to backtracking.
- `\ddd`, matching the character with octal code `ddd`: we already have `\x{...}` and the syntax is confusingly close to what we could have used for backreferences (`\1`, `\2`, ...), making it harder to produce useful error message.
- `\cx`, similar to $\mathrm{T}_{\mathrm{E}}\mathrm{X}$'s own `\^^x`.
- Comments: $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ already has its own system for comments.
- `\Q...\E` escaping: this would require to read the argument verbatim, which is not in the scope of this module.
- `\C` single byte in UTF-8 mode: $\mathrm{X}_{\mathrm{Y}}\mathrm{T}_{\mathrm{E}}\mathrm{X}$ and $\mathrm{LuaT}_{\mathrm{E}}\mathrm{X}$ serve us characters directly, and splitting those into bytes is tricky, encoding dependent, and most likely not useful anyways.

Part XXIX

The l3box package

Boxes

There are three kinds of box operations: horizontal mode denoted with prefix `\hbox_`, vertical mode with prefix `\vbox_`, and the generic operations working in both modes with prefix `\box_`.

1 Creating and initialising boxes

<code>\box_new:N</code>	<code>\box_new:N <box></code>
<code>\box_new:c</code>	Creates a new <code><box></code> or raises an error if the name is already taken. The declaration is global. The <code><box></code> is initially void.

<code>\box_clear:N</code>	<code>\box_clear:N <box></code>
<code>\box_clear:c</code>	Clears the content of the <code><box></code> by setting the box equal to <code>\c_empty_box</code> .
<code>\box_gclear:N</code>	
<code>\box_gclear:c</code>	

<code>\box_clear_new:N</code>	<code>\box_clear_new:N <box></code>
<code>\box_clear_new:c</code>	Ensures that the <code><box></code> exists globally by applying <code>\box_new:N</code> if necessary, then applies <code>\box_(g)clear:N</code> to leave the <code><box></code> empty.
<code>\box_gclear_new:N</code>	
<code>\box_gclear_new:c</code>	

<code>\box_set_eq:NN</code>	<code>\box_set_eq:NN <box₁> <box₂></code>
<code>\box_set_eq:(cN Nc cc)</code>	Sets the content of <code><box₁></code> equal to that of <code><box₂></code> .
<code>\box_gset_eq:NN</code>	
<code>\box_gset_eq:(cN Nc cc)</code>	

<code>\box_if_exist_p:N *</code>	<code>\box_if_exist_p:N <box></code>
<code>\box_if_exist_p:c *</code>	<code>\box_if_exist:NTF <box> {(true code)} {(false code)}</code>
<code>\box_if_exist:NTF *</code>	Tests whether the <code><box></code> is currently defined. This does not check that the <code><box></code> really is a box.
<code>\box_if_exist:cTF *</code>	

New: 2012-03-03

2 Using boxes

<code>\box_use:N</code>	<code>\box_use:N <box></code>
<code>\box_use:c</code>	Inserts the current content of the <code><box></code> onto the current list for typesetting. An error is raised if the variable does not exist or if it is invalid.

T_EXhackers note: This is the T_EX primitive `\copy`.

<hr/> <code>\box_move_right:nn</code> <hr/>	<code>\box_move_right:nn {<dimexpr>} {<box function>}</code>
<code>\box_move_left:nn</code> <hr/>	This function operates in vertical mode, and inserts the material specified by the $\langle box \text{ function} \rangle$ such that its reference point is displaced horizontally by the given $\langle dimexpr \rangle$ from the reference point for typesetting, to the right or left as appropriate. The $\langle box \text{ function} \rangle$ should be a box operation such as <code>\box_use:N \<box></code> or a “raw” box specification such as <code>\vbox:n { xyz }</code> .

<hr/> <code>\box_move_up:nn</code> <hr/>	<code>\box_move_up:nn {<dimexpr>} {<box function>}</code>
<code>\box_move_down:nn</code> <hr/>	This function operates in horizontal mode, and inserts the material specified by the $\langle box \text{ function} \rangle$ such that its reference point is displaced vertically by the given $\langle dimexpr \rangle$ from the reference point for typesetting, up or down as appropriate. The $\langle box \text{ function} \rangle$ should be a box operation such as <code>\box_use:N \<box></code> or a “raw” box specification such as <code>\vbox:n { xyz }</code> .

3 Measuring and setting box dimensions

<hr/> <code>\box_dp:N</code> <hr/>	<code>\box_dp:N <box></code>
<code>\box_dp:c</code> <hr/>	Calculates the depth (below the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension \text{ expression} \rangle$.

TeXhackers note: This is the TeX primitive `\dp`.

<hr/> <code>\box_ht:N</code> <hr/>	<code>\box_ht:N <box></code>
<code>\box_ht:c</code> <hr/>	Calculates the height (above the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension \text{ expression} \rangle$.

TeXhackers note: This is the TeX primitive `\ht`.

<hr/> <code>\box_wd:N</code> <hr/>	<code>\box_wd:N <box></code>
<code>\box_wd:c</code> <hr/>	Calculates the width of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension \text{ expression} \rangle$.

TeXhackers note: This is the TeX primitive `\wd`.

<hr/> <code>\box_set_dp:Nn</code> <hr/>	<code>\box_set_dp:Nn <box> {<dimension expression>}</code>
<code>\box_set_dp:cn</code> <hr/>	Set the depth (below the baseline) of the $\langle box \rangle$ to the value of the $\{<dimension \text{ expression}>\}$.
<code>\box_gset_dp:Nn</code> <hr/>	
<code>\box_gset_dp:cn</code> <hr/>	

Updated: 2019-01-22

<hr/> <code>\box_set_ht:Nn</code> <hr/>	<code>\box_set_ht:Nn <box> {<dimension expression>}</code>
<code>\box_set_ht:cn</code> <hr/>	Set the height (above the baseline) of the $\langle box \rangle$ to the value of the $\{<dimension \text{ expression}>\}$.
<code>\box_gset_ht:Nn</code> <hr/>	
<code>\box_gset_ht:cn</code> <hr/>	

Updated: 2019-01-22

<code>\box_set_wd:Nn</code>	<code>\box_set_wd:Nn <box> {<dimension expression>}</code>
<code>\box_set_wd:cn</code>	
<code>\box_gset_wd:Nn</code>	Set the width of the <code><box></code> to the value of the <code>{<dimension expression>}</code> .
<code>\box_gset_wd:cn</code>	

Updated: 2019-01-22

4 Box conditionals

<code>\box_if_empty_p:N</code> *	<code>\box_if_empty_p:N <box></code>
<code>\box_if_empty_p:c</code> *	<code>\box_if_empty:NTF <box> {<true code>} {<false code>}</code>
<code>\box_if_empty:NTF</code> *	
<code>\box_if_empty:cTF</code> *	Tests if <code><box></code> is a empty (equal to <code>\c_empty_box</code>).

<code>\box_if_horizontal_p:N</code> *	<code>\box_if_horizontal_p:N <box></code>
<code>\box_if_horizontal_p:c</code> *	<code>\box_if_horizontal:NTF <box> {<true code>} {<false code>}</code>
<code>\box_if_horizontal:NTF</code> *	
<code>\box_if_horizontal:cTF</code> *	Tests if <code><box></code> is a horizontal box.

<code>\box_if_vertical_p:N</code> *	<code>\box_if_vertical_p:N <box></code>
<code>\box_if_vertical_p:c</code> *	<code>\box_if_vertical:NTF <box> {<true code>} {<false code>}</code>
<code>\box_if_vertical:NTF</code> *	
<code>\box_if_vertical:cTF</code> *	Tests if <code><box></code> is a vertical box.

5 The last box inserted

<code>\box_set_to_last:N</code>	<code>\box_set_to_last:N <box></code>
<code>\box_set_to_last:c</code>	
<code>\box_gset_to_last:N</code>	Sets the <code><box></code> equal to the last item (box) added to the current partial list, removing the item from the list at the same time. When applied to the main vertical list, the <code><box></code> is always void as it is not possible to recover the last added item.
<code>\box_gset_to_last:c</code>	

6 Constant boxes

<code>\c_empty_box</code>	This is a permanently empty box, which is neither set as horizontal nor vertical.
---------------------------	---

Updated: 2012-11-04

T_EXhackers note: At the T_EX level this is a void box.

7 Scratch boxes

`\l_tmpa_box`
`\l_tmpb_box`

Updated: 2012-11-04

Scratch boxes for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_box`
`\g_tmpb_box`

Scratch boxes for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

8 Viewing box contents

`\box_show:N`
`\box_show:c`

Updated: 2012-05-11

`\box_show:N` $\langle box \rangle$

Shows full details of the content of the $\langle box \rangle$ in the terminal.

`\box_show:Nnn`
`\box_show:cnn`

New: 2012-05-11

`\box_show:Nnn` $\langle box \rangle$ $\{\langle intexpr_1 \rangle\}$ $\{\langle intexpr_2 \rangle\}$

Display the contents of $\langle box \rangle$ in the terminal, showing the first $\langle intexpr_1 \rangle$ items of the box, and descending into $\langle intexpr_2 \rangle$ group levels.

`\box_log:N`
`\box_log:c`

New: 2012-05-11

`\box_log:N` $\langle box \rangle$

Writes full details of the content of the $\langle box \rangle$ to the log.

`\box_log:Nnn`
`\box_log:cnn`

New: 2012-05-11

`\box_log:Nnn` $\langle box \rangle$ $\{\langle intexpr_1 \rangle\}$ $\{\langle intexpr_2 \rangle\}$

Writes the contents of $\langle box \rangle$ to the log, showing the first $\langle intexpr_1 \rangle$ items of the box, and descending into $\langle intexpr_2 \rangle$ group levels.

9 Boxes and color

All L^AT_EX3 boxes are “color safe”: a color set inside the box stops applying after the end of the box has occurred.

10 Horizontal mode boxes

`\hbox:n`

Updated: 2017-04-05

`\hbox:n` $\{\langle contents \rangle\}$

Typesets the $\langle contents \rangle$ into a horizontal box of natural width and then includes this box in the current list for typesetting.

<hr/> <code>\hbox_to_wd:nn</code> <hr/>	<code>\hbox_to_wd:nn {<dimexpr>} {<contents>}</code>
Updated: 2017-04-05	Typesets the $\langle contents \rangle$ into a horizontal box of width $\langle dimexpr \rangle$ and then includes this box in the current list for typesetting.
<hr/> <code>\hbox_to_zero:n</code> <hr/>	<code>\hbox_to_zero:n {<contents>}</code>
Updated: 2017-04-05	Typesets the $\langle contents \rangle$ into a horizontal box of zero width and then includes this box in the current list for typesetting.
<hr/> <code>\hbox_set:Nn</code> <code>\hbox_set:cn</code> <code>\hbox_gset:Nn</code> <code>\hbox_gset:cn</code> <hr/>	<code>\hbox_set:Nn <box> {<contents>}</code> Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$.
Updated: 2017-04-05	
<hr/> <code>\hbox_set_to_wd:Nnn</code> <code>\hbox_set_to_wd:cnn</code> <code>\hbox_gset_to_wd:Nnn</code> <code>\hbox_gset_to_wd:cnn</code> <hr/>	<code>\hbox_set_to_wd:Nnn <box> {<dimexpr>} {<contents>}</code> Typesets the $\langle contents \rangle$ to the width given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$.
Updated: 2017-04-05	
<hr/> <code>\hbox_overlap_right:n</code> <hr/>	<code>\hbox_overlap_right:n {<contents>}</code>
Updated: 2017-04-05	Typesets the $\langle contents \rangle$ into a horizontal box of zero width such that material protrudes to the right of the insertion point.
<hr/> <code>\hbox_overlap_left:n</code> <hr/>	<code>\hbox_overlap_left:n {<contents>}</code>
Updated: 2017-04-05	Typesets the $\langle contents \rangle$ into a horizontal box of zero width such that material protrudes to the left of the insertion point.
<hr/> <code>\hbox_set:Nw</code> <code>\hbox_set:cw</code> <code>\hbox_set_end:</code> <code>\hbox_gset:Nw</code> <code>\hbox_gset:cw</code> <code>\hbox_gset_end:</code> <hr/>	<code>\hbox_set:Nw <box> <contents> \hbox_set_end:</code> Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$. In contrast to <code>\hbox_set:Nn</code> this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument.
Updated: 2017-04-05	
<hr/> <code>\hbox_set_to_wd:Nnw</code> <code>\hbox_set_to_wd:cnw</code> <code>\hbox_gset_to_wd:Nnw</code> <code>\hbox_gset_to_wd:cnw</code> <hr/>	<code>\hbox_set_to_wd:Nnw <box> {<dimexpr>} <contents> \hbox_set_end:</code> Typesets the $\langle contents \rangle$ to the width given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$. In contrast to <code>\hbox_set_to_wd:Nnn</code> this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument
New: 2017-06-08	
<hr/> <code>\hbox_unpack:N</code> <code>\hbox_unpack:c</code> <hr/>	<code>\hbox_unpack:N <box></code> Unpacks the content of the horizontal $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set.

T_EXhackers note: This is the T_EX primitive `\unhcopy`.

11 Vertical mode boxes

Vertical boxes inherit their baseline from their contents. The standard case is that the baseline of the box is at the same position as that of the last item added to the box. This means that the box has no depth unless the last item added to it had depth. As a result most vertical boxes have a large height value and small or zero depth. The exception are `_top` boxes, where the reference point is that of the first item added. These tend to have a large depth and small height, although the latter is typically non-zero.

<hr/> <code>\vbox:n</code> <hr/>	<code>\vbox:n {⟨contents⟩}</code>
<hr/> Updated: 2017-04-05 <hr/>	Typesets the <code>⟨contents⟩</code> into a vertical box of natural height and includes this box in the current list for typesetting.
<hr/> <code>\vbox_top:n</code> <hr/>	<code>\vbox_top:n {⟨contents⟩}</code>
<hr/> Updated: 2017-04-05 <hr/>	Typesets the <code>⟨contents⟩</code> into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box is equal to that of the <i>first</i> item added to the box.
<hr/> <code>\vbox_to_ht:nn</code> <hr/>	<code>\vbox_to_ht:nn {⟨dimexpr⟩} {⟨contents⟩}</code>
<hr/> Updated: 2017-04-05 <hr/>	Typesets the <code>⟨contents⟩</code> into a vertical box of height <code>⟨dimexpr⟩</code> and then includes this box in the current list for typesetting.
<hr/> <code>\vbox_to_zero:n</code> <hr/>	<code>\vbox_to_zero:n {⟨contents⟩}</code>
<hr/> Updated: 2017-04-05 <hr/>	Typesets the <code>⟨contents⟩</code> into a vertical box of zero height and then includes this box in the current list for typesetting.
<hr/> <code>\vbox_set:Nn</code> <code>\vbox_set:cn</code> <code>\vbox_gset:Nn</code> <code>\vbox_gset:cn</code> <hr/>	<code>\vbox_set:Nn ⟨box⟩ {⟨contents⟩}</code> Typesets the <code>⟨contents⟩</code> at natural height and then stores the result inside the <code>⟨box⟩</code> .
<hr/> Updated: 2017-04-05 <hr/>	
<hr/> <code>\vbox_set_top:Nn</code> <code>\vbox_set_top:cn</code> <code>\vbox_gset_top:Nn</code> <code>\vbox_gset_top:cn</code> <hr/>	<code>\vbox_set_top:Nn ⟨box⟩ {⟨contents⟩}</code> Typesets the <code>⟨contents⟩</code> at natural height and then stores the result inside the <code>⟨box⟩</code> . The baseline of the box is equal to that of the <i>first</i> item added to the box.
<hr/> Updated: 2017-04-05 <hr/>	
<hr/> <code>\vbox_set_to_ht:Nnn</code> <code>\vbox_set_to_ht:cnn</code> <code>\vbox_gset_to_ht:Nnn</code> <code>\vbox_gset_to_ht:cnn</code> <hr/>	<code>\vbox_set_to_ht:Nnn ⟨box⟩ {⟨dimexpr⟩} {⟨contents⟩}</code> Typesets the <code>⟨contents⟩</code> to the height given by the <code>⟨dimexpr⟩</code> and then stores the result inside the <code>⟨box⟩</code> .
<hr/> Updated: 2017-04-05 <hr/>	

```

\ vbox_set:Nw
\ vbox_set:cw
\ vbox_set:end:
\ vbox_gset:Nw
\ vbox_gset:cw
\ vbox_gset:end:

```

Updated: 2017-04-05

```

\ vbox_set_to_ht:Nnw
\ vbox_set_to_ht:cnw
\ vbox_gset_to_ht:Nnw
\ vbox_gset_to_ht:cnw

```

New: 2017-06-08

`\vbox_set:Nw` $\langle box \rangle$ $\langle contents \rangle$ `\vbox_set:end:`

Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. In contrast to `\vbox_set:Nn` this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument.

`\vbox_set_to_ht:Nnw` $\langle box \rangle$ $\{\langle dimexpr \rangle\}$ $\langle contents \rangle$ `\vbox_set:end:`

Typesets the $\langle contents \rangle$ to the height given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$. In contrast to `\vbox_set_to_ht:Nnn` this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument

```

\ vbox_set_split_to_ht:NNn
\ vbox_set_split_to_ht:(cNn|Ncn|ccn)
\ vbox_gset_split_to_ht:NNn
\ vbox_gset_split_to_ht:(cNn|Ncn|ccn)

```

Updated: 2018-12-29

`\vbox_set_split_to_ht:NNn` $\langle box_1 \rangle$ $\langle box_2 \rangle$ $\{\langle dimexpr \rangle\}$

Sets $\langle box_1 \rangle$ to contain material to the height given by the $\langle dimexpr \rangle$ by removing content from the top of $\langle box_2 \rangle$ (which must be a vertical box).

```

\ vbox_unpack:N
\ vbox_unpack:c

```

`\vbox_unpack:N` $\langle box \rangle$

Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set.

T_EXhackers note: This is the T_EX primitive `\unvcopy`.

12 Using boxes efficiently

The functions above for using box contents work in exactly the same way as for any other `expl3` variable. However, for efficiency reasons, it is also useful to have functions which *drop* box contents on use. When a box is dropped, the box becomes empty at the group level *where the box was originally set* rather than necessarily *at the current group level*. For example, with

```

\ hbox_set:Nn \l_tmpa_box { A }
\ group_begin:
  \ hbox_set:Nn \l_tmpa_box { B }
  \ group_begin:
    \ box_use_drop:N \l_tmpa_box
  \ group_end:
    \ box_show:N \l_tmpa_box
\ group_end:
\ box_show:N \l_tmpa_box

```

the first use of `\box_show:N` will show an entirely cleared (void) box, and the second will show the letter **A** in the box.

These functions should be preferred when the content of the box is no longer required after use. Note that due to the unusual scoping behaviour of `drop` functions they may be applied to both local and global boxes: the latter will naturally be set and thus cleared at a global level.

`\box_use_drop:N`
`\box_use_drop:c`

`\box_use_drop:N` $\langle box \rangle$

Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting then drops the box content. An error is raised if the variable does not exist or if it is invalid. This function may be applied to local or global boxes.

T_EXhackers note: This is the `\box` primitive.

`\box_set_eq_drop:NN`
`\box_set_eq_drop:(cN|Nc|cc)`

New: 2019-01-17

`\box_set_eq_drop:NN` $\langle box_1 \rangle$ $\langle box_2 \rangle$

Sets the content of $\langle box_1 \rangle$ equal to that of $\langle box_2 \rangle$, then drops $\langle box_2 \rangle$.

`\box_gset_eq_drop:NN`
`\box_gset_eq_drop:(cN|Nc|cc)`

New: 2019-01-17

`\box_gset_eq_drop:NN` $\langle box_1 \rangle$ $\langle box_2 \rangle$

Sets the content of $\langle box_1 \rangle$ globally equal to that of $\langle box_2 \rangle$, then drops $\langle box_2 \rangle$.

`\hbox_unpack_drop:N`
`\hbox_unpack_drop:c`

New: 2019-01-17

`\hbox_unpack_drop:N` $\langle box \rangle$

Unpacks the content of the horizontal $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set. The original $\langle box \rangle$ is then dropped.

T_EXhackers note: This is the T_EX primitive `\unhbox`.

`\vbox_unpack_drop:N`
`\vbox_unpack_drop:c`

New: 2019-01-17

`\vbox_unpack_drop:N` $\langle box \rangle$

Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set. The original $\langle box \rangle$ is then dropped.

T_EXhackers note: This is the T_EX primitive `\unvbox`.

13 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in T_EX by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing of boxed material can best be handled by modifying boxes. These transformations are described here.

<code>\box_autosize_to_wd_and_ht:Nnn</code>	<code>\box_autosize_to_wd_and_ht:Nnn <box> {<x-size>} {<y-size>}</code>
<code>\box_autosize_to_wd_and_ht:cnn</code>	
<code>\box_gautosize_to_wd_and_ht:Nnn</code>	
<code>\box_gautosize_to_wd_and_ht:cnn</code>	

New: 2017-04-04

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to fit within the given $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically); both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the height only: it does not include any depth. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. The final size of the $\langle box \rangle$ is the smaller of $\{ \langle x-size \rangle \}$ and $\{ \langle y-size \rangle \}$, *i.e.* the result fits within the dimensions specified. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_autosize_to_wd_and_ht_plus_dp:Nnn</code>	<code>\box_autosize_to_wd_and_ht_plus_dp:Nnn <box> {<x-size>}</code>
<code>\box_autosize_to_wd_and_ht_plus_dp:cnn</code>	<code>{<y-size>}</code>
<code>\box_gautosize_to_wd_and_ht_plus_dp:Nnn</code>	
<code>\box_gautosize_to_wd_and_ht_plus_dp:cnn</code>	

New: 2017-04-04

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to fit within the given $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically); both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. The final size of the $\langle box \rangle$ is the smaller of $\{ \langle x-size \rangle \}$ and $\{ \langle y-size \rangle \}$, *i.e.* the result fits within the dimensions specified. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_resize_to_ht:Nn</code>	<code>\box_resize_to_ht:Nn <box> {<y-size>}</code>
<code>\box_resize_to_ht:cn</code>	
<code>\box_gresize_to_ht:Nn</code>	
<code>\box_gresize_to_ht:cn</code>	

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle y-size \rangle$ (vertically), scaling the horizontal size by the same amount; $\langle y-size \rangle$ is a dimension expression. The $\langle y-size \rangle$ is the height only: it does not include any depth. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle y-size \rangle$ causes the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_resize_to_ht_plus_dp:Nn</code>	<code>\box_resize_to_ht_plus_dp:Nn <box> {<y-size>}</code>
<code>\box_resize_to_ht_plus_dp:cn</code>	
<code>\box_gresize_to_ht_plus_dp:Nn</code>	
<code>\box_gresize_to_ht_plus_dp:cn</code>	

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle y-size \rangle$ (vertically), scaling the horizontal size by the same amount; $\langle y-size \rangle$ is a dimension expression. The $\langle y-size \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle y-size \rangle$ causes the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_resize_to_wd:Nn</code>	<code>\box_resize_to_wd:Nn <box> {<x-size>}</code>
<code>\box_resize_to_wd:cn</code>	
<code>\box_gresize_to_wd:Nn</code>	
<code>\box_gresize_to_wd:cn</code>	

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle x-size \rangle$ (horizontally), scaling the vertical size by the same amount; $\langle x-size \rangle$ is a dimension expression. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle x-size \rangle$ causes the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle x-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_resize_to_wd_and_ht:Nnn</code>	<code>\box_resize_to_wd_and_ht:Nnn <box> {<x-size>} {<y-size>}</code>
<code>\box_resize_to_wd_and_ht:cnn</code>	
<code>\box_gresize_to_wd_and_ht:Nnn</code>	
<code>\box_gresize_to_wd_and_ht:cnn</code>	

New: 2014-07-03

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically): both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the height only and does not include any depth. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_resize_to_wd_and_ht_plus_dp:Nnn</code>	<code>\box_resize_to_wd_and_ht_plus_dp:Nnn <box> {<x-size>} {<y-size>}</code>
<code>\box_resize_to_wd_and_ht_plus_dp:cnn</code>	
<code>\box_gresize_to_wd_and_ht_plus_dp:Nnn</code>	
<code>\box_gresize_to_wd_and_ht_plus_dp:cnn</code>	

New: 2017-04-06

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically): both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_rotate:Nn</code>	<code>\box_rotate:Nn <box> {<angle>}</code>
<code>\box_rotate:cn</code>	
<code>\box_grotate:Nn</code>	Rotates the $\langle box \rangle$ by $\langle angle \rangle$ (in degrees) anti-clockwise about its reference point. The reference point of the updated box is moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated $\langle box \rangle$ is an hbox , irrespective of the nature of the $\langle box \rangle$ before the rotation is applied.
<code>\box_grotate:cn</code>	
<hr/> Updated: 2019-01-22 <hr/>	

<code>\box_scale:Nnn</code>	<code>\box_scale:Nnn <box> {<x-scale>} {<y-scale>}</code>
<code>\box_scale:cnn</code>	
<code>\box_gscale:Nnn</code>	Scales the $\langle box \rangle$ by factors $\langle x-scale \rangle$ and $\langle y-scale \rangle$ in the horizontal and vertical directions, respectively (both scales are integer expressions). The updated $\langle box \rangle$ is an hbox , irrespective of the nature of the $\langle box \rangle$ before the scaling is applied. Negative scalings cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-scale \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and <i>vice versa</i> .
<code>\box_gscale:cnn</code>	
<hr/> Updated: 2019-01-22 <hr/>	

14 Primitive box conditionals

<code>\if_hbox:N *</code>	<code>\if_hbox:N <box></code> <code> <true code></code> <code>\else:</code> <code> <false code></code> <code>\fi:</code>
	Tests is $\langle box \rangle$ is a horizontal box.

T_EXhackers note: This is the T_EX primitive `\ifhbox`.

<code>\if_vbox:N *</code>	<code>\if_vbox:N <box></code> <code> <true code></code> <code>\else:</code> <code> <false code></code> <code>\fi:</code>
	Tests is $\langle box \rangle$ is a vertical box.

T_EXhackers note: This is the T_EX primitive `\ifvbox`.

<code>\if_box_empty:N *</code>	<code>\if_box_empty:N <box></code> <code> <true code></code> <code>\else:</code> <code> <false code></code> <code>\fi:</code>
	Tests is $\langle box \rangle$ is an empty (void) box.

T_EXhackers note: This is the T_EX primitive `\ifvoid`.

Part XXX

The l3coffins package

Coffin code layer

The material in this module provides the low-level support system for coffins. For details about the design concept of a coffin, see the xcoffins module (in the l3experimental bundle).

1 Creating and initialising coffins

<code>\coffin_new:N</code>
<code>\coffin_new:c</code>
<code>New: 2011-08-17</code>

`\coffin_new:N` $\langle coffin \rangle$

Creates a new $\langle coffin \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle coffin \rangle$ is initially empty.

<code>\coffin_clear:N</code>
<code>\coffin_clear:c</code>
<code>\coffin_gclear:N</code>
<code>\coffin_gclear:c</code>
<code>New: 2011-08-17</code>
<code>Updated: 2019-01-21</code>

`\coffin_clear:N` $\langle coffin \rangle$

Clears the content of the $\langle coffin \rangle$.

<code>\coffin_set_eq:NN</code>
<code>\coffin_set_eq:(Nc cN cc)</code>
<code>\coffin_gset_eq:NN</code>
<code>\coffin_gset_eq:(Nc cN cc)</code>
<code>New: 2011-08-17</code>
<code>Updated: 2019-01-21</code>

`\coffin_set_eq:NN` $\langle coffin_1 \rangle$ $\langle coffin_2 \rangle$

Sets both the content and poles of $\langle coffin_1 \rangle$ equal to those of $\langle coffin_2 \rangle$.

<code>\coffin_if_exist_p:N *</code>
<code>\coffin_if_exist_p:c *</code>
<code>\coffin_if_exist:N\overline{TF} *</code>
<code>\coffin_if_exist:c\overline{TF} *</code>
<code>New: 2012-06-20</code>

`\coffin_if_exist_p:N` $\langle box \rangle$

`\coffin_if_exist:NTF` $\langle box \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests whether the $\langle coffin \rangle$ is currently defined.

2 Setting coffin content and poles

<code>\hcoffin_set:Nn</code>
<code>\hcoffin_set:cn</code>
<code>\hcoffin_gset:Nn</code>
<code>\hcoffin_gset:cn</code>
<code>New: 2011-08-17</code>
<code>Updated: 2019-01-21</code>

`\hcoffin_set:Nn` $\langle coffin \rangle$ $\{\langle material \rangle\}$

Typesets the $\langle material \rangle$ in horizontal mode, storing the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material.

```

\hcoffin_set:Nw
\hcoffin_set:cw
\hcoffin_set_end:
\hcoffin_gset:Nw
\hcoffin_gset:cw
\hcoffin_gset_end:

```

New: 2011-09-10
Updated: 2019-01-21

```

\vcoffin_set:Nnn
\vcoffin_set:cnn
\vcoffin_gset:Nnn
\vcoffin_gset:cnn

```

New: 2011-08-17
Updated: 2019-01-21

```

\vcoffin_set:Nnw
\vcoffin_set:cnw
\vcoffin_set_end:
\vcoffin_gset:Nnw
\vcoffin_gset:cnw
\vcoffin_gset_end:

```

New: 2011-09-10
Updated: 2019-01-21

`\hcoffin_set:Nw <coffin> <material> \hcoffin_set_end:`

Typesets the $\langle material \rangle$ in horizontal mode, storing the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

`\vcoffin_set:Nnn <coffin> {\<width>} {\<material>}`

Typesets the $\langle material \rangle$ in vertical mode constrained to the given $\langle width \rangle$ and stores the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material.

`\vcoffin_set:Nnw <coffin> {\<width>} <material> \vcoffin_set_end:`

Typesets the $\langle material \rangle$ in vertical mode constrained to the given $\langle width \rangle$ and stores the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

```

\coffin_set_horizontal_pole:Nnn
\coffin_set_horizontal_pole:cnn
\coffin_gset_horizontal_pole:Nnn
\coffin_gset_horizontal_pole:cnn

```

New: 2012-07-20
Updated: 2019-01-21

`\coffin_set_horizontal_pole:Nnn <coffin> {\<pole>} {\<offset>}`

Sets the $\langle pole \rangle$ to run horizontally through the $\langle coffin \rangle$. The $\langle pole \rangle$ is placed at the $\langle offset \rangle$ from the bottom edge of the bounding box of the $\langle coffin \rangle$. The $\langle offset \rangle$ should be given as a dimension expression.

```

\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnn
\coffin_gset_vertical_pole:Nnn
\coffin_gset_vertical_pole:cnn

```

New: 2012-07-20
Updated: 2019-01-21

`\coffin_set_vertical_pole:Nnn <coffin> {\<pole>} {\<offset>}`

Sets the $\langle pole \rangle$ to run vertically through the $\langle coffin \rangle$. The $\langle pole \rangle$ is placed at the $\langle offset \rangle$ from the left-hand edge of the bounding box of the $\langle coffin \rangle$. The $\langle offset \rangle$ should be given as a dimension expression.

3 Coffin affine transformations

<code>\coffin_resize:Nnn</code>	<code>\coffin_resize:Nnn <coffin> {<width>} {<total-height>}</code>
<code>\coffin_resize:cnn</code>	
<code>\coffin_gresize:Nnn</code>	Resized the $\langle coffin \rangle$ to $\langle width \rangle$ and $\langle total-height \rangle$, both of which should be given as dimension expressions.
<code>\coffin_gresize:cnn</code>	
Updated: 2019-01-23	
<code>\coffin_rotate:Nn</code>	<code>\coffin_rotate:Nn <coffin> {<angle>}</code>
<code>\coffin_rotate:cn</code>	
<code>\coffin_grotate:Nn</code>	Rotates the $\langle coffin \rangle$ by the given $\langle angle \rangle$ (given in degrees counter-clockwise). This process rotates both the coffin content and poles. Multiple rotations do not result in the bounding box of the coffin growing unnecessarily.
<code>\coffin_grotate:cn</code>	
<code>\coffin_scale:Nnn</code>	<code>\coffin_scale:Nnn <coffin> {<x-scale>} {<y-scale>}</code>
<code>\coffin_scale:cnn</code>	
<code>\coffin_gscale:Nnn</code>	Scales the $\langle coffin \rangle$ by a factors $\langle x-scale \rangle$ and $\langle y-scale \rangle$ in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.
<code>\coffin_gscale:cnn</code>	
Updated: 2019-01-23	

4 Joining and using coffins

<code>\coffin_attach:NnnNnnnn</code>	<code>\coffin_attach:NnnNnnnn</code>
<code>\coffin_attach:(cnnNnnnn Nnnncnnnn cnnncnnnn)</code>	$\langle coffin_1 \rangle$ { $\langle coffin_1-pole_1 \rangle$ } { $\langle coffin_1-pole_2 \rangle$ }
<code>\coffin_gattach:NnnNnnnn</code>	$\langle coffin_2 \rangle$ { $\langle coffin_2-pole_1 \rangle$ } { $\langle coffin_2-pole_2 \rangle$ }
<code>\coffin_gattach:(cnnNnnnn Nnnncnnnn cnnncnnnn)</code>	{ $\langle x-offset \rangle$ } { $\langle y-offset \rangle$ }
Updated: 2019-01-22	
<p>This function attaches $\langle coffin_2 \rangle$ to $\langle coffin_1 \rangle$ such that the bounding box of $\langle coffin_1 \rangle$ is not altered, <i>i.e.</i> $\langle coffin_2 \rangle$ can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating $\langle handle_1 \rangle$, the point of intersection of $\langle coffin_1-pole_1 \rangle$ and $\langle coffin_1-pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2-pole_1 \rangle$ and $\langle coffin_2-pole_2 \rangle$. $\langle coffin_2 \rangle$ is then attached to $\langle coffin_1 \rangle$ such that the relationship between $\langle handle_1 \rangle$ and $\langle handle_2 \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions.</p>	

<code>\coffin_join:NnnNnnnn</code>	<code>\coffin_join:NnnNnnnn</code>
<code>\coffin_join:(cnnNnnnn Nnnncnnnn cnnncnnnn)</code>	$\langle coffin_1 \rangle$ { $\langle coffin_1-pole_1 \rangle$ } { $\langle coffin_1-pole_2 \rangle$ }
<code>\coffin_gjoin:NnnNnnnn</code>	$\langle coffin_2 \rangle$ { $\langle coffin_2-pole_1 \rangle$ } { $\langle coffin_2-pole_2 \rangle$ }
<code>\coffin_gjoin:(cnnNnnnn Nnnncnnnn cnnncnnnn)</code>	{ $\langle x-offset \rangle$ } { $\langle y-offset \rangle$ }
Updated: 2019-01-22	

This function joins $\langle coffin_2 \rangle$ to $\langle coffin_1 \rangle$ such that the bounding box of $\langle coffin_1 \rangle$ may expand. The new bounding box covers the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating $\langle handle_1 \rangle$, the point of intersection of $\langle coffin_1-pole_1 \rangle$ and $\langle coffin_1-pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2-pole_1 \rangle$ and $\langle coffin_2-pole_2 \rangle$. $\langle coffin_2 \rangle$ is then attached to $\langle coffin_1 \rangle$ such that the relationship between $\langle handle_1 \rangle$ and $\langle handle_2 \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions.

```
\coffin_typeset:Nnnnn
\coffin_typeset:cnnnn
```

Updated: 2012-07-20

```
\coffin_typeset:Nnnnn <coffin> {\pole_1} {\pole_2}
{\<x-offset>} {\<y-offset>}
```

Typesetting is carried out by first calculating $\langle handle \rangle$, the point of intersection of $\langle pole_1 \rangle$ and $\langle pole_2 \rangle$. The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the $\langle handle \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions. Typesetting a coffin is therefore analogous to carrying out an alignment where the “parent” coffin is the current insertion point.

5 Measuring coffins

```
\coffin_dp:N
\coffin_dp:c
```

```
\coffin_dp:N <coffin>
```

Calculates the depth (below the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

```
\coffin_ht:N
\coffin_ht:c
```

```
\coffin_ht:N <coffin>
```

Calculates the height (above the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

```
\coffin_wd:N
\coffin_wd:c
```

```
\coffin_wd:N <coffin>
```

Calculates the width of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

6 Coffin diagnostics

```
\coffin_display_handles:Nn
\coffin_display_handles:cn
```

Updated: 2011-09-02

```
\coffin_display_handles:Nn <coffin> {\color}
```

This function first calculates the intersections between all of the $\langle poles \rangle$ of the $\langle coffin \rangle$ to give a set of $\langle handles \rangle$. It then prints the $\langle coffin \rangle$ at the current location in the source, with the position of the $\langle handles \rangle$ marked on the coffin. The $\langle handles \rangle$ are labelled as part of this process: the locations of the $\langle handles \rangle$ and the labels are both printed in the $\langle color \rangle$ specified.

```
\coffin_mark_handle:Nnnn
\coffin_mark_handle:cnnn
```

Updated: 2011-09-02

```
\coffin_mark_handle:Nnnn <coffin> {\pole_1} {\pole_2} {\color}
```

This function first calculates the $\langle handle \rangle$ for the $\langle coffin \rangle$ as defined by the intersection of $\langle pole_1 \rangle$ and $\langle pole_2 \rangle$. It then marks the position of the $\langle handle \rangle$ on the $\langle coffin \rangle$. The $\langle handle \rangle$ are labelled as part of this process: the location of the $\langle handle \rangle$ and the label are both printed in the $\langle color \rangle$ specified.

```
\coffin_show_structure:N
\coffin_show_structure:c
```

Updated: 2015-08-01

```
\coffin_show_structure:N <coffin>
```

This function shows the structural information about the $\langle coffin \rangle$ in the terminal. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin.

Notice that the poles of a coffin are defined by four values: the x and y co-ordinates of a point that the pole passes through and the x - and y -components of a vector denoting the direction of the pole. It is the ratio between the later, rather than the absolute values, which determines the direction of the pole.

`\coffin_log_structure:N`
`\coffin_log_structure:c`

New: 2014-08-22
Updated: 2015-08-01

`\coffin_log_structure:N` $\langle coffin \rangle$

This function writes the structural information about the $\langle coffin \rangle$ in the log file. See also

`\coffin_show_structure:N` which displays the result in the terminal.

7 Constants and variables

`\c_empty_coffin`

A permanently empty coffin.

`\l_tmpa_coffin`
`\l_tmpb_coffin`

New: 2012-06-19

Scratch coffins for local assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_coffin`
`\g_tmpb_coffin`

New: 2019-01-24

Scratch coffins for global assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Part XXXI

The l3color-base package

Color support

This module provides support for color in L^AT_EX3. At present, the material here is mainly intended to support a small number of low-level requirements in other l3kernel modules.

1 Color in boxes

Controlling the color of text in boxes requires a small number of control functions, so that the boxed material uses the color at the point where it is set, rather than where it is used.

```
\color_group_begin:
\color_group_end:
```

New: 2011-09-03

```
\color_group_begin:
...
\color_group_end:
```

Creates a color group: one used to “trap” color settings.

```
\color_ensure_current:
```

New: 2011-09-03

```
\color_ensure_current:
```

Ensures that material inside a box uses the foreground color at the point where the box is set, rather than that in force when the box is used. This function should usually be used within a `\color_group_begin: ... \color_group_end:` group.

Part XXXII

The l3luatex package: LuaTeX-specific functions

The LuaTeX engine provides access to the Lua programming language, and with it access to the “internals” of TeX. In order to use this within the framework provided here, a family of functions is available. When used with pdfTeX, pTeX, upTeX or XeTeX these raise an error: use `\sys_if_engine_luatex:T` to avoid this. Details on using Lua with the LuaTeX engine are given in the LuaTeX manual.

1 Breaking out to Lua

<code>\lua_now:n</code>	★	<code>\lua_now:n</code>	{ <i><token list></i> }
-------------------------	---	-------------------------	-------------------------------

<code>\lua_now:e</code>	★
-------------------------	---

New: 2018-06-18

The *<token list>* is first tokenized by TeX, which includes converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting *<Lua input>* is passed to the Lua interpreter for processing. Each `\lua_now:n` block is treated by Lua as a separate chunk. The Lua interpreter executes the *<Lua input>* immediately, and in an expandable manner.

TeXhackers note: `\lua_now:e` is a macro wrapper around `\directlua:` when LuaTeX is in use two expansions are required to yield the result of the Lua code.

<code>\lua_shipout_e:n</code>	★
-------------------------------	---

<code>\lua_shipout:n</code>

New: 2018-06-18

<code>\lua_shipout:n</code>	{ <i><token list></i> }
-----------------------------	-------------------------------

The *<token list>* is first tokenized by TeX, which includes converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting *<Lua input>* is passed to the Lua interpreter when the current page is finalised (*i.e.* at shipout). Each `\lua_shipout:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the *<Lua input>* during the page-building routine: no TeX expansion of the *<Lua input>* will occur at this stage.

In the case of the `\lua_shipout_e:n` version the input is fully expanded by TeX in an e-type manner during the shipout operation.

TeXhackers note: At a TeX level, the *<Lua input>* is stored as a “whatsit”.

<code>\lua_escape:n</code>	★
----------------------------	---

<code>\lua_escape:e</code>	★
----------------------------	---

New: 2015-06-29

<code>\lua_escape:n</code>	{ <i><token list></i> }
----------------------------	-------------------------------

Converts the *<token list>* such that it can safely be passed to Lua: embedded backslashes, double and single quotes, and newlines and carriage returns are escaped. This is done by prepending an extra token consisting of a backslash with category code 12, and for the line endings, converting them to `\n` and `\r`, respectively.

TeXhackers note: `\lua_escape:e` is a macro wrapper around `\luaescapestring:` when LuaTeX is in use two expansions are required to yield the result of the Lua code.

2 Lua interfaces

As well as interfaces for \TeX , there are a small number of Lua functions provided here.

<u>13kernel</u>	All public interfaces provided by the module are stored within the <code>13kernel</code> table.
<u>13kernel.charcat</u>	<p><code>13kernel.charcat($\langle charcode \rangle$, $\langle catcode \rangle$)</code></p> <p>Constructs a character of $\langle charcode \rangle$ and $\langle catcode \rangle$ and returns the result to \TeX.</p>
<u>13kernel.elapsedtime</u>	<p><code>13kernel.elapsedtime()</code></p> <p>Returns the CPU time in $\langle scaled\ seconds \rangle$ since the start of the \TeX run or since <code>13kernel.resettimer</code> was issued. This only measures the time used by the CPU, not the real time, e.g., waiting for user input.</p>
<u>13kernel.filedump</u>	<p><code>13kernel.filedump($\langle file \rangle$, $\langle offset \rangle$, $\langle length \rangle$)</code></p> <p>Returns the uppercase hexadecimal representation of the content of the $\langle file \rangle$ read as bytes. If the $\langle length \rangle$ is given, only this part of the file is returned; similarly, one may specify the $\langle offset \rangle$ from the start of the file. If the $\langle length \rangle$ is not given, the entire file is read starting at the $\langle offset \rangle$.</p>
<u>13kernel.filemdfivesum</u>	<p><code>13kernel.filemdfivesum($\langle file \rangle$)</code></p> <p>Returns the MD5 sum of the file contents read as bytes; note that the result will depend on the nature of the line endings used in the file, in contrast to normal \TeX behaviour. If the $\langle file \rangle$ is not found, nothing is returned with <i>no error raised</i>.</p>
<u>13kernel.filemoddate</u>	<p><code>13kernel.filemoddate($\langle file \rangle$)</code></p> <p>Returns the date/time of last modification of the $\langle file \rangle$ in the format</p> <p style="text-align: center;">$D:\langle year \rangle\langle month \rangle\langle day \rangle\langle hour \rangle\langle minute \rangle\langle second \rangle\langle offset \rangle$</p> <p>where the latter may be Z (UTC) or $\langle plus-minus \rangle\langle hours \rangle'\langle minutes \rangle'$. If the $\langle file \rangle$ is not found, nothing is returned with <i>no error raised</i>.</p>
<u>13kernel.filesize</u>	<p><code>13kernel.filesize($\langle file \rangle$)</code></p> <p>Returns the size of the $\langle file \rangle$ in bytes. If the $\langle file \rangle$ is not found, nothing is returned with <i>no error raised</i>.</p>
<u>13kernel.resettimer</u>	<p><code>13kernel.resettimer()</code></p> <p>Resets the timer used by <code>13kernel.elapsedtime</code>.</p>
<u>13kernel.shellescape</u>	<p><code>13kernel.shellescape($\langle cmd \rangle$)</code></p> <p>Executes the $\langle cmd \rangle$ and prints to the log as for pdf\TeX.</p>
<u>13kernel.strcmp</u>	<p><code>13kernel.strcmp($\langle str\ one \rangle$, $\langle str\ two \rangle$)</code></p> <p>Compares the two strings and returns 0 to \TeX if the two are identical.</p>

Part XXXIII

The l3unicode package: Unicode support functions

This module provides Unicode-specific functions along with loading data from a range of Unicode Consortium files. At present, it provides no public functions.

Part XXXIV

The l3text package: text processing

1 l3text documentation

This module deals with manipulation of (formatted) text; such material is comprised of a restricted set of token list content. The functions provided here concern conversion of textual content for example in case changing, generation of bookmarks and extraction to tags. All of the major functions operate by expansion. Begin-group and end-group tokens in the $\langle text \rangle$ are normalized and become { and }, respectively.

1.1 Expanding text

<code>\text_expand:n</code> *	<code>\text_expand:n {$\langle text \rangle$}</code>
-------------------------------	---

New: 2020-01-02

Takes user input $\langle text \rangle$ and expands the content. Protected commands (typically formatting) are left in place, and no processing takes place of math mode material (as delimited by pairs given in `\l_text_math_delims_tl` or as the argument to commands listed in `\l_text_math_arg_tl`). Commands which are neither engine- nor L^AT_EX protected are expanded exhaustively. Any commands listed in `\l_text_expand_exclude_tl`, `\l_text_accents_tl` and `\l_text_letterlike_tl` are excluded from expansion.

<code>\text_declare_expand_equivalent:Nn</code>	<code>\text_declare_expand_equivalent:Nn $\langle cmd \rangle$ {$\langle replacement \rangle$}</code>
<code>\text_declare_expand_equivalent:cn</code>	

New: 2020-01-22

Declares that the $\langle replacement \rangle$ tokens should be used whenever the $\langle cmd \rangle$ (a single token) is encountered. The $\langle replacement \rangle$ tokens should be expandable.

1.2 Case changing

<code>\text_lowercase:n</code>	*
<code>\text_uppercase:n</code>	*
<code>\text_titlecase:n</code>	*
<code>\text_titlecase_first:n</code>	*
<code>\text_lowercase:nn</code>	*
<code>\text_uppercase:nn</code>	*
<code>\text_titlecase:nn</code>	*
<code>\text_titlecase_first:nn</code>	*

New: 2019-11-20
Updated: 2020-02-24

`\text_uppercase:n` $\{\langle tokens \rangle\}$
`\text_uppercase:nn` $\{\langle language \rangle\} \{\langle tokens \rangle\}$
Takes user input $\langle text \rangle$ first applies `\text_expand`, then transforms the case of character tokens as specified by the function name. The category code of letters are not changed by this process (at least where they can be represented by the engine as a single token: 8-bit engines may require active characters).

Upper- and lowercase have the obvious meanings. Titlecasing may be regarded informally as converting the first character of the $\langle tokens \rangle$ to uppercase and the rest to lowercase. However, the process is more complex than this as there are some situations where a single lowercase character maps to a special form, for example *ij* in Dutch which becomes *IJ*. The `titlecase_first` variant does not attempt any case changing at all after the first letter has been processed.

Importantly, notice that these functions are intended for working with user *text for typesetting*. For case changing programmatic data see the `l3str` module and discussion there of `\str_lowercase:n`, `\str_uppercase:n` and `\str_foldcase:n`.

Case changing does not take place within math mode material so for example

```
\text_uppercase:n { Some~text~$y = mx + c$~with~{Braces} }
```

becomes

```
SOME TEXT $y = mx + c$ WITH {BRACES}
```

The arguments of commands listed in `\l_text_case_exclude_arg_tl` are excluded from case changing; the latter are entirely non-textual content (such as labels).

As is generally true for `expl3`, these functions are designed to work with Unicode input only. As such, UTF-8 input is assumed for *all* engines. When used with Xe_{La}TeX or Lua_{La}TeX a full range of Unicode transformations are enabled. Specifically, the standard mappings here follow those defined by the [Unicode Consortium](#) in `UnicodeData.txt` and `SpecialCasing.txt`. In the case of 8-bit engines, mappings are provided for characters which can be represented in output typeset using the T1, T2 and LGR font encodings. Thus for example *ä* can be case-changed using pdf_{La}TeX. For p_{La}TeX only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Language-sensitive conversions are enabled using the $\langle language \rangle$ argument, and follow Unicode Consortium guidelines. Currently, the languages recognised for special handling are as follows.

- Azeri and Turkish (**az** and **tr**). The case pairs I/i-dotless and I-dot/i are activated for these languages. The combining dot mark is removed when lowercasing I-dot and introduced when upper casing i-dotless.
- German (**de-alt**). An alternative mapping for German in which the lowercase *Eszett* maps to a *großes Eszett*. Since there is a T1 slot for the *großes Eszett* in T1, this tailoring *is* available with pdf_{La}TeX as well as in the Unicode _{La}TeX engines.
- Greek (**el**). Removes accents from Greek letters when uppercasing; titlecasing leaves accents in place.
- Lithuanian (**lt**). The lowercase letters i and j should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lowercasing of the relevant uppercase letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when uppercasing in these cases. Note that *only* the accents used in Lithuanian are covered: the behaviour of other accents are not modified.
- Dutch (**nl**). Capitalisation of *ij* at the beginning of titlecased input produces *IJ* rather than *Ij*. The output retains two separate letters, thus this transformation *is* available using pdf_{La}TeX.

For titlecasing, note that there are two functions available. The function `\text_`
`titlecase:n` applies (broadly) uppercasing to the first letter of the input, then lower-

1.3 Removing formatting from text

<code>\text_purify:n</code> *	<code>\text_purify:n {<text>}</code>
-------------------------------	--

New: 2020-03-05
Updated: 2020-05-14

Takes user input $\langle text \rangle$ and expands as described for `\text_expand:n`, then removes all functions from the resulting text. Math mode material (as delimited by pairs given in `\l_text_math_delims_tl` or as the argument to commands listed in `\l_text_math_arg_tl`) is left contained in a pair of \$ delimiters. Non-expandable functions present in the $\langle text \rangle$ must either have a defined equivalent (see `\text_declare_purify_equivalent:Nn`) or will be removed from the result. Implicit tokens are converted to their explicit equivalent.

<code>\text_declare_purify_equivalent:Nn</code>	<code>\text_declare_purify_equivalent:Nn <cmd> {<replacement>}</code>
<code>\text_declare_purify_equivalent:Nx</code>	

New: 2020-03-05

Declares that the $\langle replacement \rangle$ tokens should be used whenever the $\langle cmd \rangle$ (a single token) is encountered. The $\langle replacement \rangle$ tokens should be expandable.

1.4 Control variables

<code>\l_text_accents_tl</code>

Lists commands which represent accents, and which are left unchanged by expansion. (Defined only for the L^AT_EX 2_ε package.)

<code>\l_text_letterlike_tl</code>

Lists commands which represent letters; these are left unchanged by expansion. (Defined only for the L^AT_EX 2_ε package.)

<code>\l_text_math_arg_tl</code>

Lists commands present in the $\langle text \rangle$ where the argument of the command should be treated as math mode material. The treatment here is similar to `\l_text_math_delims_tl` but for a command rather than paired delimiters.

<code>\l_text_math_delims_tl</code>

Lists pairs of tokens which delimit (in-line) math mode content; such content *may* be excluded from processing.

<code>\l_text_case_exclude_arg_tl</code>
--

Lists commands which are excluded from case changing.

<code>\l_text_expand_exclude_tl</code>
--

Lists commands which are excluded from expansion.

<code>\l_text_titlecase_check_letter_bool</code>
--

Controls how the start of titlecasing is handled: when **true**, the first *letter* in text is considered. The standard setting is **true**.

Part XXXV

The l3legacy package

Interfaces to legacy concepts

There are a small number of T_EX or L^AT_EX 2_ε concepts which are not used in `expl3` code but which need to be manipulated when working as a L^AT_EX 2_ε package. To allow these to be integrated cleanly into `expl3` code, a set of legacy interfaces are provided here.

<code>\legacy_if_p:n</code> *	<code>\legacy_if:nTF</code> { <i><name></i> } { <i><true code></i> } { <i><false code></i> }
<code>\legacy_if:nTF</code> *	Tests if the L ^A T _E X 2 _ε /plain T _E X conditional (generated by <code>\newif</code>) if <code>true</code> or <code>false</code> and branches accordingly. The <i><name></i> of the conditional should <i>omit</i> the leading <code>if</code> .

Part XXXVI

The l3candidates package

Experimental additions to l3kernel

1 Important notice

This module provides a space in which functions can be added to l3kernel (expl3) while still being experimental.

As such, the functions here may not remain in their current form, or indeed at all, in l3kernel in the future.

In contrast to the material in l3experimental, the functions here are all *small* additions to the kernel. We encourage programmers to test them out and report back on the LaTeX-L mailing list.

Thus, if you intend to use any of these functions from the candidate module in a public package offered to others for productive use (e.g., being placed on CTAN) please consider the following points carefully:

- Be prepared that your public packages might require updating when such functions are being finalized.
- Consider informing us that you use a particular function in your public package, e.g., by discussing this on the LaTeX-L mailing list. This way it becomes easier to coordinate any updates necessary without issues for the users of your package.
- Discussing and understanding use cases for a particular addition or concept also helps to ensure that we provide the right interfaces in the final version so please give us feedback if you consider a certain candidate function useful (or not).

We only add functions in this space if we consider them being serious candidates for a final inclusion into the kernel. However, real use sometimes leads to better ideas, so functions from this module are **not necessarily stable** and we may have to adjust them!

2 Additions to l3box

2.1 Viewing part of a box

```
\box_clip:N
\box_clip:c
\box_gclip:N
\box_gclip:c
```

Updated: 2019-01-23

```
\box_clip:N <box>
```

Clips the $\langle box \rangle$ in the output so that only material inside the bounding box is displayed in the output. The updated $\langle box \rangle$ is an hbox, irrespective of the nature of the $\langle box \rangle$ before the clipping is applied.

These functions require the L^AT_EX3 native drivers: they do not work with the L^AT_EX 2_ε graphics drivers!

T_EXhackers note: Clipping is implemented by the driver, and as such the full content of the box is placed in the output file. Thus clipping does not remove any information from the raw output, and hidden material can therefore be viewed by direct examination of the file.

```
\box_set_trim:Nnnnn
\box_set_trim:cnnnn
\box_gset_trim:Nnnnn
\box_gset_trim:cnnnn
```

New: 2019-01-23

```
\box_set_trim:Nnnnn <box> {\left} {\bottom} {\right} {\top}
```

Adjusts the bounding box of the $\langle box \rangle$ $\langle left \rangle$ is removed from the left-hand edge of the bounding box, $\langle right \rangle$ from the right-hand edge and so fourth. All adjustments are *$\langle dimension expressions \rangle$* . Material outside of the bounding box is still displayed in the output unless `\box_clip:N` is subsequently applied. The updated $\langle box \rangle$ is an hbox, irrespective of the nature of the $\langle box \rangle$ before the trim operation is applied. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

```
\box_set_viewport:Nnnnn
\box_set_viewport:cnnnn
\box_gset_viewport:Nnnnn
\box_gset_viewport:cnnnn
```

New: 2019-01-23

```
\box_set_viewport:Nnnnn <box> {\llx} {\lly} {\urx} {\ury}
```

Adjusts the bounding box of the $\langle box \rangle$ such that it has lower-left co-ordinates ($\langle llx \rangle$, $\langle lly \rangle$) and upper-right co-ordinates ($\langle urx \rangle$, $\langle ury \rangle$). All four co-ordinate positions are *$\langle dimension expressions \rangle$* . Material outside of the bounding box is still displayed in the output unless `\box_clip:N` is subsequently applied. The updated $\langle box \rangle$ is an hbox, irrespective of the nature of the $\langle box \rangle$ before the viewport operation is applied.

3 Additions to l3expan

```
\exp_args_generate:n
```

New: 2018-04-04
Updated: 2019-02-08

```
\exp_args_generate:n {\variant argument specifiers}
```

Defines `\exp_args:N<variant>` functions for each $\langle variant \rangle$ given in the comma list $\{\langle variant argument specifiers \rangle\}$. Each $\langle variant \rangle$ should consist of the letters N, c, n, V, v, o, f, e, x, p and the resulting function is protected if the letter x appears in the $\langle variant \rangle$. This is only useful for cases where `\cs_generate_variant:Nn` is not applicable.

4 Additions to l3fp

```
\fp_if_nan_p:n ★
\fp_if_nan:nTF ★
```

New: 2019-08-25

```
\fp_if_nan:n {\fpexpr}
```

Evaluates the $\langle fpexpr \rangle$ and tests whether the result is exactly NaN. The test returns **false** for any other result, even a tuple containing NaN.

5 Additions to l3file

```
\iow_allow_break:
```

New: 2018-12-29

```
\iow_allow_break:
```

In the first argument of `\iow_wrap:nnnn` (for instance in messages), inserts a break-point that allows a line break. In other words this is a zero-width breaking space.

`\ior_get_term:nN`
`\ior_str_get_term:nN`

New: 2019-03-23

`\ior_get_term:nN` $\langle prompt \rangle$ $\langle token\ list\ variable \rangle$

Function that reads one or more lines (until an equal number of left and right braces are found) from the terminal and stores the result locally in the $\langle token\ list \rangle$ variable. Tokenization occurs as described for `\ior_get:NN` or `\ior_str_get:NN`, respectively. When the $\langle prompt \rangle$ is empty, \TeX will wait for input without any other indication: typically the programmer will have provided a suitable text using e.g. `\iow_term:n`. Where the $\langle prompt \rangle$ is given, it will appear in the terminal followed by an =, e.g.

prompt=

`\ior_shell_open:Nn`

New: 2019-05-08

`\ior_shell_open:nN` $\langle stream \rangle$ $\{ \langle shell\ command \rangle \}$

Opens the *pseudo*-file created by the output of the $\langle shell\ command \rangle$ for reading using $\langle stream \rangle$ as the control sequence for access. If the $\langle stream \rangle$ was already open it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately and will remain allocated to $\langle shell\ command \rangle$ until a `\ior_close:N` instruction is given or the \TeX run ends. If piped system calls are disabled an error is raised.

For details of handling of the $\langle shell\ command \rangle$, see `\sys_get_shell:nnN(TF)`.

6 Additions to `\l3flag`

`\flag_raise_if_clear:n` ☆

New: 2018-04-02

`\flag_raise_if_clear:n` $\{ \langle flag\ name \rangle \}$

Ensures the $\langle flag \rangle$ is raised by making its height at least 1, locally.

7 Additions to `\l3intarray`

`\intarray_gset_rand:Nnn`
`\intarray_gset_rand:cnn`
`\intarray_gset_rand:Nn`
`\intarray_gset_rand:cn`

New: 2018-05-05

`\intarray_gset_rand:Nnn` $\langle intarray\ var \rangle$ $\{ \langle minimum \rangle \}$ $\{ \langle maximum \rangle \}$
`\intarray_gset_rand:Nn` $\langle intarray\ var \rangle$ $\{ \langle maximum \rangle \}$

Evaluates the integer expressions $\langle minimum \rangle$ and $\langle maximum \rangle$ then sets each entry (independently) of the $\langle integer\ array\ variable \rangle$ to a pseudo-random number between the two (with bounds included). If the absolute value of either bound is bigger than $2^{30} - 1$, an error occurs. Entries are generated in the same way as repeated calls to `\int_rand:nn` or `\int_rand:n` respectively, in particular for the second function the $\langle minimum \rangle$ is 1. Assignments are always global. This is not available in older versions of \LaTeX .

7.1 Working with contents of integer arrays

`\intarray_to_clist:N` ☆

New: 2018-05-04

`\intarray_to_clist:N` $\langle intarray\ var \rangle$

Converts the $\langle intarray \rangle$ to integer denotations separated by commas. All tokens have category code other. If the $\langle intarray \rangle$ has no entry the result is empty; otherwise the result has one fewer comma than the number of items.

8 Additions to l3msg

```
\msg_show_eval:Nn
\msg_log_eval:Nn
```

New: 2017-12-04

```
\msg_show_eval:Nn <function> {<expression>}
```

Shows or logs the $\langle expression \rangle$ (turned into a string), an equal sign, and the result of applying the $\langle function \rangle$ to the $\{ \langle expression \rangle \}$ (with f-expansion). For instance, if the $\langle function \rangle$ is `\int_eval:n` and the $\langle expression \rangle$ is `1+2` then this logs `> 1+2=3.`

```
\msg_show:nnnnnn
\msg_show:nnxxxx
\msg_show:nnnnn
\msg_show:nnxxx
\msg_show:nnnn
\msg_show:nnxx
\msg_show:nnn
\msg_show:nnx
\msg_show:nn
```

New: 2017-12-04

```
\msg_show:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}
```

Issues $\langle module \rangle$ information $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The information text is shown on the terminal and the T_EX run is interrupted in a manner similar to `\tl_show:n`. This is used in conjunction with `\msg_show_item:n` and similar functions to print complex variable contents completely. If the formatted text does not contain `>~` at the start of a line, an additional line `>~.` will be put at the end. In addition, a final period is added if not present.

```
\msg_show_item:n          * \seq_map_function:NN <seq> \msg_show_item:n
\msg_show_item_unbraced:n * \prop_map_function:NN <prop> \msg_show_item:nn
\msg_show_item:nn         *
\msg_show_item_unbraced:nn *
```

New: 2017-12-04

Used in the text of messages for `\msg_show:nnxxxx` to show or log a list of items or key-value pairs. The one-argument functions are used for sequences, clist or token lists and the others for property lists. These functions turn their arguments to strings.

9 Additions to l3prg

```
\bool_set_inverse:N
\bool_set_inverse:c
\bool_gset_inverse:N
\bool_gset_inverse:c
```

New: 2018-05-10

```
\bool_set_inverse:N <boolean>
```

Toggles the $\langle boolean \rangle$ from true to false and conversely: sets it to the inverse of its current value.

<hr/>	
<code>\bool_case_true:n</code>	★
<code>\bool_case_true:nTF</code>	★
<code>\bool_case_false:n</code>	★
<code>\bool_case_false:nTF</code>	★
<hr/>	
New: 2019-02-10	
<hr/>	
<pre> \bool_case_true:nTF { {<boolexpr case₁>} {<code case₁>} {<boolexpr case₂>} {<code case₂>} ... {<boolexpr case_n>} {<code case_n>} } {<true code>} {<false code>} </pre>	

Evaluates in turn each of the *<boolean expression cases>* until the first one that evaluates to true or to false, for `\bool_case_true:n` and `\bool_case_false:n`, respectively. The *<code>* associated to this first case is left in the input stream, followed by the *<true code>*, and other cases are discarded. If none of the cases match then only the *<false code>* is inserted. The functions `\bool_case_true:n` and `\bool_case_false:n`, which do nothing if there is no match, are also available. For example

```

\bool_case_true:nF
{
  { \dim_compare_p:n { \l__mypkg_wd_dim <= 10pt } }
    { Fits }
  { \int_compare_p:n { \l__mypkg_total_int >= 10 } }
    { Many }
  { \l__mypkg_special_bool }
    { Special }
}
{ No idea! }

```

leaves “Fits” or “Many” or “Special” or “No idea!” in the input stream, in a way similar to some other language’s “if ... elseif ... elseif ... else ...”.

10 Additions to l3prop

<hr/>	
<code>\prop_rand_key_value:N</code>	★
<code>\prop_rand_key_value:c</code>	★
<hr/>	
New: 2016-12-06	
<hr/>	
<pre> \prop_rand_key_value:N <prop var> </pre>	

Selects a pseudo-random key–value pair from the *<property list>* and returns *{<key>}* and *{<value>}*. If the *<property list>* is empty the result is empty. This is not available in older versions of Xe_{La}TeX.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<value>* does not expand further when appearing in an x-type argument expansion.

11 Additions to l3seq

<code>\seq_mapthread_function:NNN</code>	☆	<code>\seq_mapthread_function:NNN <seq₁> <seq₂> <function></code>
<code>\seq_mapthread_function:(NcN cNN ccN)</code>	☆	

Applies *<function>* to every pair of items *<seq₁-item>*–*<seq₂-item>* from the two sequences, returning items from both sequences from left to right. The *<function>* receives two *n*-type arguments for each iteration. The mapping terminates when the end of either sequence is reached (*i.e.* whichever sequence has fewer items determines how many iterations occur).

<code>\seq_set_filter:NNn</code>	<code>\seq_set_filter:NNn <sequence₁> <sequence₂> {<inline boolexpr>}</code>
<code>\seq_gset_filter:NNn</code>	

Evaluates the *<inline boolexpr>* for every *<item>* stored within the *<sequence₂>*. The *<inline boolexpr>* receives the *<item>* as #1. The sequence of all *<items>* for which the *<inline boolexpr>* evaluated to `true` is assigned to *<sequence₁>*.

TeXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level TeX errors.

<code>\seq_set_from_function:NnN</code>	<code>\seq_set_from_function:NnN <seq var> {<loop code>} <function></code>
<code>\seq_gset_from_function:NnN</code>	

New: 2018-04-06

Sets the *<seq var>* equal to a sequence whose items are obtained by *x*-expanding *<loop code>* *<function>*. This expansion must result in successive calls to the *<function>* with no nonexpandable tokens in between. More precisely the *<function>* is replaced by a wrapper function that inserts the appropriate separators between items in the sequence. The *<loop code>* must be expandable; it can be for example `\tl_map_function:NN <tl var>` or `\clist_map_function:nN {<clist>}` or `\int_step_function:nnnN {<initial value>} {<step>} {<final value>}`.

<code>\seq_set_from_inline_x:Nnn</code>	<code>\seq_set_from_inline_x:Nnn <seq var> {<loop code>} {<inline code>}</code>
<code>\seq_gset_from_inline_x:Nnn</code>	

New: 2018-04-06

Sets the *<seq var>* equal to a sequence whose items are obtained by *x*-expanding *<loop code>* applied to a *<function>* derived from the *<inline code>*. A *<function>* is defined, that takes one argument, *x*-expands the *<inline code>* with that argument as #1, then adds appropriate separators to turn the result into an item of the sequence. The *x*-expansion of *<loop code>* *<function>* must result in successive calls to the *<function>* with no nonexpandable tokens in between. The *<loop code>* must be expandable; it can be for example `\tl_map_function:NN <tl var>` or `\clist_map_function:nN {<clist>}` or `\int_step_function:nnnN {<initial value>} {<step>} {<final value>}`, but not the analogous “inline” mappings.

12 Additions to l3sys

`\c_sys_engine_version_str`

New: 2018-05-02

The version string of the current engine, in the same form as given in the banner issued when running a job. For pdfTeX and LuaTeX this is of the form

$$\langle major \rangle . \langle minor \rangle . \langle revision \rangle$$

For XeTeX, the form is

$$\langle major \rangle . \langle minor \rangle$$

For pTeX and upTeX, only releases since TeX Live 2018 make the data available, and the form is more complex, as it comprises the pTeX version, the upTeX version and the e-pTeX version.

$$p \langle major \rangle . \langle minor \rangle . \langle revision \rangle - u \langle major \rangle . \langle minor \rangle - \langle epTeX \rangle$$

where the u part is only present for upTeX.

`\sys_if_rand_exist_p: *`
`\sys_if_rand_exist:TF TF *`

New: 2017-05-27

`\sys_if_rand_exist_p:`
`\sys_if_rand_exist:TF {true code} {false code}`

Tests if the engine has a pseudo-random number generator. Currently this is the case in pdfTeX, LuaTeX, pTeX, upTeX and recent releases of XeTeX.

13 Additions to l3tl

<code>\tl_range_braced:Nnn</code>	★	<code>\tl_range_braced:Nnn <tl var> {<start index>} {<end index>}</code>
<code>\tl_range_braced:cnn</code>	★	<code>\tl_range_braced:nnn {<token list>} {<start index>} {<end index>}</code>
<code>\tl_range_braced:nnn</code>	★	<code>\tl_range_unbraced:Nnn <tl var> {<start index>} {<end index>}</code>
<code>\tl_range_unbraced:Nnn</code>	★	<code>\tl_range_unbraced:nnn {<token list>} {<start index>} {<end index>}</code>
<code>\tl_range_unbraced:cnn</code>	★	Leaves in the input stream the items from the <i><start index></i> to the <i><end index></i> inclusive, using the same indexing as <code>\tl_range:nnn</code> . Spaces are ignored. Regardless of whether items appear with or without braces in the <i><token list></i> , the <code>\tl_range_braced:nnn</code> function wraps each item in braces, while <code>\tl_range_unbraced:nnn</code> does not (overall it removes an outer set of braces). For instance,
<code>\tl_range_unbraced:nnn</code>	★	

New: 2017-07-15

```

\tl_iow_term:x { \tl_range_braced:nnn { abcd~{e}}f } { 2 } { 5 } }
\tl_iow_term:x { \tl_range_braced:nnn { abcd~{e}}f } { -4 } { -1 } }
\tl_iow_term:x { \tl_range_braced:nnn { abcd~{e}}f } { -2 } { -1 } }
\tl_iow_term:x { \tl_range_braced:nnn { abcd~{e}}f } { 0 } { -1 } }

```

prints `{b}{c}{d}{e}}`, `{c}{d}{e}}{f}`, `{e}}{f}`, and an empty line to the terminal, while

```

\tl_iow_term:x { \tl_range_unbraced:nnn { abcd~{e}}f } { 2 } { 5 } }
\tl_iow_term:x { \tl_range_unbraced:nnn { abcd~{e}}f } { -4 } { -1 } }
\tl_iow_term:x { \tl_range_unbraced:nnn { abcd~{e}}f } { -2 } { -1 } }
\tl_iow_term:x { \tl_range_unbraced:nnn { abcd~{e}}f } { 0 } { -1 } }

```

prints `bcde{}`, `cde{f}`, `e{f}`, and an empty line to the terminal. Because braces are removed, the result of `\tl_range_unbraced:nnn` may have a different number of items as for `\tl_range:nnn` or `\tl_range_braced:nnn`. In cases where preserving spaces is important, consider the slower function `\tl_range:nnn`.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<item>* does not expand further when appearing in an x-type argument expansion.

<code>\tl_build_begin:N</code>	<code>\tl_build_begin:N <tl var></code>
<code>\tl_build_gbegin:N</code>	Clears the <i><tl var></i> and sets it up to support other <code>\tl_build_...</code> functions, which allow accumulating large numbers of tokens piece by piece much more efficiently than standard l3tl functions. Until <code>\tl_build_end:N <tl var></code> is called, applying any function from l3tl other than <code>\tl_build_...</code> will lead to incorrect results. The <code>begin</code> and <code>gbegin</code> functions must be used for local and global <i><tl var></i> respectively.

New: 2018-04-01

<code>\tl_build_clear:N</code>	<code>\tl_build_clear:N <tl var></code>
<code>\tl_build_gclear:N</code>	Clears the <i><tl var></i> and sets it up to support other <code>\tl_build_...</code> functions. The <code>clear</code> and <code>gclear</code> functions must be used for local and global <i><tl var></i> respectively.

New: 2018-04-01

```

\tl_build_put_left:Nn
\tl_build_put_left:Nx
\tl_build_gput_left:Nn
\tl_build_gput_left:Nx
\tl_build_put_right:Nn
\tl_build_put_right:Nx
\tl_build_gput_right:Nn
\tl_build_gput_right:Nx

```

New: 2018-04-01

```

\tl_build_get:NN

```

New: 2018-04-01

```

\tl_build_end:N
\tl_build_gend:N

```

New: 2018-04-01

```

\tl_build_put_left:Nn <tl var> {<tokens>}
\tl_build_put_right:Nn <tl var> {<tokens>}

```

Adds *<tokens>* to the left or right side of the current contents of *<tl var>*. The *<tl var>* must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `put` and `gput` functions must be used for local and global *<tl var>* respectively. The `right` functions are about twice faster than the `left` functions.

```

\tl_build_get:N <tl var1> <tl var2>

```

Stores the contents of the *<tl var₁>* in the *<tl var₂>*. The *<tl var₁>* must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The *<tl var₂>* is a “normal” token list variable, assigned locally using `\tl_set:Nn`.

```

\tl_build_end:N <tl var>

```

Gets the contents of *<tl var>* and stores that into the *<tl var>* using `\tl_set:Nn` or `\tl_gset:Nn`. The *<tl var>* must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `end` and `gend` functions must be used for local and global *<tl var>* respectively. These functions completely remove the setup code that enabled *<tl var>* to be used for other `\tl_build_...` functions.

14 Additions to l3token

```

\c_catcode_active_space_tl

```

New: 2017-08-07

```

\char_to_utfviii_bytes:n ★

```

New: 2020-01-09

Token list containing one character with category code 13, (“active”), and character code 32 (space).

```

\char_to_utfviii_bytes:n {<codepoint>}

```

Converts the (Unicode) *<codepoint>* to UTF-8 bytes. The expansion of this function comprises four brace groups, each of which will contain a hexadecimal value: the appropriate byte. As UTF-8 is a variable-length, one or more of the groups may be empty: the bytes read in the logical order, such that a two-byte codepoint will have groups **#1** and **#2** filled and **#3** and **#4** empty.

```

\char_to_nfd:N ☆

```

New: 2020-01-02

```

\char_to_nfd:N <char>

```

Converts the *<char>* to the Unicode Normalization Form Canonical Decomposition. The category code of the generated character is the same as the *<char>*. With 8-bit engines, no change is made to the character.

<code>\peek_catcode_collect_inline:Nn</code>	<code>\peek_catcode_collect_inline:Nn <test token> {<inline code>}</code>
<code>\peek_charcode_collect_inline:Nn</code>	<code>\peek_charcode_collect_inline:Nn <test token> {<inline code>}</code>
<code>\peek_meaning_collect_inline:Nn</code>	<code>\peek_meaning_collect_inline:Nn <test token> {<inline code>}</code>

New: 2018-09-23

Collects and removes tokens from the input stream until finding a token that does not match the `<test token>` (as defined by the test `\token_if_eq_catcode:NNTF` or `\token_if_eq_charcode:NNTF` or `\token_if_eq_meaning:NNTF`). The collected tokens are passed to the `<inline code>` as #1. When begin-group or end-group tokens (usually `{` or `}`) are collected they are replaced by implicit `\c_group_begin_token` and `\c_group_end_token`, and when spaces (including `\c_space_token`) are collected they are replaced by explicit spaces.

For example the following code prints “Hello” to the terminal and leave “, world!” in the input stream.

```
\peek_catcode_collect_inline:Nn A { \iow_term:n {#1} } Hello,~world!
```

Another example is that the following code tests if the next token is `*`, ignoring intervening spaces, but putting them back using `#1` if there is no `*`.

```
\peek_meaning_collect_inline:Nn \c_space_token
{ \peek_charcode:NNTF * { star } { no~star #1 } }
```

<code>\peek_remove_spaces:n</code>	<code>\peek_remove_spaces:n {<code>}</code>
------------------------------------	---

New: 2018-10-01

Removes explicit and implicit space tokens (category code 10 and character code 32) from the input stream, then inserts `<code>`.

Part XXXVII

Implementation

1 l3bootstrap implementation

```
1 <*package>
2 <@@=kernel>
```

1.1 LuaTeX-specific code

Depending on the versions available, the L^AT_EX format may not have the raw `\Umath` primitive names available. We fix that globally: it should cause no issues. Older LuaTeX versions do not have a pre-built table of the primitive names here so sort one out ourselves. These end up globally-defined but at that is true with a newer format anyway and as they all start `\U` this should be reasonably safe.

```
3 \begingroup
4   \expandafter\ifx\csname directlua\endcsname\relax
5   \else
6     \directlua{%
7       local i
8       local t = { }
```

```

9      for _,i in pairs(tex.extraprimatives("luatex")) do
10         if string.match(i,"^U") then
11             if not string.match(i,"^Uchar$") then %$
12                 table.insert(t,i)
13             end
14         end
15     end
16     tex.enableprimitives("", t)
17 }%
18 \fi
19 \endgroup

```

1.2 The `\pdfstrcmp` primitive in \XeTeX

Only \pdfTeX has a primitive called `\pdfstrcmp`. The \XeTeX version is just `\strcmp`, so there is some shuffling to do. As this is still a real primitive, using the \pdfTeX name is “safe”.

```

20 \begingroup\expandafter\expandafter\expandafter\endgroup
21 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
22 \let\pdfstrcmp\strcmp
23 \fi

```

1.3 Loading support Lua code

When \LuaTeX is used there are various pieces of Lua code which need to be loaded. The code itself is defined in `l3luatex` and is extracted into a separate file. Thus here the task is to load the Lua code both now and (if required) at the start of each job.

```

24 \begingroup\expandafter\expandafter\expandafter\endgroup
25 \expandafter\ifx\csname directlua\endcsname\relax
26 \else
27     \ifnum\luatexversion<95 %
28     \else

```

For \LuaTeX we make sure the basic support is loaded: this is only necessary in plain.

```

29     \begingroup\expandafter\expandafter\expandafter\endgroup
30     \expandafter\ifx\csname newcatcodetable\endcsname\relax
31         \input{ltluatex}%
32     \fi
33     \directlua{require("expl3")}%

```

As the user might be making a custom format, no assumption is made about matching package mode with only loading the Lua code once. Instead, a query to Lua reveals what mode is in operation.

```

34     \ifnum 0%
35         \directlua{
36             if status.ini_version then
37                 tex.write("1")
38             end
39         }>0 %
40     \everyjob\expandafter{%
41         \the\expandafter\everyjob
42         \csname\detokenize{lua_now:n}\endcsname{require("expl3")}%
43     }%

```

```

44 \fi
45 \fi
46 \fi

```

1.4 Engine requirements

The code currently requires ε -TeX and functionality equivalent to `\pdfstrcmp`, and also driver and Unicode character support. This is available in a reasonably-wide range of engines.

```

47 \begingroup
48 \def\next{\endgroup}%
49 \def\ShortText{Required primitives not found}%
50 \def\LongText%
51 {%
52 LaTeX3 requires the e-TeX primitives and additional functionality as
53 described in the README file.
54 \LineBreak
55 These are available in the engines\LineBreak
56 - pdfTeX v1.40\LineBreak
57 - XeTeX v0.99992\LineBreak
58 - LuaTeX v0.95\LineBreak
59 - e-(u)pTeX mid-2012\LineBreak
60 or later.\LineBreak
61 \LineBreak
62}%
63 \ifnum0%
64 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
65 \else
66 \expandafter\ifx\csname pdftexversion\endcsname\relax
67 \expandafter\ifx\csname Ucharcat\endcsname\relax
68 \expandafter\ifx\csname kanjiskip\endcsname\relax
69 \else
70 1%
71 \fi
72 \else
73 1%
74 \fi
75 \else
76 \ifnum\pdftexversion<140 \else 1\fi
77 \fi
78 \fi
79 \expandafter\ifx\csname directlua\endcsname\relax
80 \else
81 \ifnum\luatexversion<76 \else 1\fi
82 \fi
83 =0 %
84 \newlinechar'\^^J %
85 \def\LineBreak{\noexpand\MessageBreak}%
86 \expandafter\ifx\csname PackageError\endcsname\relax
87 \def\LineBreak{\^^J}%
88 \def\PackageError#1#2#3%
89 {%
90 \errhelp{#3}%

```

```

91         \errmessage{#1 Error: #2}%
92     }%
93 \fi
94 \edef\next
95     {%
96         \noexpand\PackageError{expl3}{\ShortText}
97         {\LongText Loading of expl3 will abort!}%
98         \endgroup
99         \noexpand\endinput
100     }%
101 \fi
102 \next

```

1.5 Extending allocators

The ability to extend \TeX ’s allocation routine to allow for $\varepsilon\text{-}\text{\TeX}$ has been around since 1997 in the `etex` package. Loading this support is delayed until here as we are now sure that the $\varepsilon\text{-}\text{\TeX}$ extensions and `\pdfstrcmp` or equivalent are available. Thus there is no danger of an “uncontrolled” error if the engine requirements are not met.

For $\text{\LaTeX}2_{\varepsilon}$ we need to make sure that the extended pool is being used: `expl3` uses a lot of registers. For formats from 2015 onward there is nothing to do as this is automatic. For older formats, the `etex` package needs to be loaded to do the job. In that case, some inserts are reserved also as these have to be from the standard pool. Note that `\reserveinserts` is `\outer` and so is accessed here by `csname`. In earlier versions, loading `etex` was done directly and so `\reserveinserts` appeared in the code: this then required a `\relax` after `\RequirePackage` to prevent an error with “unsafe” definitions as seen for example with `capoptions`. The optional loading here is done using a group and `\ifx` test as we are not quite in the position to have a single name for `\pdfstrcmp` just yet.

```

103 \begingroup
104   \def\@tempa{LaTeX2e}%
105   \def\next{}%
106   \ifx\fmtname\@tempa
107     \expandafter\ifx\csname extrafloats\endcsname\relax
108       \def\next
109         {%
110           \RequirePackage{etex}%
111           \csname reserveinserts\endcsname{32}%
112         }%
113   \fi
114 \fi
115 \expandafter\endgroup
116 \next

```

1.6 The $\text{\LaTeX}3$ code environment

The code environment is now set up.

`\ExplSyntaxOff` Before changing any category codes, in package mode we need to save the situation before loading. Note the set up here means that once applied `\ExplSyntaxOff` becomes a “do nothing” command until `\ExplSyntaxOn` is used.

```

117 \protected\edef\ExplSyntaxOff

```

```

118 {%
119   \protected\def\noexpand\ExplSyntaxOff{%
120     \catcode 9 = \the\catcode 9\relax
121     \catcode 32 = \the\catcode 32\relax
122     \catcode 34 = \the\catcode 34\relax
123     \catcode 38 = \the\catcode 38\relax
124     \catcode 58 = \the\catcode 58\relax
125     \catcode 94 = \the\catcode 94\relax
126     \catcode 95 = \the\catcode 95\relax
127     \catcode 124 = \the\catcode 124\relax
128     \catcode 126 = \the\catcode 126\relax
129     \endlinechar = \the\endlinechar\relax
130     \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0\relax
131   }%

```

(End definition for `\ExplSyntaxOff`. This function is documented on page 7.)

The code environment is now set up.

```

132 \catcode 9 = 9\relax
133 \catcode 32 = 9\relax
134 \catcode 34 = 12\relax
135 \catcode 38 = 4\relax
136 \catcode 58 = 11\relax
137 \catcode 94 = 7\relax
138 \catcode 95 = 11\relax
139 \catcode 124 = 12\relax
140 \catcode 126 = 10\relax
141 \endlinechar = 32\relax

```

`\l__kernel_expl_bool` The status for experimental code syntax: this is on at present.

```

142 \chardef\l__kernel_expl_bool = 1\relax

```

(End definition for `\l__kernel_expl_bool`.)

`\ExplSyntaxOn` The idea here is that multiple `\ExplSyntaxOn` calls are not going to mess up category codes, and that multiple calls to `\ExplSyntaxOff` are also not wasting time. Applying `\ExplSyntaxOn` alters the definition of `\ExplSyntaxOff` and so in package mode this function should not be used until after the end of the loading process!

```

143 \protected \def \ExplSyntaxOn
144 {
145   \bool_if:NF \l__kernel_expl_bool
146   {
147     \cs_set_protected:Npx \ExplSyntaxOff
148     {
149       \char_set_catcode:nn { 9 } { \char_value_catcode:n { 9 } }
150       \char_set_catcode:nn { 32 } { \char_value_catcode:n { 32 } }
151       \char_set_catcode:nn { 34 } { \char_value_catcode:n { 34 } }
152       \char_set_catcode:nn { 38 } { \char_value_catcode:n { 38 } }
153       \char_set_catcode:nn { 58 } { \char_value_catcode:n { 58 } }
154       \char_set_catcode:nn { 94 } { \char_value_catcode:n { 94 } }
155       \char_set_catcode:nn { 95 } { \char_value_catcode:n { 95 } }
156       \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
157       \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
158       \tex_endlinechar:D =
159       \tex_the:D \tex_endlinechar:D \scan_stop:

```

```

160         \bool_set_false:N \l__kernel_expl_bool
161         \cs_set_protected:Npn \ExplSyntaxOff { }
162     }
163 }
164 \char_set_catcode_ignore:n { 9 } % tab
165 \char_set_catcode_ignore:n { 32 } % space
166 \char_set_catcode_other:n { 34 } % double quote
167 \char_set_catcode_alignment:n { 38 } % ampersand
168 \char_set_catcode_letter:n { 58 } % colon
169 \char_set_catcode_math_superscript:n { 94 } % circumflex
170 \char_set_catcode_letter:n { 95 } % underscore
171 \char_set_catcode_other:n { 124 } % pipe
172 \char_set_catcode_space:n { 126 } % tilde
173 \tex_endlinechar:D = 32 \scan_stop:
174 \bool_set_true:N \l__kernel_expl_bool
175 }

```

(End definition for `\ExplSyntaxOn`. This function is documented on page 7.)

```
176 \</package>
```

2 l3names implementation

```
177 \<package>
```

The prefix here is `kernel`. A few places need `@@` to be left as is; this is obtained as `@@@`.

```
178 \<@@=kernel>
```

The code here simply renames all of the primitives to new, internal, names.

The `\let` primitive is renamed by hand first as it is essential for the entire process to follow. This also uses `\global`, as that way we avoid leaving an unneeded csname in the hash table.

```

179 \let \tex_global:D \global
180 \let \tex_let:D \let

```

Everything is inside a (rather long) group, which keeps `__kernel_primitive:NN` trapped.

```
181 \beginingroup
```

`__kernel_primitive:NN` A temporary function to actually do the renaming.

```

182 \long \def \__kernel_primitive:NN #1#2
183 { \tex_global:D \tex_let:D #2 #1 }

```

(End definition for `__kernel_primitive:NN`.)

To allow extracting “just the names”, a bit of DocStrip fiddling.

```

184 \</package>
185 \<names | package>

```

In the current incarnation of this package, all TeX primitives are given a new name of the form `\tex_oldname:D`. But first three special cases which have symbolic original names. These are given modified new names, so that they may be entered without catcode tricks.

```

186 \__kernel_primitive:NN \tex_space:D
187 \__kernel_primitive:NN \tex_italiccorrection:D
188 \__kernel_primitive:NN \tex_hyphen:D

```


Now all the other primitives.

189	<code>_kernel_primitive:NN</code>	<code>\above</code>	<code>\tex_above:D</code>
190	<code>_kernel_primitive:NN</code>	<code>\abovedisplayshortskip</code>	<code>\tex_abovedisplayshortskip:D</code>
191	<code>_kernel_primitive:NN</code>	<code>\abovedisplayskip</code>	<code>\tex_abovedisplayskip:D</code>
192	<code>_kernel_primitive:NN</code>	<code>\abovewithdelims</code>	<code>\tex_abovewithdelims:D</code>
193	<code>_kernel_primitive:NN</code>	<code>\accent</code>	<code>\tex_accent:D</code>
194	<code>_kernel_primitive:NN</code>	<code>\adjdemerits</code>	<code>\tex_adjdemerits:D</code>
195	<code>_kernel_primitive:NN</code>	<code>\advance</code>	<code>\tex_advance:D</code>
196	<code>_kernel_primitive:NN</code>	<code>\afterassignment</code>	<code>\tex_afterassignment:D</code>
197	<code>_kernel_primitive:NN</code>	<code>\aftergroup</code>	<code>\tex_aftergroup:D</code>
198	<code>_kernel_primitive:NN</code>	<code>\atop</code>	<code>\tex_atop:D</code>
199	<code>_kernel_primitive:NN</code>	<code>\atopwithdelims</code>	<code>\tex_atopwithdelims:D</code>
200	<code>_kernel_primitive:NN</code>	<code>\badness</code>	<code>\tex_badness:D</code>
201	<code>_kernel_primitive:NN</code>	<code>\baselineskip</code>	<code>\tex_baselineskip:D</code>
202	<code>_kernel_primitive:NN</code>	<code>\batchmode</code>	<code>\tex_batchmode:D</code>
203	<code>_kernel_primitive:NN</code>	<code>\begingroup</code>	<code>\tex_begingroup:D</code>
204	<code>_kernel_primitive:NN</code>	<code>\belowdisplayshortskip</code>	<code>\tex_belowdisplayshortskip:D</code>
205	<code>_kernel_primitive:NN</code>	<code>\belowdisplayskip</code>	<code>\tex_belowdisplayskip:D</code>
206	<code>_kernel_primitive:NN</code>	<code>\binoppenalty</code>	<code>\tex_binoppenalty:D</code>
207	<code>_kernel_primitive:NN</code>	<code>\botmark</code>	<code>\tex_botmark:D</code>
208	<code>_kernel_primitive:NN</code>	<code>\box</code>	<code>\tex_box:D</code>
209	<code>_kernel_primitive:NN</code>	<code>\boxmaxdepth</code>	<code>\tex_boxmaxdepth:D</code>
210	<code>_kernel_primitive:NN</code>	<code>\brokenpenalty</code>	<code>\tex_brokenpenalty:D</code>
211	<code>_kernel_primitive:NN</code>	<code>\catcode</code>	<code>\tex_catcode:D</code>
212	<code>_kernel_primitive:NN</code>	<code>\char</code>	<code>\tex_char:D</code>
213	<code>_kernel_primitive:NN</code>	<code>\chardef</code>	<code>\tex_chardef:D</code>
214	<code>_kernel_primitive:NN</code>	<code>\cleaders</code>	<code>\tex_cleaders:D</code>
215	<code>_kernel_primitive:NN</code>	<code>\closein</code>	<code>\tex_closein:D</code>
216	<code>_kernel_primitive:NN</code>	<code>\closeout</code>	<code>\tex_closeout:D</code>
217	<code>_kernel_primitive:NN</code>	<code>\clubpenalty</code>	<code>\tex_clubpenalty:D</code>
218	<code>_kernel_primitive:NN</code>	<code>\copy</code>	<code>\tex_copy:D</code>
219	<code>_kernel_primitive:NN</code>	<code>\count</code>	<code>\tex_count:D</code>
220	<code>_kernel_primitive:NN</code>	<code>\countdef</code>	<code>\tex_countdef:D</code>
221	<code>_kernel_primitive:NN</code>	<code>\cr</code>	<code>\tex_cr:D</code>
222	<code>_kernel_primitive:NN</code>	<code>\crcr</code>	<code>\tex_crcr:D</code>
223	<code>_kernel_primitive:NN</code>	<code>\csname</code>	<code>\tex_csname:D</code>
224	<code>_kernel_primitive:NN</code>	<code>\day</code>	<code>\tex_day:D</code>
225	<code>_kernel_primitive:NN</code>	<code>\deadcycles</code>	<code>\tex_deadcycles:D</code>
226	<code>_kernel_primitive:NN</code>	<code>\def</code>	<code>\tex_def:D</code>
227	<code>_kernel_primitive:NN</code>	<code>\defaultthyphenchar</code>	<code>\tex_defaultthyphenchar:D</code>
228	<code>_kernel_primitive:NN</code>	<code>\defaultskewchar</code>	<code>\tex_defaultskewchar:D</code>
229	<code>_kernel_primitive:NN</code>	<code>\delcode</code>	<code>\tex_delcode:D</code>
230	<code>_kernel_primitive:NN</code>	<code>\delimiter</code>	<code>\tex_delimiter:D</code>
231	<code>_kernel_primitive:NN</code>	<code>\delimiterfactor</code>	<code>\tex_delimiterfactor:D</code>
232	<code>_kernel_primitive:NN</code>	<code>\delimitershortfall</code>	<code>\tex_delimitershortfall:D</code>
233	<code>_kernel_primitive:NN</code>	<code>\dimen</code>	<code>\tex_dimen:D</code>
234	<code>_kernel_primitive:NN</code>	<code>\dimendef</code>	<code>\tex_dimendef:D</code>
235	<code>_kernel_primitive:NN</code>	<code>\discretionary</code>	<code>\tex_discretionary:D</code>
236	<code>_kernel_primitive:NN</code>	<code>\displayindent</code>	<code>\tex_displayindent:D</code>
237	<code>_kernel_primitive:NN</code>	<code>\displaylimits</code>	<code>\tex_displaylimits:D</code>
238	<code>_kernel_primitive:NN</code>	<code>\displaystyle</code>	<code>\tex_displaystyle:D</code>
239	<code>_kernel_primitive:NN</code>	<code>\displaywidowpenalty</code>	<code>\tex_displaywidowpenalty:D</code>
240	<code>_kernel_primitive:NN</code>	<code>\displaywidth</code>	<code>\tex_displaywidth:D</code>
241	<code>_kernel_primitive:NN</code>	<code>\divide</code>	<code>\tex_divide:D</code>

242	_kernel_primitive:NN	\doublehyphendemerits	\tex_doublehyphendemerits:D
243	_kernel_primitive:NN	\dp	\tex_dp:D
244	_kernel_primitive:NN	\dump	\tex_dump:D
245	_kernel_primitive:NN	\edef	\tex_edef:D
246	_kernel_primitive:NN	\else	\tex_else:D
247	_kernel_primitive:NN	\emergencystretch	\tex_emergencystretch:D
248	_kernel_primitive:NN	\end	\tex_end:D
249	_kernel_primitive:NN	\endcsname	\tex_endcsname:D
250	_kernel_primitive:NN	\endgroup	\tex_endgroup:D
251	_kernel_primitive:NN	\endinput	\tex_endinput:D
252	_kernel_primitive:NN	\endlinechar	\tex_endlinechar:D
253	_kernel_primitive:NN	\eqno	\tex_eqno:D
254	_kernel_primitive:NN	\errhelp	\tex_errhelp:D
255	_kernel_primitive:NN	\errmessage	\tex_errmessage:D
256	_kernel_primitive:NN	\errorcontextlines	\tex_errorcontextlines:D
257	_kernel_primitive:NN	\errorstopmode	\tex_errorstopmode:D
258	_kernel_primitive:NN	\escapechar	\tex_escapechar:D
259	_kernel_primitive:NN	\everycr	\tex_everycr:D
260	_kernel_primitive:NN	\everydisplay	\tex_everydisplay:D
261	_kernel_primitive:NN	\everyhbox	\tex_everyhbox:D
262	_kernel_primitive:NN	\everyjob	\tex_everyjob:D
263	_kernel_primitive:NN	\everymath	\tex_everymath:D
264	_kernel_primitive:NN	\everypar	\tex_everypar:D
265	_kernel_primitive:NN	\everyvbox	\tex_everyvbox:D
266	_kernel_primitive:NN	\exhyphenpenalty	\tex_exhyphenpenalty:D
267	_kernel_primitive:NN	\expandafter	\tex_expandafter:D
268	_kernel_primitive:NN	\fam	\tex_fam:D
269	_kernel_primitive:NN	\fi	\tex_fi:D
270	_kernel_primitive:NN	\finalhyphendemerits	\tex_finalhyphendemerits:D
271	_kernel_primitive:NN	\firstmark	\tex_firstmark:D
272	_kernel_primitive:NN	\floatingpenalty	\tex_floatingpenalty:D
273	_kernel_primitive:NN	\font	\tex_font:D
274	_kernel_primitive:NN	\fontdimen	\tex_fontdimen:D
275	_kernel_primitive:NN	\fontname	\tex_fontname:D
276	_kernel_primitive:NN	\futurelet	\tex_futurelet:D
277	_kernel_primitive:NN	\gdef	\tex_gdef:D
278	_kernel_primitive:NN	\global	\tex_global:D
279	_kernel_primitive:NN	\globaldefs	\tex_globaldefs:D
280	_kernel_primitive:NN	\halign	\tex_halign:D
281	_kernel_primitive:NN	\hangafter	\tex_hangafter:D
282	_kernel_primitive:NN	\hangindent	\tex_hangindent:D
283	_kernel_primitive:NN	\hbadness	\tex_hbadness:D
284	_kernel_primitive:NN	\hbox	\tex_hbox:D
285	_kernel_primitive:NN	\hfil	\tex_hfil:D
286	_kernel_primitive:NN	\hfill	\tex_hfill:D
287	_kernel_primitive:NN	\hfilneg	\tex_hfilneg:D
288	_kernel_primitive:NN	\hfuzz	\tex_hfuzz:D
289	_kernel_primitive:NN	\hoffset	\tex_hoffset:D
290	_kernel_primitive:NN	\holdinginserts	\tex_holdinginserts:D
291	_kernel_primitive:NN	\hrule	\tex_hrule:D
292	_kernel_primitive:NN	\hsize	\tex_hsize:D
293	_kernel_primitive:NN	\hskip	\tex_hskip:D
294	_kernel_primitive:NN	\hss	\tex_hss:D
295	_kernel_primitive:NN	\ht	\tex_ht:D

296	_kernel_primitive:NN	\hyphenation	\tex_hyphenation:D
297	_kernel_primitive:NN	\hyphenchar	\tex_hyphenchar:D
298	_kernel_primitive:NN	\hyphenpenalty	\tex_hyphenpenalty:D
299	_kernel_primitive:NN	\if	\tex_if:D
300	_kernel_primitive:NN	\ifcase	\tex_ifcase:D
301	_kernel_primitive:NN	\ifcat	\tex_ifcat:D
302	_kernel_primitive:NN	\ifdim	\tex_ifdim:D
303	_kernel_primitive:NN	\ifeof	\tex_ifeof:D
304	_kernel_primitive:NN	\iffalse	\tex_iffalse:D
305	_kernel_primitive:NN	\ifhbox	\tex_ifhbox:D
306	_kernel_primitive:NN	\ifhmode	\tex_ifhmode:D
307	_kernel_primitive:NN	\ifinner	\tex_ifinner:D
308	_kernel_primitive:NN	\ifmmode	\tex_ifmmode:D
309	_kernel_primitive:NN	\ifnum	\tex_ifnum:D
310	_kernel_primitive:NN	\ifodd	\tex_ifodd:D
311	_kernel_primitive:NN	\iftrue	\tex_iftrue:D
312	_kernel_primitive:NN	\ifvbox	\tex_ifvbox:D
313	_kernel_primitive:NN	\ifvmode	\tex_ifvmode:D
314	_kernel_primitive:NN	\ifvoid	\tex_ifvoid:D
315	_kernel_primitive:NN	\ifx	\tex_ifx:D
316	_kernel_primitive:NN	\ignorespaces	\tex_ignorespaces:D
317	_kernel_primitive:NN	\immediate	\tex_immediate:D
318	_kernel_primitive:NN	\indent	\tex_indent:D
319	_kernel_primitive:NN	\input	\tex_input:D
320	_kernel_primitive:NN	\inputlineno	\tex_inputlineno:D
321	_kernel_primitive:NN	\insert	\tex_insert:D
322	_kernel_primitive:NN	\insertpenalties	\tex_insertpenalties:D
323	_kernel_primitive:NN	\interlinepenalty	\tex_interlinepenalty:D
324	_kernel_primitive:NN	\jobname	\tex_jobname:D
325	_kernel_primitive:NN	\kern	\tex_kern:D
326	_kernel_primitive:NN	\language	\tex_language:D
327	_kernel_primitive:NN	\lastbox	\tex_lastbox:D
328	_kernel_primitive:NN	\lastkern	\tex_lastkern:D
329	_kernel_primitive:NN	\lastpenalty	\tex_lastpenalty:D
330	_kernel_primitive:NN	\lastskip	\tex_lastskip:D
331	_kernel_primitive:NN	\lccode	\tex_lccode:D
332	_kernel_primitive:NN	\leaders	\tex_leaders:D
333	_kernel_primitive:NN	\left	\tex_left:D
334	_kernel_primitive:NN	\lefthyphenmin	\tex_lefthyphenmin:D
335	_kernel_primitive:NN	\leftskip	\tex_leftskip:D
336	_kernel_primitive:NN	\leqno	\tex_leqno:D
337	_kernel_primitive:NN	\let	\tex_let:D
338	_kernel_primitive:NN	\limits	\tex_limits:D
339	_kernel_primitive:NN	\linepenalty	\tex_linepenalty:D
340	_kernel_primitive:NN	\lineskip	\tex_lineskip:D
341	_kernel_primitive:NN	\lineskiplimit	\tex_lineskiplimit:D
342	_kernel_primitive:NN	\long	\tex_long:D
343	_kernel_primitive:NN	\looseness	\tex_looseness:D
344	_kernel_primitive:NN	\lower	\tex_lower:D
345	_kernel_primitive:NN	\lowercase	\tex_lowercase:D
346	_kernel_primitive:NN	\mag	\tex_mag:D
347	_kernel_primitive:NN	\mark	\tex_mark:D
348	_kernel_primitive:NN	\mathaccent	\tex_mathaccent:D
349	_kernel_primitive:NN	\mathbin	\tex_mathbin:D

350	__kernel_primitive:NN	\mathchar	\tex_mathchar:D
351	__kernel_primitive:NN	\mathchardef	\tex_mathchardef:D
352	__kernel_primitive:NN	\mathchoice	\tex_mathchoice:D
353	__kernel_primitive:NN	\mathclose	\tex_mathclose:D
354	__kernel_primitive:NN	\mathcode	\tex_mathcode:D
355	__kernel_primitive:NN	\mathinner	\tex_mathinner:D
356	__kernel_primitive:NN	\mathop	\tex_mathop:D
357	__kernel_primitive:NN	\mathopen	\tex_mathopen:D
358	__kernel_primitive:NN	\mathord	\tex_mathord:D
359	__kernel_primitive:NN	\mathpunct	\tex_mathpunct:D
360	__kernel_primitive:NN	\mathrel	\tex_mathrel:D
361	__kernel_primitive:NN	\mathsurround	\tex_mathsurround:D
362	__kernel_primitive:NN	\maxdeadcycles	\tex_maxdeadcycles:D
363	__kernel_primitive:NN	\maxdepth	\tex_maxdepth:D
364	__kernel_primitive:NN	\meaning	\tex_meaning:D
365	__kernel_primitive:NN	\medmuskip	\tex_medmuskip:D
366	__kernel_primitive:NN	\message	\tex_message:D
367	__kernel_primitive:NN	\mkern	\tex_mkern:D
368	__kernel_primitive:NN	\month	\tex_month:D
369	__kernel_primitive:NN	\moveleft	\tex_moveleft:D
370	__kernel_primitive:NN	\moveright	\tex_moveright:D
371	__kernel_primitive:NN	\mskip	\tex_mskip:D
372	__kernel_primitive:NN	\multiply	\tex_multiply:D
373	__kernel_primitive:NN	\muskip	\tex_muskip:D
374	__kernel_primitive:NN	\muskipdef	\tex_muskipdef:D
375	__kernel_primitive:NN	\newlinechar	\tex_newlinechar:D
376	__kernel_primitive:NN	\noalign	\tex_noalign:D
377	__kernel_primitive:NN	\noboundary	\tex_noboundary:D
378	__kernel_primitive:NN	\noexpand	\tex_noexpand:D
379	__kernel_primitive:NN	\noindent	\tex_noindent:D
380	__kernel_primitive:NN	\nolimits	\tex_nolimits:D
381	__kernel_primitive:NN	\nonscript	\tex_nonscript:D
382	__kernel_primitive:NN	\nonstopmode	\tex_nonstopmode:D
383	__kernel_primitive:NN	\nulldelimiterspace	\tex_nulldelimiterspace:D
384	__kernel_primitive:NN	\nullfont	\tex_nullfont:D
385	__kernel_primitive:NN	\number	\tex_number:D
386	__kernel_primitive:NN	\omit	\tex_omit:D
387	__kernel_primitive:NN	\openin	\tex_openin:D
388	__kernel_primitive:NN	\openout	\tex_openout:D
389	__kernel_primitive:NN	\or	\tex_or:D
390	__kernel_primitive:NN	\outer	\tex_outer:D
391	__kernel_primitive:NN	\output	\tex_output:D
392	__kernel_primitive:NN	\outputpenalty	\tex_outputpenalty:D
393	__kernel_primitive:NN	\over	\tex_over:D
394	__kernel_primitive:NN	\overfullrule	\tex_overfullrule:D
395	__kernel_primitive:NN	\overline	\tex_overline:D
396	__kernel_primitive:NN	\overwithdelims	\tex_overwithdelims:D
397	__kernel_primitive:NN	\pagedepth	\tex_pagedepth:D
398	__kernel_primitive:NN	\pagefilllstretch	\tex_pagefilllstretch:D
399	__kernel_primitive:NN	\pagefillstretch	\tex_pagefillstretch:D
400	__kernel_primitive:NN	\pagefilstretch	\tex_pagefilstretch:D
401	__kernel_primitive:NN	\pagegoal	\tex_pagegoal:D
402	__kernel_primitive:NN	\pageshrink	\tex_pageshrink:D
403	__kernel_primitive:NN	\pagestretch	\tex_pagestretch:D

404	_kernel_primitive:NN	\pagetotal	\tex_pagetotal:D
405	_kernel_primitive:NN	\par	\tex_par:D
406	_kernel_primitive:NN	\parfillskip	\tex_parfillskip:D
407	_kernel_primitive:NN	\parindent	\tex_parindent:D
408	_kernel_primitive:NN	\parshape	\tex_parshape:D
409	_kernel_primitive:NN	\parskip	\tex_parskip:D
410	_kernel_primitive:NN	\patterns	\tex_patterns:D
411	_kernel_primitive:NN	\pausing	\tex_pausing:D
412	_kernel_primitive:NN	\penalty	\tex_penalty:D
413	_kernel_primitive:NN	\postdisplaypenalty	\tex_postdisplaypenalty:D
414	_kernel_primitive:NN	\predisplaypenalty	\tex_predisplaypenalty:D
415	_kernel_primitive:NN	\preplaysize	\tex_preplaysize:D
416	_kernel_primitive:NN	\pretolerance	\tex_pretolerance:D
417	_kernel_primitive:NN	\prevdepth	\tex_prevdepth:D
418	_kernel_primitive:NN	\prevgraf	\tex_prevgraf:D
419	_kernel_primitive:NN	\radical	\tex_radical:D
420	_kernel_primitive:NN	\raise	\tex_raise:D
421	_kernel_primitive:NN	\read	\tex_read:D
422	_kernel_primitive:NN	\relax	\tex_relax:D
423	_kernel_primitive:NN	\relpenalty	\tex_relpenalty:D
424	_kernel_primitive:NN	\right	\tex_right:D
425	_kernel_primitive:NN	\righthyphenmin	\tex_righthyphenmin:D
426	_kernel_primitive:NN	\rightskip	\tex_rightskip:D
427	_kernel_primitive:NN	\romannumeral	\tex_romannumeral:D
428	_kernel_primitive:NN	\scriptfont	\tex_scriptfont:D
429	_kernel_primitive:NN	\scriptscriptfont	\tex_scriptscriptfont:D
430	_kernel_primitive:NN	\scriptscriptstyle	\tex_scriptscriptstyle:D
431	_kernel_primitive:NN	\scriptspace	\tex_scriptspace:D
432	_kernel_primitive:NN	\scriptstyle	\tex_scriptstyle:D
433	_kernel_primitive:NN	\scrollmode	\tex_scrollmode:D
434	_kernel_primitive:NN	\setbox	\tex_setbox:D
435	_kernel_primitive:NN	\setlanguage	\tex_setlanguage:D
436	_kernel_primitive:NN	\sfcode	\tex_sfcode:D
437	_kernel_primitive:NN	\shipout	\tex_shipout:D
438	_kernel_primitive:NN	\show	\tex_show:D
439	_kernel_primitive:NN	\showbox	\tex_showbox:D
440	_kernel_primitive:NN	\showboxbreadth	\tex_showboxbreadth:D
441	_kernel_primitive:NN	\showboxdepth	\tex_showboxdepth:D
442	_kernel_primitive:NN	\showlists	\tex_showlists:D
443	_kernel_primitive:NN	\showthe	\tex_showthe:D
444	_kernel_primitive:NN	\skewchar	\tex_skewchar:D
445	_kernel_primitive:NN	\skip	\tex_skip:D
446	_kernel_primitive:NN	\skipdef	\tex_skipdef:D
447	_kernel_primitive:NN	\spacefactor	\tex_spacefactor:D
448	_kernel_primitive:NN	\spaceskip	\tex_spaceskip:D
449	_kernel_primitive:NN	\span	\tex_span:D
450	_kernel_primitive:NN	\special	\tex_special:D
451	_kernel_primitive:NN	\splitbotmark	\tex_splitbotmark:D
452	_kernel_primitive:NN	\splitfirstmark	\tex_splitfirstmark:D
453	_kernel_primitive:NN	\splitmaxdepth	\tex_splitmaxdepth:D
454	_kernel_primitive:NN	\splittopskip	\tex_splittopskip:D
455	_kernel_primitive:NN	\string	\tex_string:D
456	_kernel_primitive:NN	\tabskip	\tex_tabskip:D
457	_kernel_primitive:NN	\textfont	\tex_textfont:D

458	_kernel_primitive:NN	\textstyle	\tex_textstyle:D
459	_kernel_primitive:NN	\the	\tex_the:D
460	_kernel_primitive:NN	\thickmuskip	\tex_thickmuskip:D
461	_kernel_primitive:NN	\thinmuskip	\tex_thinmuskip:D
462	_kernel_primitive:NN	\time	\tex_time:D
463	_kernel_primitive:NN	\toks	\tex_toks:D
464	_kernel_primitive:NN	\toksdef	\tex_toksdef:D
465	_kernel_primitive:NN	\tolerance	\tex_tolerance:D
466	_kernel_primitive:NN	\topmark	\tex_topmark:D
467	_kernel_primitive:NN	\topskip	\tex_topskip:D
468	_kernel_primitive:NN	\tracingcommands	\tex_tracingcommands:D
469	_kernel_primitive:NN	\tracinglostchars	\tex_tracinglostchars:D
470	_kernel_primitive:NN	\tracingmacros	\tex_tracingmacros:D
471	_kernel_primitive:NN	\tracingonline	\tex_tracingonline:D
472	_kernel_primitive:NN	\tracingoutput	\tex_tracingoutput:D
473	_kernel_primitive:NN	\tracingpages	\tex_tracingpages:D
474	_kernel_primitive:NN	\tracingparagraphs	\tex_tracingparagraphs:D
475	_kernel_primitive:NN	\tracingrestores	\tex_tracingrestores:D
476	_kernel_primitive:NN	\tracingstats	\tex_tracingstats:D
477	_kernel_primitive:NN	\uccode	\tex_uccode:D
478	_kernel_primitive:NN	\uchyph	\tex_uchyph:D
479	_kernel_primitive:NN	\underline	\tex_underline:D
480	_kernel_primitive:NN	\unhbox	\tex_unhbox:D
481	_kernel_primitive:NN	\unhcopy	\tex_unhcopy:D
482	_kernel_primitive:NN	\unkern	\tex_unkern:D
483	_kernel_primitive:NN	\unpenalty	\tex_unpenalty:D
484	_kernel_primitive:NN	\unskip	\tex_unskip:D
485	_kernel_primitive:NN	\unvbox	\tex_unvbox:D
486	_kernel_primitive:NN	\unvcopy	\tex_unvcopy:D
487	_kernel_primitive:NN	\uppercase	\tex_uppercase:D
488	_kernel_primitive:NN	\vadjust	\tex_vadjust:D
489	_kernel_primitive:NN	\valign	\tex_valign:D
490	_kernel_primitive:NN	\vbadness	\tex_vbadness:D
491	_kernel_primitive:NN	\vbox	\tex_vbox:D
492	_kernel_primitive:NN	\vcenter	\tex_vcenter:D
493	_kernel_primitive:NN	\vfil	\tex_vfil:D
494	_kernel_primitive:NN	\vfill	\tex_vfill:D
495	_kernel_primitive:NN	\vfilneg	\tex_vfilneg:D
496	_kernel_primitive:NN	\vfuzz	\tex_vfuzz:D
497	_kernel_primitive:NN	\voffset	\tex_voffset:D
498	_kernel_primitive:NN	\vrule	\tex_vrule:D
499	_kernel_primitive:NN	\vsize	\tex_vsize:D
500	_kernel_primitive:NN	\vskip	\tex_vskip:D
501	_kernel_primitive:NN	\vsplit	\tex_vsplit:D
502	_kernel_primitive:NN	\vss	\tex_vss:D
503	_kernel_primitive:NN	\vtop	\tex_vtop:D
504	_kernel_primitive:NN	\wd	\tex_wd:D
505	_kernel_primitive:NN	\widowpenalty	\tex_widowpenalty:D
506	_kernel_primitive:NN	\write	\tex_write:D
507	_kernel_primitive:NN	\xdef	\tex_xdef:D
508	_kernel_primitive:NN	\xleaders	\tex_xleaders:D
509	_kernel_primitive:NN	\xspaceskip	\tex_xspaceskip:D
510	_kernel_primitive:NN	\year	\tex_year:D

Primitives introduced by ϵ -TeX.

511	<code>_kernel_primitive:NN</code>	<code>\beginL</code>	<code>\tex_beginL:D</code>
512	<code>_kernel_primitive:NN</code>	<code>\beginR</code>	<code>\tex_beginR:D</code>
513	<code>_kernel_primitive:NN</code>	<code>\botmarks</code>	<code>\tex_botmarks:D</code>
514	<code>_kernel_primitive:NN</code>	<code>\clubpenalties</code>	<code>\tex_clubpenalties:D</code>
515	<code>_kernel_primitive:NN</code>	<code>\currentgrouplevel</code>	<code>\tex_currentgrouplevel:D</code>
516	<code>_kernel_primitive:NN</code>	<code>\currentgrouptype</code>	<code>\tex_currentgrouptype:D</code>
517	<code>_kernel_primitive:NN</code>	<code>\currentifbranch</code>	<code>\tex_currentifbranch:D</code>
518	<code>_kernel_primitive:NN</code>	<code>\currentiflevel</code>	<code>\tex_currentiflevel:D</code>
519	<code>_kernel_primitive:NN</code>	<code>\currentifttype</code>	<code>\tex_currentifttype:D</code>
520	<code>_kernel_primitive:NN</code>	<code>\detokenize</code>	<code>\tex_detokenize:D</code>
521	<code>_kernel_primitive:NN</code>	<code>\dimexpr</code>	<code>\tex_dimexpr:D</code>
522	<code>_kernel_primitive:NN</code>	<code>\displaywidowpenalties</code>	<code>\tex_displaywidowpenalties:D</code>
523	<code>_kernel_primitive:NN</code>	<code>\endL</code>	<code>\tex_endL:D</code>
524	<code>_kernel_primitive:NN</code>	<code>\endR</code>	<code>\tex_endR:D</code>
525	<code>_kernel_primitive:NN</code>	<code>\eTeXrevision</code>	<code>\tex_eTeXrevision:D</code>
526	<code>_kernel_primitive:NN</code>	<code>\eTeXversion</code>	<code>\tex_eTeXversion:D</code>
527	<code>_kernel_primitive:NN</code>	<code>\everyeof</code>	<code>\tex_everyeof:D</code>
528	<code>_kernel_primitive:NN</code>	<code>\firstmarks</code>	<code>\tex_firstmarks:D</code>
529	<code>_kernel_primitive:NN</code>	<code>\fontchardp</code>	<code>\tex_fontchardp:D</code>
530	<code>_kernel_primitive:NN</code>	<code>\fontcharht</code>	<code>\tex_fontcharht:D</code>
531	<code>_kernel_primitive:NN</code>	<code>\fontcharic</code>	<code>\tex_fontcharic:D</code>
532	<code>_kernel_primitive:NN</code>	<code>\fontcharwd</code>	<code>\tex_fontcharwd:D</code>
533	<code>_kernel_primitive:NN</code>	<code>\glueexpr</code>	<code>\tex_glueexpr:D</code>
534	<code>_kernel_primitive:NN</code>	<code>\glueshrink</code>	<code>\tex_glueshrink:D</code>
535	<code>_kernel_primitive:NN</code>	<code>\glueshrinkorder</code>	<code>\tex_glueshrinkorder:D</code>
536	<code>_kernel_primitive:NN</code>	<code>\gluestretch</code>	<code>\tex_gluestretch:D</code>
537	<code>_kernel_primitive:NN</code>	<code>\gluestretchorder</code>	<code>\tex_gluestretchorder:D</code>
538	<code>_kernel_primitive:NN</code>	<code>\gluetomu</code>	<code>\tex_gluetomu:D</code>
539	<code>_kernel_primitive:NN</code>	<code>\ifcsname</code>	<code>\tex_ifcsname:D</code>
540	<code>_kernel_primitive:NN</code>	<code>\ifdefined</code>	<code>\tex_ifdefined:D</code>
541	<code>_kernel_primitive:NN</code>	<code>\iffontchar</code>	<code>\tex_iffontchar:D</code>
542	<code>_kernel_primitive:NN</code>	<code>\interactionmode</code>	<code>\tex_interactionmode:D</code>
543	<code>_kernel_primitive:NN</code>	<code>\interlinepenalties</code>	<code>\tex_interlinepenalties:D</code>
544	<code>_kernel_primitive:NN</code>	<code>\lastlinefit</code>	<code>\tex_lastlinefit:D</code>
545	<code>_kernel_primitive:NN</code>	<code>\lastnodetype</code>	<code>\tex_lastnodetype:D</code>
546	<code>_kernel_primitive:NN</code>	<code>\marks</code>	<code>\tex_marks:D</code>
547	<code>_kernel_primitive:NN</code>	<code>\middle</code>	<code>\tex_middle:D</code>
548	<code>_kernel_primitive:NN</code>	<code>\muexpr</code>	<code>\tex_muexpr:D</code>
549	<code>_kernel_primitive:NN</code>	<code>\mutoglu</code>	<code>\tex_mutoglu:D</code>
550	<code>_kernel_primitive:NN</code>	<code>\numexpr</code>	<code>\tex_numexpr:D</code>
551	<code>_kernel_primitive:NN</code>	<code>\pagediscards</code>	<code>\tex_pagediscards:D</code>
552	<code>_kernel_primitive:NN</code>	<code>\parshapedimen</code>	<code>\tex_parshapedimen:D</code>
553	<code>_kernel_primitive:NN</code>	<code>\parshapeindent</code>	<code>\tex_parshapeindent:D</code>
554	<code>_kernel_primitive:NN</code>	<code>\parshapelength</code>	<code>\tex_parshapelength:D</code>
555	<code>_kernel_primitive:NN</code>	<code>\predisplaydirection</code>	<code>\tex_predisplaydirection:D</code>
556	<code>_kernel_primitive:NN</code>	<code>\protected</code>	<code>\tex_protected:D</code>
557	<code>_kernel_primitive:NN</code>	<code>\readline</code>	<code>\tex_readline:D</code>
558	<code>_kernel_primitive:NN</code>	<code>\savinghyphcodes</code>	<code>\tex_savinghyphcodes:D</code>
559	<code>_kernel_primitive:NN</code>	<code>\savingvdiscards</code>	<code>\tex_savingvdiscards:D</code>
560	<code>_kernel_primitive:NN</code>	<code>\scantokens</code>	<code>\tex_scantokens:D</code>
561	<code>_kernel_primitive:NN</code>	<code>\showgroups</code>	<code>\tex_showgroups:D</code>
562	<code>_kernel_primitive:NN</code>	<code>\showifs</code>	<code>\tex_showifs:D</code>
563	<code>_kernel_primitive:NN</code>	<code>\showtokens</code>	<code>\tex_showtokens:D</code>

564	_kernel_primitive:NN	\splitbotmarks	\tex_splitbotmarks:D
565	_kernel_primitive:NN	\splitdiscards	\tex_splitdiscards:D
566	_kernel_primitive:NN	\splitfirstmarks	\tex_splitfirstmarks:D
567	_kernel_primitive:NN	\TeXeTstate	\tex_TeXeTstate:D
568	_kernel_primitive:NN	\topmarks	\tex_topmarks:D
569	_kernel_primitive:NN	\tracingassigns	\tex_tracingassigns:D
570	_kernel_primitive:NN	\tracinggroups	\tex_tracinggroups:D
571	_kernel_primitive:NN	\tracingifs	\tex_tracingifs:D
572	_kernel_primitive:NN	\tracingnesting	\tex_tracingnesting:D
573	_kernel_primitive:NN	\tracingscantokens	\tex_tracingscantokens:D
574	_kernel_primitive:NN	\unexpanded	\tex_unexpanded:D
575	_kernel_primitive:NN	\unless	\tex_unless:D
576	_kernel_primitive:NN	\widowpenalties	\tex_widowpenalties:D

Post- ϵ - \TeX primitives do not always end up with the same name in all engines, if indeed they are available cross-engine anyway. We therefore take the approach of preferring the shortest name that makes sense. First, we deal with the primitives introduced by pdf \TeX which directly relate to PDF output: these are copied with the names unchanged.

577	_kernel_primitive:NN	\pdfannot	\tex_pdfannot:D
578	_kernel_primitive:NN	\pdfcatalog	\tex_pdfcatalog:D
579	_kernel_primitive:NN	\pdfcompresslevel	\tex_pdfcompresslevel:D
580	_kernel_primitive:NN	\pdfcolorstack	\tex_pdfcolorstack:D
581	_kernel_primitive:NN	\pdfcolorstackinit	\tex_pdfcolorstackinit:D
582	_kernel_primitive:NN	\pdfcreationdate	\tex_pdfcreationdate:D
583	_kernel_primitive:NN	\pdfdecimaldigits	\tex_pdfdecimaldigits:D
584	_kernel_primitive:NN	\pdfdest	\tex_pdfdest:D
585	_kernel_primitive:NN	\pdfdestmargin	\tex_pdfdestmargin:D
586	_kernel_primitive:NN	\pdfendlink	\tex_pdfendlink:D
587	_kernel_primitive:NN	\pdfendthread	\tex_pdfendthread:D
588	_kernel_primitive:NN	\pdffontattr	\tex_pdffontattr:D
589	_kernel_primitive:NN	\pdffontname	\tex_pdffontname:D
590	_kernel_primitive:NN	\pdffontobjnum	\tex_pdffontobjnum:D
591	_kernel_primitive:NN	\pdfgamma	\tex_pdfgamma:D
592	_kernel_primitive:NN	\pdfimageapplygamma	\tex_pdfimageapplygamma:D
593	_kernel_primitive:NN	\pdfimagegamma	\tex_pdfimagegamma:D
594	_kernel_primitive:NN	\pdfgentounicode	\tex_pdfgentounicode:D
595	_kernel_primitive:NN	\pdfglyphptounicode	\tex_pdfglyphptounicode:D
596	_kernel_primitive:NN	\pdfhorigin	\tex_pdfhorigin:D
597	_kernel_primitive:NN	\pdfimagehicolor	\tex_pdfimagehicolor:D
598	_kernel_primitive:NN	\pdfimageresolution	\tex_pdfimageresolution:D
599	_kernel_primitive:NN	\pdfincludechars	\tex_pdfincludechars:D
600	_kernel_primitive:NN	\pdfinclusioncopyfonts	\tex_pdfinclusioncopyfonts:D
601	_kernel_primitive:NN	\pdfinclusionerrorlevel	
602		\tex_pdfinclusionerrorlevel:D	
603	_kernel_primitive:NN	\pdfinfo	\tex_pdfinfo:D
604	_kernel_primitive:NN	\pdflastannot	\tex_pdflastannot:D
605	_kernel_primitive:NN	\pdflastlink	\tex_pdflastlink:D
606	_kernel_primitive:NN	\pdflastobj	\tex_pdflastobj:D
607	_kernel_primitive:NN	\pdflastxform	\tex_pdflastxform:D
608	_kernel_primitive:NN	\pdflastximage	\tex_pdflastximage:D
609	_kernel_primitive:NN	\pdflastximagecolordepth	
610		\tex_pdflastximagecolordepth:D	
611	_kernel_primitive:NN	\pdflastximagepages	\tex_pdflastximagepages:D
612	_kernel_primitive:NN	\pdflinkmargin	\tex_pdflinkmargin:D

613	_kernel_primitive:NN	\pdfliteral	\tex_pdfliteral:D
614	_kernel_primitive:NN	\pdfmajorversion	\tex_pdfmajorversion:D
615	_kernel_primitive:NN	\pdfminorversion	\tex_pdfminorversion:D
616	_kernel_primitive:NN	\pdfnames	\tex_pdfnames:D
617	_kernel_primitive:NN	\pdfobj	\tex_pdfobj:D
618	_kernel_primitive:NN	\pdfobjcompresslevel	\tex_pdfobjcompresslevel:D
619	_kernel_primitive:NN	\pdfoutline	\tex_pdfoutline:D
620	_kernel_primitive:NN	\pdfoutput	\tex_pdfoutput:D
621	_kernel_primitive:NN	\pdfpageattr	\tex_pdfpageattr:D
622	_kernel_primitive:NN	\pdfpagesattr	\tex_pdfpagesattr:D
623	_kernel_primitive:NN	\pdfpagebox	\tex_pdfpagebox:D
624	_kernel_primitive:NN	\pdfpageref	\tex_pdfpageref:D
625	_kernel_primitive:NN	\pdfpageresources	\tex_pdfpageresources:D
626	_kernel_primitive:NN	\pdfpagesattr	\tex_pdfpagesattr:D
627	_kernel_primitive:NN	\pdfrefobj	\tex_pdfrefobj:D
628	_kernel_primitive:NN	\pdfrefxform	\tex_pdfrefxform:D
629	_kernel_primitive:NN	\pdfrefximage	\tex_pdfrefximage:D
630	_kernel_primitive:NN	\pdfrestore	\tex_pdfrestore:D
631	_kernel_primitive:NN	\pdfretval	\tex_pdfretval:D
632	_kernel_primitive:NN	\pdfsave	\tex_pdfsave:D
633	_kernel_primitive:NN	\pdfsetmatrix	\tex_pdfsetmatrix:D
634	_kernel_primitive:NN	\pdfstartlink	\tex_pdfstartlink:D
635	_kernel_primitive:NN	\pdfstartthread	\tex_pdfstartthread:D
636	_kernel_primitive:NN	\pdfsuppressptexinfo	\tex_pdfsuppressptexinfo:D
637	_kernel_primitive:NN	\pdfthread	\tex_pdfthread:D
638	_kernel_primitive:NN	\pdfthreadmargin	\tex_pdfthreadmargin:D
639	_kernel_primitive:NN	\pdftrailer	\tex_pdftrailer:D
640	_kernel_primitive:NN	\pdfuniqueresname	\tex_pdfuniqueresname:D
641	_kernel_primitive:NN	\pdfvorigin	\tex_pdfvorigin:D
642	_kernel_primitive:NN	\pdfxform	\tex_pdfxform:D
643	_kernel_primitive:NN	\pdfxformname	\tex_pdfxformname:D
644	_kernel_primitive:NN	\pdfximage	\tex_pdfximage:D
645	_kernel_primitive:NN	\pdfximagebbox	\tex_pdfximagebbox:D

These are not related to PDF output and either already appear in other engines without the \pdf prefix, or might reasonably do so at some future stage. We therefore drop the leading pdf here.

646	_kernel_primitive:NN	\ifpdfabsdim	\tex_ifabsdim:D
647	_kernel_primitive:NN	\ifpdfabsnum	\tex_ifabsnum:D
648	_kernel_primitive:NN	\ifpdfprimitive	\tex_ifprimitive:D
649	_kernel_primitive:NN	\pdfadjustspacing	\tex_adjustspacing:D
650	_kernel_primitive:NN	\pdfcopyfont	\tex_copyfont:D
651	_kernel_primitive:NN	\pdfdraftmode	\tex_draftmode:D
652	_kernel_primitive:NN	\pdfeachlinedepth	\tex_eachlinedepth:D
653	_kernel_primitive:NN	\pdfeachlineheight	\tex_eachlineheight:D
654	_kernel_primitive:NN	\pdfelapsedtime	\tex_elapsedtime:D
655	_kernel_primitive:NN	\pdffiledump	\tex_filedump:D
656	_kernel_primitive:NN	\pdffilemoddate	\tex_filemoddate:D
657	_kernel_primitive:NN	\pdffilesize	\tex_filesize:D
658	_kernel_primitive:NN	\pdffirstlineheight	\tex_firstlineheight:D
659	_kernel_primitive:NN	\pdffontexpand	\tex_fontexpand:D
660	_kernel_primitive:NN	\pdffontsize	\tex_fontsize:D
661	_kernel_primitive:NN	\pdfignoreddimen	\tex_ignoreddimen:D
662	_kernel_primitive:NN	\pdfinserttht	\tex_inserttht:D

```

663 \__kernel_primitive:NN \pdfastlinedepth \tex_lastlinedepth:D
664 \__kernel_primitive:NN \pdfastxpos \tex_lastxpos:D
665 \__kernel_primitive:NN \pdfastypos \tex_lastypos:D
666 \__kernel_primitive:NN \pdfmapfile \tex_mapfile:D
667 \__kernel_primitive:NN \pdfmapline \tex_mapline:D
668 \__kernel_primitive:NN \pdfmdfivesum \tex_mdfivesum:D
669 \__kernel_primitive:NN \pdfnoligatures \tex_noligatures:D
670 \__kernel_primitive:NN \pdfnormaldeviate \tex_normaldeviate:D
671 \__kernel_primitive:NN \pdfpageheight \tex_pageheight:D
672 \__kernel_primitive:NN \pdfpagewidth \tex_pagewidth:D
673 \__kernel_primitive:NN \pdfpkmode \tex_pkmode:D
674 \__kernel_primitive:NN \pdfpkresolution \tex_pkresolution:D
675 \__kernel_primitive:NN \pdfprimitive \tex_primitive:D
676 \__kernel_primitive:NN \pdfprotrudechars \tex_protrudechars:D
677 \__kernel_primitive:NN \pdfpxdimen \tex_pxdimen:D
678 \__kernel_primitive:NN \pdfrandomseed \tex_randomseed:D
679 \__kernel_primitive:NN \pdfresettimer \tex_resettimer:D
680 \__kernel_primitive:NN \pdfsavepos \tex_savepos:D
681 \__kernel_primitive:NN \pdfstrcmp \tex_strcmp:D
682 \__kernel_primitive:NN \pdfsetrandomseed \tex_setrandomseed:D
683 \__kernel_primitive:NN \pdfshellescape \tex_shellescape:D
684 \__kernel_primitive:NN \pdftracingfonts \tex_tracingfonts:D
685 \__kernel_primitive:NN \pdfuniformdeviate \tex_uniformdeviate:D

```

The version primitives are not related to PDF mode but are pdfTeX-specific, so again are carried forward unchanged.

```

686 \__kernel_primitive:NN \pdftexbanner \tex_pdftexbanner:D
687 \__kernel_primitive:NN \pdftexrevision \tex_pdftexrevision:D
688 \__kernel_primitive:NN \pdftexversion \tex_pdftexversion:D

```

These ones appear in pdfTeX but don't have pdf in the name at all: no decisions to make.

```

689 \__kernel_primitive:NN \efcode \tex_efcode:D
690 \__kernel_primitive:NN \ifincsname \tex_ifincsname:D
691 \__kernel_primitive:NN \leftmarginkern \tex_leftmarginkern:D
692 \__kernel_primitive:NN \letterspacefont \tex_letterspacefont:D
693 \__kernel_primitive:NN \lpcode \tex_lpcode:D
694 \__kernel_primitive:NN \quitvmode \tex_quitvmode:D
695 \__kernel_primitive:NN \rightmarginkern \tex_rightmarginkern:D
696 \__kernel_primitive:NN \rprcode \tex_rprcode:D
697 \__kernel_primitive:NN \synctex \tex_synctex:D
698 \__kernel_primitive:NN \tagcode \tex_tagcode:D

```

Post pdfTeX primitive availability gets more complex. Both XeTeX and LuaTeX have varying names for some primitives from pdfTeX. Particularly for LuaTeX tracking all of that would be hard. Instead, we now check that we only save primitives if they actually exist.

```

699 </names | package>
700 <*package>
701 \tex_long:D \tex_def:D \use_ii:nn #1#2 {#2}
702 \tex_long:D \tex_def:D \use_none:n #1 { }
703 \tex_long:D \tex_def:D \__kernel_primitive:NN #1#2
704 {
705     \tex_ifdefined:D #1
706     \tex_expandafter:D \use_ii:nn

```

```

707     \tex_fi:D
708     \use_none:n { \tex_global:D \tex_let:D #2 #1 }
709   }
710 </package>
711 <*names | package>

```

X_YTeX-specific primitives. Note that X_YTeX’s `\strcmp` is handled earlier and is “rolled up” into `\pdfstrcmp`. A few cross-compatibility names which lack the `pdf` of the original are handled later.

```

712 \__kernel_primitive:NN \suppressfontnotfounderror
713 \tex_suppressfontnotfounderror:D
714 \__kernel_primitive:NN \XeTeXcharclass           \tex_XeTeXcharclass:D
715 \__kernel_primitive:NN \XeTeXcharglyph           \tex_XeTeXcharglyph:D
716 \__kernel_primitive:NN \XeTeXcountfeatures       \tex_XeTeXcountfeatures:D
717 \__kernel_primitive:NN \XeTeXcountglyphs         \tex_XeTeXcountglyphs:D
718 \__kernel_primitive:NN \XeTeXcountselectors      \tex_XeTeXcountselectors:D
719 \__kernel_primitive:NN \XeTeXcountvariations     \tex_XeTeXcountvariations:D
720 \__kernel_primitive:NN \XeTeXdefaultencoding     \tex_XeTeXdefaultencoding:D
721 \__kernel_primitive:NN \XeTeXdashbreakstate     \tex_XeTeXdashbreakstate:D
722 \__kernel_primitive:NN \XeTeXfeaturecode         \tex_XeTeXfeaturecode:D
723 \__kernel_primitive:NN \XeTeXfeaturename         \tex_XeTeXfeaturename:D
724 \__kernel_primitive:NN \XeTeXfindfeaturebyname   \tex_XeTeXfindfeaturebyname:D
725 \tex_XeTeXfindfeaturebyname:D
726 \__kernel_primitive:NN \XeTeXfindselectorbyname  \tex_XeTeXfindselectorbyname:D
727 \tex_XeTeXfindselectorbyname:D
728 \__kernel_primitive:NN \XeTeXfindvariationbyname \tex_XeTeXfindvariationbyname:D
729 \tex_XeTeXfindvariationbyname:D
730 \__kernel_primitive:NN \XeTeXfirstfontchar       \tex_XeTeXfirstfontchar:D
731 \__kernel_primitive:NN \XeTeXfonttype            \tex_XeTeXfonttype:D
732 \__kernel_primitive:NN \XeTeXgenerateactualtext  \tex_XeTeXgenerateactualtext:D
733 \tex_XeTeXgenerateactualtext:D
734 \__kernel_primitive:NN \XeTeXglyph               \tex_XeTeXglyph:D
735 \__kernel_primitive:NN \XeTeXglyphbounds         \tex_XeTeXglyphbounds:D
736 \__kernel_primitive:NN \XeTeXglyphindex          \tex_XeTeXglyphindex:D
737 \__kernel_primitive:NN \XeTeXglyphname           \tex_XeTeXglyphname:D
738 \__kernel_primitive:NN \XeTeXinputencoding       \tex_XeTeXinputencoding:D
739 \__kernel_primitive:NN \XeTeXinputnormalization  \tex_XeTeXinputnormalization:D
740 \tex_XeTeXinputnormalization:D
741 \__kernel_primitive:NN \XeTeXinterchartokenstate \tex_XeTeXinterchartokenstate:D
742 \tex_XeTeXinterchartokenstate:D
743 \__kernel_primitive:NN \XeTeXinterchartoks       \tex_XeTeXinterchartoks:D
744 \__kernel_primitive:NN \XeTeXisdefaultselector   \tex_XeTeXisdefaultselector:D
745 \tex_XeTeXisdefaultselector:D
746 \__kernel_primitive:NN \XeTeXisexclusivefeature  \tex_XeTeXisexclusivefeature:D
747 \tex_XeTeXisexclusivefeature:D
748 \__kernel_primitive:NN \XeTeXlastfontchar        \tex_XeTeXlastfontchar:D
749 \__kernel_primitive:NN \XeTeXlinebreakskip       \tex_XeTeXlinebreakskip:D
750 \__kernel_primitive:NN \XeTeXlinebreaklocale     \tex_XeTeXlinebreaklocale:D
751 \__kernel_primitive:NN \XeTeXlinebreakpenalty    \tex_XeTeXlinebreakpenalty:D
752 \__kernel_primitive:NN \XeTeXOTcountfeatures     \tex_XeTeXOTcountfeatures:D
753 \__kernel_primitive:NN \XeTeXOTcountlanguages    \tex_XeTeXOTcountlanguages:D
754 \__kernel_primitive:NN \XeTeXOTcountscripts     \tex_XeTeXOTcountscripts:D
755 \__kernel_primitive:NN \XeTeXOTfeaturetag        \tex_XeTeXOTfeaturetag:D
756 \__kernel_primitive:NN \XeTeXOTlanguagetag       \tex_XeTeXOTlanguagetag:D

```

757	<code>__kernel_primitive:NN \XeTeXOTscripttag</code>	<code>\tex_XeTeXOTscripttag:D</code>
758	<code>__kernel_primitive:NN \XeTeXpdffile</code>	<code>\tex_XeTeXpdffile:D</code>
759	<code>__kernel_primitive:NN \XeTeXpdfpagecount</code>	<code>\tex_XeTeXpdfpagecount:D</code>
760	<code>__kernel_primitive:NN \XeTeXpicfile</code>	<code>\tex_XeTeXpicfile:D</code>
761	<code>__kernel_primitive:NN \XeTeXrevision</code>	<code>\tex_XeTeXrevision:D</code>
762	<code>__kernel_primitive:NN \XeTeXselectorname</code>	<code>\tex_XeTeXselectorname:D</code>
763	<code>__kernel_primitive:NN \XeTeXtracingfonts</code>	<code>\tex_XeTeXtracingfonts:D</code>
764	<code>__kernel_primitive:NN \XeTeXupwardsmode</code>	<code>\tex_XeTeXupwardsmode:D</code>
765	<code>__kernel_primitive:NN \XeTeXuseglyphmetrics</code>	<code>\tex_XeTeXuseglyphmetrics:D</code>
766	<code>__kernel_primitive:NN \XeTeXvariation</code>	<code>\tex_XeTeXvariation:D</code>
767	<code>__kernel_primitive:NN \XeTeXvariationdefault</code>	<code>\tex_XeTeXvariationdefault:D</code>
768	<code>__kernel_primitive:NN \XeTeXvariationmax</code>	<code>\tex_XeTeXvariationmax:D</code>
769	<code>__kernel_primitive:NN \XeTeXvariationmin</code>	<code>\tex_XeTeXvariationmin:D</code>
770	<code>__kernel_primitive:NN \XeTeXvariationname</code>	<code>\tex_XeTeXvariationname:D</code>
771	<code>__kernel_primitive:NN \XeTeXversion</code>	<code>\tex_XeTeXversion:D</code>

Primitives from pdfTeX that XeTeX renames: also helps with LuaTeX.

772	<code>__kernel_primitive:NN \creationdate</code>	<code>\tex_creationdate:D</code>
773	<code>__kernel_primitive:NN \elapsedtime</code>	<code>\tex_elapsedtime:D</code>
774	<code>__kernel_primitive:NN \filedump</code>	<code>\tex_filedump:D</code>
775	<code>__kernel_primitive:NN \filemoddate</code>	<code>\tex_filemoddate:D</code>
776	<code>__kernel_primitive:NN \filesize</code>	<code>\tex_filesize:D</code>
777	<code>__kernel_primitive:NN \mdfivesum</code>	<code>\tex_mdfivesum:D</code>
778	<code>__kernel_primitive:NN \ifprimitive</code>	<code>\tex_ifprimitive:D</code>
779	<code>__kernel_primitive:NN \primitive</code>	<code>\tex_primitive:D</code>
780	<code>__kernel_primitive:NN \resettimer</code>	<code>\tex_resettimer:D</code>
781	<code>__kernel_primitive:NN \shellescape</code>	<code>\tex_shellescape:D</code>

Primitives from LuaTeX, some of which have been ported back to XeTeX.

782	<code>__kernel_primitive:NN \alignmark</code>	<code>\tex_alignmark:D</code>
783	<code>__kernel_primitive:NN \aligntab</code>	<code>\tex_aligntab:D</code>
784	<code>__kernel_primitive:NN \attribute</code>	<code>\tex_attribute:D</code>
785	<code>__kernel_primitive:NN \attributedef</code>	<code>\tex_attributedef:D</code>
786	<code>__kernel_primitive:NN \automaticdiscretionary</code>	
787	<code>\tex_automaticdiscretionary:D</code>	
788	<code>__kernel_primitive:NN \automatichyphenmode</code>	<code>\tex_automatichyphenmode:D</code>
789	<code>__kernel_primitive:NN \automatichyphenpenalty</code>	
790	<code>\tex_automatichyphenpenalty:D</code>	
791	<code>__kernel_primitive:NN \beginscname</code>	<code>\tex_beginscname:D</code>
792	<code>__kernel_primitive:NN \bodydir</code>	<code>\tex_bodydir:D</code>
793	<code>__kernel_primitive:NN \bodydirection</code>	<code>\tex_bodydirection:D</code>
794	<code>__kernel_primitive:NN \boxdir</code>	<code>\tex_boxdir:D</code>
795	<code>__kernel_primitive:NN \boxdirection</code>	<code>\tex_boxdirection:D</code>
796	<code>__kernel_primitive:NN \breakafterdirmode</code>	<code>\tex_breakafterdirmode:D</code>
797	<code>__kernel_primitive:NN \catcodetable</code>	<code>\tex_catcodetable:D</code>
798	<code>__kernel_primitive:NN \clearmarks</code>	<code>\tex_clearmarks:D</code>
799	<code>__kernel_primitive:NN \crampeddisplaystyle</code>	<code>\tex_crampeddisplaystyle:D</code>
800	<code>__kernel_primitive:NN \crampedscriptscriptstyle</code>	
801	<code>\tex_crampedscriptscriptstyle:D</code>	
802	<code>__kernel_primitive:NN \crampedscriptstyle</code>	<code>\tex_crampedscriptstyle:D</code>
803	<code>__kernel_primitive:NN \crampedtextstyle</code>	<code>\tex_crampedtextstyle:D</code>
804	<code>__kernel_primitive:NN \csstring</code>	<code>\tex_csstring:D</code>
805	<code>__kernel_primitive:NN \directlua</code>	<code>\tex_directlua:D</code>
806	<code>__kernel_primitive:NN \dviextension</code>	<code>\tex_dviextension:D</code>
807	<code>__kernel_primitive:NN \dvifedback</code>	<code>\tex_dvifedback:D</code>

808	_kernel_primitive:NN	\dvivariable	\tex_dvivariable:D
809	_kernel_primitive:NN	\eTeXglueshrinkorder	\tex_eTeXglueshrinkorder:D
810	_kernel_primitive:NN	\eTeXgluestretchorder	\tex_eTeXgluestretchorder:D
811	_kernel_primitive:NN	\etoksapp	\tex_etoksapp:D
812	_kernel_primitive:NN	\etokspre	\tex_etokspre:D
813	_kernel_primitive:NN	\exceptionpenalty	\tex_exceptionpenalty:D
814	_kernel_primitive:NN	\explicithyphenpenalty	\tex_explicithyphenpenalty:D
815	_kernel_primitive:NN	\expanded	\tex_expanded:D
816	_kernel_primitive:NN	\explicitdiscretionary	\tex_explicitdiscretionary:D
817	_kernel_primitive:NN	\firstvalidlanguage	\tex_firstvalidlanguage:D
818	_kernel_primitive:NN	\fontid	\tex_fontid:D
819	_kernel_primitive:NN	\formatname	\tex_formatname:D
820	_kernel_primitive:NN	\hjcode	\tex_hjcode:D
821	_kernel_primitive:NN	\hpack	\tex_hpack:D
822	_kernel_primitive:NN	\hyphenationbounds	\tex_hyphenationbounds:D
823	_kernel_primitive:NN	\hyphenationmin	\tex_hyphenationmin:D
824	_kernel_primitive:NN	\hyphenpenaltymode	\tex_hyphenpenaltymode:D
825	_kernel_primitive:NN	\gleaders	\tex_gleaders:D
826	_kernel_primitive:NN	\ifcondition	\tex_ifcondition:D
827	_kernel_primitive:NN	\immediateassigned	\tex_immediateassigned:D
828	_kernel_primitive:NN	\immediateassignment	\tex_immediateassignment:D
829	_kernel_primitive:NN	\initcatcodetable	\tex_initcatcodetable:D
830	_kernel_primitive:NN	\lastnamedcs	\tex_lastnamedcs:D
831	_kernel_primitive:NN	\latalua	\tex_latalua:D
832	_kernel_primitive:NN	\lataluafunction	\tex_lataluafunction:D
833	_kernel_primitive:NN	\leftghost	\tex_leftghost:D
834	_kernel_primitive:NN	\lcharcode	\tex_lcharcode:D
835	_kernel_primitive:NN	\linedir	\tex_linedir:D
836	_kernel_primitive:NN	\linedirection	\tex_linedirection:D
837	_kernel_primitive:NN	\localbrokenpenalty	\tex_localbrokenpenalty:D
838	_kernel_primitive:NN	\localinterlinepenalty	\tex_localinterlinepenalty:D
839	_kernel_primitive:NN	\luabytecode	\tex_luabytecode:D
840	_kernel_primitive:NN	\luabytecodecall	\tex_luabytecodecall:D
841	_kernel_primitive:NN	\luacopyinputnodes	\tex_luacopyinputnodes:D
842	_kernel_primitive:NN	\luaodef	\tex_luaodef:D
843	_kernel_primitive:NN	\lcalleftbox	\tex_lcalleftbox:D
844	_kernel_primitive:NN	\lcalrightbox	\tex_lcalrightbox:D
845	_kernel_primitive:NN	\luaescapestring	\tex_luaescapestring:D
846	_kernel_primitive:NN	\luaofunction	\tex_luaofunction:D
847	_kernel_primitive:NN	\luaofunctioncall	\tex_luaofunctioncall:D
848	_kernel_primitive:NN	\luatexbanner	\tex_luatexbanner:D
849	_kernel_primitive:NN	\luatexrevision	\tex_luatexrevision:D
850	_kernel_primitive:NN	\luatexversion	\tex_luatexversion:D
851	_kernel_primitive:NN	\mathdelimitersmode	\tex_mathdelimitersmode:D
852	_kernel_primitive:NN	\mathdir	\tex_mathdir:D
853	_kernel_primitive:NN	\mathdirection	\tex_mathdirection:D
854	_kernel_primitive:NN	\mathdisplayskipmode	\tex_mathdisplayskipmode:D
855	_kernel_primitive:NN	\matheqnogapstep	\tex_matheqnogapstep:D
856	_kernel_primitive:NN	\mathnolimitsmode	\tex_mathnolimitsmode:D
857	_kernel_primitive:NN	\mathoption	\tex_mathoption:D
858	_kernel_primitive:NN	\mathpenaltiesmode	\tex_mathpenaltiesmode:D
859	_kernel_primitive:NN	\mathrulesfam	\tex_mathrulesfam:D
860	_kernel_primitive:NN	\mathscrtsmode	\tex_mathscrtsmode:D
861	_kernel_primitive:NN	\mathscriptboxmode	\tex_mathscriptboxmode:D

862	<code>__kernel_primitive:NN \mathscriptcharmode</code>	<code>\tex_mathscriptcharmode:D</code>
863	<code>__kernel_primitive:NN \mathstyle</code>	<code>\tex_mathstyle:D</code>
864	<code>__kernel_primitive:NN \mathsurroundmode</code>	<code>\tex_mathsurroundmode:D</code>
865	<code>__kernel_primitive:NN \mathsurroundskip</code>	<code>\tex_mathsurroundskip:D</code>
866	<code>__kernel_primitive:NN \nohrule</code>	<code>\tex_nohrule:D</code>
867	<code>__kernel_primitive:NN \nokerns</code>	<code>\tex_nokerns:D</code>
868	<code>__kernel_primitive:NN \noligs</code>	<code>\tex_noligs:D</code>
869	<code>__kernel_primitive:NN \nospaces</code>	<code>\tex_nospaces:D</code>
870	<code>__kernel_primitive:NN \novrule</code>	<code>\tex_novrule:D</code>
871	<code>__kernel_primitive:NN \outputbox</code>	<code>\tex_outputbox:D</code>
872	<code>__kernel_primitive:NN \pagebottomoffset</code>	<code>\tex_pagebottomoffset:D</code>
873	<code>__kernel_primitive:NN \pagedir</code>	<code>\tex_pagedir:D</code>
874	<code>__kernel_primitive:NN \pagedirection</code>	<code>\tex_pagedirection:D</code>
875	<code>__kernel_primitive:NN \pageleftoffset</code>	<code>\tex_pageleftoffset:D</code>
876	<code>__kernel_primitive:NN \pagerightoffset</code>	<code>\tex_pagerightoffset:D</code>
877	<code>__kernel_primitive:NN \pagetopoffset</code>	<code>\tex_pagetopoffset:D</code>
878	<code>__kernel_primitive:NN \pardir</code>	<code>\tex_pardir:D</code>
879	<code>__kernel_primitive:NN \pardirection</code>	<code>\tex_pardirection:D</code>
880	<code>__kernel_primitive:NN \pdfextension</code>	<code>\tex_pdfextension:D</code>
881	<code>__kernel_primitive:NN \pdffeedback</code>	<code>\tex_pdffeedback:D</code>
882	<code>__kernel_primitive:NN \pdfvariable</code>	<code>\tex_pdfvariable:D</code>
883	<code>__kernel_primitive:NN \postexhyphenchar</code>	<code>\tex_postexhyphenchar:D</code>
884	<code>__kernel_primitive:NN \posthyphenchar</code>	<code>\tex_posthyphenchar:D</code>
885	<code>__kernel_primitive:NN \prebinoppenalty</code>	<code>\tex_prebinoppenalty:D</code>
886	<code>__kernel_primitive:NN \predisplaygapfactor</code>	<code>\tex_predisplaygapfactor:D</code>
887	<code>__kernel_primitive:NN \preexhyphenchar</code>	<code>\tex_preexhyphenchar:D</code>
888	<code>__kernel_primitive:NN \prehyphenchar</code>	<code>\tex_prehyphenchar:D</code>
889	<code>__kernel_primitive:NN \prerelpenalty</code>	<code>\tex_prerelpenalty:D</code>
890	<code>__kernel_primitive:NN \rightghost</code>	<code>\tex_rightghost:D</code>
891	<code>__kernel_primitive:NN \savecatcodetable</code>	<code>\tex_savecatcodetable:D</code>
892	<code>__kernel_primitive:NN \scantextokens</code>	<code>\tex_scantextokens:D</code>
893	<code>__kernel_primitive:NN \setfontid</code>	<code>\tex_setfontid:D</code>
894	<code>__kernel_primitive:NN \shapemode</code>	<code>\tex_shapemode:D</code>
895	<code>__kernel_primitive:NN \suppressifcsnameerror</code>	<code>\tex_suppressifcsnameerror:D</code>
896	<code>__kernel_primitive:NN \suppresslongerror</code>	<code>\tex_suppresslongerror:D</code>
897	<code>__kernel_primitive:NN \suppressmathparerror</code>	<code>\tex_suppressmathparerror:D</code>
898	<code>__kernel_primitive:NN \suppressoutererror</code>	<code>\tex_suppressoutererror:D</code>
899	<code>__kernel_primitive:NN \suppressprimitiveerror</code>	
900	<code>\tex_suppressprimitiveerror:D</code>	
901	<code>__kernel_primitive:NN \textdir</code>	<code>\tex_textdir:D</code>
902	<code>__kernel_primitive:NN \textdirection</code>	<code>\tex_textdirection:D</code>
903	<code>__kernel_primitive:NN \toksapp</code>	<code>\tex_toksapp:D</code>
904	<code>__kernel_primitive:NN \tokspre</code>	<code>\tex_tokspre:D</code>
905	<code>__kernel_primitive:NN \tpack</code>	<code>\tex_tpack:D</code>
906	<code>__kernel_primitive:NN \vpack</code>	<code>\tex_vpack:D</code>

Primitives from pdfTeX that LuaTeX renames.

907	<code>__kernel_primitive:NN \adjustspacing</code>	<code>\tex_adjustspacing:D</code>
908	<code>__kernel_primitive:NN \copyfont</code>	<code>\tex_copyfont:D</code>
909	<code>__kernel_primitive:NN \draftmode</code>	<code>\tex_draftmode:D</code>
910	<code>__kernel_primitive:NN \expandglyphsinfont</code>	<code>\tex_fontexpand:D</code>
911	<code>__kernel_primitive:NN \ifabsdim</code>	<code>\tex_ifabsdim:D</code>
912	<code>__kernel_primitive:NN \ifabsnum</code>	<code>\tex_ifabsnum:D</code>
913	<code>__kernel_primitive:NN \ignoreligaturesinfont</code>	<code>\tex_ignoreligaturesinfont:D</code>
914	<code>__kernel_primitive:NN \insertht</code>	<code>\tex_insertht:D</code>

```

915 \__kernel_primitive:NN \lastsavedboxresourceindex
916 \tex_pdflastxform:D
917 \__kernel_primitive:NN \lastsavedimageresourceindex
918 \tex_pdflastximage:D
919 \__kernel_primitive:NN \lastsavedimageresourcepages
920 \tex_pdflastximagepages:D
921 \__kernel_primitive:NN \lastxpos \tex_lastxpos:D
922 \__kernel_primitive:NN \lastypos \tex_lastypos:D
923 \__kernel_primitive:NN \normaldeviate \tex_normaldeviate:D
924 \__kernel_primitive:NN \outputmode \tex_pdfoutput:D
925 \__kernel_primitive:NN \pageheight \tex_pageheight:D
926 \__kernel_primitive:NN \pagewidth \tex_pagewidth:D
927 \__kernel_primitive:NN \protrudechars \tex_protrudechars:D
928 \__kernel_primitive:NN \pxdimen \tex_pxdimen:D
929 \__kernel_primitive:NN \randomseed \tex_randomseed:D
930 \__kernel_primitive:NN \useboxresource \tex_pdfrefxform:D
931 \__kernel_primitive:NN \useimageresource \tex_pdfrefximage:D
932 \__kernel_primitive:NN \savepos \tex_savepos:D
933 \__kernel_primitive:NN \saveboxresource \tex_pdfxform:D
934 \__kernel_primitive:NN \saveimageresource \tex_pdfximage:D
935 \__kernel_primitive:NN \setrandomseed \tex_setrandomseed:D
936 \__kernel_primitive:NN \tracingfonts \tex_tracingfonts:D
937 \__kernel_primitive:NN \uniformdeviate \tex_uniformdeviate:D

```

The set of Unicode math primitives were introduced by X_YTeX and LuaTeX in a somewhat complex fashion: a few first as \XeTeX... which were then renamed with LuaTeX having a lot more. These names now all start \U... and mainly \Umath....

```

938 \__kernel_primitive:NN \Uchar \tex_Uchar:D
939 \__kernel_primitive:NN \Ucharcat \tex_Ucharcat:D
940 \__kernel_primitive:NN \Udelcode \tex_Udelcode:D
941 \__kernel_primitive:NN \Udelcodenum \tex_Udelcodenum:D
942 \__kernel_primitive:NN \Udelimiter \tex_Udelimiter:D
943 \__kernel_primitive:NN \Udelimiterover \tex_Udelimiterover:D
944 \__kernel_primitive:NN \Udelimiterunder \tex_Udelimiterunder:D
945 \__kernel_primitive:NN \Uhextensible \tex_Uhextensible:D
946 \__kernel_primitive:NN \Umathaccent \tex_Umathaccent:D
947 \__kernel_primitive:NN \Umathaxis \tex_Umathaxis:D
948 \__kernel_primitive:NN \Umathbinbinspacing \tex_Umathbinbinspacing:D
949 \__kernel_primitive:NN \Umathbinclonespacing \tex_Umathbinclonespacing:D
950 \__kernel_primitive:NN \Umathbininnerspacing \tex_Umathbininnerspacing:D
951 \__kernel_primitive:NN \Umathbinopenspacing \tex_Umathbinopenspacing:D
952 \__kernel_primitive:NN \Umathbinopspacing \tex_Umathbinopspacing:D
953 \__kernel_primitive:NN \Umathbinordspacing \tex_Umathbinordspacing:D
954 \__kernel_primitive:NN \Umathbinpunctspacing \tex_Umathbinpunctspacing:D
955 \__kernel_primitive:NN \Umathbinrelspacing \tex_Umathbinrelspacing:D
956 \__kernel_primitive:NN \Umathchar \tex_Umathchar:D
957 \__kernel_primitive:NN \Umathcharclass \tex_Umathcharclass:D
958 \__kernel_primitive:NN \Umathchardef \tex_Umathchardef:D
959 \__kernel_primitive:NN \Umathcharfam \tex_Umathcharfam:D
960 \__kernel_primitive:NN \Umathcharnum \tex_Umathcharnum:D
961 \__kernel_primitive:NN \Umathcharnumdef \tex_Umathcharnumdef:D
962 \__kernel_primitive:NN \Umathcharslot \tex_Umathcharslot:D
963 \__kernel_primitive:NN \Umathclosebinspacing \tex_Umathclosebinspacing:D
964 \__kernel_primitive:NN \Umathcloseclonespacing

```

```

965 \tex_Umathcloseclosespacing:D
966 \__kernel_primitive:NN \Umathcloseinnerspacing
967 \tex_Umathcloseinnerspacing:D
968 \__kernel_primitive:NN \Umathcloseopenspacing \tex_Umathcloseopenspacing:D
969 \__kernel_primitive:NN \Umathcloseopspacing \tex_Umathcloseopspacing:D
970 \__kernel_primitive:NN \Umathcloseordspacing \tex_Umathcloseordspacing:D
971 \__kernel_primitive:NN \Umathclosepunctspacing
972 \tex_Umathclosepunctspacing:D
973 \__kernel_primitive:NN \Umathcloserelspacing \tex_Umathcloserelspacing:D
974 \__kernel_primitive:NN \Umathcode \tex_Umathcode:D
975 \__kernel_primitive:NN \Umathcodenum \tex_Umathcodenum:D
976 \__kernel_primitive:NN \Umathconnectoroverlapmin
977 \tex_Umathconnectoroverlapmin:D
978 \__kernel_primitive:NN \Umathfractiondelsize \tex_Umathfractiondelsize:D
979 \__kernel_primitive:NN \Umathfractiondenomdown
980 \tex_Umathfractiondenomdown:D
981 \__kernel_primitive:NN \Umathfractiondenomvgap
982 \tex_Umathfractiondenomvgap:D
983 \__kernel_primitive:NN \Umathfractionnumup \tex_Umathfractionnumup:D
984 \__kernel_primitive:NN \Umathfractionnumvgap \tex_Umathfractionnumvgap:D
985 \__kernel_primitive:NN \Umathfractionrule \tex_Umathfractionrule:D
986 \__kernel_primitive:NN \Umathinnerbinspacing \tex_Umathinnerbinspacing:D
987 \__kernel_primitive:NN \Umathinnerclosespacing
988 \tex_Umathinnerclosespacing:D
989 \__kernel_primitive:NN \Umathinnerinnerspacing
990 \tex_Umathinnerinnerspacing:D
991 \__kernel_primitive:NN \Umathinneropenspacing \tex_Umathinneropenspacing:D
992 \__kernel_primitive:NN \Umathinneropspacing \tex_Umathinneropspacing:D
993 \__kernel_primitive:NN \Umathinnerordspacing \tex_Umathinnerordspacing:D
994 \__kernel_primitive:NN \Umathinnerpunctspacing
995 \tex_Umathinnerpunctspacing:D
996 \__kernel_primitive:NN \Umathinnerrelspacing \tex_Umathinnerrelspacing:D
997 \__kernel_primitive:NN \Umathlimitabovebgap \tex_Umathlimitabovebgap:D
998 \__kernel_primitive:NN \Umathlimitabovekern \tex_Umathlimitabovekern:D
999 \__kernel_primitive:NN \Umathlimitabovevgap \tex_Umathlimitabovevgap:D
1000 \__kernel_primitive:NN \Umathlimitbelowbgap \tex_Umathlimitbelowbgap:D
1001 \__kernel_primitive:NN \Umathlimitbelowkern \tex_Umathlimitbelowkern:D
1002 \__kernel_primitive:NN \Umathlimitbelowvgap \tex_Umathlimitbelowvgap:D
1003 \__kernel_primitive:NN \Umathnolimitsubfactor \tex_Umathnolimitsubfactor:D
1004 \__kernel_primitive:NN \Umathnolimitsupfactor \tex_Umathnolimitsupfactor:D
1005 \__kernel_primitive:NN \Umathopbinspacing \tex_Umathopbinspacing:D
1006 \__kernel_primitive:NN \Umathopclosespacing \tex_Umathopclosespacing:D
1007 \__kernel_primitive:NN \Umathopenbinspacing \tex_Umathopenbinspacing:D
1008 \__kernel_primitive:NN \Umathopenclosespacing \tex_Umathopenclosespacing:D
1009 \__kernel_primitive:NN \Umathopeninnerspacing \tex_Umathopeninnerspacing:D
1010 \__kernel_primitive:NN \Umathopenopenspacing \tex_Umathopenopenspacing:D
1011 \__kernel_primitive:NN \Umathopenopspacing \tex_Umathopenopspacing:D
1012 \__kernel_primitive:NN \Umathopenordspacing \tex_Umathopenordspacing:D
1013 \__kernel_primitive:NN \Umathopenpunctspacing \tex_Umathopenpunctspacing:D
1014 \__kernel_primitive:NN \Umathopenrelspacing \tex_Umathopenrelspacing:D
1015 \__kernel_primitive:NN \Umathoperatorsize \tex_Umathoperatorsize:D
1016 \__kernel_primitive:NN \Umathopinnerspacing \tex_Umathopinnerspacing:D
1017 \__kernel_primitive:NN \Umathopopenspacing \tex_Umathopopenspacing:D
1018 \__kernel_primitive:NN \Umathopopspacing \tex_Umathopopspacing:D

```


1019 __kernel_primitive:NN \Umathopordspacing \tex_Umathopordspacing:D
1020 __kernel_primitive:NN \Umathoppunctspacing \tex_Umathoppunctspacing:D
1021 __kernel_primitive:NN \Umathoprelspacing \tex_Umathoprelspacing:D
1022 __kernel_primitive:NN \Umathordbinspacing \tex_Umathordbinspacing:D
1023 __kernel_primitive:NN \Umathordclosespacing \tex_Umathordclosespacing:D
1024 __kernel_primitive:NN \Umathordinnerspacing \tex_Umathordinnerspacing:D
1025 __kernel_primitive:NN \Umathordopenspacing \tex_Umathordopenspacing:D
1026 __kernel_primitive:NN \Umathordopspacing \tex_Umathordopspacing:D
1027 __kernel_primitive:NN \Umathordordspacing \tex_Umathordordspacing:D
1028 __kernel_primitive:NN \Umathordpunctspacing \tex_Umathordpunctspacing:D
1029 __kernel_primitive:NN \Umathordrelspacing \tex_Umathordrelspacing:D
1030 __kernel_primitive:NN \Umathoverbarkern \tex_Umathoverbarkern:D
1031 __kernel_primitive:NN \Umathoverbarrule \tex_Umathoverbarrule:D
1032 __kernel_primitive:NN \Umathoverbarvgap \tex_Umathoverbarvgap:D
1033 __kernel_primitive:NN \Umathoverdelimterbgap
1034 \tex_Umathoverdelimterbgap:D
1035 __kernel_primitive:NN \Umathoverdelimtervgap
1036 \tex_Umathoverdelimtervgap:D
1037 __kernel_primitive:NN \Umathpunctbinspacing \tex_Umathpunctbinspacing:D
1038 __kernel_primitive:NN \Umathpunctclosespacing
1039 \tex_Umathpunctclosespacing:D
1040 __kernel_primitive:NN \Umathpunctinnerspacing
1041 \tex_Umathpunctinnerspacing:D
1042 __kernel_primitive:NN \Umathpunctopenspacing \tex_Umathpunctopenspacing:D
1043 __kernel_primitive:NN \Umathpunctopspacing \tex_Umathpunctopspacing:D
1044 __kernel_primitive:NN \Umathpunctordspacing \tex_Umathpunctordspacing:D
1045 __kernel_primitive:NN \Umathpunctpunctspacing
1046 \tex_Umathpunctpunctspacing:D
1047 __kernel_primitive:NN \Umathpunctrelspacing \tex_Umathpunctrelspacing:D
1048 __kernel_primitive:NN \Umathquad \tex_Umathquad:D
1049 __kernel_primitive:NN \Umathradicaldegreeafter
1050 \tex_Umathradicaldegreeafter:D
1051 __kernel_primitive:NN \Umathradicaldegreebefore
1052 \tex_Umathradicaldegreebefore:D
1053 __kernel_primitive:NN \Umathradicaldegreeraise
1054 \tex_Umathradicaldegreeraise:D
1055 __kernel_primitive:NN \Umathradicalkern \tex_Umathradicalkern:D
1056 __kernel_primitive:NN \Umathradicalrule \tex_Umathradicalrule:D
1057 __kernel_primitive:NN \Umathradicalvgap \tex_Umathradicalvgap:D
1058 __kernel_primitive:NN \Umathrelbinspacing \tex_Umathrelbinspacing:D
1059 __kernel_primitive:NN \Umathrelclosespacing \tex_Umathrelclosespacing:D
1060 __kernel_primitive:NN \Umathrelinnerspacing \tex_Umathrelinnerspacing:D
1061 __kernel_primitive:NN \Umathrelopenspacing \tex_Umathrelopenspacing:D
1062 __kernel_primitive:NN \Umathrelopspacing \tex_Umathrelopspacing:D
1063 __kernel_primitive:NN \Umathrelordspacing \tex_Umathrelordspacing:D
1064 __kernel_primitive:NN \Umathrelpunctspacing \tex_Umathrelpunctspacing:D
1065 __kernel_primitive:NN \Umathrelrelspacing \tex_Umathrelrelspacing:D
1066 __kernel_primitive:NN \Umathskewedfractionhgap
1067 \tex_Umathskewedfractionhgap:D
1068 __kernel_primitive:NN \Umathskewedfractionvgap
1069 \tex_Umathskewedfractionvgap:D
1070 __kernel_primitive:NN \Umathspaceafterscript \tex_Umathspaceafterscript:D
1071 __kernel_primitive:NN \Umathstackdenomdown \tex_Umathstackdenomdown:D
1072 __kernel_primitive:NN \Umathstacknumup \tex_Umathstacknumup:D

1073	<code>__kernel_primitive:NN</code>	<code>\Umathstackvgap</code>	<code>\tex_Umathstackvgap:D</code>
1074	<code>__kernel_primitive:NN</code>	<code>\Umathsubshiftdown</code>	<code>\tex_Umathsubshiftdown:D</code>
1075	<code>__kernel_primitive:NN</code>	<code>\Umathsubshiftdrop</code>	<code>\tex_Umathsubshiftdrop:D</code>
1076	<code>__kernel_primitive:NN</code>	<code>\Umathsubsupshiftdown</code>	<code>\tex_Umathsubsupshiftdown:D</code>
1077	<code>__kernel_primitive:NN</code>	<code>\Umathsubsupvgap</code>	<code>\tex_Umathsubsupvgap:D</code>
1078	<code>__kernel_primitive:NN</code>	<code>\Umathsubtopmax</code>	<code>\tex_Umathsubtopmax:D</code>
1079	<code>__kernel_primitive:NN</code>	<code>\Umathsupbottommin</code>	<code>\tex_Umathsupbottommin:D</code>
1080	<code>__kernel_primitive:NN</code>	<code>\Umathsupshiftdrop</code>	<code>\tex_Umathsupshiftdrop:D</code>
1081	<code>__kernel_primitive:NN</code>	<code>\Umathsupshiftup</code>	<code>\tex_Umathsupshiftup:D</code>
1082	<code>__kernel_primitive:NN</code>	<code>\Umathsupsubbottommax</code>	<code>\tex_Umathsupsubbottommax:D</code>
1083	<code>__kernel_primitive:NN</code>	<code>\Umathunderbarkern</code>	<code>\tex_Umathunderbarkern:D</code>
1084	<code>__kernel_primitive:NN</code>	<code>\Umathunderbarrule</code>	<code>\tex_Umathunderbarrule:D</code>
1085	<code>__kernel_primitive:NN</code>	<code>\Umathunderbarvgap</code>	<code>\tex_Umathunderbarvgap:D</code>
1086	<code>__kernel_primitive:NN</code>	<code>\Umathunderdelimitervgap</code>	
1087		<code>\tex_Umathunderdelimitervgap:D</code>	
1088	<code>__kernel_primitive:NN</code>	<code>\Umathunderdelimitervgap</code>	
1089		<code>\tex_Umathunderdelimitervgap:D</code>	
1090	<code>__kernel_primitive:NN</code>	<code>\Unosubscript</code>	<code>\tex_Unosubscript:D</code>
1091	<code>__kernel_primitive:NN</code>	<code>\Unosuperscript</code>	<code>\tex_Unosuperscript:D</code>
1092	<code>__kernel_primitive:NN</code>	<code>\Uoverdelimiter</code>	<code>\tex_Uoverdelimiter:D</code>
1093	<code>__kernel_primitive:NN</code>	<code>\Uradical</code>	<code>\tex_Uradical:D</code>
1094	<code>__kernel_primitive:NN</code>	<code>\Uroot</code>	<code>\tex_Uroot:D</code>
1095	<code>__kernel_primitive:NN</code>	<code>\Uskewed</code>	<code>\tex_Uskewed:D</code>
1096	<code>__kernel_primitive:NN</code>	<code>\Uskewedwithdelims</code>	<code>\tex_Uskewedwithdelims:D</code>
1097	<code>__kernel_primitive:NN</code>	<code>\Ustack</code>	<code>\tex_Ustack:D</code>
1098	<code>__kernel_primitive:NN</code>	<code>\Ustartdisplaymath</code>	<code>\tex_Ustartdisplaymath:D</code>
1099	<code>__kernel_primitive:NN</code>	<code>\Ustartmath</code>	<code>\tex_Ustartmath:D</code>
1100	<code>__kernel_primitive:NN</code>	<code>\Ustopdisplaymath</code>	<code>\tex_Ustopdisplaymath:D</code>
1101	<code>__kernel_primitive:NN</code>	<code>\Ustopmath</code>	<code>\tex_Ustopmath:D</code>
1102	<code>__kernel_primitive:NN</code>	<code>\Usubscript</code>	<code>\tex_Usubscript:D</code>
1103	<code>__kernel_primitive:NN</code>	<code>\Usuperscript</code>	<code>\tex_Usuperscript:D</code>
1104	<code>__kernel_primitive:NN</code>	<code>\Uunderdelimiter</code>	<code>\tex_Uunderdelimiter:D</code>
1105	<code>__kernel_primitive:NN</code>	<code>\Uvextensible</code>	<code>\tex_Uvextensible:D</code>

Primitives from pTeX.

1106	<code>__kernel_primitive:NN</code>	<code>\autospacing</code>	<code>\tex_autospacing:D</code>
1107	<code>__kernel_primitive:NN</code>	<code>\autoxspacing</code>	<code>\tex_autoxspacing:D</code>
1108	<code>__kernel_primitive:NN</code>	<code>\currentcjktoken</code>	<code>\tex_currentcjktoken:D</code>
1109	<code>__kernel_primitive:NN</code>	<code>\currentspacingmode</code>	<code>\tex_currentspacingmode:D</code>
1110	<code>__kernel_primitive:NN</code>	<code>\currentxspacingmode</code>	<code>\tex_currentxspacingmode:D</code>
1111	<code>__kernel_primitive:NN</code>	<code>\disinhibitglue</code>	<code>\tex_disinhibitglue:D</code>
1112	<code>__kernel_primitive:NN</code>	<code>\dtou</code>	<code>\tex_dtou:D</code>
1113	<code>__kernel_primitive:NN</code>	<code>\epTeXinputencoding</code>	<code>\tex_epTeXinputencoding:D</code>
1114	<code>__kernel_primitive:NN</code>	<code>\epTeXversion</code>	<code>\tex_epTeXversion:D</code>
1115	<code>__kernel_primitive:NN</code>	<code>\euc</code>	<code>\tex_euc:D</code>
1116	<code>__kernel_primitive:NN</code>	<code>\hfi</code>	<code>\tex_hfi:D</code>
1117	<code>__kernel_primitive:NN</code>	<code>\ifdbbox</code>	<code>\tex_ifdbbox:D</code>
1118	<code>__kernel_primitive:NN</code>	<code>\ifddir</code>	<code>\tex_ifddir:D</code>
1119	<code>__kernel_primitive:NN</code>	<code>\ifjfont</code>	<code>\tex_ifjfont:D</code>
1120	<code>__kernel_primitive:NN</code>	<code>\ifmbox</code>	<code>\tex_ifmbox:D</code>
1121	<code>__kernel_primitive:NN</code>	<code>\ifmdir</code>	<code>\tex_ifmdir:D</code>
1122	<code>__kernel_primitive:NN</code>	<code>\iftbox</code>	<code>\tex_iftbox:D</code>
1123	<code>__kernel_primitive:NN</code>	<code>\iftfont</code>	<code>\tex_iftfont:D</code>
1124	<code>__kernel_primitive:NN</code>	<code>\iftdir</code>	<code>\tex_iftdir:D</code>
1125	<code>__kernel_primitive:NN</code>	<code>\ifybox</code>	<code>\tex_ifybox:D</code>

1126	<code>_kernel_primitive:NN \ifypdir</code>	<code>\tex_ifypdir:D</code>
1127	<code>_kernel_primitive:NN \inhibitglue</code>	<code>\tex_inhibitglue:D</code>
1128	<code>_kernel_primitive:NN \inhibitxspcode</code>	<code>\tex_inhibitxspcode:D</code>
1129	<code>_kernel_primitive:NN \jcharwidowpenalty</code>	<code>\tex_jcharwidowpenalty:D</code>
1130	<code>_kernel_primitive:NN \jfam</code>	<code>\tex_jfam:D</code>
1131	<code>_kernel_primitive:NN \jfont</code>	<code>\tex_jfont:D</code>
1132	<code>_kernel_primitive:NN \jis</code>	<code>\tex_jis:D</code>
1133	<code>_kernel_primitive:NN \kanjiskip</code>	<code>\tex_kanjiskip:D</code>
1134	<code>_kernel_primitive:NN \kansuji</code>	<code>\tex_kansuji:D</code>
1135	<code>_kernel_primitive:NN \kansujichar</code>	<code>\tex_kansujichar:D</code>
1136	<code>_kernel_primitive:NN \kcatcode</code>	<code>\tex_kcatcode:D</code>
1137	<code>_kernel_primitive:NN \kuten</code>	<code>\tex_kuten:D</code>
1138	<code>_kernel_primitive:NN \lastnodechar</code>	<code>\tex_lastnodechar:D</code>
1139	<code>_kernel_primitive:NN \lastnodesubtype</code>	<code>\tex_lastnodesubtype:D</code>
1140	<code>_kernel_primitive:NN \noautospadding</code>	<code>\tex_noautospadding:D</code>
1141	<code>_kernel_primitive:NN \noautoxspacing</code>	<code>\tex_noautoxspacing:D</code>
1142	<code>_kernel_primitive:NN \pagefistretch</code>	<code>\tex_pagefistretch:D</code>
1143	<code>_kernel_primitive:NN \postbreakpenalty</code>	<code>\tex_postbreakpenalty:D</code>
1144	<code>_kernel_primitive:NN \prebreakpenalty</code>	<code>\tex_prebreakpenalty:D</code>
1145	<code>_kernel_primitive:NN \ptexminorversion</code>	<code>\tex_ptexminorversion:D</code>
1146	<code>_kernel_primitive:NN \ptexrevision</code>	<code>\tex_ptexrevision:D</code>
1147	<code>_kernel_primitive:NN \ptexversion</code>	<code>\tex_ptexversion:D</code>
1148	<code>_kernel_primitive:NN \readpapersizespecial</code>	<code>\tex_readpapersizespecial:D</code>
1149	<code>_kernel_primitive:NN \scriptbaselineshiftfactor</code>	
1150	<code>\tex_scriptbaselineshiftfactor:D</code>	
1151	<code>_kernel_primitive:NN \scriptscriptbaselineshiftfactor</code>	
1152	<code>\tex_scriptscriptbaselineshiftfactor:D</code>	
1153	<code>_kernel_primitive:NN \showmode</code>	<code>\tex_showmode:D</code>
1154	<code>_kernel_primitive:NN \sjis</code>	<code>\tex_sjis:D</code>
1155	<code>_kernel_primitive:NN \tate</code>	<code>\tex_tate:D</code>
1156	<code>_kernel_primitive:NN \tbaselineshift</code>	<code>\tex_tbaselineshift:D</code>
1157	<code>_kernel_primitive:NN \textbaselineshiftfactor</code>	
1158	<code>\tex_textbaselineshiftfactor:D</code>	
1159	<code>_kernel_primitive:NN \tfont</code>	<code>\tex_tfont:D</code>
1160	<code>_kernel_primitive:NN \xkanjiskip</code>	<code>\tex_xkanjiskip:D</code>
1161	<code>_kernel_primitive:NN \xspcode</code>	<code>\tex_xspcode:D</code>
1162	<code>_kernel_primitive:NN \ybaselineshift</code>	<code>\tex_ybaselineshift:D</code>
1163	<code>_kernel_primitive:NN \yoko</code>	<code>\tex_yoko:D</code>
1164	<code>_kernel_primitive:NN \vfi</code>	<code>\tex_vfi:D</code>

Primitives from upTeX.

1165	<code>_kernel_primitive:NN \currentcjktoken</code>	<code>\tex_currentcjktoken:D</code>
1166	<code>_kernel_primitive:NN \disablecjktoken</code>	<code>\tex_disablecjktoken:D</code>
1167	<code>_kernel_primitive:NN \enablecjktoken</code>	<code>\tex_enablecjktoken:D</code>
1168	<code>_kernel_primitive:NN \forcecjktoken</code>	<code>\tex_forcecjktoken:D</code>
1169	<code>_kernel_primitive:NN \kchar</code>	<code>\tex_kchar:D</code>
1170	<code>_kernel_primitive:NN \kchardef</code>	<code>\tex_kchardef:D</code>
1171	<code>_kernel_primitive:NN \kuten</code>	<code>\tex_kuten:D</code>
1172	<code>_kernel_primitive:NN \ucs</code>	<code>\tex_ucs:D</code>
1173	<code>_kernel_primitive:NN \uptexrevision</code>	<code>\tex_uptexrevision:D</code>
1174	<code>_kernel_primitive:NN \uptexversion</code>	<code>\tex_uptexversion:D</code>

Omega primitives provided by pTeX (listed separately mainly to allow understanding of their source).

1175	<code>_kernel_primitive:NN \odelcode</code>	<code>\tex_odelcode:D</code>
------	---	-------------------------------

```

1176 \__kernel_primitive:NN \odelimiter \tex_odelimiter:D
1177 \__kernel_primitive:NN \omathaccent \tex_omathaccent:D
1178 \__kernel_primitive:NN \omathchar \tex_omathchar:D
1179 \__kernel_primitive:NN \omathchardef \tex_omathchardef:D
1180 \__kernel_primitive:NN \omathcode \tex_omathcode:D
1181 \__kernel_primitive:NN \oradical \tex_oradical:D

```

End of the “just the names” part of the source.

```

1182 </names | package>
1183 <*package>

```

The job is done: close the group (using the primitive renamed!).

```

1184 \tex_endgroup:D

```

L^AT_EX 2_ε moves a few primitives, so these are sorted out. In newer versions of L^AT_EX 2_ε, expl3 is loaded rather early, so only some primitives are already renamed, so we need two tests here. At the beginning of the L^AT_EX 2_ε format, the primitives `\end` and `\input` are renamed, and only later on the other ones.

```

1185 \tex_ifdefined:D \@@end
1186 \tex_let:D \tex_end:D \@@end
1187 \tex_let:D \tex_input:D \@@input
1188 \tex_fi:D

```

If `\@@@hyph` is defined, we are loading expl3 in a pre-2020/10/01 release of L^AT_EX 2_ε, so a few other primitives have to be tested as well.

```

1189 \tex_ifdefined:D \@@hyph
1190 \tex_let:D \tex_everydisplay:D \frozen@everydisplay
1191 \tex_let:D \tex_everymath:D \frozen@everymath
1192 \tex_let:D \tex_hyphen:D \@@hyph
1193 \tex_let:D \tex_italiccorrection:D \@@italiccorr
1194 \tex_let:D \tex_underline:D \@@underline

```

The `\shipout` primitive is particularly tricky as a number of packages want to hook in here. First, we see if a sufficiently-new kernel has saved a copy: if it has, just use that. Otherwise, we need to check each of the possible packages/classes that might move it: here, we are looking for those which do *not* delay action to the `\AtBeginDocument` hook. (We cannot use `\primitive` as that doesn’t allow us to make a direct copy of the primitive *itself*.) As we know that L^AT_EX 2_ε is in use, we use its `\@tfor` loop here.

```

1195 \tex_ifdefined:D \@@shipout
1196 \tex_let:D \tex_shipout:D \@@shipout
1197 \tex_fi:D
1198 \tex_begingroup:D
1199 \tex_edef:D \l_tmpa_tl { \tex_string:D \shipout }
1200 \tex_edef:D \l_tmpb_tl { \tex_meaning:D \shipout }
1201 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1202 \tex_else:D
1203 \tex_expandafter:D \@tfor \tex_expandafter:D \@tempa \tex_string:D :=
1204 \CROP@shipout
1205 \dup@shipout
1206 \GPTorg@shipout
1207 \LL@shipout
1208 \mem@oldshipout
1209 \opem@shipout
1210 \pgfpages@originalshipout
1211 \pr@shipout

```

```

1212     \Shipout
1213     \verso@orig@shipout
1214     \do
1215     {
1216         \tex_edef:D \l_tmpb_tl
1217         { \tex_expandafter:D \tex_meaning:D \@tempa }
1218         \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1219         \tex_global:D \tex_expandafter:D \tex_let:D
1220         \tex_expandafter:D \tex_shipout:D \@tempa
1221         \tex_fi:D
1222     }
1223     \tex_fi:D
1224     \tex_endgroup:D

```

Some tidying up is needed for `\(pdf)tracingfonts`. Newer LuaTeX has this simply as `\tracingfonts`, but that is overwritten by the L^AT_EX 2_ε kernel. So any spurious definition has to be removed, then the real version saved either from the pdfTeX name or from LuaTeX. In the latter case, we leave `\@@tracingfonts` available: this might be useful and almost all L^AT_EX 2_ε users will have `expl3` loaded by `fontspec`. (We follow the usual kernel convention that `@@` is used for saved primitives.)

```

1225     \tex_let:D \tex_tracingfonts:D \tex_undefined:D
1226     \tex_ifdefined:D \pdftracingfonts
1227     \tex_let:D \tex_tracingfonts:D \pdftracingfonts
1228     \tex_else:D
1229     \tex_ifdefined:D \tex_directlua:D
1230     \tex_directlua:D { tex.enableprimitives("@@", {"tracingfonts"}) }
1231     \tex_let:D \tex_tracingfonts:D \@@tracingfonts
1232     \tex_fi:D
1233     \tex_fi:D
1234     \tex_fi:D

```

That is also true for the LuaTeX primitives under L^AT_EX 2_ε (depending on the format-building date). There are a few primitives that get the right names anyway so are missing here!

```

1235     \tex_ifdefined:D \luatexsuppressfontnotfounderror
1236     \tex_let:D \tex_alignmark:D          \luatexalignmark
1237     \tex_let:D \tex_aligntab:D           \luatexaligntab
1238     \tex_let:D \tex_attribute:D          \luatexattribute
1239     \tex_let:D \tex_attributedef:D       \luatexattributedef
1240     \tex_let:D \tex_catcodetable:D       \luatexcatcodetable
1241     \tex_let:D \tex_clearmarks:D         \luatexclearmarks
1242     \tex_let:D \tex_crampeddisplaystyle:D \luatexcrampeddisplaystyle
1243     \tex_let:D \tex_crampedscriptscriptstyle:D
1244     \luatexcrampedscriptscriptstyle
1245     \tex_let:D \tex_crampedscriptstyle:D \luatexcrampedscriptstyle
1246     \tex_let:D \tex_crampedtextstyle:D  \luatexcrampedtextstyle
1247     \tex_let:D \tex_fontid:D             \luatexfontid
1248     \tex_let:D \tex_formatname:D         \luatexformatname
1249     \tex_let:D \tex_gleaders:D           \luatexgleaders
1250     \tex_let:D \tex_initcatcodetable:D   \luatexinitcatcodetable
1251     \tex_let:D \tex_latelua:D            \luatexlatelua
1252     \tex_let:D \tex_luaescapestring:D    \luatexluaescapestring
1253     \tex_let:D \tex_luafunction:D        \luatexluafunction
1254     \tex_let:D \tex_mathstyle:D          \luatexmathstyle

```

```

1255 \tex_let:D \tex_nokerns:D          \luatexnokerns
1256 \tex_let:D \tex_noligs:D          \luatexnoligs
1257 \tex_let:D \tex_outputbox:D        \luatexoutputbox
1258 \tex_let:D \tex_pageleftoffset:D   \luatexpageleftoffset
1259 \tex_let:D \tex_pagetopoffset:D     \luatexpagetopoffset
1260 \tex_let:D \tex_postexhyphenchar:D \luatexpostexhyphenchar
1261 \tex_let:D \tex_posthyphenchar:D    \luatexposthyphenchar
1262 \tex_let:D \tex_preexhyphenchar:D   \luatexpreehyphenchar
1263 \tex_let:D \tex_prehyphenchar:D     \luatexprehyphenchar
1264 \tex_let:D \tex_savecatcodetable:D  \luatexsavecatcodetable
1265 \tex_let:D \tex_scantextokens:D     \luatexscantextokens
1266 \tex_let:D \tex_suppressifcsnameerror:D
1267 \luatexsuppressifcsnameerror
1268 \tex_let:D \tex_suppresslongerror:D \luatexsuppresslongerror
1269 \tex_let:D \tex_suppressmathparerror:D
1270 \luatexsuppressmathparerror
1271 \tex_let:D \tex_suppressoutererror:D \luatexsuppressoutererror
1272 \tex_let:D \tex_Uchar:D             \luatexUchar
1273 \tex_let:D \tex_suppressfontnotfounderror:D
1274 \luatexsuppressfontnotfounderror

```

Which also covers those slightly odd ones.

```

1275 \tex_let:D \tex_bodydir:D          \luatexbodydir
1276 \tex_let:D \tex_boxdir:D           \luatexboxdir
1277 \tex_let:D \tex_leftghost:D         \luatexleftghost
1278 \tex_let:D \tex_localbrokenpenalty:D \luatexlocalbrokenpenalty
1279 \tex_let:D \tex_localinterlinepenalty:D
1280 \luatexlocalinterlinepenalty
1281 \tex_let:D \tex_localleftbox:D       \luatexlocalleftbox
1282 \tex_let:D \tex_localrightbox:D      \luatexlocalrightbox
1283 \tex_let:D \tex_mathdir:D            \luatexmathdir
1284 \tex_let:D \tex_pagebottomoffset:D   \luatexpagebottomoffset
1285 \tex_let:D \tex_pagedir:D            \luatexpagedir
1286 \tex_let:D \tex_pageheight:D         \luatexpageheight
1287 \tex_let:D \tex_pagerightoffset:D    \luatexpagerightoffset
1288 \tex_let:D \tex_pagewidth:D          \luatexpagewidth
1289 \tex_let:D \tex_pardir:D             \luatexpardir
1290 \tex_let:D \tex_rightghost:D         \luatexrightghost
1291 \tex_let:D \tex_textdir:D            \luatextextdir
1292 \tex_fi:D

```

Only pdfTeX and LuaTeX define \pdfmapfile and \pdfmapline: Tidy up the fact that some format-building processes leave a couple of questionable decisions about that!

```

1293 \tex_ifnum:D 0
1294 \tex_ifdefined:D \tex_pdftexversion:D 1 \tex_fi:D
1295 \tex_ifdefined:D \tex_luatexversion:D 1 \tex_fi:D
1296 = 0 %
1297 \tex_let:D \tex_mapfile:D \tex_undefined:D
1298 \tex_let:D \tex_mapline:D \tex_undefined:D
1299 \tex_fi:D

```

A few packages do unfortunate things to date-related primitives.

```

1300 \tex_begingroup:D
1301 \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_time:D }
1302 \tex_edef:D \l_tmpb_tl { \tex_string:D \time }

```

```

1303 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1304 \tex_else:D
1305   \tex_global:D \tex_let:D \tex_time:D \tex_undefined:D
1306 \tex_fi:D
1307 \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_day:D }
1308 \tex_edef:D \l_tmpb_tl { \tex_string:D \day }
1309 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1310 \tex_else:D
1311   \tex_global:D \tex_let:D \tex_day:D \tex_undefined:D
1312 \tex_fi:D
1313 \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_month:D }
1314 \tex_edef:D \l_tmpb_tl { \tex_string:D \month }
1315 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1316 \tex_else:D
1317   \tex_global:D \tex_let:D \tex_month:D \tex_undefined:D
1318 \tex_fi:D
1319 \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_year:D }
1320 \tex_edef:D \l_tmpb_tl { \tex_string:D \year }
1321 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1322 \tex_else:D
1323   \tex_global:D \tex_let:D \tex_year:D \tex_undefined:D
1324 \tex_fi:D
1325 \tex_endgroup:D

```

Up to v0.80, LuaTeX defines the pdfTeX version data: rather confusing. Removing them means that `\tex_pdftexversion:D` is a marker for pdfTeX alone: useful in engine-dependent code later.

```

1326 \tex_ifdefined:D \tex luatexversion:D
1327   \tex_let:D \tex_pdftexbanner:D \tex_undefined:D
1328   \tex_let:D \tex_pdftexrevision:D \tex_undefined:D
1329   \tex_let:D \tex_pdftexversion:D \tex_undefined:D
1330 \tex_fi:D

```

For ConTeXt, two tests are needed. Both Mark II and Mark IV move several primitives: these are all covered by the first test, again using `\end` as a marker. For Mark IV, a few more primitives are moved: they are implemented using some Lua code in the current ConTeXt.

```

1331 \tex_ifdefined:D \normalend
1332   \tex_let:D \tex_end:D \normalend
1333   \tex_let:D \tex_everyjob:D \normaleveryjob
1334   \tex_let:D \tex_input:D \normalinput
1335   \tex_let:D \tex_language:D \normallanguage
1336   \tex_let:D \tex_mathop:D \normalmathop
1337   \tex_let:D \tex_month:D \normalmonth
1338   \tex_let:D \tex_outer:D \normalouter
1339   \tex_let:D \tex_over:D \normalover
1340   \tex_let:D \tex_vcenter:D \normalvcenter
1341   \tex_let:D \tex_unexpanded:D \normalunexpanded
1342   \tex_let:D \tex_expanded:D \normalexpanded
1343 \tex_fi:D
1344 \tex_ifdefined:D \normalitaliccorrection
1345   \tex_let:D \tex_hoffset:D \normalhoffset
1346   \tex_let:D \tex_italiccorrection:D \normalitaliccorrection
1347   \tex_let:D \tex_voffset:D \normalvoffset

```

```

1348 \tex_let:D \tex_showtokens:D \normalshowtokens
1349 \tex_let:D \tex_bodydir:D \spac_directions_normal_body_dir
1350 \tex_let:D \tex_pagedir:D \spac_directions_normal_page_dir
1351 \tex_fi:D
1352 \tex_ifdefined:D \normalleft
1353 \tex_let:D \tex_left:D \normalleft
1354 \tex_let:D \tex_middle:D \normalmiddle
1355 \tex_let:D \tex_right:D \normalright
1356 \tex_fi:D
1357 \</package>

```

3 Internal kernel functions

`__kernel_chk_cs_exist:N`
`__kernel_chk_cs_exist:c`

`__kernel_chk_cs_exist:N` $\langle cs \rangle$

This function is only created if debugging is enabled. It checks that $\langle cs \rangle$ exists according to the criteria for `\cs_if_exist_p:N`, and if not raises a kernel-level error.

`__kernel_chk_defined:NT`

`__kernel_chk_defined:NT` $\langle variable \rangle$ $\{ \langle true\ code \rangle \}$

If $\langle variable \rangle$ is not defined (according to `\cs_if_exist:NTF`), this triggers an error, otherwise the $\langle true\ code \rangle$ is run.

`__kernel_chk_expr:nNnN`

`__kernel_chk_expr:nNnN` $\{ \langle expr \rangle \}$ $\langle eval \rangle$ $\{ \langle convert \rangle \}$ $\langle caller \rangle$

This function is only created if debugging is enabled. By default it is equivalent to `\use_i:nmmn`. When expression checking is enabled, it leaves in the input stream the result of `\tex_the:D` $\langle eval \rangle$ $\langle expr \rangle$ `\tex_relax:D` after checking that no token was left over. If any token was not taken as part of the expression, there is an error message displaying the result of the evaluation as well as the $\langle caller \rangle$. For instance $\langle eval \rangle$ can be `__int_eval:w` and $\langle caller \rangle$ can be `\int_eval:n` or `\int_set:Nn`. The argument $\langle convert \rangle$ is empty except for mu expressions where it is `\tex_mutoglu:D`, used for internal purposes.

`__kernel_cs_parm_from_arg_count:nnF`

`__kernel_cs_parm_from_arg_count:nnF` $\{ \langle follow-on \rangle \}$ $\{ \langle args \rangle \}$
 $\{ \langle false\ code \rangle \}$

Evaluates the number of $\langle args \rangle$ and leaves the $\langle follow-on \rangle$ code followed by a brace group containing the required number of primitive parameter markers (`#1`, *etc.*). If the number of $\langle args \rangle$ is outside the range $[0, 9]$, the $\langle false\ code \rangle$ is inserted *instead* of the $\langle follow-on \rangle$.

`__kernel_deprecation_code:nn`

`__kernel_deprecation_code:nn` $\{ \langle error\ code \rangle \}$ $\{ \langle working\ code \rangle \}$

Stores both an $\langle error \rangle$ and $\langle working \rangle$ definition for given material such that they can be exchanged by `\debug_on:` and `\debug_off:`.

`__kernel_exp_not:w` ★

`__kernel_exp_not:w` $\langle expandable\ tokens \rangle$ $\{ \langle content \rangle \}$

Carries out expansion on the $\langle expandable\ tokens \rangle$ before preventing further expansion of the $\langle content \rangle$ as for `\exp_not:n`. Typically, the $\langle expandable\ tokens \rangle$ will alter the nature of the $\langle content \rangle$, *i.e.* allow it to be generated in some way.

`\l__kernel_expl_bool` A boolean which records the current code syntax status: `true` if currently inside a code environment. This variable should only be set by `\ExplSyntaxOn/\ExplSyntaxOff`.

(End definition for `\l__kernel_expl_bool`.)

`__kernel_file_missing:n` `__kernel_file_missing:n <{name}>`

Expands the `<name>` as per `__kernel_file_name_sanitiz:nN` then produces an error message indicating that this file was not found.

`__kernel_file_name_sanitiz:nN` `__kernel_file_name_sanitiz:nN <{name}> <str var>`

For converting a `<name>` to a string where active characters are treated as strings.

`__kernel_file_input_push:n` `__kernel_file_input_push:n <{name}>`
`__kernel_file_input_pop:` `__kernel_file_input_pop:`

Used to push and pop data from the internal file stack: needed only in package mode, where interfacing with the L^AT_EX 2_ε kernel is necessary.

`__kernel_int_add:nnn *` `__kernel_int_add:nnn <{integer}_1> <{integer}_2> <{integer}_3>`

Expands to the result of adding the three `<integers>` (which must be suitable input for `\int_eval:w`), avoiding intermediate overflow. Overflow occurs only if the overall result is outside $[-2^{31}+1, 2^{31}-1]$. The `<integers>` may be of the form `\int_eval:w ... \scan_stop:` but may be evaluated more than once.

`__kernel_intarray_gset:Nnn` `__kernel_intarray_gset:Nnn <intarray var> <{index}> <{value}>`

New: 2018-03-31

Faster version of `\intarray_gset:Nnn`. Stores the `<value>` into the `<integer array variable>` at the `<position>`. The `<index>` and `<value>` must be suitable for a direct assignment to a T_EX count register, for instance expanding to an integer denotation or obtained through the primitive `\numexpr` (which may be un-terminated). No bound checking is performed: the caller is responsible for ensuring that the `<position>` is between 1 and the `\intarray_count:N`, and the `<value>`'s absolute value is at most $2^{30}-1$. Assignments are always global.

`__kernel_intarray_item:Nn *` `__kernel_intarray_item:Nn <intarray var> <{index}>`

New: 2018-03-31

Faster version of `\intarray_item:Nn`. Expands to the integer entry stored at the `<index>` in the `<integer array variable>`. The `<index>` must be suitable for a direct assignment to a T_EX count register and must be between 1 and the `\intarray_count:N`, lest a low-level T_EX error occur.

`__kernel_ior_open:Nn` `__kernel_ior_open:Nn <stream> <{file name}>`
`__kernel_ior_open:No`

This function has identical syntax to the public version. However, it does not take precautions against active characters in the `<file name>`, and it does not attempt to add a `<path>` to the `<file name>`: it is therefore intended to be used by higher-level functions which have already fully expanded the `<file name>` and which need to perform multiple open or close operations. See for example the implementation of `\file_get_full_name:nN`,

<code>_kernel_iow_with:Nnn</code>	<code>_kernel_iow_with:Nnn <integer> {<value>} {<code>}</code>
------------------------------------	---

If the *<integer>* is equal to the *<value>* then this function simply runs the *<code>*. Otherwise it saves the current value of the *<integer>*, sets it to the *<value>*, runs the *<code>*, and restores the *<integer>* to its former value. This is used to ensure that the `\newlinechar` is 10 when writing to a stream, which lets `\iow_newline:` work, and that `\errorcontextlines` is `-1` when displaying a message.

<code>_kernel_msg_new:nnnn</code>	<code>_kernel_msg_new:nnnn {<module>} {<message>} {<text>} {<more text>}</code>
------------------------------------	--

Creates a kernel *<message>* for a given *<module>*. The message is defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (*#1* to *#4*) can be used: these will be supplied and expanded at the time the message is used. An error is raised if the *<message>* already exists.

<code>_kernel_msg_set:nnnn</code>	<code>_kernel_msg_set:nnnn {<module>} {<message>} {<text>} {<more text>}</code>
------------------------------------	--

Sets up the text for a kernel *<message>* for a given *<module>*. The message is defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (*#1* to *#4*) can be used: these will be supplied and expanded at the time the message is used.

<code>_kernel_msg_fatal:nnnnnn</code>	<code>_kernel_msg_fatal:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
--	--

Issues kernel *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. After issuing a fatal error the T_EX run halts. Cannot be redirected.

<code>_kernel_msg_critical:nnnnnn</code>	<code>_kernel_msg_critical:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
---	---

Issues kernel *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. After issuing a critical error, T_EX stops reading the current input file. Cannot be redirected.

```

\__kernel_msg_error:nnnnnn
\__kernel_msg_error:nnxxxx
\__kernel_msg_error:nnnnn
\__kernel_msg_error:nnxxx
\__kernel_msg_error:nnnn
\__kernel_msg_error:nnxx
\__kernel_msg_error:nnn
\__kernel_msg_error:nnx
\__kernel_msg_error:nn

```

```

\__kernel_msg_error:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg
three} {\arg four}

```

Issues kernel *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The error stops processing and issues the text at the terminal. After user input, the run continues. Cannot be redirected.

```

\__kernel_msg_warning:nnnnnn
\__kernel_msg_warning:nnxxxx
\__kernel_msg_warning:nnnnn
\__kernel_msg_warning:nnxxx
\__kernel_msg_warning:nnnn
\__kernel_msg_warning:nnxx
\__kernel_msg_warning:nnn
\__kernel_msg_warning:nnx
\__kernel_msg_warning:nn

```

```

\__kernel_msg_warning:nnnnnn {\module} {\message} {\arg one} {\arg
two} {\arg three} {\arg four}

```

Issues kernel *<module>* warning *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The warning text is added to the log file, but the T_EX run is not interrupted.

```

\__kernel_msg_info:nnnnnn
\__kernel_msg_info:nnxxxx
\__kernel_msg_info:nnnnn
\__kernel_msg_info:nnxxx
\__kernel_msg_info:nnnn
\__kernel_msg_info:nnxx
\__kernel_msg_info:nnn
\__kernel_msg_info:nnx
\__kernel_msg_info:nn

```

```

\__kernel_msg_info:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg
three} {\arg four}

```

Issues kernel *<module>* information *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The information text is added to the log file.

```

\__kernel_msg_expandable_error:nnnnnn *
\__kernel_msg_expandable_error:nnffff *
\__kernel_msg_expandable_error:nnnnn *
\__kernel_msg_expandable_error:nnffff *
\__kernel_msg_expandable_error:nnnn *
\__kernel_msg_expandable_error:nnff *
\__kernel_msg_expandable_error:nnn *
\__kernel_msg_expandable_error:nnf *
\__kernel_msg_expandable_error:nn *

```

```

\__kernel_msg_expandable_error:nnnnnn {\module} {\message}
{\arg one} {\arg two} {\arg three} {\arg four}

```

Issues an error, passing *<arg one>* to *<arg four>* to the text-creating functions. The resulting string must be much shorter than a line, otherwise it is cropped.

`\g__kernel_prg_map_int`

This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions `\<type>_map_1:w`, `\<type>_map_2:w`, *etc.*, labelled by `\g__kernel_prg_map_int` hold functions to be mapped over various list datatypes in inline and variable mappings.

(End definition for `\g__kernel_prg_map_int.`)

`__kernel_quark_new_test:N`

`__kernel_quark_new_test:N \<name>:<arg spec>`

Defines a quark-test function `\<name>:<arg spec>` which tests if its argument is `\q__-<namespace>_recursion_tail`, then acts accordingly, as described below for each possible `<arg spec>`.

The `<namespace>` is determined as the first (nonempty) `_`-delimited word in `<name>` and is used internally in the definition of auxiliaries. The function `__kernel_quark_new_test:N` does *not* define the `\q__<namespace>_recursion_tail` and `\q__<namespace>_recursion_stop` quarks. They should be manually defined with `\quark_new:N`.

There are 6 different types of quark-test functions. Which one is defined depends on the `<arg spec>`, which *must* be one of the options listed now. Four of them are modeled after `\quark_if_recursion_tail:(N|n)` and `\quark_if_recursion_tail_do:(N|n)n`.

`n` defines `\<name>:n` such that it checks if #1 contains only `\q__<namespace>_recursion_tail`, and if so consumes all tokens up to `\q__<namespace>_recursion_stop` (c.f. `\quark_if_recursion_tail_stop:n`).

`nn` defines `\<name>:nn` such that it checks if #1 contains only `\q__<namespace>_recursion_tail`, and if so consumes all tokens up to `\q__<namespace>_recursion_stop`, then executes the code #2 after that (c.f. `\quark_if_recursion_tail_stop_do:nn`).

`N` defines `\<name>:N` such that it checks if #1 is `\q__<namespace>_recursion_tail`, and if so consumes all tokens up to `\q__<namespace>_recursion_stop` (c.f. `\quark_if_recursion_tail_stop:N`).

`Nn` defines `\<name>:Nn` such that it checks if #1 is `\q__<namespace>_recursion_tail`, and if so consumes all tokens up to `\q__<namespace>_recursion_stop`, then executes the code #2 after that (c.f. `\quark_if_recursion_tail_stop_do:Nn`).

The last two are modeled after `\quark_if_recursion_tail_break:(n|N)N`, and in those cases the quark `\q__<namespace>_recursion_stop` is not used (and thus needs not be defined).

`nN` defines `\<name>:nN` such that it checks if #1 contains only `\q__<namespace>_recursion_tail`, and if so uses the `\<type>_map_break:` function #2.

`NN` defines `\<name>:NN` such that it checks if #1 is `\q__<namespace>_recursion_tail`, and if so uses the `\<type>_map_break:` function #2.

Any other signature, as well as a function without signature are errors, and in such case the definition is aborted.

<u><code>__kernel_quark_new_conditional:Nn</code></u>	<code>__kernel_quark_new_conditional:Nn</code> <code>__<namespace>_quark_if_<name>:<arg spec> {<conditions>}</code>
--	--

Defines a collection of quark conditionals that test if their argument is the quark `\q_`
`__<namespace>_<name>` and perform suitable actions. The `<conditions>` are a comma-separated list of one or more of p, T, F, and TF, and one conditional is defined for each `<condition>` in the list, as described for `\prg_new_conditional:Npnn`. The conditionals are defined using `\prg_new_conditional:Npnn`, so that their name is obtained by adding p, T, F, or TF to the base name `__<namespace>_quark_if_<name>:<arg spec>`.

The first argument of `__kernel_quark_new_conditional:Nn` must contain `_quark_if_` and `:`, as these markers are used to determine the `<name>` of the quark `\q_`
`__<namespace>_<name>` to be tested. This quark should be manually defined with `\quark_new:N`, as `__kernel_quark_new_conditional:Nn` does *not* define it.

The function `__kernel_quark_new_conditional:Nn` can define 2 different types of quark conditionals. Which one is defined depends on the `<arg spec>`, which *must* be one of the following options, modeled after `\quark_if_nil:(N|n)(TF)`.

`n` defines `__<namespace>_quark_if_<name>:n(TF)` such that it checks if #1 contains only `\q_`
`__<namespace>_<name>`, and executes the proper conditional branch.

`N` defines `__<namespace>_quark_if_<name>:N(TF)` such that it checks if #1 is `\q_`
`__<namespace>_<name>`, and executes the proper conditional branch.

Any other signature, as well as a function without signature are errors, and in such case the definition is aborted.

`\c__kernel_randint_max_int` Maximal allowed argument to `__kernel_randint:n`. Equal to $2^{17} - 1$.

(End definition for `\c__kernel_randint_max_int`.)

<u><code>__kernel_randint:n</code></u>	<code>__kernel_randint:n {<max>}</code>
---	--

Used in an integer expression this gives a pseudo-random number between 1 and `<max>` included. One must have $\langle max \rangle \leq 2^{17} - 1$. The `<max>` must be suitable for `\int_value:w` (and any `\int_eval:w` must be terminated by `\scan_stop:` or equivalent).

<u><code>__kernel_randint:nn</code></u>	<code>__kernel_randint:nn {<min>} {<max>}</code>
--	---

Used in an integer expression this gives a pseudo-random number between `<min>` and `<max>` included. The `<min>` and `<max>` must be suitable for `\int_value:w` (and any `\int_eval:w` must be terminated by `\scan_stop:` or equivalent). For small ranges $R = \langle max \rangle - \langle min \rangle + 1 \leq 2^{17} - 1$, `<min> - 1 + __kernel_randint:n{R}` is faster.

<u><code>__kernel_register_show:N</code></u> <u><code>__kernel_register_show:c</code></u>	<code>__kernel_register_show:N <register></code>
--	---

Used to show the contents of a T_EX register at the terminal, formatted such that internal parts of the mechanism are not visible.

<u><code>__kernel_register_log:N</code></u> <u><code>__kernel_register_log:c</code></u>	<code>__kernel_register_log:N <register></code>
--	--

Used to write the contents of a T_EX register to the log file in a form similar to `__kernel_register_show:N`.

<code>_kernel_str_to_other:n</code> ★	<code>_kernel_str_to_other:n</code> { $\langle token list \rangle$ }
--	---

Converts the $\langle token list \rangle$ to a $\langle other string \rangle$, where spaces have category code “other”. This function can be f-expanded without fear of losing a leading space, since spaces do not have category code 10 in its result. It takes a time quadratic in the character count of the string.

<code>_kernel_str_to_other_fast:n</code> ☆	<code>_kernel_str_to_other_fast:n</code> { $\langle token list \rangle$ }
---	--

Same behaviour `_kernel_str_to_other:n` but only restricted-expandable. It takes a time linear in the character count of the string.

<code>_kernel_tl_to_str:w</code> ★	<code>_kernel_tl_to_str:w</code> $\langle expandable tokens \rangle$ { $\langle tokens \rangle$ }
-------------------------------------	--

Carries out expansion on the $\langle expandable tokens \rangle$ before conversion of the $\langle tokens \rangle$ to a string as describe for `\tl_to_str:n`. Typically, the $\langle expandable tokens \rangle$ will alter the nature of the $\langle tokens \rangle$, *i.e.* allow it to be generated in some way. This function requires only a single expansion.

4 Kernel backend functions

These functions are required to pass information to the backend. The nature of these means that they are defined only when the relevant backend is in use.

<code>_kernel_backend_literal:n</code>	<code>_kernel_backend_literal:n</code> { $\langle content \rangle$ }
<code>_kernel_backend_literal:(e x)</code>	

Adds the $\langle content \rangle$ literally to the current vertical list as a whatsit. The nature of the $\langle content \rangle$ will depend on the backend in use.

<code>_kernel_backend_literal_postscript:n</code>	<code>_kernel_backend_literal_postscript:n</code> { $\langle PostScript \rangle$ }
<code>_kernel_backend_literal_postscript:x</code>	

Adds the $\langle PostScript \rangle$ literally to the current vertical list as a whatsit. No positioning is applied.

<code>_kernel_backend_literal_pdf:n</code>	<code>_kernel_backend_literal_pdf:n</code> { $\langle PDF instructions \rangle$ }
<code>_kernel_backend_literal_pdf:x</code>	

Adds the $\langle PDF instructions \rangle$ literally to the current vertical list as a whatsit. No positioning is applied.

<code>_kernel_backend_literal_svg:n</code>	<code>_kernel_backend_literal_svg:n</code> { $\langle SVG instructions \rangle$ }
<code>_kernel_backend_literal_svg:x</code>	

Adds the $\langle SVG instructions \rangle$ literally to the current vertical list as a whatsit. No positioning is applied.

<code>_kernel_backend_postscript:n</code>	<code>_kernel_backend_postscript:n</code> { $\langle PostScript \rangle$ }
<code>_kernel_backend_postscript:x</code>	

Adds the $\langle PostScript \rangle$ to the current vertical list as a whatsit. The PostScript reference point is adjusted to match the current position. The PostScript is inserted inside a `SDict begin/end` pair.

<code>_kernel_backend_align_begin:</code>	<code>_kernel_backend_align_begin:</code>
<code>_kernel_backend_align_end:</code>	<code>_kernel_backend_align_end:</code>

Arranges to align the PostScript and DVI current positions and scales.

<code>_kernel_backend_scope_begin:</code>	<code>_kernel_backend_scope_begin:</code>
<code>_kernel_backend_scope_end:</code>	<code>_kernel_backend_scope_end:</code>

Creates a scope for instructions at the backend level.

<code>_kernel_backend_matrix:n</code>	<code>_kernel_backend_matrix:n {<matrix>}</code>
<code>_kernel_backend_matrix:x</code>	Applies the <code><matrix></code> to the current transformation matrix.

`\g_kernel_backend_header_bool`

Specifies whether to write headers for the backend.

<code>\l_kernel_color_stack_int</code>	The color stack used in pdf \TeX and Lua \TeX for the main color.
--	---

5 l3basics implementation

1358 `<*package>`

5.1 Renaming some \TeX primitives (again)

Having given all the \TeX primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but we do a few now, just to get started.⁶

<code>\if_true:</code>	Then some conditionals.
<code>\if_false:</code>	1359 <code>\tex_let:D \if_true:</code> <code>\tex_iftrue:D</code>
<code>\or:</code>	1360 <code>\tex_let:D \if_false:</code> <code>\tex_iffalse:D</code>
<code>\else:</code>	1361 <code>\tex_let:D \or:</code> <code>\tex_or:D</code>
<code>\fi:</code>	1362 <code>\tex_let:D \else:</code> <code>\tex_else:D</code>
<code>\reverse_if:N</code>	1363 <code>\tex_let:D \fi:</code> <code>\tex_fi:D</code>
<code>\if:w</code>	1364 <code>\tex_let:D \reverse_if:N</code> <code>\tex_unless:D</code>
<code>\if_charcode:w</code>	1365 <code>\tex_let:D \if:w</code> <code>\tex_if:D</code>
<code>\if_catcode:w</code>	1366 <code>\tex_let:D \if_charcode:w</code> <code>\tex_if:D</code>
<code>\if_meaning:w</code>	1367 <code>\tex_let:D \if_catcode:w</code> <code>\tex_ifcat:D</code>
	1368 <code>\tex_let:D \if_meaning:w</code> <code>\tex_ifx:D</code>
	1369 <code>\tex_let:D \if_bool:N</code> <code>\tex_ifodd:D</code>

(End definition for `\if_true:` and others. These functions are documented on page 23.)

<code>\if_mode_math:</code>	\TeX lets us detect some if its modes.
<code>\if_mode_horizontal:</code>	1370 <code>\tex_let:D \if_mode_math:</code> <code>\tex_ifmmode:D</code>
<code>\if_mode_vertical:</code>	1371 <code>\tex_let:D \if_mode_horizontal:</code> <code>\tex_ifhmode:D</code>
<code>\if_mode_inner:</code>	1372 <code>\tex_let:D \if_mode_vertical:</code> <code>\tex_ifvmode:D</code>
	1373 <code>\tex_let:D \if_mode_inner:</code> <code>\tex_ifinner:D</code>

⁶This renaming gets expensive in terms of csname usage, an alternative scheme would be to just use the `\tex_...:D` name in the cases where no good alternative exists.

(End definition for `\if_mode_math:` and others. These functions are documented on page 23.)

`\if_cs_exist:N` Building csnames and testing if control sequences exist.
`\if_cs_exist:w` 1374 `\tex_let:D \if_cs_exist:N \tex_ifdefined:D`
`\cs:w` 1375 `\tex_let:D \if_cs_exist:w \tex_ifcsname:D`
`\cs_end:` 1376 `\tex_let:D \cs:w \tex_csname:D`
1377 `\tex_let:D \cs_end: \tex_endcsname:D`

(End definition for `\if_cs_exist:N` and others. These functions are documented on page 23.)

`\exp_after:wN` The five `\exp_` functions are used in the `l3expan` module where they are described.
`\exp_not:N` 1378 `\tex_let:D \exp_after:wN \tex_expandafter:D`
`\exp_not:n` 1379 `\tex_let:D \exp_not:N \tex_noexpand:D`
1380 `\tex_let:D \exp_not:n \tex_unexpanded:D`
1381 `\tex_let:D \exp:w \tex_romannumeral:D`
1382 `\tex_chardef:D \exp_end: = 0 ~`

(End definition for `\exp_after:wN`, `\exp_not:N`, and `\exp_not:n`. These functions are documented on page 33.)

`\token_to_meaning:N` Examining a control sequence or token.
`\cs_meaning:N` 1383 `\tex_let:D \token_to_meaning:N \tex_meaning:D`
1384 `\tex_let:D \cs_meaning:N \tex_meaning:D`

(End definition for `\token_to_meaning:N` and `\cs_meaning:N`. These functions are documented on page 135.)

`\tl_to_str:n` Making strings.
`\token_to_str:N` 1385 `\tex_let:D \tl_to_str:n \tex_detokenize:D`
`__kernel_tl_to_str:w` 1386 `\tex_let:D \token_to_str:N \tex_string:D`
1387 `\tex_let:D __kernel_tl_to_str:w \tex_detokenize:D`

(End definition for `\tl_to_str:n`, `\token_to_str:N`, and `__kernel_tl_to_str:w`. These functions are documented on page 51.)

`\scan_stop:` The next three are basic functions for which there also exist versions that are safe inside
`\group_begin:` alignments. These safe versions are defined in the `l3prg` module.
`\group_end:` 1388 `\tex_let:D \scan_stop: \tex_relax:D`
1389 `\tex_let:D \group_begin: \tex_begingroup:D`
1390 `\tex_let:D \group_end: \tex_endgroup:D`

(End definition for `\scan_stop:`, `\group_begin:`, and `\group_end:`. These functions are documented on page 9.)

1391 `<@@=int>`

`\if_int_compare:w` For integers.
`__int_to_roman:w` 1392 `\tex_let:D \if_int_compare:w \tex_ifnum:D`
1393 `\tex_let:D __int_to_roman:w \tex_romannumeral:D`

(End definition for `\if_int_compare:w` and `__int_to_roman:w`. This function is documented on page 102.)

`\group_insert_after:N` Adding material after the end of a group.
1394 `\tex_let:D \group_insert_after:N \tex_aftergroup:D`

(End definition for `\group_insert_after:N`. This function is documented on page 9.)

`\exp_args:Nc` Discussed in `l3expan`, but needed much earlier.

```
\exp_args:cc 1395 \tex_long:D \tex_def:D \exp_args:Nc #1#2
              1396 { \exp_after:wN #1 \cs:w #2 \cs_end: }
              1397 \tex_long:D \tex_def:D \exp_args:cc #1#2
              1398 { \cs:w #1 \exp_after:wN \cs_end: \cs:w #2 \cs_end: }
```

(End definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 29.)

`\token_to_meaning:c` A small number of variants defined by hand. Some of the necessary functions (`\use_i:nn`, `\use_ii:nn`, and `\exp_args:Nnc`) are not defined at that point yet, but will be defined before those variants are used. The `\cs_meaning:c` command must check for an undefined control sequence to avoid defining it mistakenly.

```
1399 \tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
1400 \tex_long:D \tex_def:D \cs_meaning:c #1
1401 {
1402   \if_cs_exist:w #1 \cs_end:
1403     \exp_after:wN \use_i:nn
1404   \else:
1405     \exp_after:wN \use_ii:nn
1406   \fi:
1407   { \exp_args:Nc \cs_meaning:N {#1} }
1408   { \tl_to_str:n {undefined} }
1409 }
1410 \tex_let:D \token_to_meaning:c = \cs_meaning:c
```

(End definition for `\token_to_meaning:N`. This function is documented on page 135.)

5.2 Defining some constants

`\c_zero_int` We need the constant `\c_zero_int` which is used by some functions in the `l3alloc` module. The rest are defined in the `l3int` module – at least for the ones that can be defined with `\tex_chardef:D` or `\tex_mathchardef:D`. For other constants the `l3int` module is required but it can't be used until the allocation has been set up properly!

```
1411 \tex_chardef:D \c_zero_int = 0 ~
```

(End definition for `\c_zero_int`. This variable is documented on page 101.)

`\c_max_register_int` This is here as this particular integer is needed both in package mode and to bootstrap `l3alloc`, and is documented in `l3int`. LuaTeX and those which contain parts of the Omega extensions have more registers available than ε -TeX.

```
1412 \tex_ifdefined:D \tex_luatexversion:D
1413   \tex_chardef:D \c_max_register_int = 65 535 ~
1414 \tex_else:D
1415   \tex_ifdefined:D \tex_omathchardef:D
1416     \tex_omathchardef:D \c_max_register_int = 65535 ~
1417   \tex_else:D
1418     \tex_mathchardef:D \c_max_register_int = 32767 ~
1419   \tex_fi:D
1420 \tex_fi:D
```

(End definition for `\c_max_register_int`. This variable is documented on page 101.)

5.3 Defining functions

We start by providing functions for the typical definition functions. First the local ones.

`\cs_set_nopar:Npn` All assignment functions in L^AT_EX3 should be naturally protected; after all, the T_EX primitives for assignments are and it can be a cause of problems if others aren't.

`\cs_set_nopar:Npx`

`\cs_set:Npn` 1421 `\tex_let:D \cs_set_nopar:Npn \tex_def:D`

`\cs_set:Npx` 1422 `\tex_let:D \cs_set_nopar:Npx \tex_edef:D`

`\cs_set_protected_nopar:Npn` 1423 `\tex_protected:D \tex_long:D \tex_def:D \cs_set:Npn`

`\cs_set_protected_nopar:Npx` 1424 `{ \tex_long:D \tex_def:D }`

`\cs_set_protected:Npn` 1425 `\tex_protected:D \tex_long:D \tex_def:D \cs_set:Npx`

`\cs_set_protected:Npx` 1426 `{ \tex_long:D \tex_edef:D }`

1427 `\tex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npn`

1428 `{ \tex_protected:D \tex_def:D }`

1429 `\tex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npx`

1430 `{ \tex_protected:D \tex_edef:D }`

1431 `\tex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npn`

1432 `{ \tex_protected:D \tex_long:D \tex_def:D }`

1433 `\tex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npx`

1434 `{ \tex_protected:D \tex_long:D \tex_edef:D }`

(End definition for `\cs_set_nopar:Npn` and others. These functions are documented on page 11.)

`\cs_gset_nopar:Npn` Global versions of the above functions.

`\cs_gset_nopar:Npx` 1435 `\tex_let:D \cs_gset_nopar:Npn \tex_gdef:D`

`\cs_gset:Npn` 1436 `\tex_let:D \cs_gset_nopar:Npx \tex_xdef:D`

`\cs_gset:Npx` 1437 `\cs_set_protected:Npn \cs_gset:Npn`

`\cs_gset_protected_nopar:Npn` 1438 `{ \tex_long:D \tex_gdef:D }`

`\cs_gset_protected_nopar:Npx` 1439 `\cs_set_protected:Npn \cs_gset:Npx`

`\cs_gset_protected:Npn` 1440 `{ \tex_long:D \tex_xdef:D }`

`\cs_gset_protected:Npx` 1441 `\cs_set_protected:Npn \cs_gset_protected_nopar:Npn`

1442 `{ \tex_protected:D \tex_gdef:D }`

1443 `\cs_set_protected:Npn \cs_gset_protected_nopar:Npx`

1444 `{ \tex_protected:D \tex_xdef:D }`

1445 `\cs_set_protected:Npn \cs_gset_protected:Npn`

1446 `{ \tex_protected:D \tex_long:D \tex_gdef:D }`

1447 `\cs_set_protected:Npn \cs_gset_protected:Npx`

1448 `{ \tex_protected:D \tex_long:D \tex_xdef:D }`

(End definition for `\cs_gset_nopar:Npn` and others. These functions are documented on page 12.)

5.4 Selecting tokens

1449 `<@@=exp>`

`\l__exp_internal_tl` Scratch token list variable for l3expan, used by `\use:x`, used in defining conditionals. We don't use `tl` methods because l3basics is loaded earlier.

1450 `\cs_set_nopar:Npn \l__exp_internal_tl { }`

(End definition for `\l__exp_internal_tl`.)

`\use:c` This macro grabs its argument and returns a csname from it.

1451 `\cs_set:Npn \use:c #1 { \cs:w #1 \cs_end: }`

(End definition for `\use:c`. This function is documented on page 16.)

\use:x Fully expands its argument and passes it to the input stream. Uses the reserved `\l__exp_internal_tl` which we’ve set up above.

```

1452 \cs_set_protected:Npn \use:x #1
1453 {
1454   \cs_set_nopar:Npx \l__exp_internal_tl {#1}
1455   \l__exp_internal_tl
1456 }

```

(End definition for `\use:x`. This function is documented on page 20.)

```

1457 \@@=use

```

\use:e In non-LuaTeX engines older than 2019, `\expanded` is emulated.

```

1458 \cs_set:Npn \use:e #1 { \tex_expanded:D {#1} }
1459 \tex_ifdefined:D \tex_expanded:D \tex_else:D
1460   \cs_set:Npn \use:e #1 { \exp_args:Ne \use:n {#1} }
1461 \tex_fi:D

```

(End definition for `\use:e`. This function is documented on page 20.)

```

1462 \@@=exp

```

\use:n These macros grab their arguments and return them back to the input (with outer braces removed).

```

1463 \cs_set:Npn \use:n #1 {#1}
1464 \cs_set:Npn \use:nn #1#2 {#1#2}
1465 \cs_set:Npn \use:nnn #1#2#3 {#1#2#3}
1466 \cs_set:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}

```

(End definition for `\use:n` and others. These functions are documented on page 19.)

\use_i:nn The equivalent to L^AT_EX 2_ε’s `\@firstoftwo` and `\@secondoftwo`.

```

1467 \cs_set:Npn \use_i:nn #1#2 {#1}
1468 \cs_set:Npn \use_ii:nn #1#2 {#2}

```

(End definition for `\use_i:nn` and `\use_ii:nn`. These functions are documented on page 19.)

\use_i:nnn We also need something for picking up arguments from a longer list.

```

1469 \cs_set:Npn \use_i:nnn #1#2#3 {#1}
1470 \cs_set:Npn \use_ii:nnn #1#2#3 {#2}
1471 \cs_set:Npn \use_iii:nnn #1#2#3 {#3}
1472 \cs_set:Npn \use_i_ii:nnn #1#2#3 {#1#2}
1473 \cs_set:Npn \use_i:nnnn #1#2#3#4 {#1}
1474 \cs_set:Npn \use_ii:nnnn #1#2#3#4 {#2}
1475 \cs_set:Npn \use_iii:nnnn #1#2#3#4 {#3}
1476 \cs_set:Npn \use_iv:nnnn #1#2#3#4 {#4}

```

(End definition for `\use_i:nnn` and others. These functions are documented on page 19.)

\use_ii_i:nn

```

1477 \cs_set:Npn \use_ii_i:nn #1#2 { #2 #1 }

```

(End definition for `\use_ii_i:nn`. This function is documented on page 20.)

`\use_none_delimit_by_q_nil:w` Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_recursion_stop`, respectively.

```

\use_none_delimit_by_q_stop:w
\use_none_delimit_by_q_recursion_stop:w
1478 \cs_set:Npn \use_none_delimit_by_q_nil:w #1 \q_nil { }
1479 \cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
1480 \cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop { }

```

(End definition for `\use_none_delimit_by_q_nil:w`, `\use_none_delimit_by_q_stop:w`, and `\use_none_delimit_by_q_recursion_stop:w`. These functions are documented on page 21.)

`\use_i_delimit_by_q_nil:nw` Same as above but execute first argument after gobbling. Very useful when you need to skip the rest of a mapping sequence but want an easy way to control what should be expanded next.

```

\use_i_delimit_by_q_stop:nw
\use_i_delimit_by_q_recursion_stop:nw
1481 \cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2 \q_nil {#1}
1482 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
1483 \cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw
1484 #1#2 \q_recursion_stop {#1}

```

(End definition for `\use_i_delimit_by_q_nil:nw`, `\use_i_delimit_by_q_stop:nw`, and `\use_i_delimit_by_q_recursion_stop:nw`. These functions are documented on page 21.)

5.5 Gobbling tokens from input

`\use_none:n` To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of n's following the : in the name. Although we could define functions to remove ten arguments or more using separate calls of `\use_none:nnnnn`, this is very non-intuitive to the programmer who will assume that expanding such a function once takes care of gobbling all the tokens in one go.

```

\use_none:nn
\use_none:nnn
\use_none:nnnn
\use_none:nnnnn
\use_none:nnnnnn
\use_none:nnnnnnn
\use_none:nnnnnnnn
1485 \cs_set:Npn \use_none:n #1 { }
1486 \cs_set:Npn \use_none:nn #1#2 { }
1487 \cs_set:Npn \use_none:nnn #1#2#3 { }
1488 \cs_set:Npn \use_none:nnnn #1#2#3#4 { }
1489 \cs_set:Npn \use_none:nnnnn #1#2#3#4#5 { }
1490 \cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6 { }
1491 \cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7 { }
1492 \cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8 { }
1493 \cs_set:Npn \use_none:nnnnnnnnn #1#2#3#4#5#6#7#8#9 { }

```

(End definition for `\use_none:n` and others. These functions are documented on page 20.)

5.6 Debugging and patching later definitions

```

1494 <@@=debug>

```

`__kernel_if_debug:TF` A more meaningful test of whether debugging is enabled than messing up with guards. We can also more easily change the logic in one place then. This is needed primarily for deprecations.

```

1495 \cs_set_protected:Npn \__kernel_if_debug:TF #1#2 {#2}

```

(End definition for `__kernel_if_debug:TF`.)

`\debug_on:n` Stubs.

```
\debug_off:n 1496 \cs_set_protected:Npn \debug_on:n #1
1497 {
1498   \__kernel_msg_error:nnx { kernel } { enable-debug }
1499   { \tl_to_str:n { \debug_on:n {#1} } }
1500 }
1501 \cs_set_protected:Npn \debug_off:n #1
1502 {
1503   \__kernel_msg_error:nnx { kernel } { enable-debug }
1504   { \tl_to_str:n { \debug_off:n {#1} } }
1505 }
```

(End definition for `\debug_on:n` and `\debug_off:n`. These functions are documented on page 24.)

`\debug_suspend:`

`\debug_resume:`

```
1506 \cs_set_protected:Npn \debug_suspend: { }
1507 \cs_set_protected:Npn \debug_resume: { }
```

(End definition for `\debug_suspend:` and `\debug_resume:`. These functions are documented on page 24.)

`__kernel_deprecation_code:nn`

`\g__debug_deprecation_on_tl`

`\g__debug_deprecation_off_tl`

Some commands were more recently deprecated and not yet removed; only make these into errors if the user requests it. This relies on two token lists, filled up in `l3deprecation`.

```
1508 \cs_set_nopar:Npn \g__debug_deprecation_on_tl { }
1509 \cs_set_nopar:Npn \g__debug_deprecation_off_tl { }
1510 \cs_set_protected:Npn \__kernel_deprecation_code:nn #1#2
1511 {
1512   \tl_gput_right:Nn \g__debug_deprecation_on_tl {#1}
1513   \tl_gput_right:Nn \g__debug_deprecation_off_tl {#2}
1514 }
```

(End definition for `__kernel_deprecation_code:nn`, `\g__debug_deprecation_on_tl`, and `\g__debug_deprecation_off_tl`.)

5.7 Conditional processing and definitions

```
1515 <@@=prg>
```

Underneath any predicate function (`_p`) or other conditional forms (TF, etc.) is a built-in logic saying that it after all of the testing and processing must return the *state* this leaves \TeX in. Therefore, a simple user interface could be something like

```
\if_meaning:w #1#2
\prg_return_true:
\else:
\if_meaning:w #1#3
\prg_return_true:
\else:
\prg_return_false:
\fi:
\fi:
```

Usually, a \TeX programmer would have to insert a number of `\exp_after:wN`s to ensure the state value is returned at exactly the point where the last conditional is finished. However, that obscures the code and forces the \TeX programmer to prove that he/she knows the $2^n - 1$ table. We therefore provide the simpler interface.

`\prg_return_true:` The idea here is that `\exp:w` expands fully any `\else:` and `\fi:` that are waiting to be discarded, before reaching the `\exp_end:` which leaves an empty expansion. The code can then leave either the first or second argument in the input stream. This means that all of the branching code has to contain at least two tokens: see how the logical tests are actually implemented to see this.

```

1516 \cs_set:Npn \prg_return_true:
1517   { \exp_after:wN \use_i:nn \exp:w }
1518 \cs_set:Npn \prg_return_false:
1519   { \exp_after:wN \use_ii:nn \exp:w}

```

An extended state space could be implemented by including a more elaborate function in place of `\use_i:nn/\use_ii:nn`. Provided two arguments are absorbed then the code would work.

(End definition for \prg_return_true: and \prg_return_false:. These functions are documented on page 108.)

`\prg_use_none_delimit_by_q_recursion_stop:w` Private version of `\use_none_delimit_by_q_recursion_stop:w`.

```

1520 \cs_set:Npn \prg_use_none_delimit_by_q_recursion_stop:w
1521   #1 \q__prg_recursion_stop { }

```

(End definition for \prg_use_none_delimit_by_q_recursion_stop:w.)

`\prg_set_conditional:Npnn` The user functions for the types using parameter text from the programmer. The various functions only differ by which function is used for the assignment. For those `Npnn` type functions, we must grab the parameter text, reading everything up to a left brace before continuing. Then split the base function into name and signature, and feed `{\name}` `{\signature}` `\boolean` `{\set or new}` `{\maybe protected}` `{\parameters}` `{TF,...}` `{\code}` to the auxiliary function responsible for defining all conditionals. Note that `e` stands for expandable and `p` for protected.

```

1522 \cs_set_protected:Npn \prg_set_conditional:Npnn
1523   { \prg_generate_conditional_parm:NNNpnn \cs_set:Npn e }
1524 \cs_set_protected:Npn \prg_new_conditional:Npnn
1525   { \prg_generate_conditional_parm:NNNpnn \cs_new:Npn e }
1526 \cs_set_protected:Npn \prg_set_protected_conditional:Npnn
1527   { \prg_generate_conditional_parm:NNNpnn \cs_set_protected:Npn p }
1528 \cs_set_protected:Npn \prg_new_protected_conditional:Npnn
1529   { \prg_generate_conditional_parm:NNNpnn \cs_new_protected:Npn p }
1530 \cs_set_protected:Npn \prg_generate_conditional_parm:NNNpnn #1#2#3#4#
1531   {
1532     \use:x
1533     {
1534       \prg_generate_conditional:nnNNNnnn
1535       \cs_split_function:N #3
1536     }
1537     #1 #2 {#4}
1538   }

```

(End definition for \prg_set_conditional:Npnn and others. These functions are documented on page 106.)

`\prg_set_conditional:Nnn` The user functions for the types automatically inserting the correct parameter text based on the signature. The various functions only differ by which function is used for the assignment. Split the base function into name and signature. The second auxiliary

`\prg_generate_conditional_count:NNNnn`
`\prg_generate_conditional_count:nnNNNnn`

generates the parameter text from the number of letters in the signature. Then feed $\langle name \rangle$ $\langle signature \rangle$ $\langle boolean \rangle$ $\langle set \text{ or } new \rangle$ $\langle maybe \text{ protected} \rangle$ $\langle parameters \rangle$ $\{TF, \dots\}$ $\langle code \rangle$ to the auxiliary function responsible for defining all conditionals. If the $\langle signature \rangle$ has more than 9 letters, the definition is aborted since T_EX macros have at most 9 arguments. The erroneous case where the function name contains no colon is captured later.

```

1539 \cs_set_protected:Npn \prg_set_conditional:Nnn
1540   { \prg_generate_conditional_count:NNNnn \cs_set:Npn e }
1541 \cs_set_protected:Npn \prg_new_conditional:Nnn
1542   { \prg_generate_conditional_count:NNNnn \cs_new:Npn e }
1543 \cs_set_protected:Npn \prg_set_protected_conditional:Nnn
1544   { \prg_generate_conditional_count:NNNnn \cs_set_protected:Npn p }
1545 \cs_set_protected:Npn \prg_new_protected_conditional:Nnn
1546   { \prg_generate_conditional_count:NNNnn \cs_new_protected:Npn p }
1547 \cs_set_protected:Npn \prg_generate_conditional_count:NNNnn #1#2#3
1548   {
1549     \use:x
1550     {
1551       \prg_generate_conditional_count:nnNNNnn
1552       \cs_split_function:N #3
1553     }
1554     #1 #2
1555   }
1556 \cs_set_protected:Npn \prg_generate_conditional_count:nnNNNnn #1#2#3#4#5
1557   {
1558     \kernel_cs_parm_from_arg_count:nnF
1559     { \prg_generate_conditional:nnNNNnn {#1} {#2} #3 #4 #5 }
1560     { \tl_count:n {#2} }
1561     {
1562       \kernel_msg_error:nxxx { kernel } { bad-number-of-arguments }
1563       { \token_to_str:c { #1 : #2 } }
1564       { \tl_count:n {#2} }
1565       \use_none:nn
1566     }
1567   }

```

(End definition for $\backslash\text{prg_set_conditional:Nnn}$ and others. These functions are documented on page 106.)

$\backslash\text{prg_generate_conditional:nnNNNnnn}$
 $\backslash\text{prg_generate_conditional:NNnnnnNw}$
 $\backslash\text{prg_generate_conditional_test:w}$
 $\backslash\text{prg_generate_conditional_fast:nw}$

The workhorse here is going through a list of desired forms, *i.e.*, p, TF, T and F. The first three arguments come from splitting up the base form of the conditional, which gives the name, signature and a boolean to signal whether or not there was a colon in the name. In the absence of a colon, we throw an error and don't define any conditional. The fourth and fifth arguments build up the defining function. The sixth is the parameters to use (possibly empty), the seventh is the list of forms to define, the eighth is the replacement text which we will augment when defining the forms. The use of $\backslash\text{tl_to_str:n}$ makes the later loop more robust.

A large number of our low-level conditionals look like $\langle code \rangle \backslash\text{prg_return_true:} \backslash\text{else:} \backslash\text{prg_return_false:} \backslash\text{fi:}$ so we optimize this special case by calling $\backslash\text{prg_generate_conditional_fast:nw} \langle code \rangle$. This passes $\backslash\text{use_i:nn}$ instead of $\backslash\text{use_i_ii:nnn}$ to functions such as $\backslash\text{prg_generate_p_form:wNNnnnnN}$.

```

1568 \cs_set_protected:Npn \prg_generate_conditional:nnNNNnnn #1#2#3#4#5#6#7#8
1569   {

```

```

1570 \if_meaning:w \c_false_bool #3
1571 \__kernel_msg_error:nxx { kernel } { missing-colon }
1572 { \token_to_str:c {#1} }
1573 \exp_after:wN \use_none:nn
1574 \fi:
1575 \use:x
1576 {
1577 \exp_not:N \__prg_generate_conditional:NNnnnnNw
1578 \exp_not:n { #4 #5 {#1} {#2} {#6} }
1579 \__prg_generate_conditional_test:w
1580 #8 \s__prg_mark
1581 \__prg_generate_conditional_fast:nw
1582 \prg_return_true: \else: \prg_return_false: \fi: \s__prg_mark
1583 \use_none:n
1584 \exp_not:n { {#8} \use_i_ii:nnn }
1585 \tl_to_str:n {#7}
1586 \exp_not:n { , \q__prg_recursion_tail , \q__prg_recursion_stop }
1587 }
1588 }
1589 \cs_set:Npn \__prg_generate_conditional_test:w
1590 #1 \prg_return_true: \else: \prg_return_false: \fi: \s__prg_mark #2
1591 { #2 {#1} }
1592 \cs_set:Npn \__prg_generate_conditional_fast:nw #1#2 \exp_not:n #3
1593 { \exp_not:n { {#1} \use_i:nn } }

```

Looping through the list of desired forms. First are six arguments and seventh is the form. Use the form to call the correct type. If the form does not exist, the `\use:c` construction results in `\relax`, and the error message is displayed (unless the form is empty, to allow for `{T, , F}`), then `\use_none:nnnnnnnn` cleans up. Otherwise, the error message is removed by the variant form.

```

1594 \cs_set_protected:Npn \__prg_generate_conditional:NNnnnnNw #1#2#3#4#5#6#7#8 ,
1595 {
1596 \if_meaning:w \q__prg_recursion_tail #8
1597 \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
1598 \fi:
1599 \use:c { __prg_generate_ #8 _form:wNNnnnnN }
1600 \tl_if_empty:nF {#8}
1601 {
1602 \__kernel_msg_error:nxxx
1603 { kernel } { conditional-form-unknown }
1604 {#8} { \token_to_str:c { #3 : #4 } }
1605 }
1606 \use_none:nnnnnnnn
1607 \s__prg_stop
1608 #1 #2 {#3} {#4} {#5} {#6} #7
1609 \__prg_generate_conditional:NNnnnnNw #1 #2 {#3} {#4} {#5} {#6} #7
1610 }

```

(End definition for `__prg_generate_conditional:nnNNnnnn` and others.)

`__prg_generate_p_form:wNNnnnnN` How to generate the various forms. Those functions take the following arguments: 1: junk, 2: `\cs_set:Npn` or similar, 3: p (for protected conditionals) or e, 4: function name, 5: signature, 6: parameter text, 7: replacement (possibly trimmed by `__prg_generate_conditional_fast:nw`), 8: `\use_i_ii:nnn` or `\use_i:nn` (for “fast” conditionals). Remember that the logic-returning functions expect two arguments to be present

after `\exp_end::` notice the construction of the different variants relies on this, and that the `TF` and `F` variants will be slightly faster than the `T` version. The `p` form is only valid for expandable tests, we check for that by making sure that the second argument is empty. For “fast” conditionals, `#7` has an extra `\if_...` To optimize a bit further we could replace `\exp_after:wN \use_ii:nnn` and similar by a single macro similar to `__prg_p_true:w`. The drawback is that if the `T` or `F` arguments are actually missing, the recovery from the runaway argument would not insert `\fi:` back, messing up nesting of conditionals.

```

1611 \cs_set_protected:Npn \__prg_generate_p_form:wNNnnnnN
1612   #1 \s__prg_stop #2#3#4#5#6#7#8
1613   {
1614     \if_meaning:w e #3
1615     \exp_after:wN \use_i:nn
1616   \else:
1617     \exp_after:wN \use_ii:nn
1618   \fi:
1619     {
1620       #8
1621       { \exp_args:Nc #2 { #4 _p: #5 } #6 }
1622       { { #7 \exp_end: \c_true_bool \c_false_bool } }
1623       { #7 \__prg_p_true:w \fi: \c_false_bool }
1624     }
1625     {
1626       \__kernel_msg_error:nxx { kernel } { protected-predicate }
1627       { \token_to_str:c { #4 _p: #5 } }
1628     }
1629   }
1630 \cs_set_protected:Npn \__prg_generate_T_form:wNNnnnnN
1631   #1 \s__prg_stop #2#3#4#5#6#7#8
1632   {
1633     #8
1634     { \exp_args:Nc #2 { #4 : #5 T } #6 }
1635     { { #7 \exp_end: \use:n \use_none:n } }
1636     { #7 \exp_after:wN \use_ii:nn \fi: \use_none:n }
1637   }
1638 \cs_set_protected:Npn \__prg_generate_F_form:wNNnnnnN
1639   #1 \s__prg_stop #2#3#4#5#6#7#8
1640   {
1641     #8
1642     { \exp_args:Nc #2 { #4 : #5 F } #6 }
1643     { { #7 \exp_end: { } } }
1644     { #7 \exp_after:wN \use_none:nn \fi: \use:n }
1645   }
1646 \cs_set_protected:Npn \__prg_generate_TF_form:wNNnnnnN
1647   #1 \s__prg_stop #2#3#4#5#6#7#8
1648   {
1649     #8
1650     { \exp_args:Nc #2 { #4 : #5 TF } #6 }
1651     { { #7 \exp_end: { } } }
1652     { #7 \exp_after:wN \use_ii:nnn \fi: \use_ii:nn }
1653   }
1654 \cs_set:Npn \__prg_p_true:w \fi: \c_false_bool { \fi: \c_true_bool }

```

(End definition for `__prg_generate_p_form:wNNnnnnN` and others.)

`\prg_set_eq_conditional:NNn` The setting-equal functions. Split both functions and feed $\{\langle name_1 \rangle\}$ $\{\langle signature_1 \rangle\}$
`\prg_new_eq_conditional:NNn` $\langle boolean_1 \rangle$ $\{\langle name_2 \rangle\}$ $\{\langle signature_2 \rangle\}$ $\langle boolean_2 \rangle$ $\langle copying\ function \rangle$ $\langle conditions \rangle$, `\q__prg_set_eq_conditional:NNNn` `prg_recursion_tail` , `\q__prg_recursion_stop` to a first auxiliary.

```

1655 \cs_set_protected:Npn \prg_set_eq_conditional:NNn
1656   { \__prg_set_eq_conditional:NNNn \cs_set_eq:cc }
1657 \cs_set_protected:Npn \prg_new_eq_conditional:NNn
1658   { \__prg_set_eq_conditional:NNNn \cs_new_eq:cc }
1659 \cs_set_protected:Npn \__prg_set_eq_conditional:NNNn #1#2#3#4
1660   {
1661     \use:x
1662     {
1663       \exp_not:N \__prg_set_eq_conditional:nnNnnNWw
1664       \cs_split_function:N #2
1665       \cs_split_function:N #3
1666       \exp_not:N #1
1667       \tl_to_str:n {#4}
1668       \exp_not:n { , \q__prg_recursion_tail , \q__prg_recursion_stop }
1669     }
1670   }

```

(End definition for `\prg_set_eq_conditional:NNn`, `\prg_new_eq_conditional:NNn`, and `__prg_set_eq_conditional:NNNn`. These functions are documented on page 107.)

`__prg_set_eq_conditional:nnNnnNWw`
`__prg_set_eq_conditional_loop:nnnnNW`
`__prg_set_eq_conditional_p_form:nnn`
`__prg_set_eq_conditional_TF_form:nnn`
`__prg_set_eq_conditional_T_form:nnn`
`__prg_set_eq_conditional_F_form:nnn`

Split the function to be defined, and setup a manual clist loop over argument #6 of the first auxiliary. The second auxiliary receives twice three arguments coming from splitting the function to be defined and the function to copy. Make sure that both functions contained a colon, otherwise we don't know how to build conditionals, hence abort. Call the looping macro, with arguments $\{\langle name_1 \rangle\}$ $\{\langle signature_1 \rangle\}$ $\{\langle name_2 \rangle\}$ $\{\langle signature_2 \rangle\}$ $\langle copying\ function \rangle$ and followed by the comma list. At each step in the loop, make sure that the conditional form we copy is defined, and copy it, otherwise abort.

```

1671 \cs_set_protected:Npn \__prg_set_eq_conditional:nnNnnNWw #1#2#3#4#5#6
1672   {
1673     \if_meaning:w \c_false_bool #3
1674       \__kernel_msg_error:nnx { kernel } { missing-colon }
1675       { \token_to_str:c {#1} } }
1676     \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
1677     \fi:
1678     \if_meaning:w \c_false_bool #6
1679       \__kernel_msg_error:nnx { kernel } { missing-colon }
1680       { \token_to_str:c {#4} } }
1681     \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
1682     \fi:
1683     \__prg_set_eq_conditional_loop:nnnnNW {#1} {#2} {#4} {#5}
1684   }
1685 \cs_set_protected:Npn \__prg_set_eq_conditional_loop:nnnnNW #1#2#3#4#5#6 ,
1686   {
1687     \if_meaning:w \q__prg_recursion_tail #6
1688       \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
1689       \fi:
1690     \use:c { __prg_set_eq_conditional_ #6 _form:wNnnnn }
1691     \tl_if_empty:nF {#6}
1692     {
1693       \__kernel_msg_error:nnxx

```

```

1694         { kernel } { conditional-form-unknown }
1695         {#6} { \token_to_str:c { #1 : #2 } }
1696     }
1697     \use_none:nnnnnn
1698     \s__prg_stop
1699     #5 {#1} {#2} {#3} {#4}
1700     \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
1701 }
1702 \cs_set:Npn \__prg_set_eq_conditional_p_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1703 { #2 { #3 _p : #4 } { #5 _p : #6 } }
1704 \cs_set:Npn \__prg_set_eq_conditional_TF_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1705 { #2 { #3 : #4 TF } { #5 : #6 TF } }
1706 \cs_set:Npn \__prg_set_eq_conditional_T_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1707 { #2 { #3 : #4 T } { #5 : #6 T } }
1708 \cs_set:Npn \__prg_set_eq_conditional_F_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1709 { #2 { #3 : #4 F } { #5 : #6 F } }

```

(End definition for `__prg_set_eq_conditional:nnNnnNNw` and others.)

All that is left is to define the canonical boolean true and false. I think Michael originated the idea of expandable boolean tests. At first these were supposed to expand into either TT or TF to be tested using `\if:w` but this was later changed to 00 and 01, so they could be used in logical operations. Later again they were changed to being numerical constants with values of 1 for true and 0 for false. We need this from the get-go.

```

\c_true_bool Here are the canonical boolean values.
\c_false_bool
1710 \tex_chardef:D \c_true_bool = 1 ~
1711 \tex_chardef:D \c_false_bool = 0 ~

```

(End definition for `\c_true_bool` and `\c_false_bool`. These variables are documented on page 22.)

5.8 Dissecting a control sequence

```

1712 <@@=cs>

```

```

\__cs_count_signature:N \__cs_count_signature:N <function>

```

Splits the *<function>* into the *<name>* (i.e. the part before the colon) and the *<signature>* (i.e. after the colon). The *<number>* of tokens in the *<signature>* is then left in the input stream. If there was no *<signature>* then the result is the marker value `-1`.

```

\__cs_get_function_name:N * \__cs_get_function_name:N <function>

```

Splits the *<function>* into the *<name>* (i.e. the part before the colon) and the *<signature>* (i.e. after the colon). The *<name>* is then left in the input stream without the escape character present made up of tokens with category code 12 (other).

```

\__cs_get_function_signature:N * \__cs_get_function_signature:N <function>

```

Splits the *<function>* into the *<name>* (i.e. the part before the colon) and the *<signature>* (i.e. after the colon). The *<signature>* is then left in the input stream made up of tokens with category code 12 (other).

<code>__cs_tmp:w</code>	Function used for various short-term usages, for instance defining functions whose definition involves tokens which are hard to insert normally (spaces, characters with category other).
<code>\cs_to_str:N</code> <code>__cs_to_str:N</code> <code>__cs_to_str:w</code>	This converts a control sequence into the character string of its name, removing the leading escape character. This turns out to be a non-trivial matter as there are different cases:

- The usual case of a printable escape character;
- the case of a non-printable escape characters, e.g., when the value of the `\escapechar` is negative;
- when the escape character is a space.

One approach to solve this is to test how many tokens result from `\token_to_str:N \a`. If there are two tokens, then the escape character is printable, while if it is non-printable then only one is present.

However, there is an additional complication: the control sequence itself may start with a space. Clearly that should *not* be lost in the process of converting to a string. So the approach adopted is a little more intricate still. When the escape character is printable, `\token_to_str:N _` yields the escape character itself and a space. The character codes are different, thus the `\if:w` test is false, and TeX reads `__cs_to_str:N` after turning the following control sequence into a string; this auxiliary removes the escape character, and stops the expansion of the initial `\tex_romannumeral:D`. The second case is that the escape character is not printable. Then the `\if:w` test is unfinished after reading a the space from `\token_to_str:N _`, and the auxiliary `__cs_to_str:w` is expanded, feeding – as a second character for the test; the test is false, and TeX skips to `\fi:`, then performs `\token_to_str:N`, and stops the `\tex_romannumeral:D` with `\c_zero_int`. The last case is that the escape character is itself a space. In this case, the `\if:w` test is true, and the auxiliary `__cs_to_str:w` comes into play, inserting `-\int_value:w`, which expands `\c_zero_int` to the character 0. The initial `\tex_romannumeral:D` then sees 0, which is not a terminated number, followed by the escape character, a space, which is removed, terminating the expansion of `\tex_romannumeral:D`. In all three cases, `\cs_to_str:N` takes two expansion steps to be fully expanded.

```

1713 \cs_set:Npn \cs_to_str:N
1714 {

```

We implement the expansion scheme using `\tex_romannumeral:D` terminating it with `\c_zero_int` rather than using `\exp:w` and `\exp_end:` as we normally do. The reason is that the code heavily depends on terminating the expansion with `\c_zero_int` so we make this dependency explicit.

```

1715     \tex_romannumeral:D
1716     \if:w \token_to_str:N \ \__cs_to_str:w \fi:
1717     \exp_after:wN \__cs_to_str:N \token_to_str:N
1718 }
1719 \cs_set:Npn \__cs_to_str:N #1 { \c_zero_int }
1720 \cs_set:Npn \__cs_to_str:w #1 \__cs_to_str:N
1721 { - \int_value:w \fi: \exp_after:wN \c_zero_int }

```

If speed is a concern we could use `\csstring` in LuaTeX. For the empty csname that primitive gives an empty result while the current `\cs_to_str:N` gives incorrect results in all engines (this is impossible to fix without huge performance hit).

(End definition for `\cs_to_str:N`, `__cs_to_str:N`, and `__cs_to_str:w`. This function is documented on page 17.)

`\cs_split_function:N`

This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean $\langle true \rangle$ or $\langle false \rangle$ is returned with $\langle true \rangle$ for when there is a colon in the function and $\langle false \rangle$ if there is not.

We cannot use `:` directly as it has the wrong category code so an `x`-type expansion is used to force the conversion.

First ensure that we actually get a properly evaluated string by expanding `\cs_to_str:N` twice. If the function contained a colon, the auxiliary takes as `#1` the function name, delimited by the first colon, then the signature `#2`, delimited by `\s__cs_mark`, then `\c_true_bool` as `#3`, and `#4` cleans up until `\s__cs_stop`. Otherwise, the `#1` contains the function name and `\s__cs_mark` `\c_true_bool`, `#2` is empty, `#3` is `\c_false_bool`, and `#4` cleans up. The second auxiliary trims the trailing `\s__cs_mark` from the function name if present (that is, if the original function had no colon).

```

1722 \cs_set_protected:Npn \__cs_tmp:w #1
1723 {
1724   \cs_set:Npn \cs_split_function:N ##1
1725   {
1726     \exp_after:wN \exp_after:wN \exp_after:wN
1727     \__cs_split_function_auxi:w
1728     \cs_to_str:N ##1 \s__cs_mark \c_true_bool
1729     #1 \s__cs_mark \c_false_bool \s__cs_stop
1730   }
1731   \cs_set:Npn \__cs_split_function_auxi:w
1732     ##1 #1 ##2 \s__cs_mark ##3##4 \s__cs_stop
1733   { \__cs_split_function_auxii:w ##1 \s__cs_mark \s__cs_stop {##2} ##3 }
1734   \cs_set:Npn \__cs_split_function_auxii:w ##1 \s__cs_mark ##2 \s__cs_stop
1735   { {##1} }
1736 }
1737 \exp_after:wN \__cs_tmp:w \token_to_str:N :
```

(End definition for `\cs_split_function:N`, `__cs_split_function_auxi:w`, and `__cs_split_function_auxii:w`. This function is documented on page 17.)

5.9 Exist or free

A control sequence is said to *exist* (to be used) if has an entry in the hash table and its meaning is different from the primitive `\relax` token. A control sequence is said to be *free* (to be defined) if it does not already exist.

`\cs_if_exist_p:N`

`\cs_if_exist_p:c`

`\cs_if_exist:NTF`

`\cs_if_exist:cTF`

Two versions for checking existence. For the `N` form we firstly check for `\scan_stop:` and then if it is in the hash table. There is no problem when inputting something like `\else:` or `\fi:` as \TeX will only ever skip input in case the token tested against is `\scan_stop:`.

```

1738 \prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
1739 {
1740   \if_meaning:w #1 \scan_stop:
1741   \prg_return_false:
1742   \else:
1743     \if_cs_exist:N #1
1744     \prg_return_true:
1745     \else:
```

```

1746     \prg_return_false:
1747     \fi:
1748     \fi:
1749 }

```

For the `c` form we firstly check if it is in the hash table and then for `\scan_stop:` so that we do not add it to the hash table unless it was already there. Here we have to be careful as the text to be skipped if the first test is false may contain tokens that disturb the scanner. Therefore, we ensure that the second test is performed after the first one has concluded completely.

```

1750 \prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
1751 {
1752     \if_cs_exist:w #1 \cs_end:
1753     \exp_after:wN \use_i:nn
1754     \else:
1755     \exp_after:wN \use_ii:nn
1756     \fi:
1757     {
1758     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1759     \prg_return_false:
1760     \else:
1761     \prg_return_true:
1762     \fi:
1763     }
1764     \prg_return_false:
1765 }

```

(End definition for `\cs_if_exist:NTF`. This function is documented on page 22.)

`\cs_if_free_p:N`

The logical reversal of the above.

`\cs_if_free_p:c`

`\cs_if_free:NTF`

`\cs_if_free:cTF`

```

1766 \prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
1767 {
1768     \if_meaning:w #1 \scan_stop:
1769     \prg_return_true:
1770     \else:
1771     \if_cs_exist:N #1
1772     \prg_return_false:
1773     \else:
1774     \prg_return_true:
1775     \fi:
1776     \fi:
1777 }
1778 \prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
1779 {
1780     \if_cs_exist:w #1 \cs_end:
1781     \exp_after:wN \use_i:nn
1782     \else:
1783     \exp_after:wN \use_ii:nn
1784     \fi:
1785     {
1786     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1787     \prg_return_true:
1788     \else:
1789     \prg_return_false:

```

```

1790     \fi:
1791   }
1792   { \prg_return_true: }
1793 }

```

(End definition for `\cs_if_free:NTF`. This function is documented on page 22.)

`\cs_if_exist_use:N` The `\cs_if_exist_use:...` functions cannot be implemented as conditionals because the true branch must leave both the control sequence itself and the true code in the input stream. For the `c` variants, we are careful not to put the control sequence in the hash table if it does not exist. In LuaTeX we could use the `\lastnamedcs` primitive.

```

1794 \cs_set:Npn \cs_if_exist_use:NTF #1#2
1795 { \cs_if_exist:NTF #1 { #1 #2 } }
1796 \cs_set:Npn \cs_if_exist_use:NF #1
1797 { \cs_if_exist:NTF #1 { #1 } }
1798 \cs_set:Npn \cs_if_exist_use:NT #1 #2
1799 { \cs_if_exist:NTF #1 { #1 #2 } { } }
1800 \cs_set:Npn \cs_if_exist_use:N #1
1801 { \cs_if_exist:NTF #1 { #1 } { } }
1802 \cs_set:Npn \cs_if_exist_use:cTF #1#2
1803 { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } }
1804 \cs_set:Npn \cs_if_exist_use:cF #1
1805 { \cs_if_exist:cTF {#1} { \use:c {#1} } }
1806 \cs_set:Npn \cs_if_exist_use:cT #1#2
1807 { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } }
1808 \cs_set:Npn \cs_if_exist_use:c #1
1809 { \cs_if_exist:cTF {#1} { \use:c {#1} } { } }

```

(End definition for `\cs_if_exist_use:NTF`. This function is documented on page 16.)

5.10 Preliminaries for new functions

We provide two kinds of functions that can be used to define control sequences. On the one hand we have functions that check if their argument doesn't already exist, they are called `\..._new`. The second type of defining functions doesn't check if the argument is already defined.

Before we can define them, we need some auxiliary macros that allow us to generate error messages. The next few definitions here are only temporary, they will be redefined later on.

`__kernel_msg_error:nxxx` If an internal error occurs before L^AT_EX3 has loaded `l3msg` then the code should issue a usable if terse error message and halt. This can only happen if a coding error is made by the team, so this is a reasonable response. Setting the `\newlinechar` is needed, to turn `^^J` into a proper line break in plain T_EX.

```

1810 \cs_set_protected:Npn \__kernel_msg_error:nxxx #1#2#3#4
1811 {
1812   \tex_newlinechar:D = '\^^J \scan_stop:
1813   \tex_errmessage:D
1814   {
1815     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^^J
1816     Argh,~internal~LaTeX3~error! ^^J ^^J
1817     Module ~ #1 , ~ message~name~"#2": ^^J
1818     Arguments~'#3'~and~'#4' ^^J ^^J

```

```

1819         This~is~one~for~The~LaTeX3~Project:~bailing~out
1820     }
1821     \tex_end:D
1822 }
1823 \cs_set_protected:Npn \__kernel_msg_error:nxx #1#2#3
1824 { \__kernel_msg_error:nxxx {#1} {#2} {#3} { } }
1825 \cs_set_protected:Npn \__kernel_msg_error:nn #1#2
1826 { \__kernel_msg_error:nxxx {#1} {#2} { } { } }

```

(End definition for `__kernel_msg_error:nxxx`, `__kernel_msg_error:nxx`, and `__kernel_msg_error:nn`.)

`\msg_line_context:` Another one from `l3msg` which will be altered later.

```

1827 \cs_set:Npn \msg_line_context:
1828 { on~line~ \tex_the:D \tex_inputlineno:D }

```

(End definition for `\msg_line_context:`. This function is documented on page 153.)

`\iow_log:x` We define a routine to write only to the log file. And a similar one for writing to both
`\iow_term:x` the log file and the terminal. These will be redefined later by `l3io`.

```

1829 \cs_set_protected:Npn \iow_log:x
1830 { \tex_immediate:D \tex_write:D -1 }
1831 \cs_set_protected:Npn \iow_term:x
1832 { \tex_immediate:D \tex_write:D 16 }

```

(End definition for `\iow_log:n`. This function is documented on page 163.)

`__kernel_chk_if_free_cs:N` This command is called by `\cs_new_nopar:Npn` and `\cs_new_eq:NN` etc. to make sure
`__kernel_chk_if_free_cs:c` that the argument sequence is not already in use. If it is, an error is signalled. It checks
if `<csname>` is undefined or `\scan_stop:`. Otherwise an error message is issued. We have
to make sure we don't put the argument into the conditional processing since it may be
an `\if...` type function!

```

1833 \cs_set_protected:Npn \__kernel_chk_if_free_cs:N #1
1834 {
1835     \cs_if_free:NF #1
1836     {
1837         \__kernel_msg_error:nxxx { kernel } { command-already-defined }
1838         { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1839     }
1840 }
1841 \cs_set_protected:Npn \__kernel_chk_if_free_cs:c
1842 { \exp_args:Nc \__kernel_chk_if_free_cs:N }

```

(End definition for `__kernel_chk_if_free_cs:N`.)

5.11 Defining new functions

```

1843 \@@=cs

```

`\cs_new_nopar:Npn` Function which check that the control sequence is free before defining it.

```

\cs_new_nopar:Npx 1844 \cs_set:Npn \__cs_tmp:w #1#2
\cs_new:Npn       1845 {
\cs_new:Npx       1846     \cs_set_protected:Npn #1 ##1
\cs_new_protected_nopar:Npn 1847 {
\cs_new_protected_nopar:Npx 1848     \__kernel_chk_if_free_cs:N ##1
\cs_new_protected:Npn       1849     #2 ##1
\cs_new_protected:Npx

```

```

\__cs_tmp:w

```



```

1850     }
1851   }
1852   \__cs_tmp:w \cs_new_nopar:Npn          \cs_gset_nopar:Npn
1853   \__cs_tmp:w \cs_new_nopar:Npx          \cs_gset_nopar:Npx
1854   \__cs_tmp:w \cs_new:Npn                \cs_gset:Npn
1855   \__cs_tmp:w \cs_new:Npx                \cs_gset:Npx
1856   \__cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
1857   \__cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
1858   \__cs_tmp:w \cs_new_protected:Npn       \cs_gset_protected:Npn
1859   \__cs_tmp:w \cs_new_protected:Npx       \cs_gset_protected:Npx

```

(End definition for `\cs_new_nopar:Npn` and others. These functions are documented on page 11.)

`\cs_set_nopar:cpn` Like `\cs_set_nopar:Npn` and `\cs_new_nopar:Npn`, except that the first argument consists of the sequence of characters that should be used to form the name of the desired control sequence (the `c` stands for `csname` argument, see the expansion module). Global versions are also provided.

`\cs_set_nopar:cpn` $\langle string \rangle \langle rep-text \rangle$ turns $\langle string \rangle$ into a `csname` and then assigns $\langle rep-text \rangle$ to it by using `\cs_set_nopar:Npn`. This means that there might be a parameter string between the two arguments.

```

1860 \cs_set:Npn \__cs_tmp:w #1#2
1861   { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
1862 \__cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
1863 \__cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx
1864 \__cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
1865 \__cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
1866 \__cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
1867 \__cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx

```

(End definition for `\cs_set_nopar:Npn`. This function is documented on page 11.)

`\cs_set:cpn` Variants of the `\cs_set:Npn` versions which make a `csname` out of the first arguments.
`\cs_set:cpx` We may also do this globally.

```

1868 \__cs_tmp:w \cs_set:cpn \cs_set:Npn
1869 \__cs_tmp:w \cs_set:cpx \cs_set:Npx
1870 \__cs_tmp:w \cs_gset:cpn \cs_gset:Npn
1871 \__cs_tmp:w \cs_gset:cpx \cs_gset:Npx
1872 \__cs_tmp:w \cs_new:cpn \cs_new:Npn
1873 \__cs_tmp:w \cs_new:cpx \cs_new:Npx

```

(End definition for `\cs_set:Npn`. This function is documented on page 11.)

`\cs_set_protected_nopar:cpn` Variants of the `\cs_set_protected_nopar:Npn` versions which make a `csname` out of the first arguments. We may also do this globally.

```

1874 \__cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn
1875 \__cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx
1876 \__cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
1877 \__cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
1878 \__cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn
1879 \__cs_tmp:w \cs_new_protected_nopar:cpx \cs_new_protected_nopar:Npx

```

(End definition for `\cs_set_protected_nopar:Npn`. This function is documented on page 12.)

`\cs_set_protected:cpn`
`\cs_set_protected:cpx`
`\cs_gset_protected:cpn`
`\cs_gset_protected:cpx`
`\cs_new_protected:cpn`
`\cs_new_protected:cpx`

Variants of the `\cs_set_protected:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```

1880 \__cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn
1881 \__cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
1882 \__cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
1883 \__cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
1884 \__cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn
1885 \__cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx

```

(End definition for `\cs_set_protected:Npn`. This function is documented on page 11.)

5.12 Copying definitions

`\cs_set_eq:NN`
`\cs_set_eq:cN`
`\cs_set_eq:Nc`
`\cs_set_eq:cc`
`\cs_gset_eq:NN`
`\cs_gset_eq:cN`
`\cs_gset_eq:Nc`
`\cs_gset_eq:cc`
`\cs_new_eq:NN`
`\cs_new_eq:cN`
`\cs_new_eq:Nc`
`\cs_new_eq:cc`

These macros allow us to copy the definition of a control sequence to another control sequence.

The = sign allows us to define funny char tokens like = itself or `_` with this function. For the definition of `\c_space_char{~}` to work we need the ~ after the =.

`\cs_set_eq:NN` is long to avoid problems with a literal argument of `\par`. While `\cs_new_eq:NN` will probably never be correct with a first argument of `\par`, define it long in order to throw an “already defined” error rather than “runaway argument”.

```

1886 \cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }
1887 \cs_new_protected:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
1888 \cs_new_protected:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
1889 \cs_new_protected:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
1890 \cs_new_protected:Npn \cs_gset_eq:NN { \tex_global:D \cs_set_eq:NN }
1891 \cs_new_protected:Npn \cs_gset_eq:Nc { \exp_args:NNc \cs_gset_eq:NN }
1892 \cs_new_protected:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }
1893 \cs_new_protected:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }
1894 \cs_new_protected:Npn \cs_new_eq:NN #1
1895 {
1896   \__kernel_chk_if_free_cs:N #1
1897   \tex_global:D \cs_set_eq:NN #1
1898 }
1899 \cs_new_protected:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }
1900 \cs_new_protected:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
1901 \cs_new_protected:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }

```

(End definition for `\cs_set_eq:NN`, `\cs_gset_eq:NN`, and `\cs_new_eq:NN`. These functions are documented on page 15.)

5.13 Undefined functions

`\cs_undefine:N`
`\cs_undefine:c`

The following function is used to free the main memory from the definition of some function that isn’t in use any longer. The c variant is careful not to add the control sequence to the hash table if it isn’t there yet, and it also avoids nesting TeX conditionals in case #1 is unbalanced in this matter.

```

1902 \cs_new_protected:Npn \cs_undefine:N #1
1903 { \cs_gset_eq:NN #1 \tex_undefined:D }
1904 \cs_new_protected:Npn \cs_undefine:c #1
1905 {
1906   \if_cs_exist:w #1 \cs_end:
1907     \exp_after:wN \use:n
1908   \else:

```

```

1909     \exp_after:wN \use_none:n
1910   \fi:
1911   { \cs_gset_eq:cN {#1} \tex_undefined:D }
1912 }

```

(End definition for `\cs_undefine:N`. This function is documented on page 15.)

5.14 Generating parameter text from argument count

`_kernel_cs_parm_from_arg_count:nnF`
`_cs_parm_from_arg_count_test:nnF`

L^AT_EX3 provides shorthands to define control sequences and conditionals with a simple parameter text, derived directly from the signature, or more generally from knowing the number of arguments, between 0 and 9. This function expands to its first argument, untouched, followed by a brace group containing the parameter text `{#1...#n}`, where n is the result of evaluating the second argument (as described in `\int_eval:n`). If the second argument gives a result outside the range $[0, 9]$, the third argument is returned instead, normally an error message. Some of the functions use here are not defined yet, but will be defined before this function is called.

```

1914 \cs_set_protected:Npn \_kernel_cs_parm_from_arg_count:nnF #1#2
1915 {
1916   \exp_args:Nx \_cs_parm_from_arg_count_test:nnF
1917   {
1918     \exp_after:wN \exp_not:n
1919     \if_case:w \int_eval:n {#2}
1920       { }
1921       \or: { ##1 }
1922       \or: { ##1##2 }
1923       \or: { ##1##2##3 }
1924       \or: { ##1##2##3##4 }
1925       \or: { ##1##2##3##4##5 }
1926       \or: { ##1##2##3##4##5##6 }
1927       \or: { ##1##2##3##4##5##6##7 }
1928       \or: { ##1##2##3##4##5##6##7##8 }
1929       \or: { ##1##2##3##4##5##6##7##8##9 }
1930       \else: { \c_false_bool }
1931     \fi:
1932   }
1933   {#1}
1934 }
1935 \cs_set_protected:Npn \_cs_parm_from_arg_count_test:nnF #1#2
1936 {
1937   \if_meaning:w \c_false_bool #1
1938     \exp_after:wN \use_ii:nn
1939   \else:
1940     \exp_after:wN \use_i:nn
1941   \fi:
1942   { #2 {#1} }
1943 }

```

(End definition for `_kernel_cs_parm_from_arg_count:nnF` and `_cs_parm_from_arg_count_test:nnF`.)

5.15 Defining functions from a given number of arguments

1944 `<@@=cs>`

`_cs_count_signature:N` Counting the number of tokens in the signature, *i.e.*, the number of arguments the function should take. Since this is not used in any time-critical function, we simply use `_cs_count_signature:c` `\tl_count:n` if there is a signature, otherwise `-1` arguments to signal an error. We need `_cs_count_signature:n` a variant form right away.

```

1945 \cs_new:Npn \_cs_count_signature:N #1
1946 { \exp_args:Nf \_cs_count_signature:n { \cs_split_function:N #1 } }
1947 \cs_new:Npn \_cs_count_signature:n #1
1948 { \int_eval:n { \_cs_count_signature:nnN #1 } }
1949 \cs_new:Npn \_cs_count_signature:nnN #1#2#3
1950 {
1951   \if_meaning:w \c_true_bool #3
1952     \tl_count:n {#2}
1953   \else:
1954     -1
1955   \fi:
1956 }
1957 \cs_new:Npn \_cs_count_signature:c
1958 { \exp_args:Nc \_cs_count_signature:N }
```

(End definition for `_cs_count_signature:N`, `_cs_count_signature:n`, and `_cs_count_signature:nnN`.)

`\cs_generate_from_arg_count:NNnn`
`\cs_generate_from_arg_count:cNnn`
`\cs_generate_from_arg_count:Ncnn`

We provide a constructor function for defining functions with a given number of arguments. For this we need to choose the correct parameter text and then use that when defining. Since \TeX supports from zero to nine arguments, we use a simple switch to choose the correct parameter text, ensuring the result is returned after finishing the conditional. If it is not between zero and nine, we throw an error.

1: function to define, 2: with what to define it, 3: the number of args it requires and 4: the replacement text

```

1959 \cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
1960 {
1961   \_kernel_cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
1962   {
1963     \_kernel_msg_error:nnxx { kernel } { bad-number-of-arguments }
1964     { \token_to_str:N #1 } { \int_eval:n {#3} }
1965     \use_none:n
1966   }
1967   {#4}
1968 }
```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```

1969 \cs_new_protected:Npn \cs_generate_from_arg_count:cNnn
1970 { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
1971 \cs_new_protected:Npn \cs_generate_from_arg_count:Ncnn
1972 { \exp_args:NNc \cs_generate_from_arg_count:NNnn }
```

(End definition for `\cs_generate_from_arg_count:NNnn`. This function is documented on page 14.)

5.16 Using the signature to define functions

1973 <@@=cs>

We can now combine some of the tools we have to provide a simple interface for defining functions, where the number of arguments is read from the signature. For instance, `\cs_set:Nn \foo_bar:nn {#1,#2}`.

We want to define `\cs_set:Nn` as

```

\cs_set:Nn
\cs_set:Nx
\cs_set_nopar:Nn
\cs_set_nopar:Nx
\cs_set_protected:Nn
\cs_set_protected:Nx
\cs_set_protected_nopar:Nn
\cs_set_protected_nopar:Nx
\cs_gset:Nn
\cs_gset:Nx
\cs_gset_nopar:Nn
\cs_gset_nopar:Nx
\cs_gset_protected:Nn
\cs_gset_protected:Nx
\cs_new:Nn
\cs_new:Nx
\cs_new_nopar:Nn
\cs_new_nopar:Nx
\cs_new_protected:Nn
\cs_new_protected:Nx
\cs_new_protected_nopar:Nn
\cs_new_protected_nopar:Nx

```

In short, to define `\cs_set:Nn` we need just use `\cs_set:Npn`, everything else is the same for each variant. Therefore, we can make it simpler by temporarily defining a function to do this for us.

```

1974 \cs_set:Npn \__cs_tmp:w #1#2#3
1975 {
1976   \cs_new_protected:cpx { cs_ #1 : #2 }
1977   {
1978     \exp_not:N \__cs_generate_from_signature:NNn
1979     \exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
1980   }
1981 }
1982 \cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
1983 {
1984   \use:x
1985   {
1986     \__cs_generate_from_signature:nnNNNn
1987     \cs_split_function:N #2
1988   }
1989   #1 #2
1990 }
1991 \cs_new_protected:Npn \__cs_generate_from_signature:nnNNNn #1#2#3#4#5#6
1992 {
1993   \bool_if:NTF #3
1994   {
1995     \str_if_eq:eeF { }
1996     { \tl_map_function:nN {#2} \__cs_generate_from_signature:n }
1997     {
1998       \__kernel_msg_error:nnx { kernel } { non-base-function }
1999       { \token_to_str:N #5 }
2000     }
2001     \cs_generate_from_arg_count:NNnn
2002     #5 #4 { \tl_count:n {#2} } {#6}
2003   }
2004   {
2005     \__kernel_msg_error:nnx { kernel } { missing-colon }
2006     { \token_to_str:N #5 }
2007   }
2008 }
2009 \cs_new:Npn \__cs_generate_from_signature:n #1

```

```

2010 {
2011     \if:w n #1 \else: \if:w N #1 \else:
2012     \if:w T #1 \else: \if:w F #1 \else: #1 \fi: \fi: \fi: \fi:
2013 }

```

Then we define the 24 variants beginning with N.

```

2014 \__cs_tmp:w { set } { Nn } { Npn }
2015 \__cs_tmp:w { set } { Nx } { Npx }
2016 \__cs_tmp:w { set_nopar } { Nn } { Npn }
2017 \__cs_tmp:w { set_nopar } { Nx } { Npx }
2018 \__cs_tmp:w { set_protected } { Nn } { Npn }
2019 \__cs_tmp:w { set_protected } { Nx } { Npx }
2020 \__cs_tmp:w { set_protected_nopar } { Nn } { Npn }
2021 \__cs_tmp:w { set_protected_nopar } { Nx } { Npx }
2022 \__cs_tmp:w { gset } { Nn } { Npn }
2023 \__cs_tmp:w { gset } { Nx } { Npx }
2024 \__cs_tmp:w { gset_nopar } { Nn } { Npn }
2025 \__cs_tmp:w { gset_nopar } { Nx } { Npx }
2026 \__cs_tmp:w { gset_protected } { Nn } { Npn }
2027 \__cs_tmp:w { gset_protected } { Nx } { Npx }
2028 \__cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
2029 \__cs_tmp:w { gset_protected_nopar } { Nx } { Npx }
2030 \__cs_tmp:w { new } { Nn } { Npn }
2031 \__cs_tmp:w { new } { Nx } { Npx }
2032 \__cs_tmp:w { new_nopar } { Nn } { Npn }
2033 \__cs_tmp:w { new_nopar } { Nx } { Npx }
2034 \__cs_tmp:w { new_protected } { Nn } { Npn }
2035 \__cs_tmp:w { new_protected } { Nx } { Npx }
2036 \__cs_tmp:w { new_protected_nopar } { Nn } { Npn }
2037 \__cs_tmp:w { new_protected_nopar } { Nx } { Npx }

```

(End definition for \cs_set:Nn and others. These functions are documented on page 13.)

\cs_set:cn The 24 c variants simply use \exp_args:Nc.

\cs_set:cx 2038 \cs_set:Npn __cs_tmp:w #1#2

\cs_set_nopar:cn 2039 {

\cs_set_nopar:cx 2040 \cs_new_protected:cpx { cs_ #1 : c #2 }

\cs_set_protected:cn 2041 {

\cs_set_protected:cx 2042 \exp_not:N \exp_args:Nc

\cs_set_protected_nopar:cn 2043 \exp_after:wN \exp_not:N \cs:w cs_ #1 : N #2 \cs_end:

\cs_set_protected_nopar:cx 2044 }

\cs_gset:cn 2045 }

\cs_gset:cx 2046 __cs_tmp:w { set } { n }

\cs_gset_nopar:cn 2047 __cs_tmp:w { set } { x }

\cs_gset_nopar:cx 2048 __cs_tmp:w { set_nopar } { n }

\cs_gset_protected:cn 2049 __cs_tmp:w { set_nopar } { x }

\cs_gset_protected:cx 2050 __cs_tmp:w { set_protected } { n }

\cs_gset_protected_nopar:cn 2051 __cs_tmp:w { set_protected } { x }

\cs_gset_protected_nopar:cx 2052 __cs_tmp:w { set_protected_nopar } { n }

\cs_new:cn 2053 __cs_tmp:w { set_protected_nopar } { x }

\cs_new:cx 2054 __cs_tmp:w { gset } { n }

\cs_new_nopar:cn 2055 __cs_tmp:w { gset } { x }

\cs_new_nopar:cx 2056 __cs_tmp:w { gset_nopar } { n }

\cs_new_protected:cn 2057 __cs_tmp:w { gset_nopar } { x }

\cs_new_protected:cx 2058 __cs_tmp:w { gset_protected } { n }

\cs_new_protected_nopar:cn

\cs_new_protected_nopar:cx

```

2059 \__cs_tmp:w { gset_protected } { x }
2060 \__cs_tmp:w { gset_protected_nopar } { n }
2061 \__cs_tmp:w { gset_protected_nopar } { x }
2062 \__cs_tmp:w { new } { n }
2063 \__cs_tmp:w { new } { x }
2064 \__cs_tmp:w { new_nopar } { n }
2065 \__cs_tmp:w { new_nopar } { x }
2066 \__cs_tmp:w { new_protected } { n }
2067 \__cs_tmp:w { new_protected } { x }
2068 \__cs_tmp:w { new_protected_nopar } { n }
2069 \__cs_tmp:w { new_protected_nopar } { x }

```

(End definition for \cs_set:Nn. This function is documented on page 13.)

5.17 Checking control sequence equality

\cs_if_eq_p:NN Check if two control sequences are identical.

```

\cs_if_eq_p:cN 2070 \prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
\cs_if_eq_p:Nc 2071 {
\cs_if_eq_p:cc 2072   \if_meaning:w #1#2
\cs_if_eq:NNTF 2073   \prg_return_true: \else: \prg_return_false: \fi:
\cs_if_eq:cNTF 2074 }
\cs_if_eq:NcTF 2075 \cs_new:Npn \cs_if_eq_p:cN { \exp_args:Nc \cs_if_eq_p:NN }
\cs_if_eq:NcTF 2076 \cs_new:Npn \cs_if_eq:cNTF { \exp_args:Nc \cs_if_eq:NNTF }
\cs_if_eq:ccTF 2077 \cs_new:Npn \cs_if_eq:cNT { \exp_args:Nc \cs_if_eq:NNT }
\cs_if_eq:ccTF 2078 \cs_new:Npn \cs_if_eq:cNF { \exp_args:Nc \cs_if_eq:NNF }
\cs_if_eq:ccTF 2079 \cs_new:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
\cs_if_eq:ccTF 2080 \cs_new:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
\cs_if_eq:ccTF 2081 \cs_new:Npn \cs_if_eq:NcT { \exp_args:NNc \cs_if_eq:NNT }
\cs_if_eq:ccTF 2082 \cs_new:Npn \cs_if_eq:NcF { \exp_args:NNc \cs_if_eq:NNF }
\cs_if_eq:ccTF 2083 \cs_new:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
\cs_if_eq:ccTF 2084 \cs_new:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
\cs_if_eq:ccTF 2085 \cs_new:Npn \cs_if_eq:ccT { \exp_args:Ncc \cs_if_eq:NNT }
\cs_if_eq:ccTF 2086 \cs_new:Npn \cs_if_eq:ccF { \exp_args:Ncc \cs_if_eq:NNF }

```

(End definition for \cs_if_eq:NNTF. This function is documented on page 22.)

5.18 Diagnostic functions

```

2087 <@@=kernel>
\__kernel_chk_defined:NT Error if the variable #1 is not defined.
2088 \cs_new_protected:Npn \__kernel_chk_defined:NT #1#2
2089 {
2090   \cs_if_exist:NTF #1
2091   {#2}
2092   {
2093     \__kernel_msg_error:nxx { kernel } { variable-not-defined }
2094     { \token_to_str:N #1 }
2095   }
2096 }

```

(End definition for __kernel_chk_defined:NT.)

Simply using the `\showthe` primitive does not allow for line-wrapping, so instead use `\tl_show:n` and `\tl_log:n` (defined in `l3tl` and that performs line-wrapping). This displays `>~⟨variable⟩=⟨value⟩`. We expand the value before-hand as otherwise some integers (such as `\currentgrouplevel` or `\currentgrouptype`) altered by the line-wrapping code would show wrong values.

```

\__kernel_register_show:N
\__kernel_register_show:c
\__kernel_register_log:N
\__kernel_register_log:c
  \__kernel_register_show_aux:NN
  \__kernel_register_show_aux:nNN
2097 \cs_new_protected:Npn \__kernel_register_show:N
2098   { \__kernel_register_show_aux:NN \tl_show:n }
2099 \cs_new_protected:Npn \__kernel_register_show:c
2100   { \exp_args:Nc \__kernel_register_show:N }
2101 \cs_new_protected:Npn \__kernel_register_log:N
2102   { \__kernel_register_show_aux:NN \tl_log:n }
2103 \cs_new_protected:Npn \__kernel_register_log:c
2104   { \exp_args:Nc \__kernel_register_log:N }
2105 \cs_new_protected:Npn \__kernel_register_show_aux:NN #1#2
2106   {
2107     \__kernel_chk_defined:NT #2
2108     {
2109       \exp_args:No \__kernel_register_show_aux:nNN
2110       { \tex_the:D #2 } #2 #1
2111     }
2112   }
2113 \cs_new_protected:Npn \__kernel_register_show_aux:nNN #1#2#3
2114   { \exp_args:No #3 { \token_to_str:N #2 = #1 } }

```

(End definition for `__kernel_register_show:N` and others.)

`\cs_show:N` Some control sequences have a very long name or meaning. Thus, simply using TeX's primitive `\show` could lead to overlong lines. The output of this primitive is mimicked to some extent, then the re-built string is given to `\tl_show:n` or `\tl_log:n` for line-wrapping. We must expand the meaning before passing it to the wrapping code as otherwise we would wrongly see the definitions that are in place there. To get correct escape characters, set the `\escapechar` in a group; this also localizes the assignment performed by x-expansion. The `\cs_show:c` and `\cs_log:c` commands convert their argument to a control sequence within a group to avoid showing `\relax` for undefined control sequences.

```

\cs_show:N
\cs_show:c
\cs_log:N
\cs_log:c
\__kernel_show:NN
2115 \cs_new_protected:Npn \cs_show:N { \__kernel_show:NN \tl_show:n }
2116 \cs_new_protected:Npn \cs_show:c
2117   { \group_begin: \exp_args:NNc \group_end: \cs_show:N }
2118 \cs_new_protected:Npn \cs_log:N { \__kernel_show:NN \tl_log:n }
2119 \cs_new_protected:Npn \cs_log:c
2120   { \group_begin: \exp_args:NNc \group_end: \cs_log:N }
2121 \cs_new_protected:Npn \__kernel_show:NN #1#2
2122   {
2123     \group_begin:
2124       \int_set:Nn \tex_escapechar:D { '\ }
2125       \exp_args:NNx
2126       \group_end:
2127       #1 { \token_to_str:N #2 = \cs_meaning:N #2 }
2128   }

```

(End definition for `\cs_show:N`, `\cs_log:N`, and `__kernel_show:NN`. These functions are documented on page 16.)

5.19 Decomposing a macro definition

`\cs_prefix_spec:N` We sometimes want to test if a control sequence can be expanded to reveal a hidden value.
`\cs_argument_spec:N` However, we cannot just expand the macro blindly as it may have arguments and none
`\cs_replacement_spec:N` might be present. Therefore we define these functions to pick either the prefix(es), the
`__kernel_prefix_arg_replacement:wN` argument specification, or the replacement text from a macro. All of this information is
returned as characters with catcode 12. If the token in question isn't a macro, the token
`\scan_stop:` is returned instead.

```

2129 \use:x
2130 {
2131   \exp_not:n { \cs_new:Npn \__kernel_prefix_arg_replacement:wN #1 }
2132   \tl_to_str:n { macro : } \exp_not:n { #2 -> #3 \s__kernel_stop #4 }
2133 }
2134 { #4 {#1} {#2} {#3} }
2135 \cs_new:Npn \cs_prefix_spec:N #1
2136 {
2137   \token_if_macro:NTF #1
2138   {
2139     \exp_after:wN \__kernel_prefix_arg_replacement:wN
2140     \token_to_meaning:N #1 \s__kernel_stop \use_i:nnn
2141   }
2142   { \scan_stop: }
2143 }
2144 \cs_new:Npn \cs_argument_spec:N #1
2145 {
2146   \token_if_macro:NTF #1
2147   {
2148     \exp_after:wN \__kernel_prefix_arg_replacement:wN
2149     \token_to_meaning:N #1 \s__kernel_stop \use_ii:nnn
2150   }
2151   { \scan_stop: }
2152 }
2153 \cs_new:Npn \cs_replacement_spec:N #1
2154 {
2155   \token_if_macro:NTF #1
2156   {
2157     \exp_after:wN \__kernel_prefix_arg_replacement:wN
2158     \token_to_meaning:N #1 \s__kernel_stop \use_iii:nnn
2159   }
2160   { \scan_stop: }
2161 }

```

(End definition for `\cs_prefix_spec:N` and others. These functions are documented on page 18.)

5.20 Doing nothing functions

`\prg_do_nothing:` This does not fit anywhere else!

```

2162 \cs_new:Npn \prg_do_nothing: { }

```

(End definition for `\prg_do_nothing:`. This function is documented on page 9.)

5.21 Breaking out of mapping functions

2163 `<@@=prg>`

`\prg_break_point:Nn` In inline mappings, the nesting level must be reset at the end of the mapping, even when the user decides to break out. This is done by putting the code that must be performed as an argument of `__prg_break_point:Nn`. The breaking functions are then defined to jump to that point and perform the argument of `__prg_break_point:Nn`, before the user's code (if any). There is a check that we close the correct loop, otherwise we continue breaking.

```
2164 \cs_new_eq:NN \prg_break_point:Nn \use_ii:nn
2165 \cs_new:Npn \prg_map_break:Nn #1#2#3 \prg_break_point:Nn #4#5
2166 {
2167   #5
2168   \if_meaning:w #1 #4
2169     \exp_after:wN \use_iii:nnn
2170   \fi:
2171   \prg_map_break:Nn #1 {#2}
2172 }
```

(End definition for `\prg_break_point:Nn` and `\prg_map_break:Nn`. These functions are documented on page 114.)

`\prg_break_point:` Very simple analogues of `\prg_break_point:Nn` and `\prg_map_break:Nn`, for use in fast short-term recursions which are not mappings, do not need to support nesting, and in `\prg_break:` which nothing has to be done at the end of the loop.

```
2173 \cs_new_eq:NN \prg_break_point: \prg_do_nothing:
2174 \cs_new:Npn \prg_break: #1 \prg_break_point: { }
2175 \cs_new:Npn \prg_break:n #1#2 \prg_break_point: {#1}
```

(End definition for `\prg_break_point:`, `\prg_break:`, and `\prg_break:n`. These functions are documented on page 115.)

5.22 Starting a paragraph

`\mode_leave_vertical:` The approach here is different to that used by L^AT_EX 2_ε or plain T_EX, which unbox a void box to force horizontal mode. That inserts the `\everypar` tokens *before* the re-inserted unboxing tokens. The approach here uses either the `\quitvmode` primitive or the equivalent protected macro. In vertical mode, the `\indent` primitive is inserted: this will switch to horizontal mode and insert `\everypar` tokens and nothing else. Unlike the L^AT_EX 2_ε version, the availability of ε-T_EX means using a mode test can be done at for example the start of an `\halign`.

```
2176 \cs_new_protected:Npn \mode_leave_vertical:
2177 {
2178   \if_mode_vertical:
2179     \exp_after:wN \tex_indent:D
2180   \fi:
2181 }
```

(End definition for `\mode_leave_vertical:`. This function is documented on page 24.)

2182 `</package>`

6 l3expan implementation

2183 `*package`

2184 `\@@=exp`

`\l__exp_internal_tl` The `\exp_` module has its private variable to temporarily store the result of `x`-type argument expansion. This is done to avoid interference with other functions using temporary variables.

(End definition for `\l__exp_internal_tl`.)

`\exp_after:wN` These are defined in `l3basics`, as they are needed “early”. This is just a reminder of that fact!

`\exp_not:N`

`\exp_not:n`

(End definition for `\exp_after:wN`, `\exp_not:N`, and `\exp_not:n`. These functions are documented on page 33.)

6.1 General expansion

In this section a general mechanism for defining functions that handle arguments is defined. These general expansion functions are expandable unless `x` is used. (Any version of `x` is going to have to use one of the L^AT_EX3 names for `\cs_set:Npx` at some point, and so is never going to be expandable.)

The definition of expansion functions with this technique happens in section 6.8. In section 6.2 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

`\l__exp_internal_tl` This scratch token list variable is defined in `l3basics`.

(End definition for `\l__exp_internal_tl`.)

This code uses internal functions with names that start with `\::` to perform the expansions. All macros are `long` since the tokens undergoing expansion may be arbitrary user input.

An argument manipulator `\::⟨Z⟩` always has signature `#1\:::#2#3` where `#1` holds the remaining argument manipulations to be performed, `\:::` serves as an end marker for the list of manipulations, `#2` is the carried over result of the previous expansion steps and `#3` is the argument about to be processed. One exception to this rule is `\::p`, which has to grab an argument delimited by a left brace.

`__exp_arg_next:nnn` `#1` is the result of an expansion step, `#2` is the remaining argument manipulations and `#3` is the current result of the expansion chain. This auxiliary function moves `#1` back after `#3` in the input stream and checks if any expansion is left to be done by calling `#2`. In by far the most cases we need to add a set of braces to the result of an argument manipulation so it is more effective to do it directly here. Actually, so far only the `c` of the final argument manipulation variants does not require a set of braces.

2185 `\cs_new:Npn __exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }`

2186 `\cs_new:Npn __exp_arg_next:Nnn #1#2#3 { #2 \::: { #3 #1 } }`

(End definition for `__exp_arg_next:nnn` and `__exp_arg_next:Nnn`.)

`\:::` The end marker is just another name for the identity function.

2187 `\cs_new:Npn \::: #1 {#1}`

(End definition for `\:::`. This function is documented on page 37.)

\::n This function is used to skip an argument that doesn't need to be expanded.

```
2188 \cs_new:Npn \::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(End definition for \::n. This function is documented on page 37.)

\::N This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```
2189 \cs_new:Npn \::N #1 \::: #2#3 { #1 \::: {#2#3} }
```

(End definition for \::N. This function is documented on page 37.)

\::p This function is used to skip an argument that is delimited by a left brace and doesn't need to be expanded. It is not wrapped in braces in the result.

```
2190 \cs_new:Npn \::p #1 \::: #2#3# { #1 \::: {#2#3} }
```

(End definition for \::p. This function is documented on page 37.)

\::c This function is used to skip an argument that is turned into a control sequence without expansion.

```
2191 \cs_new:Npn \::c #1 \::: #2#3
2192 { \exp_after:wN \__exp_arg_next:Nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(End definition for \::c. This function is documented on page 37.)

\::o This function is used to expand an argument once.

```
2193 \cs_new:Npn \::o #1 \::: #2#3
2194 { \exp_after:wN \__exp_arg_next:nnn \exp_after:wN {#3} {#1} {#2} }
```

(End definition for \::o. This function is documented on page 37.)

\::e With the `\expanded` primitive available, just expand. Otherwise defer to `\exp_args:Ne` implemented later.

```
2195 \cs_if_exist:NTF \tex_expanded:D
2196 {
2197   \cs_new:Npn \::e #1 \::: #2#3
2198   { \tex_expanded:D { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } } }
2199 }
2200 {
2201   \cs_new:Npn \::e #1 \::: #2#3
2202   { \exp_args:Ne \__exp_arg_next:nnn {#3} {#1} {#2} }
2203 }
```

(End definition for \::e. This function is documented on page 37.)

\::f This function is used to expand a token list until the first unexpandable token is found. This is achieved through `\exp:w \exp_end_continue_f:w` that expands everything in its way following it. This scanning procedure is terminated once the expansion hits something non-expandable (if that is a space it is removed). We introduce `\exp_stop_f:` to mark such an end-of-expansion marker. For example, f-expanding `\cs_set_eq:Nc \aaa { b \l_tmpa_tl b }` where `\l_tmpa_tl` contains the characters `lur` gives `\tex_let:D \aaa = \blurb` which then turns out to start with the non-expandable token `\tex_let:D`. Since the expansion of `\exp:w \exp_end_continue_f:w` is empty, we wind up with a fully expanded list, only `TEX` has not tried to execute any of

the non-expandable tokens. This is what differentiates this function from the `x` argument type.

```

2204 \cs_new:Npn \::f #1 \::: #2#3
2205 {
2206   \exp_after:wN \__exp_arg_next:nnn
2207   \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2208   {#1} {#2}
2209 }
2210 \use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }
```

(End definition for `\::f` and `\exp_stop_f:`. These functions are documented on page 37.)

\::x This function is used to expand an argument fully. We build in the expansion of `__exp_arg_next:nnn`.

```

2211 \cs_new_protected:Npn \::x #1 \::: #2#3
2212 {
2213   \cs_set_nopar:Npx \l__exp_internal_tl
2214   { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } }
2215   \l__exp_internal_tl
2216 }
```

(End definition for `\::x`. This function is documented on page 37.)

\::v These functions return the value of a register, i.e., one of `tl`, `clist`, `int`, `skip`, `dim`, **\::V** `muskip`, or built-in TeX register. The `V` version expects a single token whereas `v` like `c` creates a `csname` from its argument given in braces and then evaluates it as if it was a `V`. The `\exp:w` sets off an expansion similar to an `f`-type expansion, which we terminate using `\exp_end:`. The argument is returned in braces.

```

2217 \cs_new:Npn \::V #1 \::: #2#3
2218 {
2219   \exp_after:wN \__exp_arg_next:nnn
2220   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2221   {#1} {#2}
2222 }
2223 \cs_new:Npn \::v #1 \::: #2#3
2224 {
2225   \exp_after:wN \__exp_arg_next:nnn
2226   \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2227   {#1} {#2}
2228 }
```

(End definition for `\::v` and `\::V`. These functions are documented on page 37.)

`__exp_eval_register:N` This function evaluates a register. Now a register might exist as one of two things: A parameter-less macro or a built-in TeX register such as `\count`. For the TeX registers we have to utilize a `\the` whereas for the macros we merely have to expand them once. The trick is to find out when to use `\the` and when not to. What we want here is to find out whether the token expands to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the token in question when it has been prefixed with `\exp_not:N` and the token itself. If it is a macro, the prefixed `\exp_not:N` temporarily turns it into the primitive `\scan_stop:`.

```

2229 \cs_new:Npn \__exp_eval_register:N #1
2230 {
2231   \exp_after:wN \if_meaning:w \exp_not:N #1 #1
```

If the token was not a macro it may be a malformed variable from a `c` expansion in which case it is equal to the primitive `\scan_stop:`. In that case we throw an error. We could let `TeX` do it for us but that would result in the rather obscure

```
! You can't use '\relax' after \the.
```

which while quite true doesn't give many hints as to what actually went wrong. We provide something more sensible.

```
2232     \if_meaning:w \scan_stop: #1
2233     \__exp_eval_error_msg:w
2234     \fi:
```

The next bit requires some explanation. The function must be initiated by `\exp:w` and we want to terminate this expansion chain by inserting the `\exp_end:` token. However, we have to expand the register `#1` before we do that. If it is a `TeX` register, we need to execute the sequence `\exp_after:wN \exp_end: \tex_the:D #1` and if it is a macro we need to execute `\exp_after:wN \exp_end: #1`. We therefore issue the longer of the two sequences and if the register is a macro, we remove the `\tex_the:D`.

```
2235     \else:
2236     \exp_after:wN \use_i_ii:nnn
2237     \fi:
2238     \exp_after:wN \exp_end: \tex_the:D #1
2239   }
2240 \cs_new:Npn \__exp_eval_register:c #1
2241 { \exp_after:wN \__exp_eval_register:N \cs:w #1 \cs_end: }
```

Clean up nicely, then call the undefined control sequence. The result is an error message looking like this:

```
! Undefined control sequence.
<argument> \LaTeX3 error:
                               Erroneous variable used!
1.55 \tl_set:Nv \l_tmpa_tl {undefined_tl}

2242 \cs_new:Npn \__exp_eval_error_msg:w #1 \tex_the:D #2
2243 {
2244   \fi:
2245   \fi:
2246   \__kernel_msg_expandable_error:nnn { kernel } { bad-variable } {#2}
2247   \exp_end:
2248 }
```

(End definition for `__exp_eval_register:N` and `__exp_eval_error_msg:w`.)

6.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable.

`\exp_args:Nc` In `l3basics`.

`\exp_args:cc` *(End definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 29.)*

`\exp_args:Nnc` Here are the functions that turn their argument into csnames but are expandable.

```

2249 \cs_new:Npn \exp_args:Nnc #1#2#3
2250 { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
2251 \cs_new:Npn \exp_args:Ncc #1#2#3
2252 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
2253 \cs_new:Npn \exp_args:Nccc #1#2#3#4
2254 {
2255   \exp_after:wN #1
2256   \cs:w #2 \exp_after:wN \cs_end:
2257   \cs:w #3 \exp_after:wN \cs_end:
2258   \cs:w #4 \cs_end:
2259 }
```

(End definition for `\exp_args:Nnc`, `\exp_args:Ncc`, and `\exp_args:Nccc`. These functions are documented on page 31.)

`\exp_args:No` Those lovely runs of expansion!

```

2260 \cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
2261 \cs_new:Npn \exp_args:NNo #1#2#3
2262 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
2263 \cs_new:Npn \exp_args:NNNo #1#2#3#4
2264 { \exp_after:wN #1 \exp_after:wN#2 \exp_after:wN #3 \exp_after:wN {#4} }
```

(End definition for `\exp_args:No`, `\exp_args:NNo`, and `\exp_args:NNNo`. These functions are documented on page 30.)

`\exp_args:Ne` When the `\expanded` primitive is available, use it. Otherwise use `__exp_e:nn`, defined later, to fully expand tokens.

```

2265 \cs_if_exist:NTF \tex_expanded:D
2266 {
2267   \cs_new:Npn \exp_args:Ne #1#2
2268   { \exp_after:wN #1 \tex_expanded:D { {#2} } }
2269 }
2270 {
2271   \cs_new:Npn \exp_args:Ne #1#2
2272   {
2273     \exp_after:wN #1 \exp_after:wN
2274     { \exp:w \__exp_e:nn {#2} { } }
2275   }
2276 }
```

(End definition for `\exp_args:Ne`. This function is documented on page 30.)

`\exp_args:Nf`
`\exp_args:Nv`
`\exp_args:Nv`

```

2277 \cs_new:Npn \exp_args:Nf #1#2
2278 { \exp_after:wN #1 \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } }
2279 \cs_new:Npn \exp_args:Nv #1#2
2280 {
2281   \exp_after:wN #1 \exp_after:wN
2282   { \exp:w \__exp_eval_register:c {#2} }
2283 }
2284 \cs_new:Npn \exp_args:NV #1#2
2285 {
2286   \exp_after:wN #1 \exp_after:wN
2287   { \exp:w \__exp_eval_register:N #2 }
2288 }
```

(End definition for `\exp_args:Nf`, `\exp_args:Nv`, and `\exp_args:Nv`. These functions are documented on page 30.)

`\exp_args:NNV` Some more hand-tuned function with three arguments. If we forced that an `o` argument always has braces, we could implement `\exp_args:Nco` with less tokens and only two arguments.

```

2289 \cs_new:Npn \exp_args:NNV #1#2#3
2290 {
2291   \exp_after:wN #1
2292   \exp_after:wN #2
2293   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2294 }
2295 \cs_new:Npn \exp_args:NNv #1#2#3
2296 {
2297   \exp_after:wN #1
2298   \exp_after:wN #2
2299   \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2300 }
2301 \cs_if_exist:NTF \tex_expanded:D
2302 {
2303   \cs_new:Npn \exp_args:NNe #1#2#3
2304   {
2305     \exp_after:wN #1
2306     \exp_after:wN #2
2307     \tex_expanded:D { {#3} }
2308   }
2309 }
2310 { \cs_new:Npn \exp_args:NNe { \::N \::e \::: } }
2311 \cs_new:Npn \exp_args:NNf #1#2#3
2312 {
2313   \exp_after:wN #1
2314   \exp_after:wN #2
2315   \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2316 }
2317 \cs_new:Npn \exp_args:Nco #1#2#3
2318 {
2319   \exp_after:wN #1
2320   \cs:w #2 \exp_after:wN \cs_end:
2321   \exp_after:wN {#3}
2322 }
2323 \cs_new:Npn \exp_args:NcV #1#2#3
2324 {
2325   \exp_after:wN #1
2326   \cs:w #2 \exp_after:wN \cs_end:
2327   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2328 }
2329 \cs_new:Npn \exp_args:Ncv #1#2#3
2330 {
2331   \exp_after:wN #1
2332   \cs:w #2 \exp_after:wN \cs_end:
2333   \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2334 }
2335 \cs_new:Npn \exp_args:Ncf #1#2#3
2336 {

```



```

2337     \exp_after:wN #1
2338     \cs:w #2 \exp_after:wN \cs_end:
2339     \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2340   }
2341 \cs_new:Npn \exp_args:NVV #1#2#3
2342 {
2343   \exp_after:wN #1
2344   \exp_after:wN { \exp:w \exp_after:wN
2345     \__exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
2346   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2347 }

```

(End definition for `\exp_args:NNV` and others. These functions are documented on page 31.)

`\exp_args:NNNV` A few more that we can hand-tune.

```

\exp_args:NcNc 2348 \cs_new:Npn \exp_args:NNNV #1#2#3#4
\exp_args:NcNo 2349 {
\exp_args:Ncco 2350   \exp_after:wN #1
2351   \exp_after:wN #2
2352   \exp_after:wN #3
2353   \exp_after:wN { \exp:w \__exp_eval_register:N #4 }
2354 }
2355 \cs_new:Npn \exp_args:NcNc #1#2#3#4
2356 {
2357   \exp_after:wN #1
2358   \cs:w #2 \exp_after:wN \cs_end:
2359   \exp_after:wN #3
2360   \cs:w #4 \cs_end:
2361 }
2362 \cs_new:Npn \exp_args:NcNo #1#2#3#4
2363 {
2364   \exp_after:wN #1
2365   \cs:w #2 \exp_after:wN \cs_end:
2366   \exp_after:wN #3
2367   \exp_after:wN {#4}
2368 }
2369 \cs_new:Npn \exp_args:Ncco #1#2#3#4
2370 {
2371   \exp_after:wN #1
2372   \cs:w #2 \exp_after:wN \cs_end:
2373   \cs:w #3 \exp_after:wN \cs_end:
2374   \exp_after:wN {#4}
2375 }

```

(End definition for `\exp_args:NNNV` and others. These functions are documented on page 32.)

`\exp_args:Nx`

```

2376 \cs_new_protected:Npn \exp_args:Nx #1#2
2377 { \use:x { \exp_not:N #1 {#2} } }

```

(End definition for `\exp_args:Nx`. This function is documented on page 31.)

6.3 Last-unbraced versions

There are a few places where the last argument needs to be available unbraced. First some helper macros.

```

\__exp_arg_last_unbraced:nn
\::o_unbraced
\::V_unbraced
\::v_unbraced
\::e_unbraced
\::f_unbraced
\::x_unbraced
2378 \cs_new:Npn \__exp_arg_last_unbraced:nn #1#2 { #2#1 }
2379 \cs_new:Npn \::o_unbraced \::: #1#2
2380 { \exp_after:wN \__exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
2381 \cs_new:Npn \::V_unbraced \::: #1#2
2382 {
2383   \exp_after:wN \__exp_arg_last_unbraced:nn
2384   \exp_after:wN { \exp:w \__exp_eval_register:N #2 } {#1}
2385 }
2386 \cs_new:Npn \::v_unbraced \::: #1#2
2387 {
2388   \exp_after:wN \__exp_arg_last_unbraced:nn
2389   \exp_after:wN { \exp:w \__exp_eval_register:c {#2} } {#1}
2390 }
2391 \cs_if_exist:NTF \tex_expanded:D
2392 {
2393   \cs_new:Npn \::e_unbraced \::: #1#2
2394   { \tex_expanded:D { \exp_not:n {#1} #2 } }
2395 }
2396 {
2397   \cs_new:Npn \::e_unbraced \::: #1#2
2398   { \exp:w \__exp_e:nn {#2} {#1} }
2399 }
2400 \cs_new:Npn \::f_unbraced \::: #1#2
2401 {
2402   \exp_after:wN \__exp_arg_last_unbraced:nn
2403   \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } {#1}
2404 }
2405 \cs_new_protected:Npn \::x_unbraced \::: #1#2
2406 {
2407   \cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n {#1} #2 }
2408   \l__exp_internal_tl
2409 }

```

(End definition for `__exp_arg_last_unbraced:nn` and others. These functions are documented on page 37.)

Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```

\exp_last_unbraced:No
\exp_last_unbraced:Nv
\exp_last_unbraced:Nv
\exp_last_unbraced:Ne
\exp_last_unbraced:Nf
\exp_last_unbraced:NNo
\exp_last_unbraced:NNv
\exp_last_unbraced:NNf
\exp_last_unbraced:Nco
\exp_last_unbraced:NcV
\exp_last_unbraced:NNNo
\exp_last_unbraced:NNNV
\exp_last_unbraced:NNNf
\exp_last_unbraced:Nno
\exp_last_unbraced:Noo
\exp_last_unbraced:Nfo
\exp_last_unbraced:NnNo
\exp_last_unbraced:NNNNo
\exp_last_unbraced:NNNNf
\exp_last_unbraced:Nx
2410 \cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
2411 \cs_new:Npn \exp_last_unbraced:Nv #1#2
2412 { \exp_after:wN #1 \exp:w \__exp_eval_register:N #2 }
2413 \cs_new:Npn \exp_last_unbraced:Nv #1#2
2414 { \exp_after:wN #1 \exp:w \__exp_eval_register:c {#2} }
2415 \cs_if_exist:NTF \tex_expanded:D
2416 {
2417   \cs_new:Npn \exp_last_unbraced:Ne #1#2
2418   { \exp_after:wN #1 \tex_expanded:D {#2} }
2419 }
2420 { \cs_new:Npn \exp_last_unbraced:Ne { \::e_unbraced \::: } }
2421 \cs_new:Npn \exp_last_unbraced:Nf #1#2

```

```

2422 { \exp_after:wN #1 \exp:w \exp_end_continue_f:w #2 }
2423 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3
2424 { \exp_after:wN #1 \exp_after:wN #2 #3 }
2425 \cs_new:Npn \exp_last_unbraced:NNV #1#2#3
2426 {
2427   \exp_after:wN #1
2428   \exp_after:wN #2
2429   \exp:w \__exp_eval_register:N #3
2430 }
2431 \cs_new:Npn \exp_last_unbraced:NNf #1#2#3
2432 {
2433   \exp_after:wN #1
2434   \exp_after:wN #2
2435   \exp:w \exp_end_continue_f:w #3
2436 }
2437 \cs_new:Npn \exp_last_unbraced:Nco #1#2#3
2438 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
2439 \cs_new:Npn \exp_last_unbraced:NcV #1#2#3
2440 {
2441   \exp_after:wN #1
2442   \cs:w #2 \exp_after:wN \cs_end:
2443   \exp:w \__exp_eval_register:N #3
2444 }
2445 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
2446 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
2447 \cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
2448 {
2449   \exp_after:wN #1
2450   \exp_after:wN #2
2451   \exp_after:wN #3
2452   \exp:w \__exp_eval_register:N #4
2453 }
2454 \cs_new:Npn \exp_last_unbraced:NNNf #1#2#3#4
2455 {
2456   \exp_after:wN #1
2457   \exp_after:wN #2
2458   \exp_after:wN #3
2459   \exp:w \exp_end_continue_f:w #4
2460 }
2461 \cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \:: }
2462 \cs_new:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \:: }
2463 \cs_new:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \:: }
2464 \cs_new:Npn \exp_last_unbraced:NnNo { \::n \::N \::o_unbraced \:: }
2465 \cs_new:Npn \exp_last_unbraced:NNNNo #1#2#3#4#5
2466 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 \exp_after:wN #4 #5 }
2467 \cs_new:Npn \exp_last_unbraced:NNNNf #1#2#3#4#5
2468 {
2469   \exp_after:wN #1
2470   \exp_after:wN #2
2471   \exp_after:wN #3
2472   \exp_after:wN #4
2473   \exp:w \exp_end_continue_f:w #5
2474 }
2475 \cs_new_protected:Npn \exp_last_unbraced:Nx { \::x_unbraced \:: }

```

(End definition for `\exp_last_unbraced:No` and others. These functions are documented on page 33.)

If #2 is a single token then this can be implemented as

```
\cs_new:Npn \exp_last_two_unbraced:Noo #1 #2 #3
{ \exp_after:wN \exp_after:wN \exp_after:wN #1 \exp_after:wN #2 #3 }
```

However, for robustness this is not suitable. Instead, a bit of a shuffle is used to ensure that #2 can be multiple tokens.

```
2476 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
2477 { \exp_after:wN \__exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
2478 \cs_new:Npn \__exp_last_two_unbraced:noN #1#2#3
2479 { \exp_after:wN #3 #2 #1 }
```

(End definition for `\exp_last_two_unbraced:Noo` and `__exp_last_two_unbraced:noN`. This function is documented on page 33.)

6.4 Preventing expansion

`__kernel_exp_not:w` At the kernel level, we need the primitive behaviour to allow expansion *before* the brace group.

```
2480 \cs_new_eq:NN \__kernel_exp_not:w \tex_unexpanded:D
```

(End definition for `__kernel_exp_not:w`.)

`\exp_not:c` All these except `\exp_not:c` call the kernel-internal `__kernel_exp_not:w` namely `\tex_unexpanded:D`.

```
\exp_not:o \tex_unexpanded:D.
\exp_not:e 2481 \cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
\exp_not:f 2482 \cs_new:Npn \exp_not:o #1 { \__kernel_exp_not:w \exp_after:wN {#1} }
\exp_not:V 2483 \cs_if_exist:NTF \tex_expanded:D
\exp_not:v 2484 {
2485   \cs_new:Npn \exp_not:e #1
2486   { \__kernel_exp_not:w \tex_expanded:D { {#1} } }
2487 }
2488 {
2489   \cs_new:Npn \exp_not:e
2490   { \__kernel_exp_not:w \exp_args:Ne \prg_do_nothing: }
2491 }
2492 \cs_new:Npn \exp_not:f #1
2493 { \__kernel_exp_not:w \exp_after:wN { \exp:w \exp_end_continue_f:w #1 } }
2494 \cs_new:Npn \exp_not:V #1
2495 {
2496   \__kernel_exp_not:w \exp_after:wN
2497   { \exp:w \__exp_eval_register:N #1 }
2498 }
2499 \cs_new:Npn \exp_not:v #1
2500 {
2501   \__kernel_exp_not:w \exp_after:wN
2502   { \exp:w \__exp_eval_register:c {#1} }
2503 }
```

(End definition for `\exp_not:c` and others. These functions are documented on page 34.)

6.5 Controlled expansion

`\exp:w`
`\exp_end:`
`\exp_end_continue_f:w`
`\exp_end_continue_f:nw`

To trigger a sequence of “arbitrarily” many expansions we need a method to invoke T_EX’s expansion mechanism in such a way that (a) we are able to stop it in a controlled manner and (b) the result of what triggered the expansion in the first place is null, i.e., that we do not get any unwanted side effects. There aren’t that many possibilities in T_EX; in fact the one explained below might well be the only one (as normally the result of expansion is not null).

The trick here is to make use of the fact that `\tex_romannumeral:D` expands the tokens following it when looking for a number and that its expansion is null if that number turns out to be zero or negative. So we use that to start the expansion sequence: `\exp:w` is set equal to `\tex_romannumeral:D` in `l3basics`. To stop the expansion sequence in a controlled way all we need to provide is a constant integer zero as part of expanded tokens. As this is an integer constant it immediately stops `\tex_romannumeral:D`’s search for a number. Again, the definition of `\exp_end:` as the integer constant zero is in `l3basics`. (Note that according to our specification all tokens we expand initiated by `\exp:w` are supposed to be expandable (as well as their replacement text in the expansion) so we will not encounter a “number” that actually result in a roman numeral being generated. Or if we do then the programmer made a mistake.)

If on the other hand we want to stop the initial expansion sequence but continue with an f-type expansion we provide the alphabetic constant `^^@` that also represents 0 but this time T_EX’s syntax for a *⟨number⟩* continues searching for an optional space (and it continues expansion doing that) — see T_EXbook page 269 for details.

```
2504 \group_begin:
2505   \tex_catcode:D ‘^^@ = 13
2506   \cs_new_protected:Npn \exp_end_continue_f:w { ‘^^@ }
```

If the above definition ever appears outside its proper context the active character `^^@` will be executed so we turn this into an error. The test for existence covers the (unlikely) case that some other code has already defined `^^@`: this is true for example for `xmltex.tex`.

```
2507   \if_cs_exist:N ^^@
2508   \else:
2509     \cs_new:Npn ^^@
2510       { \__kernel_msg_expandable_error:nn { kernel } { bad-exp-end-f } }
2511   \fi:
```

The same but grabbing an argument to remove spaces and braces.

```
2512   \cs_new:Npn \exp_end_continue_f:nw #1 { ‘^^@ #1 }
2513 \group_end:
```

(End definition for `\exp:w` and others. These functions are documented on page 36.)

6.6 Emulating e-type expansion

When the `\expanded` primitive is available it is used to implement e-type expansion; otherwise we emulate it.

```
2514 \cs_if_exist:NF \tex_expanded:D
2515 {
```

`__exp_e:nn`
`__exp_e_end:nn`

Repeatedly expand tokens, keeping track of fully-expanded tokens in the second argument to `__exp_e:nn`; this function eventually calls `__exp_e_end:nn` to leave `\exp_end:` in the input stream, followed by the result of the expansion. There are many special cases:

spaces, brace groups, `\noexpand`, `\unexpanded`, `\the`, `\primitive`. While we use brace tricks `\if_false: { \fi:`, the expansion of this function is always triggered by `\exp:w` so brace balance is eventually restored after that is hit with a single step of expansion. Otherwise we could not nest e-type expansions within each other.

```

2516 \cs_new:Npn \__exp_e:nn #1
2517 {
2518   \if_false: { \fi:
2519     \tl_if_head_is_N_type:nTF {#1}
2520     { \__exp_e:N }
2521     {
2522       \tl_if_head_is_group:nTF {#1}
2523       { \__exp_e_group:n }
2524       {
2525         \tl_if_empty:nTF {#1}
2526         { \exp_after:wN \__exp_e_end:nn }
2527         { \exp_after:wN \__exp_e_space:nn }
2528         \exp_after:wN { \if_false: } \fi:
2529       }
2530     }
2531     #1
2532   }
2533 }
2534 \cs_new:Npn \__exp_e_end:nn #1#2 { \exp_end: #2 }

```

(End definition for `__exp_e:nn` and `__exp_e_end:nn`.)

`__exp_e_space:nn` For an explicit space character, remove it by f-expansion and put it in the (future) output.

```

2535 \cs_new:Npn \__exp_e_space:nn #1#2
2536 { \exp_args:Nf \__exp_e:nn {#1} { #2 ~ } }

```

(End definition for `__exp_e_space:nn`.)

`__exp_e_group:n` For a group, expand its contents, wrap it in two pairs of braces, and call `__exp_e_put:nn`. This function places the first item (the double-brace wrapped result) into the output. Importantly, `\tl_head:n` works even if the input contains quarks.

```

2537 \cs_new:Npn \__exp_e_group:n #1
2538 {
2539   \exp_after:wN \__exp_e_put:nn
2540   \exp_after:wN { \exp_after:wN { \exp_after:wN {
2541     \exp:w \if_false: } \fi: \__exp_e:nn {#1} { } } }
2542 }
2543 \cs_new:Npn \__exp_e_put:nn #1
2544 {
2545   \exp_args:NNo \exp_args:No \__exp_e_put:nnn
2546   { \tl_head:n {#1} } {#1}
2547 }
2548 \cs_new:Npn \__exp_e_put:nnn #1#2#3
2549 { \exp_args:No \__exp_e:nn { \use_none:n #2 } { #3 #1 } }

```

(End definition for `__exp_e_group:n`, `__exp_e_put:nn`, and `__exp_e_put:nnn`.)

`__exp_e:N` For an N-type token, call `__exp_e:Nnn` with arguments the *⟨first token⟩*, the remain-
`__exp_e:Nnn` ing tokens to expand and what’s already been expanded. If the *⟨first token⟩* is non-
`__exp_e_protected:Nnn` expandable, including `\protected` (`\long` or not) macros, it is put in the result by
`__exp_e_expandable:Nnn` `__exp_e_protected:Nnn`. The four special primitives `\unexpanded`, `\noexpand`, `\the`,
`\primitive` are detected; otherwise the token is expanded by `__exp_e_expandable:Nnn`.

```

2550 \cs_new:Npn \__exp_e:N #1
2551 {
2552   \exp_after:wN \__exp_e:Nnn
2553   \exp_after:wN #1
2554   \exp_after:wN { \if_false: } \fi:
2555 }
2556 \cs_new:Npn \__exp_e:Nnn #1
2557 {
2558   \if_case:w
2559     \exp_after:wN \if_meaning:w \exp_not:N #1 #1 1 ~ \fi:
2560     \token_if_protected_macro:NT #1 { 1 ~ }
2561     \token_if_protected_long_macro:NT #1 { 1 ~ }
2562     \if_meaning:w \exp_not:n #1 2 ~ \fi:
2563     \if_meaning:w \exp_not:N #1 3 ~ \fi:
2564     \if_meaning:w \tex_the:D #1 4 ~ \fi:
2565     \if_meaning:w \tex_primitive:D #1 5 ~ \fi:
2566     0 ~
2567     \exp_after:wN \__exp_e_expandable:Nnn
2568     \or: \exp_after:wN \__exp_e_protected:Nnn
2569     \or: \exp_after:wN \__exp_e_unexpanded:Nnn
2570     \or: \exp_after:wN \__exp_e_noexpand:Nnn
2571     \or: \exp_after:wN \__exp_e_the:Nnn
2572     \or: \exp_after:wN \__exp_e_primitive:Nnn
2573     \fi:
2574     #1
2575   }
2576   \cs_new:Npn \__exp_e_protected:Nnn #1#2#3
2577     { \__exp_e:nn {#2} { #3 #1 } }
2578   \cs_new:Npn \__exp_e_expandable:Nnn #1#2
2579     { \exp_args:No \__exp_e:nn { #1 #2 } }

```

(End definition for `__exp_e:N` and others.)

`__exp_e_primitive:Nnn` We don’t try hard to make sensible error recovery since the error recovery of `\tex_`
`__exp_e_primitive_aux:NNw` `primitive:D` when followed by something else than a primitive depends on the engine.
`__exp_e_primitive_aux:NNnn` The only valid case is when what follows is N-type. Then distinguish special primitives
`__exp_e_primitive_other:NNnn` `\unexpanded`, `\noexpand`, `\the`, `\primitive` from other primitives. In the “other” case,
`__exp_e_primitive_other_aux:nNNnn` the only reasonable way to check if the primitive that follows `\tex_primitive:D` is
expandable is to expand and compare the before-expansion and after-expansion results.
If they coincide then probably the primitive is non-expandable and should be put in the
output together with `\tex_primitive:D` (one can cook up contrived counter-examples
where the true `\expanded` would have an infinite loop), and otherwise one should continue
expanding.

```

2580 \cs_new:Npn \__exp_e_primitive:Nnn #1#2
2581 {
2582   \if_false: { \fi:
2583   \tl_if_head_is_N_type:nTF {#2}
2584     { \__exp_e_primitive_aux:NNw #1 }

```

```

2585         {
2586             \__kernel_msg_expandable_error:nnn { kernel } { e-type }
2587             { Missing~primitive~name }
2588             \__exp_e_primitive_aux:NNw #1 \c_empty_tl
2589         }
2590     #2
2591 }
2592 }
2593 \cs_new:Npn \__exp_e_primitive_aux:NNw #1#2
2594 {
2595     \exp_after:wN \__exp_e_primitive_aux:NNnn
2596     \exp_after:wN #1
2597     \exp_after:wN #2
2598     \exp_after:wN { \if_false: } \fi:
2599 }
2600 \cs_new:Npn \__exp_e_primitive_aux:NNnn #1#2
2601 {
2602     \exp_args:Nf \str_case_e:nnTF { \cs_to_str:N #2 }
2603     {
2604         { unexpanded } { \__exp_e_unexpanded:NNn \exp_not:n }
2605         { noexpand } { \__exp_e_noexpand:NNn \exp_not:N }
2606         { the } { \__exp_e_the:NNn \tex_the:D }
2607         {
2608             \sys_if_engine_xetex:T { pdf }
2609             \sys_if_engine luatex:T { pdf }
2610             primitive
2611         } { \__exp_e_primitive:NNn #1 }
2612     }
2613     { \__exp_e_primitive_other:NNnn #1 #2 }
2614 }
2615 \cs_new:Npn \__exp_e_primitive_other:NNnn #1#2#3
2616 {
2617     \exp_args:No \__exp_e_primitive_other_aux:nNNnn
2618     { #1 #2 #3 }
2619     #1 #2 {#3}
2620 }
2621 \cs_new:Npn \__exp_e_primitive_other_aux:nNNnn #1#2#3#4#5
2622 {
2623     \str_if_eq:nnTF {#1} { #2 #3 #4 }
2624     { \__exp_e:nn {#4} { #5 #2 #3 } }
2625     { \__exp_e:nn {#1} {#5} }
2626 }

```

(End definition for __exp_e_primitive:NNn and others.)

__exp_e_noexpand:NNn The \noexpand primitive has no effect when followed by a token that is not N-type; otherwise __exp_e_put:nn can grab the next token and put it in the result unchanged.

```

2627 \cs_new:Npn \__exp_e_noexpand:NNn #1#2
2628 {
2629     \tl_if_head_is_N_type:nTF {#2}
2630     { \__exp_e_put:nn } { \__exp_e:nn } {#2}
2631 }

```

(End definition for __exp_e_noexpand:NNn.)

`__exp_e_unexpanded:Nnn`
`__exp_e_unexpanded:nn`
`__exp_e_unexpanded:nN`
`__exp_e_unexpanded:N`

The `\unexpanded` primitive expands and ignores any space, `\scan_stop:`, or token affected by `\exp_not:N`, then expects a brace group. Since we only support brace-balanced token lists it is impossible to support the case where the argument of `\unexpanded` starts with an implicit brace. Even though we want to expand and ignore spaces we cannot blindly f-expand because tokens affected by `\exp_not:N` should be discarded without being expanded further.

As usual distinguish four cases: brace group (the normal case, where we just put the item in the result), space (just f-expand to remove the space), empty (an error), or N-type *token*. In the last case call `__exp_e_unexpanded:nN` triggered by an f-expansion. Having a non-expandable *token* after `\unexpanded` is an error (we recover by passing `{}` to `\unexpanded`; this is different from T_EX because the error recovery of `\unexpanded` changes the balance of braces), unless that *token* is `\scan_stop:` or a space (recall that we don't implement the case of an implicit begin-group token). An expandable *token* is instead expanded, unless it is `\noexpand`. The latter primitive can be followed by an expandable N-type token (removed), by a non-expandable one (kept and later causing an error), by a space (removed by f-expansion), or by a brace group or nothing (later causing an error).

```

2632 \cs_new:Npn \__exp_e_unexpanded:Nnn #1 { \__exp_e_unexpanded:nn }
2633 \cs_new:Npn \__exp_e_unexpanded:nn #1
2634 {
2635   \tl_if_head_is_N_type:nTF {#1}
2636   {
2637     \exp_args:Nf \__exp_e_unexpanded:nn
2638     { \__exp_e_unexpanded:nN {#1} #1 }
2639   }
2640   {
2641     \tl_if_head_is_group:nTF {#1}
2642     { \__exp_e_put:nn }
2643     {
2644       \tl_if_empty:nTF {#1}
2645       {
2646         \__kernel_msg_expandable_error:nnn
2647         { kernel } { e-type }
2648         { \unexpanded missing~brace }
2649         \__exp_e_end:nn
2650       }
2651       { \exp_args:Nf \__exp_e_unexpanded:nn }
2652     }
2653     {#1}
2654   }
2655 }
2656 \cs_new:Npn \__exp_e_unexpanded:nN #1#2
2657 {
2658   \exp_after:wN \if_meaning:w \exp_not:N #2 #2
2659   \exp_after:wN \use_i:nn
2660   \else:
2661     \exp_after:wN \use_ii:nn
2662   \fi:
2663   {
2664     \token_if_eq_catcode:NNTF #2 \c_space_token
2665     { \exp_stop_f: }
2666     {

```

```

2667         \token_if_eq_meaning:NNTF #2 \scan_stop:
2668         { \exp_stop_f: }
2669         {
2670             \__kernel_msg_expandable_error:nnn
2671             { kernel } { e-type }
2672             { \unexpanded missing-brace }
2673             { }
2674         }
2675     }
2676 }
2677 {
2678     \token_if_eq_meaning:NNTF #2 \exp_not:N
2679     {
2680         \exp_args:No \tl_if_head_is_N_type:nT { \use_none:n #1 }
2681         { \__exp_e_unexpanded:N }
2682     }
2683     { \exp_after:wN \exp_stop_f: #2 }
2684 }
2685 }
2686 \cs_new:Npn \__exp_e_unexpanded:N #1
2687 {
2688     \exp_after:wN \if_meaning:w \exp_not:N #1 #1 \else:
2689     \exp_after:wN \use_i:nn
2690     \fi:
2691     \exp_stop_f: #1
2692 }

```

(End definition for `__exp_e_unexpanded:Nnn` and others.)

```

\__exp_e_the:Nnn
\__exp_e_the:N
\__exp_e_the_toks_reg:N

```

Finally implement `\the`. Followed by anything other than an N-type *<token>* this causes an error (we just let T_EX make one), otherwise we test the *<token>*. If the *<token>* is expandable, expand it. Otherwise it could be any kind of register, or things like `\numexpr`, so there is no way to deal with all cases. Thankfully, only `\toks` data needs to be protected from expansion since everything else gives a string of characters. If the *<token>* is `\toks` we find a number and unpack using the `the_toks` functions. If it is a token register we unpack it in a brace group and call `__exp_e_put:nn` to move it to the result. Otherwise we unpack and continue expanding (useless but safe) since it is basically impossible to have a handle on where the result of `\the` ends.

```

2693 \cs_new:Npn \__exp_e_the:Nnn #1#2
2694 {
2695     \tl_if_head_is_N_type:nTF {#2}
2696     { \if_false: { \fi: \__exp_e_the:N #2 } }
2697     { \exp_args:No \__exp_e:nn { \tex_the:D #2 } }
2698 }
2699 \cs_new:Npn \__exp_e_the:N #1
2700 {
2701     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
2702     \exp_after:wN \use_i:nn
2703     \else:
2704     \exp_after:wN \use_ii:nn
2705     \fi:
2706     {
2707         \if_meaning:w \tex_toks:D #1
2708         \exp_after:wN \__exp_e_the_toks:wnn \int_value:w

```

```

2709         \exp_after:wN \_\_exp_e_the_toks:n
2710         \exp_after:wN { \int_value:w \if_false: } \fi:
2711     \else:
2712         \_\_exp_e_if_toks_register:NTF #1
2713         { \exp_after:wN \_\_exp_e_the_toks_reg:N }
2714         {
2715             \exp_after:wN \_\_exp_e:nn \exp_after:wN {
2716                 \tex_the:D \if_false: } \fi:
2717         }
2718         \exp_after:wN #1
2719     \fi:
2720 }
2721 {
2722     \exp_after:wN \_\_exp_e_the:Nnn \exp_after:wN ?
2723     \exp_after:wN { \exp:w \if_false: } \fi:
2724     \exp_after:wN \exp_end: #1
2725 }
2726 }
2727 \cs_new:Npn \_\_exp_e_the_toks_reg:N #1
2728 {
2729     \exp_after:wN \_\_exp_e_put:nn \exp_after:wN {
2730         \exp_after:wN {
2731             \tex_the:D \if_false: } \fi: #1 }
2732 }

```

(End definition for `__exp_e_the:Nnn`, `__exp_e_the:N`, and `__exp_e_the_toks_reg:N`.)

`__exp_e_the_toks:wnn` The calling function has applied `\int_value:w` so we collect digits with `__exp_e_the_toks:n` (which gets the token list as an argument) and `__exp_e_the_toks:N` (which gets the first token in case it is N-type). The digits are themselves collected into an `\int_value:w` argument to `__exp_e_the_toks:wnn`. Then that function unpacks the `\toks<number>` into the result. We include `?` because `__exp_e_put:nnn` removes one item from its second argument. Note that our approach is rather crude: in cases like `\the\toks12~34` the first `\int_value:w` removes the space and we will incorrectly unpack the `\the\toks1234`.

```

2733 \cs_new:Npn \_\_exp_e_the_toks:wnn #1; #2
2734 {
2735     \exp_args:No \_\_exp_e_put:nnn
2736     { \tex_the:D \tex_toks:D #1 } { ? #2 }
2737 }
2738 \cs_new:Npn \_\_exp_e_the_toks:n #1
2739 {
2740     \tl_if_head_is_N_type:NTF {#1}
2741     { \exp_after:wN \_\_exp_e_the_toks:N \if_false: { \fi: #1 } }
2742     { ; {#1} }
2743 }
2744 \cs_new:Npn \_\_exp_e_the_toks:N #1
2745 {
2746     \if_int_compare:w 10 < 9 \token_to_str:N #1 \exp_stop_f:
2747     \exp_after:wN \use_i:nn
2748     \else:
2749         \exp_after:wN \use_ii:nn
2750     \fi:
2751 {

```

```

2752         #1
2753         \exp_after:wN \__exp_e_the_toks:n
2754         \exp_after:wN { \if_false: } \fi:
2755     }
2756     {
2757         \exp_after:wN ;
2758         \exp_after:wN { \if_false: } \fi: #1
2759     }
2760 }

```

(End definition for `__exp_e_the_toks:wnn`, `__exp_e_the_toks:n`, and `__exp_e_the_toks:N`.)

`__exp_e_if_toks_register:NTF` We need to detect both `\toks` registers like `\toks@` in L^AT_EX 2_ε and parameters such as `\everypar`, as the result of unpacking the register should not expand further. Registers are found by `\token_if_toks_register:NTF` by inspecting the meaning. The list of parameters is finite so we just use a `\cs_if_exist:cTF` test to look up in a table. We abuse `\cs_to_str:N`'s ability to remove a leading escape character whatever it is.

```

\__exp_e_the_everydisplay:
\__exp_e_the_everyeof: 2761 \prg_new_conditional:Npnn \__exp_e_if_toks_register:N #1 { TF }
\__exp_e_the_everyhbox: 2762 {
\__exp_e_the_everyjob: 2763     \token_if_toks_register:NTF #1 { \prg_return_true: }
\__exp_e_the_everymath: 2764     {
\__exp_e_the_everypar: 2765         \cs_if_exist:cTF
\__exp_e_the_everyvbox: 2766         {
\__exp_e_the_output: 2767             __exp_e_the_
2768             \exp_after:wN \cs_to_str:N
2769             \token_to_meaning:N #1
2770             :
2771             } { \prg_return_true: } { \prg_return_false: }
\__exp_e_the_pdfpageattr: 2772         }
\__exp_e_the_pdfpagesattr: 2773     }
\__exp_e_the_pdfpkmode: 2774     \cs_new_eq:NN \__exp_e_the_XeTeXinterchartoks: ?
2775     \cs_new_eq:NN \__exp_e_the_errhelp: ?
2776     \cs_new_eq:NN \__exp_e_the_everycr: ?
2777     \cs_new_eq:NN \__exp_e_the_everydisplay: ?
2778     \cs_new_eq:NN \__exp_e_the_everyeof: ?
2779     \cs_new_eq:NN \__exp_e_the_everyhbox: ?
2780     \cs_new_eq:NN \__exp_e_the_everyjob: ?
2781     \cs_new_eq:NN \__exp_e_the_everymath: ?
2782     \cs_new_eq:NN \__exp_e_the_everypar: ?
2783     \cs_new_eq:NN \__exp_e_the_everyvbox: ?
2784     \cs_new_eq:NN \__exp_e_the_output: ?
2785     \cs_new_eq:NN \__exp_e_the_pdfpageattr: ?
2786     \cs_new_eq:NN \__exp_e_the_pdfpageresources: ?
2787     \cs_new_eq:NN \__exp_e_the_pdfpagesattr: ?
2788     \cs_new_eq:NN \__exp_e_the_pdfpkmode: ?

```

(End definition for `__exp_e_if_toks_register:NTF` and others.)

We are done emulating e-type argument expansion when `\expanded` is unavailable.

```

2789 }

```

6.7 Defining function variants

```

2790 <@@=cs>

```

`\s__cs_mark` Internal scan marks. No l3quark yet, so do things by hand.

`\s__cs_stop` 2791 `\cs_new_eq:NN \s__cs_mark \scan_stop:`
 2792 `\cs_new_eq:NN \s__cs_stop \scan_stop:`

(End definition for `\s__cs_mark` and `\s__cs_stop`.)

`\q__cs_recursion_stop` Internal recursion quarks. No l3quark yet, so do things by hand.

2793 `\cs_new:Npn \q__cs_recursion_stop { \q__cs_recursion_stop }`

(End definition for `\q__cs_recursion_stop`.)

`__cs_use_none_delimit_by_s_stop:w` Internal scan marks.

`__cs_use_i_delimit_by_s_stop:nw` 2794 `\cs_new:Npn __cs_use_none_delimit_by_s_stop:w #1 \s__cs_stop { }`
`__cs_use_none_delimit_by_q_recursion_stop:w` 2795 `\cs_new:Npn __cs_use_i_delimit_by_s_stop:nw #1 #2 \s__cs_stop {#1}`
 2796 `\cs_new:Npn __cs_use_none_delimit_by_q_recursion_stop:w`
 2797 `#1 \q__cs_recursion_stop { }`

(End definition for `__cs_use_none_delimit_by_s_stop:w`, `__cs_use_i_delimit_by_s_stop:nw`, and `__cs_use_none_delimit_by_q_recursion_stop:w`.)

`\cs_generate_variant:Nn` #1 : Base form of a function; e.g., `\tl_set:Nn`

`\cs_generate_variant:cn` #2 : One or more variant argument specifiers; e.g., `{Nx,c,cx}`

After making sure that the base form exists, test whether it is protected or not and define `__cs_tmp:w` as either `\cs_new:Npx` or `\cs_new_protected:Npx`, which is then used to define all the variants (except those involving x-expansion, always protected). Split up the original base function only once, to grab its name and signature. Then we wish to iterate through the comma list of variant argument specifiers, which we first convert to a string: the reason is explained later.

2798 `\cs_new_protected:Npn \cs_generate_variant:Nn #1#2`
 2799 `{`
 2800 `__cs_generate_variant:N #1`
 2801 `\use:x`
 2802 `{`
 2803 `__cs_generate_variant:nnNN`
 2804 `\cs_split_function:N #1`
 2805 `\exp_not:N #1`
 2806 `\tl_to_str:n {#2} ,`
 2807 `\exp_not:N \scan_stop: ,`
 2808 `\exp_not:N \q__cs_recursion_stop`
 2809 `}`
 2810 `}`
 2811 `\cs_new_protected:Npn \cs_generate_variant:cn`
 2812 `{ \exp_args:Nc \cs_generate_variant:Nn }`

(End definition for `\cs_generate_variant:Nn`. This function is documented on page 27.)

`__cs_generate_variant:N` The goal here is to pick up protected parent functions. There are four cases: the parent function can be a primitive or a macro, and can be expandable or not. For non-expandable primitives, all variants should be protected; skipping the `\else:` branch is safe because non-expandable primitives cannot be T_EX conditionals.

The other case where variants should be protected is when the parent function is a protected macro: then `protected` appears in the meaning before the first occurrence of `macro`. The `ww` auxiliary removes everything in the meaning string after the first `ma`. We use `ma` rather than the full `macro` because the meaning of the `\firstmark` primitive (and

four others) can contain an arbitrary string after a leading `firstmark:`. Then, look for `pr` in the part we extracted: no need to look for anything longer: the only strings we can have are an empty string, `\long`, `\protected`, `\protected\long`, `\first`, `\top`, `\bot`, `\splittop`, or `\splitbot`, with `\` replaced by the appropriate escape character. If `pr` appears in the part before `ma`, the first `\s__cs_mark` is taken as an argument of the `wwNw` auxiliary, and `#3` is `\cs_new_protected:Npx`, otherwise it is `\cs_new:Npx`.

```

2813 \cs_new_protected:Npx \__cs_generate_variant:N #1
2814 {
2815   \exp_not:N \exp_after:wN \exp_not:N \if_meaning:w
2816   \exp_not:N \exp_not:N #1 #1
2817   \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx
2818   \exp_not:N \else:
2819   \exp_not:N \exp_after:wN \exp_not:N \__cs_generate_variant:ww
2820   \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma }
2821   \s__cs_mark
2822   \s__cs_mark \cs_new_protected:Npx
2823   \tl_to_str:n { pr }
2824   \s__cs_mark \cs_new:Npx
2825   \s__cs_stop
2826   \exp_not:N \fi:
2827 }
2828 \exp_last_unbraced:NNNNo
2829 \cs_new_protected:Npn \__cs_generate_variant:ww
2830 #1 { \tl_to_str:n { ma } } #2 \s__cs_mark
2831 { \__cs_generate_variant:wwNw #1 }
2832 \exp_last_unbraced:NNNNo
2833 \cs_new_protected:Npn \__cs_generate_variant:wwNw
2834 #1 { \tl_to_str:n { pr } } #2 \s__cs_mark #3 #4 \s__cs_stop
2835 { \cs_set_eq:NN \__cs_tmp:w #3 }

```

(End definition for `__cs_generate_variant:N`, `__cs_generate_variant:ww`, and `__cs_generate_variant:wwNw`.)

`__cs_generate_variant:nnNN` #1 : Base name.
 #2 : Base signature.
 #3 : Boolean.
 #4 : Base function.

If the boolean is `\c_false_bool`, the base function has no colon and we abort with an error; otherwise, set off a loop through the desired variant forms. The original function is retained as `#4` for efficiency.

```

2836 \cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
2837 {
2838   \if_meaning:w \c_false_bool #3
2839   \__kernel_msg_error:nxx { kernel } { missing-colon }
2840   { \token_to_str:c {#1} }
2841   \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
2842   \fi:
2843   \__cs_generate_variant:Nnnw #4 {#1}{#2}
2844 }

```

(End definition for `__cs_generate_variant:nnNN`.)

`__cs_generate_variant:Nnnw` #1 : Base function.
 #2 : Base name.

#3 : Base signature.

#4 : Beginning of variant signature.

First check whether to terminate the loop over variant forms. Then, for each variant form, construct a new function name using the original base name, the variant signature consisting of l letters and the last $k - l$ letters of the base signature (of length k). For example, for a base function `\prop_put:Nnn` which needs a `cV` variant form, we want the new signature to be `cVn`.

There are further subtleties:

- In `\cs_generate_variant:Nn \foo:nnTF {xxTF}`, we must define `\foo:xxTF` using `\exp_args:Nxx`, rather than a hypothetical `\exp_args:NxxTF`. Thus, we wish to trim a common trailing part from the base signature and the variant signature.
- In `\cs_generate_variant:Nn \foo:on {ox}`, the function `\foo:ox` must be defined using `\exp_args:Nnx`, not `\exp_args:Nox`, to avoid double `o` expansion.
- Lastly, `\cs_generate_variant:Nn \foo:on {xn}` must trigger an error, because we do not have a means to replace `o`-expansion by `x`-expansion. More generally, we can only convert `N` to `c`, or convert `n` to `V`, `v`, `o`, `f`, `x`.

All this boils down to a few rules. Only `n` and `N`-type arguments can be replaced by `\cs_generate_variant:Nn`. Other argument types are allowed to be passed unchanged from the base form to the variant: in the process they are changed to `n` except for `N` and `p`-type arguments. A common trailing part is ignored.

We compare the base and variant signatures one character at a time within `x`-expansion. The result is given to `__cs_generate_variant:wwNN` (defined later) in the form `<processed variant signature> \s__cs_mark <errors> \s__cs_stop <base function> <new function>`. If all went well, `<errors>` is empty; otherwise, it is a kernel error message and some clean-up code.

Note the space after #3 and after the following brace group. Those are ignored by `TeX` when fetching the last argument for `__cs_generate_variant_loop:nNwN`, but can be used as a delimiter for `__cs_generate_variant_loop_end:nwwwNNnn`.

```

2845 \cs_new_protected:Npn \__cs_generate_variant:Nnnw #1#2#3#4 ,
2846 {
2847   \if_meaning:w \scan_stop: #4
2848   \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
2849   \fi:
2850   \use:x
2851   {
2852     \exp_not:N \__cs_generate_variant:wwNN
2853     \__cs_generate_variant_loop:nNwN { }
2854     #4
2855     \__cs_generate_variant_loop_end:nwwwNNnn
2856     \s__cs_mark
2857     #3 ~
2858     { ~ { } \fi: \__cs_generate_variant_loop_long:wNNnn } ~
2859     { }
2860     \s__cs_stop
2861     \exp_not:N #1 {#2} {#4}
2862   }
2863   \__cs_generate_variant:Nnnw #1 {#2} {#3}
2864 }
```

(End definition for `_cs_generate_variant:Nnnw`.)

<code>_cs_generate_variant_loop:nNwN</code>	#1 :	Last few consecutive letters common between the base and variant (more precisely,
<code>_cs_generate_variant_loop_base:N</code>		<code>_cs_generate_variant_same:N</code> <i><letter></i> for each letter).
<code>_cs_generate_variant_loop_same:w</code>	#2 :	Next variant letter.
<code>_cs_generate_variant_loop_end:nwwwNNnn</code>	#3 :	Remainder of variant form.
<code>_cs_generate_variant_loop_long:wNNnn</code>	#4 :	Next base letter.
<code>_cs_generate_variant_loop_invalid:NNwNNnn</code>		
<code>_cs_generate_variant_loop_special:NNwNNnn</code>		

The first argument is populated by `_cs_generate_variant_loop_same:w` when a variant letter and a base letter match. It is flushed into the input stream whenever the two letters are different: if the loop ends before, the argument is dropped, which means that trailing common letters are ignored.

The case where the two letters are different is only allowed if the base is N and the variant is c, or when the base is n and the variant is o, V, v, f or x. Otherwise, call `_cs_generate_variant_loop_invalid:NNwNNnn` to remove the end of the loop, get arguments at the end of the loop, and place an appropriate error message as a second argument of `_cs_generate_variant:wwNN`. If the letters are distinct and the base letter is indeed n or N, leave in the input stream whatever argument #1 was collected, and the next variant letter #2, then loop by calling `_cs_generate_variant_loop:nNwN`.

The loop can stop in three ways.

- If the end of the variant form is encountered first, #2 is `_cs_generate_variant_loop_end:nwwwNNnn` (expanded by the conditional `\if:w`), which inserts some tokens to end the conditional; grabs the *<base name>* as #7, the *<variant signature>* #8, the *<next base letter>* #1 and the part #3 of the base signature that wasn't read yet; and combines those into the *<new function>* to be defined.
- If the end of the base form is encountered first, #4 is `~{} \fi:` which ends the conditional (with an empty expansion), followed by `_cs_generate_variant_loop_long:wNNnn`, which places an error as the second argument of `_cs_generate_variant:wwNN`.
- The loop can be interrupted early if the requested expansion is unavailable, namely when the variant and base letters differ and the base is not the right one (n or N to support the variant). In that case too an error is placed as the second argument of `_cs_generate_variant:wwNN`.

Note that if the variant form has the same length as the base form, #2 is as described in the first point, and #4 as described in the second point above. The `_cs_generate_variant_loop_end:nwwwNNnn` breaking function takes the empty brace group in #4 as its first argument: this empty brace group produces the correct signature for the full variant.

```

2865 \cs_new:Npn \_cs_generate_variant_loop:nNwN #1#2#3 \s_cs_mark #4
2866 {
2867   \if:w #2 #4
2868     \exp_after:wN \_cs_generate_variant_loop_same:w
2869   \else:
2870     \if:w #4 \_cs_generate_variant_loop_base:N #2 \else:
2871       \if:w 0
2872         \if:w N #4 \else: \if:w n #4 \else: 1 \fi: \fi:
2873         \if:w \scan_stop: \_cs_generate_variant_loop_base:N #2 1 \fi:
2874         0
2875         \_cs_generate_variant_loop_special:NNwNNnn #4#2

```



```

2876         \else:
2877             \__cs_generate_variant_loop_invalid:NNwNNnn #4#2
2878         \fi:
2879     \fi:
2880 \fi:
2881 #1
2882 \prg_do_nothing:
2883 #2
2884 \__cs_generate_variant_loop:nNwN { } #3 \s__cs_mark
2885 }
2886 \cs_new:Npn \__cs_generate_variant_loop_base:N #1
2887 {
2888     \if:w c #1 N \else:
2889         \if:w o #1 n \else:
2890             \if:w V #1 n \else:
2891                 \if:w v #1 n \else:
2892                     \if:w f #1 n \else:
2893                         \if:w e #1 n \else:
2894                             \if:w x #1 n \else:
2895                                 \if:w n #1 n \else:
2896                                     \if:w N #1 N \else:
2897                                         \scan_stop:
2898                                         \fi:
2899                                     \fi:
2900                                 \fi:
2901                             \fi:
2902                         \fi:
2903                     \fi:
2904                 \fi:
2905             \fi:
2906         \fi:
2907     }
2908 \cs_new:Npn \__cs_generate_variant_loop_same:w
2909     #1 \prg_do_nothing: #2#3#4
2910     { #3 { #1 \__cs_generate_variant_same:N #2 } }
2911 \cs_new:Npn \__cs_generate_variant_loop_end:nwwwNNnn
2912     #1#2 \s__cs_mark #3 ~ #4 \s__cs_stop #5#6#7#8
2913 {
2914     \scan_stop: \scan_stop: \fi:
2915     \s__cs_mark \s__cs_stop
2916     \exp_not:N #6
2917     \exp_not:c { #7 : #8 #1 #3 }
2918 }
2919 \cs_new:Npn \__cs_generate_variant_loop_long:wNNnn #1 \s__cs_stop #2#3#4#5
2920 {
2921     \exp_not:n
2922     {
2923         \s__cs_mark
2924         \__kernel_msg_error:nxxx { kernel } { variant-too-long }
2925         {#5} { \token_to_str:N #3 }
2926         \use_none:nnn
2927         \s__cs_stop
2928         #3
2929         #3

```

```

2930     }
2931   }
2932   \cs_new:Npn \__cs_generate_variant_loop_invalid:NNwNNnn
2933     #1#2 \fi: \fi: \fi: #3 \s__cs_stop #4#5#6#7
2934   {
2935     \fi: \fi: \fi:
2936     \exp_not:n
2937     {
2938       \s__cs_mark
2939       \__kernel_msg_error:nxxxxx { kernel } { invalid-variant }
2940       {#7} { \token_to_str:N #5 } {#1} {#2}
2941       \use_none:nnn
2942       \s__cs_stop
2943       #5
2944       #5
2945     }
2946   }
2947   \cs_new:Npn \__cs_generate_variant_loop_special:NNwNNnn
2948     #1#2#3 \s__cs_stop #4#5#6#7
2949   {
2950     #3 \s__cs_stop #4 #5 {#6} {#7}
2951     \exp_not:n
2952     {
2953       \__kernel_msg_error:nxxxxx
2954       { kernel } { deprecated-variant }
2955       {#7} { \token_to_str:N #5 } {#1} {#2}
2956     }
2957   }

```

(End definition for __cs_generate_variant_loop:nNwN and others.)

__cs_generate_variant_same:N When the base and variant letters are identical, don't do any expansion. For most argument types, we can use the n-type no-expansion, but the N and p types require a slightly different behaviour with respect to braces. For V-type this function could output N to avoid adding useless braces but that is not a problem.

```

2958   \cs_new:Npn \__cs_generate_variant_same:N #1
2959   {
2960     \if:w N #1 #1 \else:
2961       \if:w p #1 #1 \else:
2962         \token_to_str:N n
2963         \if:w n #1 \else:
2964           \__cs_generate_variant_loop_special:NNwNNnn #1#1
2965         \fi:
2966       \fi:
2967     \fi:
2968   }

```

(End definition for __cs_generate_variant_same:N.)

__cs_generate_variant:wwNN If the variant form has already been defined, log its existence (provided log-functions is active). Otherwise, make sure that the \exp_args:N #3 form is defined, and if it contains x, change __cs_tmp:w locally to \cs_new_protected:Npx. Then define the variant by combining the \exp_args:N #3 variant and the base function.

```

2969   \cs_new_protected:Npn \__cs_generate_variant:wwNN

```

```

2970     #1 \s__cs_mark #2 \s__cs_stop #3#4
2971   {
2972     #2
2973     \cs_if_free:NT #4
2974     {
2975       \group_begin:
2976       \__cs_generate_internal_variant:n {#1}
2977       \__cs_tmp:w #4 { \exp_not:c { exp_args:N #1 } \exp_not:N #3 }
2978       \group_end:
2979     }
2980   }

```

(End definition for __cs_generate_variant:wwNN.)

__cs_generate_internal_variant:n
__cs_generate_internal_variant_loop:n

First test for the presence of x (this is where working with strings makes our lives easier), as the result should be protected, and the next variant to be defined using that internal variant should be protected (done by setting __cs_tmp:w). Then call __cs_generate_internal_variant:NNn with arguments \cs_new_protected:cpn \use:x (for protected) or \cs_new:cpn \tex_expanded:D (expandable) and the signature. If p appears in the signature, or if the function to be defined is expandable and the primitive \expanded is not available, or if there are more than 8 arguments, call some fall-back code that just puts the appropriate \:: commands. Otherwise, call __cs_generate_internal_one_go:NNn to construct the \exp_args:N... function as a macro taking up to 9 arguments and expanding them using \use:x or \tex_expanded:D.

```

2981 \cs_new_protected:Npx \__cs_generate_internal_variant:n #1
2982 {
2983   \exp_not:N \__cs_generate_internal_variant:wwnNwn
2984   #1 \s__cs_mark
2985   { \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx }
2986   \cs_new_protected:cpn
2987   \use:x
2988   \token_to_str:N x \s__cs_mark
2989   { }
2990   \cs_new:cpn
2991   \exp_not:N \tex_expanded:D
2992   \s__cs_stop
2993   {#1}
2994 }
2995 \exp_last_unbraced:NNNNo
2996 \cs_new_protected:Npn \__cs_generate_internal_variant:wwnNwn #1
2997 { \token_to_str:N x } #2 \s__cs_mark #3#4#5#6 \s__cs_stop #7
2998 {
2999   #3
3000   \cs_if_free:cT { exp_args:N #7 }
3001   { \__cs_generate_internal_variant:NNn #4 #5 {#7} }
3002 }
3003 \cs_set_protected:Npn \__cs_tmp:w #1
3004 {
3005   \cs_new_protected:Npn \__cs_generate_internal_variant:NNn ##1##2##3
3006   {
3007     \if_catcode:w X \use_none:nnnnnnnn ##3
3008     \prg_do_nothing: \prg_do_nothing: \prg_do_nothing:
3009     \prg_do_nothing: \prg_do_nothing: \prg_do_nothing:
3010     \prg_do_nothing: \prg_do_nothing: X

```

```

3011         \exp_after:wN \_cs_generate_internal_test:Nw \exp_after:wN ##2
3012     \else:
3013         \exp_after:wN \_cs_generate_internal_test_aux:w \exp_after:wN #1
3014     \fi:
3015     ##3
3016     \s__cs_mark
3017     {
3018         \use:x
3019         {
3020             ##1 { exp_args:N ##3 }
3021             { \_cs_generate_internal_variant_loop:n ##3 { : \use_i:nn } }
3022         }
3023     }
3024     #1
3025     \s__cs_mark
3026     { \exp_not:n { \_cs_generate_internal_one_go:NNn ##1 ##2 {##3} } }
3027     \s__cs_stop
3028 }
3029 \cs_new_protected:Npn \_cs_generate_internal_test_aux:w
3030     ##1 #1 ##2 \s__cs_mark ##3 ##4 \s__cs_stop {##3}
3031 \cs_if_exist:NTF \tex_expanded:D
3032 {
3033     \cs_new_eq:NN \_cs_generate_internal_test:Nw
3034         \_cs_generate_internal_test_aux:w
3035 }
3036 {
3037     \cs_new_protected:Npn \_cs_generate_internal_test:Nw ##1
3038     {
3039         \if_meaning:w \tex_expanded:D ##1
3040         \exp_after:wN \_cs_generate_internal_test_aux:w
3041         \exp_after:wN #1
3042         \else:
3043         \exp_after:wN \_cs_generate_internal_test_aux:w
3044         \fi:
3045     }
3046 }
3047 }
3048 \exp_args:No \_cs_tmp:w { \token_to_str:N p }
3049 \cs_new_protected:Npn \_cs_generate_internal_one_go:NNn #1#2#3
3050 {
3051     \_cs_generate_internal_loop:nwnnw
3052     { \exp_not:N ##1 } 1 . { } { }
3053     #3 { ? \_cs_generate_internal_end:w } X ;
3054     23456789 { ? \_cs_generate_internal_long:w } ;
3055     #1 #2 {##3}
3056 }
3057 \cs_new_protected:Npn \_cs_generate_internal_loop:nwnnw #1#2 . #3#4#5#6 ; #7
3058 {
3059     \use_none:n #5
3060     \use_none:n #7
3061     \cs_if_exist_use:cF { __cs_generate_internal_#5:NN }
3062     { \_cs_generate_internal_other:NN }
3063     #5 #7
3064     #7 .

```

```

3065     { #3 #1 } { #4 ## #2 }
3066     #6 ;
3067 }
3068 \cs_new_protected:Npn \__cs_generate_internal_N:NN #1#2
3069 { \__cs_generate_internal_loop:nwnnw { \exp_not:N ###2 } }
3070 \cs_new_protected:Npn \__cs_generate_internal_c:NN #1#2
3071 { \exp_args:No \__cs_generate_internal_loop:nwnnw { \exp_not:c {###2} } }
3072 \cs_new_protected:Npn \__cs_generate_internal_n:NN #1#2
3073 { \__cs_generate_internal_loop:nwnnw { { \exp_not:n {###2} } } }
3074 \cs_new_protected:Npn \__cs_generate_internal_x:NN #1#2
3075 { \__cs_generate_internal_loop:nwnnw { {###2} } }
3076 \cs_new_protected:Npn \__cs_generate_internal_other:NN #1#2
3077 {
3078     \exp_args:No \__cs_generate_internal_loop:nwnnw
3079     {
3080         \exp_after:wN
3081         {
3082             \exp:w \exp_args:NNc \exp_after:wN \exp_end:
3083             { exp_not:#1 } {###2}
3084         }
3085     }
3086 }
3087 \cs_new_protected:Npn \__cs_generate_internal_end:w #1 . #2#3#4 ; #5 ; #6#7#8
3088 { #6 { exp_args:N #8 } #3 { #7 {#2} } }
3089 \cs_new_protected:Npn \__cs_generate_internal_long:w #1 N #2#3 . #4#5#6#
3090 {
3091     \exp_args:Nx \__cs_generate_internal_long:nnnNNn
3092     { \__cs_generate_internal_variant_loop:n #2 #6 { : \use_i:nn } }
3093     {#4} {#5}
3094 }
3095 \cs_new:Npn \__cs_generate_internal_long:nnnNNn #1#2#3#4 ; ; #5#6#7
3096 { #5 { exp_args:N #7 } #3 { #6 { \exp_not:n {#1} {#2} } } }

```

This command grabs char by char outputting \::#1 (not expanded further). We avoid tests by putting a trailing : \use_i:nn, which leaves \cs_end: and removes the looping macro. The colon is in fact also turned into \::: so that the required structure for \exp_args:N... commands is correctly terminated.

```

3097 \cs_new:Npn \__cs_generate_internal_variant_loop:n #1
3098 {
3099     \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
3100     \__cs_generate_internal_variant_loop:n
3101 }

```

(End definition for __cs_generate_internal_variant:n and __cs_generate_internal_variant_loop:n.)

\prg_generate_conditional_variant:Nnn

```

\__cs_generate_variant:nnNnn 3102 \cs_new_protected:Npn \prg_generate_conditional_variant:Nnn #1
\__cs_generate_variant:w      3103 {
\__cs_generate_variant:n      3104     \use:x
    \__cs_generate_variant_p_form:nnn 3105     {
    \__cs_generate_variant_T_form:nnn 3106         \__cs_generate_variant:nnNnn
    \__cs_generate_variant_F_form:nnn 3107         \cs_split_function:N #1
    \__cs_generate_variant_TF_form:nnn 3108     }
3109 }

```

```

3110 \cs_new_protected:Npn \__cs_generate_variant:nnNnn #1#2#3#4#5
3111 {
3112   \if_meaning:w \c_false_bool #3
3113     \__kernel_msg_error:nnx { kernel } { missing-colon }
3114     { \token_to_str:c {#1} }
3115     \__cs_use_i_delimit_by_s_stop:nw
3116   \fi:
3117   \exp_after:wN \__cs_generate_variant:w
3118   \tl_to_str:n {#5} , \scan_stop: , \q__cs_recursion_stop
3119   \__cs_use_none_delimit_by_s_stop:w \s__cs_mark {#1} {#2} {#4} \s__cs_stop
3120 }
3121 \cs_new_protected:Npn \__cs_generate_variant:w
3122   #1 , #2 \s__cs_mark #3#4#5
3123 {
3124   \if_meaning:w \scan_stop: #1 \scan_stop:
3125     \if_meaning:w \q__cs_nil #1 \q__cs_nil
3126     \use_i:nnn
3127   \fi:
3128   \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
3129   \else:
3130     \cs_if_exist_use:cTF { __cs_generate_variant_#1_form:nnn }
3131     { {#3} {#4} {#5} }
3132     {
3133       \__kernel_msg_error:nnxx
3134       { kernel } { conditional-form-unknown }
3135       {#1} { \token_to_str:c { #3 : #4 } }
3136     }
3137   \fi:
3138   \__cs_generate_variant:w #2 \s__cs_mark {#3} {#4} {#5}
3139 }
3140 \cs_new_protected:Npn \__cs_generate_variant_p_form:nnn #1#2
3141 { \cs_generate_variant:cn { #1_p : #2 } }
3142 \cs_new_protected:Npn \__cs_generate_variant_T_form:nnn #1#2
3143 { \cs_generate_variant:cn { #1 : #2 T } }
3144 \cs_new_protected:Npn \__cs_generate_variant_F_form:nnn #1#2
3145 { \cs_generate_variant:cn { #1 : #2 F } }
3146 \cs_new_protected:Npn \__cs_generate_variant_TF_form:nnn #1#2
3147 { \cs_generate_variant:cn { #1 : #2 TF } }

```

(End definition for \prg_generate_conditional_variant:Nnn and others. This function is documented on page 108.)

\exp_args_generate:n This function is not used in the kernel hence we can use functions that are defined in later modules. It also does not need to be fast so use inline mappings. For each requested variant we check that there are no characters besides NnpcofVvx, in particular that there are no spaces. Then we just call the internal function.

```

3148 \cs_new_protected:Npn \exp_args_generate:n #1
3149 {
3150   \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
3151   {
3152     \str_map_inline:nn {##1}
3153     {
3154       \str_if_in:nnF { NnpcofVvx } {####1}
3155       {

```

```

3156         \__kernel_msg_error:nnnn { kernel } { invalid-exp-args }
3157         {####1} {##1}
3158         \str_map_break:n { \use_none:nn }
3159     }
3160 }
3161 \__cs_generate_internal_variant:n {##1}
3162 }
3163 }

```

(End definition for `\exp_args_generate:n`. This function is documented on page 266.)

6.8 Definitions with the automated technique

Some of these could be done more efficiently, but the complexity of coding then becomes an issue. Notice that the auto-generated functions actually take no arguments themselves.

Here are the actual function definitions, using the helper functions above. The group is used because `__cs_generate_internal_variant:n` redefines `__cs_tmp:w` locally.

```

\exp_args:Nnc \exp_args:Nno \exp_args:NnV \exp_args:Nnv \exp_args:Nne
\exp_args:Nnf \exp_args:Noc \exp_args:Noo \exp_args:Nof \exp_args:NVo
\exp_args:Nfo \exp_args:Nff \exp_args:Nee \exp_args:NNx \exp_args:Ncx
\exp_args:Nnx \exp_args:Nox \exp_args:Nxo \exp_args:Nxx
3164 \cs_set_protected:Npn \__cs_tmp:w #1
3165 {
3166     \group_begin:
3167     \exp_args:No \__cs_generate_internal_variant:n
3168     { \tl_to_str:n {#1} }
3169     \group_end:
3170 }
3171 \__cs_tmp:w { nc }
3172 \__cs_tmp:w { no }
3173 \__cs_tmp:w { nV }
3174 \__cs_tmp:w { nv }
3175 \__cs_tmp:w { ne }
3176 \__cs_tmp:w { nf }
3177 \__cs_tmp:w { oc }
3178 \__cs_tmp:w { oo }
3179 \__cs_tmp:w { of }
3180 \__cs_tmp:w { Vo }
3181 \__cs_tmp:w { fo }
3182 \__cs_tmp:w { ff }
3183 \__cs_tmp:w { ee }
3184 \__cs_tmp:w { Nx }
3185 \__cs_tmp:w { cx }
3186 \__cs_tmp:w { nx }
3187 \__cs_tmp:w { ox }
3188 \__cs_tmp:w { xo }
3189 \__cs_tmp:w { xx }

```

(End definition for `\exp_args:Nnc` and others. These functions are documented on page 31.)

```

\exp_args:NNcf \exp_args:NNno \exp_args:NNnV \exp_args:NNoo
\exp_args:NNVV \exp_args:Ncno \exp_args:NcnV \exp_args:Ncoo
\exp_args:NcVV \exp_args:Nnnc \exp_args:Nnno \exp_args:Nnnf
\exp_args:Nnff \exp_args:Nooo \exp_args:Noof \exp_args:Nffo
\exp_args:Neee

```

```

3196 \__cs_tmp:w { cnV }
3197 \__cs_tmp:w { coo }
3198 \__cs_tmp:w { cVV }
3199 \__cs_tmp:w { nnc }
3200 \__cs_tmp:w { nno }
3201 \__cs_tmp:w { nnf }
3202 \__cs_tmp:w { nff }
3203 \__cs_tmp:w { ooo }
3204 \__cs_tmp:w { oof }
3205 \__cs_tmp:w { ffo }
3206 \__cs_tmp:w { eee }
3207 \__cs_tmp:w { NNx }
3208 \__cs_tmp:w { Nnx }
3209 \__cs_tmp:w { Nox }
3210 \__cs_tmp:w { nnx }
3211 \__cs_tmp:w { nox }
3212 \__cs_tmp:w { ccx }
3213 \__cs_tmp:w { cnx }
3214 \__cs_tmp:w { oox }

```

(End definition for `\exp_args:NNcf` and others. These functions are documented on page 32.)

```

3215 </package>

```

7 l3quark implementation

The following test files are used for this code: `m3quark001.lvt`.

```

3216 <*package>

```

7.1 Quarks

```

3217 <@@=quark>

```

`\quark_new:N` Allocate a new quark.

```

3218 \cs_new_protected:Npn \quark_new:N #1
3219 {
3220   \__kernel_chk_if_free_cs:N #1
3221   \cs_gset_nopar:Npn #1 {#1}
3222 }

```

(End definition for `\quark_new:N`. This function is documented on page 38.)

`\q_nil` Some “public” quarks. `\q_stop` is an “end of argument” marker, `\q_nil` is a empty value and `\q_no_value` marks an empty argument.

```

\q_mark
\q_no_value
\q_stop
3223 \quark_new:N \q_nil
3224 \quark_new:N \q_mark
3225 \quark_new:N \q_no_value
3226 \quark_new:N \q_stop

```

(End definition for `\q_nil` and others. These variables are documented on page 39.)

`\q_recursion_tail` Quarks for ending recursions. Only ever used there! `\q_recursion_tail` is appended to whatever list structure we are doing recursion on, meaning it is added as a proper list item with whatever list separator is in use. `\q_recursion_stop` is placed directly after the list.

```
3227 \quark_new:N \q_recursion_tail
3228 \quark_new:N \q_recursion_stop
```

(End definition for \q_recursion_tail and \q_recursion_stop. These variables are documented on page 39.)

`\s__quark` Private scan mark used in `l3quark`. We don't have `l3scan` yet, so we declare the scan mark here and add it to the scan mark pool later.

```
3229 \cs_new_eq:NN \s__quark \scan_stop:
```

(End definition for \s__quark.)

`\q__quark_nil` Private quark use for some tests.

```
3230 \quark_new:N \q__quark_nil
```

(End definition for \q__quark_nil.)

`\quark_if_recursion_tail_stop:N` When doing recursions, it is easy to spend a lot of time testing if the end marker has been found. To avoid this, a dedicated end marker is used each time a recursion is set up. Thus if the marker is found everything can be wrapper up and finished off. The simple case is when the test can guarantee that only a single token is being tested. In this case, there is just a dedicated copy of the standard quark test. Both a gobbling version and one inserting end code are provided.

```
3231 \cs_new:Npn \quark_if_recursion_tail_stop:N #1
3232 {
3233   \if_meaning:w \q_recursion_tail #1
3234   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
3235   \fi:
3236 }
3237 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
3238 {
3239   \if_meaning:w \q_recursion_tail #1
3240   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
3241   \else:
3242   \exp_after:wN \use_none:n
3243   \fi:
3244 }
```

(End definition for \quark_if_recursion_tail_stop:N and \quark_if_recursion_tail_stop_do:Nn. These functions are documented on page 40.)

`\quark_if_recursion_tail_stop:n` See `\quark_if_nil:nTF` for the details. Expanding `__quark_if_recursion_tail:w` once in front of the tokens chosen here gives an empty result if and only if `#1` is exactly `\q_recursion_tail`.

```
\quark_if_recursion_tail_stop:0
\quark_if_recursion_tail_stop:0n
\quark_if_recursion_tail_stop:0n
\__quark_if_recursion_tail:w
3245 \cs_new:Npn \quark_if_recursion_tail_stop:n #1
3246 {
3247   \tl_if_empty:oTF
3248   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
3249   { \use_none_delimit_by_q_recursion_stop:w }
3250   { }
```

```

3251 }
3252 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
3253 {
3254   \tl_if_empty:oTF
3255   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
3256   { \use_i_delimit_by_q_recursion_stop:nw }
3257   { \use_none:n }
3258 }
3259 \cs_new:Npn \__quark_if_recursion_tail:w
3260   #1 \q_recursion_tail #2 ? #3 ?! { #1 #2 }
3261 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
3262 \cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn { o }

```

(End definition for `\quark_if_recursion_tail_stop:n`, `\quark_if_recursion_tail_stop_do:nn`, and `__quark_if_recursion_tail:w`. These functions are documented on page 40.)

`\quark_if_recursion_tail_break:NN` Analogues of the `\quark_if_recursion_tail_stop...` functions. Break the mapping
`\quark_if_recursion_tail_break:nN` using #2.

```

3263 \cs_new:Npn \quark_if_recursion_tail_break:NN #1#2
3264 {
3265   \if_meaning:w \q_recursion_tail #1
3266   \exp_after:wN #2
3267   \fi:
3268 }
3269 \cs_new:Npn \quark_if_recursion_tail_break:nN #1#2
3270 {
3271   \tl_if_empty:oT
3272   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
3273   {#2}
3274 }

```

(End definition for `\quark_if_recursion_tail_break:NN` and `\quark_if_recursion_tail_break:nN`. These functions are documented on page 40.)

`\quark_if_nil_p:N` Here we test if we found a special quark as the first argument. We better start with
`\quark_if_nil:N \underline{TF}` `\q_no_value` as the first argument since the whole thing may otherwise loop if #1 is
`\quark_if_no_value_p:N` wrongly given a string like `aabc` instead of a single token.⁷

```

\quark_if_no_value_p:c 3275 \prg_new_conditional:Npnn \quark_if_nil:N #1 { p, T , F , TF }
\quark_if_no_value:N $\underline{TF}$  3276 {
\quark_if_no_value:c $\underline{TF}$  3277   \if_meaning:w \q_nil #1
3278   \prg_return_true:
3279   \else:
3280   \prg_return_false:
3281   \fi:
3282 }
3283 \prg_new_conditional:Npnn \quark_if_no_value:N #1 { p, T , F , TF }
3284 {
3285   \if_meaning:w \q_no_value #1
3286   \prg_return_true:
3287   \else:
3288   \prg_return_false:
3289   \fi:
3290 }

```

⁷It may still loop in special circumstances however!

```

3291 \prg_generate_conditional_variant:Nnn \quark_if_no_value:N
3292 { c } { p , T , F , TF }

```

(End definition for \quark_if_nil:NTF and \quark_if_no_value:NTF. These functions are documented on page 39.)

\quark_if_nil_p:n Let us explain \quark_if_nil:n(TF). Expanding __quark_if_nil:w once is safe thanks to the trailing \q_nil ??!. The result of expanding once is empty if and only if both delimited arguments #1 and #2 are empty and #3 is delimited by the last tokens ?!. Thanks to the leading {}, the argument #1 is empty if and only if the argument of \quark_if_nil:n starts with \q_nil. The argument #2 is empty if and only if this \q_nil is followed immediately by ? or by {}, coming either from the trailing tokens in the definition of \quark_if_nil:n, or from its argument. In the first case, __quark_if_nil:w is followed by {} \q_nil {} ? ! \q_nil ??!, hence #3 is delimited by the final ?!, and the test returns true as wanted. In the second case, the result is not empty since the first ?! in the definition of \quark_if_nil:n stop #3. The auxiliary here is the same as __tl_if_empty_if:o, with the same comments applying.

```

3293 \prg_new_conditional:Npnn \quark_if_nil:n #1 { p , T , F , TF }
3294 {
3295   \__quark_if_empty_if:o
3296   { \__quark_if_nil:w {} #1 {} ? ! \q_nil ? ? ! }
3297   \prg_return_true:
3298   \else:
3299     \prg_return_false:
3300   \fi:
3301 }
3302 \cs_new:Npn \__quark_if_nil:w #1 \q_nil #2 ? #3 ? ! { #1 #2 }
3303 \prg_new_conditional:Npnn \quark_if_no_value:n #1 { p , T , F , TF }
3304 {
3305   \__quark_if_empty_if:o
3306   { \__quark_if_no_value:w {} #1 {} ? ! \q_no_value ? ? ! }
3307   \prg_return_true:
3308   \else:
3309     \prg_return_false:
3310   \fi:
3311 }
3312 \cs_new:Npn \__quark_if_no_value:w #1 \q_no_value #2 ? #3 ? ! { #1 #2 }
3313 \prg_generate_conditional_variant:Nnn \quark_if_nil:n
3314 { V , o } { p , TF , T , F }
3315 \cs_new:Npn \__quark_if_empty_if:o #1
3316 {
3317   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
3318   \__kernel_tl_to_str:w \exp_after:wN {#1} \q_nil
3319 }

```

(End definition for \quark_if_nil:nTF and others. These functions are documented on page 39.)

__kernel_quark_new_test:N The function __kernel_quark_new_test:N defines #1 in a similar way as \quark_if_recursion_tail... functions (as described below), using \q_<namespace>-recursion_tail as the test quark and \q_<namespace>-recursion_stop as the delimiter quark, where the <namespace> is determined as the first _-delimited part in #1.

There are six possible function types which this function can define, and which is defined depends on the signature of the function being defined:

:n gives an analogue of \quark_if_recursion_tail_stop:n
:nn gives an analogue of \quark_if_recursion_tail_stop_do:nn
:nN gives an analogue of \quark_if_recursion_tail_break:nN
:N gives an analogue of \quark_if_recursion_tail_stop:N
:Nn gives an analogue of \quark_if_recursion_tail_stop_do:Nn
:NN gives an analogue of \quark_if_recursion_tail_break:NN

Any other signature causes an error, as does a function without signature.

Similar to __kernel_quark_new_test:N, but defines quark branching conditionals like __kernel_quark_new_conditional:Nn \quark_if_nil:nTF that test for the quark \q_{\langle namespace \rangle}_{\langle name \rangle}. The \langle namespace \rangle and \langle name \rangle are determined from the conditional #1, which must take the rather rigid form __{\langle namespace \rangle}_quark_if_{\langle name \rangle}:\langle arg spec \rangle. There are only two cases for the \langle arg spec \rangle here:

:n gives an analogue of \quark_if_nil:n(TF)
:N gives an analogue of \quark_if_nil:N(TF)

Any other signature causes an error, as does a function without signature. We use low-level emptiness tests as l3tl is not available yet when these functions are used; thankfully we only care about whether strings are empty so a simple \if_meaning:w \q_nil \langle string \rangle \q_nil suffices.

```

3320 \cs_new_protected:Npn \__kernel_quark_new_test:N #1
3321 { \__quark_new_test_aux:Nx #1 { \__quark_module_name:N #1 } }
\__quark_new_test:NNNn 3322 \cs_new_protected:Npn \__quark_new_test_aux:Nn #1 #2
\__quark_new_test:Nccn 3323 {
\__quark_new_test_aux:nnNNnnnn 3324 \if_meaning:w \q_nil #2 \q_nil
\__quark_new_conditional:Nnnn 3325 \__kernel_msg_error:nxx { kernel } { invalid-quark-function }
\__quark_new_conditional:Nxxn 3326 { \token_to_str:N #1 }
3327 \else:
3328 \__quark_new_test:Nccn #1
3329 { q_{#2_recursion_tail} } { q_{#2_recursion_stop} } { __#2 }
3330 \fi:
3331 }
3332 \cs_generate_variant:Nn \__quark_new_test_aux:Nn { Nx }
3333 \cs_new_protected:Npn \__quark_new_test:NNNn #1
3334 {
3335 \exp_last_unbraced:Nf \__quark_new_test_aux:nnNNnnnn
3336 { \cs_split_function:N #1 }
3337 #1 { test }
3338 }
3339 \cs_generate_variant:Nn \__quark_new_test:NNNn { Ncc }
3340 \cs_new_protected:Npn \__kernel_quark_new_conditional:Nn #1
3341 {
3342 \__quark_new_conditional:Nxxn #1
3343 { \__quark_quark_conditional_name:N #1 }
3344 { \__quark_module_name:N #1 }

```

```

3345 }
3346 \cs_new_protected:Npn \__quark_new_conditional:Nnnn #1#2#3#4
3347 {
3348   \if_meaning:w \q_nil #2 \q_nil
3349   \__kernel_msg_error:nxx { kernel } { invalid-quark-function }
3350   { \token_to_str:N #1 }
3351   \else:
3352   \if_meaning:w \q_nil #3 \q_nil
3353   \__kernel_msg_error:nxx { kernel } { invalid-quark-function }
3354   { \token_to_str:N #1 }
3355   \else:
3356   \exp_last_unbraced:Nf \__quark_new_test_aux:nnNNnnnn
3357   { \cs_split_function:N #1 }
3358   #1 { conditional }
3359   {#2} {#3} {#4}
3360   \fi:
3361   \fi:
3362 }
3363 \cs_generate_variant:Nn \__quark_new_conditional:Nnnn { Nxx }
3364 \cs_new_protected:Npn \__quark_new_test_aux:nnNNnnnn #1 #2 #3 #4 #5
3365 {
3366   \cs_if_exist_use:cTF { __quark_new_#5_#2:Nnnn } { #4 }
3367   {
3368     \__kernel_msg_error:nxxx { kernel } { invalid-quark-function }
3369     { \token_to_str:N #4 } {#2}
3370     \use_none:nnn
3371   }
3372 }

```

(End definition for __kernel_quark_new_test:N and others.)

```

\__quark_new_test_n:Nnnn
\__quark_new_test_nn:Nnnn
\__quark_new_test_N:Nnnn
\__quark_new_test_Nn:Nnnn
\__quark_new_test_NN:Nnnn
\__quark_new_test_NN:Nnnn

```

These macros implement the six possibilities mentioned above, passing the right arguments to __quark_new_test_aux_do:nnNNnnnnNNn, which defines some auxiliaries, and then to __quark_new_test_define_tl:nNnNNn (:n(n) variants) or to __quark_new_test_define_ifx:nNnNNn (:N(n)) which define the main conditionals.

```

3373 \cs_new_protected:Npn \__quark_new_test_n:Nnnn #1 #2 #3 #4
3374 {
3375   \__quark_new_test_aux_do:nnNNnnnnNNn {#4} #2 #3 { none } { } { } { }
3376   \__quark_new_test_define_tl:nNnNNn #1 { }
3377 }
3378 \cs_new_protected:Npn \__quark_new_test_nn:Nnnn #1 #2 #3 #4
3379 {
3380   \__quark_new_test_aux_do:nnNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
3381   \__quark_new_test_define_tl:nNnNNn #1 { \use_none:n }
3382 }
3383 \cs_new_protected:Npn \__quark_new_test_nN:Nnnn #1 #2 #3 #4
3384 {
3385   \__quark_new_test_aux_do:nnNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
3386   \__quark_new_test_define_break_tl:nNNNNn #1 { }
3387 }
3388 \cs_new_protected:Npn \__quark_new_test_N:Nnnn #1 #2 #3 #4
3389 {
3390   \__quark_new_test_aux_do:nnNNnnnnNNn {#4} #2 #3 { none } { } { } { }
3391   \__quark_new_test_define_ifx:nNnNNn #1 { }

```

```

3392 }
3393 \cs_new_protected:Npn \__quark_new_test_Nn:Nnnn #1 #2 #3 #4
3394 {
3395   \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
3396   \__quark_new_test_define_ifx:nNnNNn #1
3397   { \else: \exp_after:wN \use_none:n }
3398 }
3399 \cs_new_protected:Npn \__quark_new_test_NN:Nnnn #1 #2 #3 #4
3400 {
3401   \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
3402   \__quark_new_test_define_break_ifx:nNNNNn #1 { }
3403 }

```

(End definition for __quark_new_test_n:Nnnn and others.)

__quark_new_test_aux_do:nNNnnnnNNn __quark_new_test_aux_do:nNNnnnnNNn makes the control sequence names which will be used by __quark_test_define_aux:NNNNnnNNn, and then later by __quark_new_test_define_tl:nNnNNn or __quark_new_test_define_ifx:nNnNNn. The control sequences defined here are analogous to __quark_if_recursion_tail:w and to \use_-(none|i)_delimit_by_q_recursion_stop:(|n)w.

The name is composed by the name-space and the name of the quarks. Suppose __kernel_quark_new_test:N was used with:

```
\__kernel_quark_new_test:N \__test_quark_tail:n
```

then the first auxiliary will be __test_quark_recursion_tail:w, and the second one will be __test_use_none_delimit_by_q_recursion_stop:w.

Note that the actual quarks are *not* defined here. They should be defined separately using \quark_new:N.

```

3404 \cs_new_protected:Npn \__quark_new_test_aux_do:nNNnnnnNNn #1 #2 #3 #4 #5
3405 {
3406   \exp_args:Ncc \__quark_test_define_aux:NNNNnnNNn
3407   { #1 \quark_recursion_tail:w }
3408   { #1 \use_#4_delimit_by_q_recursion_stop: #5 w }
3409   #2 #3
3410 }
3411 \cs_new_protected:Npn \__quark_test_define_aux:NNNNnnNNn #1 #2 #3 #4 #5 #6 #7
3412 {
3413   \cs_gset:Npn #1 ##1 #3 ##2 ? ##3 ?! { ##1 ##2 }
3414   \cs_gset:Npn #2 ##1 #6 #4 {#5}
3415   #7 {##1} #1 #2 #3
3416 }

```

(End definition for __quark_new_test_aux_do:nNNnnnnNNn and __quark_test_define_aux:NNNNnnNNn.)

```

\__quark_new_test_define_tl:nNnNNn
\__quark_new_test_define_ifx:nNnNNn
\__quark_new_test_define_break_tl:nNNNNn
\__quark_new_test_define_break_ifx:nNNNNn

```

Finally, these two macros define the main conditional function using what's been set up before.

```

3417 \cs_new_protected:Npn \__quark_new_test_define_tl:nNnNNn #1 #2 #3 #4 #5 #6
3418 {
3419   \cs_new:Npn #5 #1
3420   {
3421     \tl_if_empty:oTF
3422     { #2 {} ##1 {} ?! #4 ??! }
3423     {#3} {#6}

```

```

3424     }
3425   }
3426 \cs_new_protected:Npn \__quark_new_test_define_ifx:nNnNNn #1 #2 #3 #4 #5 #6
3427 {
3428   \cs_new:Npn #5 #1
3429   {
3430     \if_meaning:w #4 ##1
3431     \exp_after:wN #3
3432     #6
3433     \fi:
3434   }
3435 }
3436 \cs_new_protected:Npn \__quark_new_test_define_break_tl:nNNNNn #1 #2 #3
3437 { \__quark_new_test_define_tl:nNnNNn {##1##2} #2 {##2} }
3438 \cs_new_protected:Npn \__quark_new_test_define_break_ifx:nNNNNn #1 #2 #3
3439 { \__quark_new_test_define_ifx:nNnNNn {##1##2} #2 {##2} }

```

(End definition for __quark_new_test_define_tl:nNnNNn and others.)

__quark_new_conditional_n:Nnnn These macros implement the two possibilities for branching quark conditionals, passing the right arguments to __quark_new_conditional_aux_do:NNnnn, which defines some auxiliaries and defines the main conditionals.

```

3440 \cs_new_protected:Npn \__quark_new_conditional_n:Nnnn
3441 { \__quark_new_conditional_aux_do:NNnnn \use_i:nn }
3442 \cs_new_protected:Npn \__quark_new_conditional_N:Nnnn
3443 { \__quark_new_conditional_aux_do:NNnnn \use_ii:nn }

```

(End definition for __quark_new_conditional_n:Nnnn and __quark_new_conditional_N:Nnnn.)

__quark_new_conditional_aux_do:NNnnn Similar to the previous macros, but branching conditionals only require one auxiliary, so we take a shortcut. In __quark_new_conditional_define:NNNNn, #4 is \use_i:nn to define the n-type function (which needs an auxiliary) and is \use_ii:nn to define the N-type function.

```

3444 \cs_new_protected:Npn \__quark_new_conditional_aux_do:NNnnn #1 #2 #3 #4
3445 {
3446   \exp_args:Ncc \__quark_new_conditional_define:NNNNn
3447   { __ #4 _if_quark_ #3 :w } { q__ #4 _ #3 } #2 #1
3448 }
3449 \cs_new_protected:Npn \__quark_new_conditional_define:NNNNn #1 #2 #3 #4 #5
3450 {
3451   #4 { \cs_gset:Npn #1 ##1 #2 ##2 ? ##3 ?! { ##1 ##2 } } { }
3452   \exp_args:Nno \use:n { \prg_new_conditional:Npnn #3 ##1 {#5} }
3453   {
3454     #4 { \__quark_if_empty_if:o { #1 {} ##1 {} ?! #2 ??! } }
3455     { \if_meaning:w #2 ##1 }
3456     \prg_return_true: \else: \prg_return_false: \fi:
3457   }
3458 }

```

(End definition for __quark_new_conditional_aux_do:NNnnn and __quark_new_conditional_define:NNNNn.)

__quark_module_name:N __quark_module_name:N takes a control sequence and returns its $\langle module \rangle$ name, determined as the first non-empty non-single-character word, separated by _ or :. These rules give the correct result for public functions \langle module \rangle_..., private functions __-\langle module \rangle_..., and variables such as \l_1\langle module \rangle_.... If no valid module is found the

result is an empty string. The approach is to first cut off everything after the (first) : if any is present, then repeatedly grab _-delimited words until finding one of length at least 2 (we use low-level tests as l3tl is not fully available when __kernel_quark_new_test:N is first used. If no *module* is found (such as in \::n) we get the trailing marker \use_none:n {}, which expands to nothing.

```

3459 \cs_set:Npn \__quark_tmp:w #1#2
3460 {
3461   \cs_new:Npn \__quark_module_name:N ##1
3462   {
3463     \exp_last_unbraced:Nf \__quark_module_name:w
3464     { \cs_to_str:N ##1 } #1 \s__quark
3465   }
3466   \cs_new:Npn \__quark_module_name:w ##1 #1 ##2 \s__quark
3467   { \__quark_module_name_loop:w ##1 #2 \use_none:n { } #2 \s__quark }
3468   \cs_new:Npn \__quark_module_name_loop:w ##1 #2
3469   {
3470     \use_i_ii:nnn \if_meaning:w \prg_do_nothing:
3471     ##1 \prg_do_nothing: \prg_do_nothing:
3472     \exp_after:wN \__quark_module_name_loop:w
3473     \else:
3474       \__quark_module_name_end:w ##1
3475     \fi:
3476   }
3477   \cs_new:Npn \__quark_module_name_end:w
3478   ##1 \fi: ##2 \s__quark { \fi: ##1 }
3479 }
3480 \exp_after:wN \__quark_tmp:w \tl_to_str:n { : _ }

```

(End definition for __quark_module_name:N and others.)

_quark_quark_conditional_name:N __quark_quark_conditional_name:N determines the quark name that the quark conditional function ##1 queries, as the part of the function name between _quark_if_ and the trailing :. Again we define it through __quark_tmp:w, which receives : as #1 and _quark_if_ as #2. The auxiliary __quark_quark_conditional_name:w returns the part between the first _quark_if_ and the next :, and we apply this auxiliary to the function name followed by : (in case the function name is lacking a signature), and _quark_if_: so that __quark_quark_conditional_name:N returns an empty string if _quark_if_ is not present.

```

3481 \cs_set:Npn \__quark_tmp:w #1 #2 \s__quark
3482 {
3483   \cs_new:Npn \__quark_quark_conditional_name:N ##1
3484   {
3485     \exp_last_unbraced:Nf \__quark_quark_conditional_name:w
3486     { \cs_to_str:N ##1 } #1 #2 #1 \s__quark
3487   }
3488   \cs_new:Npn \__quark_quark_conditional_name:w
3489   ##1 #2 ##2 #1 ##3 \s__quark {##2}
3490 }
3491 \exp_after:wN \__quark_tmp:w \tl_to_str:n { : _quark_if_ } \s__quark

```

(End definition for __quark_quark_conditional_name:N and __quark_quark_conditional_name:w.)

7.2 Scan marks

3492 `<@@=scan>`

`\g__scan_marks_tl` The list of all scan marks currently declared. No `l3tl` yet, so define this by hand.

3493 `\cs_gset:Npn \g__scan_marks_tl { }`

(End definition for `\g__scan_marks_tl`.)

`\scan_new:N` Check whether the variable is already a scan mark, then declare it to be equal to `\scan_stop`: globally.

```
3494 \cs_new_protected:Npn \scan_new:N #1
3495 {
3496   \tl_if_in:NnTF \g__scan_marks_tl { #1 }
3497   {
3498     \__kernel_msg_error:nxx { kernel } { scanmark-already-defined }
3499     { \token_to_str:N #1 }
3500   }
3501   {
3502     \tl_gput_right:Nn \g__scan_marks_tl {#1}
3503     \cs_new_eq:NN #1 \scan_stop:
3504   }
3505 }
```

(End definition for `\scan_new:N`. This function is documented on page 41.)

`\s_stop` We only declare one scan mark here, more can be defined by specific modules. Can't use `\scan_new:N` yet because `l3tl` isn't loaded, so define `\s_stop` by hand and add it to `\g__scan_marks_tl`. We also add `\s__quark` (declared earlier) to the pool here. Since it lives in a different namespace, a little `l3docstrip` cheating is necessary.

```
3506 \cs_new_eq:NN \s_stop \scan_stop:
3507 \cs_gset_nopar:Npx \g__scan_marks_tl
3508 {
3509   \exp_not:o \g__scan_marks_tl
3510   \s_stop
3511   <@@=quark>
3512   \s__quark
3513   <@@=scan>
3514 }
```

(End definition for `\s_stop`. This variable is documented on page 42.)

`\use_none_delimit_by_s_stop:w` Similar to `\use_none_delimit_by_q_stop:w`.

3515 `\cs_new:Npn \use_none_delimit_by_s_stop:w #1 \s_stop { }`

(End definition for `\use_none_delimit_by_s_stop:w`. This function is documented on page 42.)

3516 `</package>`

8 l3tl implementation

```
3517 \*package>
3518 \@@=tl>
```

A token list variable is a TeX macro that holds tokens. By using the ε -TeX primitive `\unexpanded` inside a TeX `\edef` it is possible to store any tokens, including #, in this way.

8.1 Functions

\tl_new:N Creating new token list variables is a case of checking for an existing definition and doing the definition.

```
\tl_new:c
3519 \cs_new_protected:Npn \tl_new:N #1
3520 {
3521   \__kernel_chk_if_free_cs:N #1
3522   \cs_gset_eq:NN #1 \c_empty_tl
3523 }
3524 \cs_generate_variant:Nn \tl_new:N { c }
```

(End definition for `\tl_new:N`. This function is documented on page 43.)

\tl_const:Nn Constants are also easy to generate.

```
\tl_const:Nx
3525 \cs_new_protected:Npn \tl_const:Nn #1#2
\tl_const:cn
3526 {
\tl_const:cx
3527   \__kernel_chk_if_free_cs:N #1
3528   \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
3529 }
3530 \cs_new_protected:Npn \tl_const:Nx #1#2
3531 {
3532   \__kernel_chk_if_free_cs:N #1
3533   \cs_gset_nopar:Npx #1 {#2}
3534 }
3535 \cs_generate_variant:Nn \tl_const:Nn { c }
3536 \cs_generate_variant:Nn \tl_const:Nx { c }
```

(End definition for `\tl_const:Nn`. This function is documented on page 43.)

\tl_clear:N Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```
\tl_clear:c
\tl_gclear:N
3537 \cs_new_protected:Npn \tl_clear:N #1
\tl_gclear:c
3538 { \tl_set_eq:NN #1 \c_empty_tl }
3539 \cs_new_protected:Npn \tl_gclear:N #1
3540 { \tl_gset_eq:NN #1 \c_empty_tl }
3541 \cs_generate_variant:Nn \tl_clear:N { c }
3542 \cs_generate_variant:Nn \tl_gclear:N { c }
```

(End definition for `\tl_clear:N` and `\tl_gclear:N`. These functions are documented on page 43.)

\tl_clear_new:N Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```
\tl_clear_new:c
\tl_gclear_new:N
3543 \cs_new_protected:Npn \tl_clear_new:N #1
3544 { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
3545 \cs_new_protected:Npn \tl_gclear_new:N #1
3546 { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
3547 \cs_generate_variant:Nn \tl_clear_new:N { c }
3548 \cs_generate_variant:Nn \tl_gclear_new:N { c }
```

(End definition for `\tl_clear_new:N` and `\tl_gclear_new:N`. These functions are documented on page 44.)

`\tl_set_eq:NN` For setting token list variables equal to each other. To allow for patching, the arguments have to be explicit.

`\tl_set_eq:Nc`

`\tl_set_eq:cN` 3549 `\cs_new_protected:Npn \tl_set_eq:NN #1#2 { \cs_set_eq:NN #1 #2 }`

`\tl_set_eq:cc` 3550 `\cs_new_protected:Npn \tl_gset_eq:NN #1#2 { \cs_gset_eq:NN #1 #2 }`

`\tl_gset_eq:NN` 3551 `\cs_generate_variant:Nn \tl_set_eq:NN { cN, Nc, cc }`

`\tl_gset_eq:Nc` 3552 `\cs_generate_variant:Nn \tl_gset_eq:NN { cN, Nc, cc }`

`\tl_gset_eq:cN`

`\tl_gset_eq:cc` (End definition for `\tl_set_eq:NN` and `\tl_gset_eq:NN`. These functions are documented on page 44.)

`\tl_concat:NNN` Concatenating token lists is easy. When checking is turned on, all three arguments must be checked: a token list #2 or #3 equal to `\scan_stop:` would lead to problems later on.

`\tl_concat:ccc`

`\tl_gconcat:NNN` 3553 `\cs_new_protected:Npn \tl_concat:NNN #1#2#3`

`\tl_gconcat:ccc` 3554 `{ \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }`

3555 `\cs_new_protected:Npn \tl_gconcat:NNN #1#2#3`

3556 `{ \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }`

3557 `\cs_generate_variant:Nn \tl_concat:NNN { ccc }`

3558 `\cs_generate_variant:Nn \tl_gconcat:NNN { ccc }`

(End definition for `\tl_concat:NNN` and `\tl_gconcat:NNN`. These functions are documented on page 44.)

`\tl_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

`\tl_if_exist_p:c` 3559 `\prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }`

`\tl_if_exist:NTF` 3560 `\prg_new_eq_conditional:NNn \tl_if_exist:c \cs_if_exist:c { TF , T , F , p }`

`\tl_if_exist:cTF`

(End definition for `\tl_if_exist:NTF`. This function is documented on page 44.)

8.2 Constant token lists

`\c_empty_tl` Never full. We need to define that constant before using `\tl_new:N`.

3561 `\tl_const:Nn \c_empty_tl { }`

(End definition for `\c_empty_tl`. This variable is documented on page 58.)

`\c_novalue_tl` A special marker: as we don't have `\char_generate:nn` yet, has to be created the old-fashioned way.

3562 `\group_begin:`

3563 `\tex_lccode:D 'A = '-`

3564 `\tex_lccode:D 'N = 'N`

3565 `\tex_lccode:D 'V = 'V`

3566 `\tex_lowercase:D`

3567 `{`

3568 `\group_end:`

3569 `\tl_const:Nn \c_novalue_tl { ANoValue-`

3570 `}`

(End definition for `\c_novalue_tl`. This variable is documented on page 58.)

`\c_space_tl` A space as a token list (as opposed to as a character).

3571 `\tl_const:Nn \c_space_tl { ~ }`

(End definition for `\c_space_tl`. This variable is documented on page 58.)

8.3 Adding to token list variables

By using `\exp_not:n` token list variables can contain `#` tokens, which makes the token list registers provided by T_EX more or less redundant. The `\tl_set:No` version is done “by hand” as it is used quite a lot.

```

\tl_set:Nn 3572 \cs_new_protected:Npn \tl_set:Nn #1#2
\tl_set:Nv 3573 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
\tl_set:Nf 3574 \cs_new_protected:Npn \tl_set:Nf #1#2
\tl_set:Nx 3575 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
\tl_set:cn 3576 \cs_new_protected:Npn \tl_set:cn #1#2
\tl_set:cV 3577 { \cs_set_nopar:Npx #1 {#2} }
\tl_set:cv 3578 \cs_new_protected:Npn \tl_gset:Nn #1#2
\tl_set:co 3579 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} } }
\tl_set:cf 3580 \cs_new_protected:Npn \tl_gset:No #1#2
\tl_set:cx 3581 { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} } }
\tl_gset:Nn 3582 \cs_new_protected:Npn \tl_gset:Nn #1#2
\tl_gset:Nv 3583 { \cs_gset_nopar:Npx #1 {#2} }
\tl_gset:Nf 3584 \cs_generate_variant:Nn \tl_set:Nn { NV , Nv , Nf }
\tl_gset:Nx 3585 \cs_generate_variant:Nn \tl_set:Nx { c }
\tl_gset:cn 3586 \cs_generate_variant:Nn \tl_set:Nn { c , co , cV , cv , cf }
\tl_gset:cV 3587 \cs_generate_variant:Nn \tl_gset:Nn { NV , Nv , Nf }
\tl_gset:cv 3588 \cs_generate_variant:Nn \tl_gset:Nx { c }
\tl_gset:co 3589 \cs_generate_variant:Nn \tl_gset:Nn { c , co , cV , cv , cf }

```

(End definition for `\tl_set:Nn` and `\tl_gset:Nn`. These functions are documented on page 44.)

Adding to the left is done directly to gain a little performance.

```

\tl_put_left:Nn 3590 \cs_new_protected:Npn \tl_put_left:Nn #1#2
\tl_put_left:Nv 3591 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\tl_put_left:Nf 3592 \cs_new_protected:Npn \tl_put_left:Nf #1#2
\tl_put_left:Nx 3593 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\tl_put_left:cn 3594 \cs_new_protected:Npn \tl_put_left:cn #1#2
\tl_put_left:cV 3595 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\tl_put_left:cv 3596 \cs_new_protected:Npn \tl_put_left:cV #1#2
\tl_put_left:co 3597 { \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 } }
\tl_put_left:cx 3598 \cs_new_protected:Npn \tl_gput_left:Nn #1#2
\tl_gput_left:Nn 3599 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\tl_gput_left:Nv 3600 \cs_new_protected:Npn \tl_gput_left:Nv #1#2
\tl_gput_left:Nf 3601 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\tl_gput_left:Nx 3602 \cs_new_protected:Npn \tl_gput_left:Nx #1#2
\tl_gput_left:cn 3603 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\tl_gput_left:cV 3604 \cs_new_protected:Npn \tl_gput_left:cV #1#2
\tl_gput_left:cv 3605 { \cs_gset_nopar:Npx #1 { #2 \exp_not:o {#1} } }
\tl_gput_left:co 3606 \cs_generate_variant:Nn \tl_put_left:Nn { c }
\tl_gput_left:cx 3607 \cs_generate_variant:Nn \tl_put_left:Nv { c }
3608 \cs_generate_variant:Nn \tl_put_left:Nf { c }
3609 \cs_generate_variant:Nn \tl_put_left:Nx { c }
3610 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
3611 \cs_generate_variant:Nn \tl_gput_left:Nv { c }
3612 \cs_generate_variant:Nn \tl_gput_left:Nf { c }
3613 \cs_generate_variant:Nn \tl_gput_left:Nx { c }

```

(End definition for `\tl_put_left:Nn` and `\tl_gput_left:Nn`. These functions are documented on page 44.)

\tl_put_right:Nn The same on the right.

```

\tl_put_right:NV 3614 \cs_new_protected:Npn \tl_put_right:Nn #1#2
\tl_put_right:No 3615 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\tl_put_right:Nx 3616 \cs_new_protected:Npn \tl_put_right:NV #1#2
\tl_put_right:cn 3617 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\tl_put_right:cV 3618 \cs_new_protected:Npn \tl_put_right:No #1#2
\tl_put_right:co 3619 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\tl_put_right:cx 3620 \cs_new_protected:Npn \tl_put_right:Nx #1#2
\tl_gput_right:Nn 3621 { \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 } }
\tl_gput_right:NV 3622 \cs_new_protected:Npn \tl_gput_right:Nn #1#2
\tl_gput_right:No 3623 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\tl_gput_right:Nx 3624 \cs_new_protected:Npn \tl_gput_right:NV #1#2
\tl_gput_right:cn 3625 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\tl_gput_right:cV 3626 \cs_new_protected:Npn \tl_gput_right:No #1#2
\tl_gput_right:co 3627 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\tl_gput_right:cx 3628 \cs_new_protected:Npn \tl_gput_right:Nx #1#2
3629 { \cs_gset_nopar:Npx #1 { \exp_not:o {#1} #2 } }
3630 \cs_generate_variant:Nn \tl_put_right:Nn { c }
3631 \cs_generate_variant:Nn \tl_put_right:NV { c }
3632 \cs_generate_variant:Nn \tl_put_right:No { c }
3633 \cs_generate_variant:Nn \tl_put_right:Nx { c }
3634 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
3635 \cs_generate_variant:Nn \tl_gput_right:NV { c }
3636 \cs_generate_variant:Nn \tl_gput_right:No { c }
3637 \cs_generate_variant:Nn \tl_gput_right:Nx { c }

```

(End definition for \tl_put_right:Nn and \tl_gput_right:Nn. These functions are documented on page 44.)

8.4 Internal quarks and quark-query functions

\q__tl_nil Internal quarks.

```

\q__tl_mark 3638 \quark_new:N \q__tl_nil
\q__tl_stop 3639 \quark_new:N \q__tl_mark
3640 \quark_new:N \q__tl_stop

```

(End definition for \q__tl_nil, \q__tl_mark, and \q__tl_stop.)

\q__tl_recursion_tail Internal recursion quarks.

```

\q__tl_recursion_stop 3641 \quark_new:N \q__tl_recursion_tail
3642 \quark_new:N \q__tl_recursion_stop

```

(End definition for \q__tl_recursion_tail and \q__tl_recursion_stop.)

_tl_if_recursion_tail_break:nN Functions to query recursion quarks.

```

\_tl_if_recursion_tail_stop_p:n 3643 \__kernel_quark_new_test:N \_tl_if_recursion_tail_break:nN
\_tl_if_recursion_tail_stop:nTF 3644 \__kernel_quark_new_conditional:Nn \_tl_quark_if_nil:n { TF }

```

(End definition for _tl_if_recursion_tail_break:nN and _tl_if_recursion_tail_stop:nTF.)

8.5 Reassigning token list category codes

`\c__tl_rescan_marker_tl`

The rescanning code needs a special token list containing the same character (chosen here to be a colon) with two different category codes: it cannot appear in the tokens being rescanned since all colons have the same category code.

```
3645 \tl_const:Nx \c__tl_rescan_marker_tl { : \token_to_str:N : }
```

(End definition for `\c__tl_rescan_marker_tl`.)

`\tl_set_rescan:Nnn`

In a group, after some initial setup explained below and the user setup #3 (followed by `\scan_stop:` to be safe), there is a call to `__tl_set_rescan:nNN`. This shared auxiliary defined later distinguishes single-line and multi-line “files”. In the simplest case of multi-line files, it calls (with the same arguments) `__tl_set_rescan_multi:nNN`, whose code is included here to help understand the approach. This function rescans its argument #1, closes the group, and performs the assignment.

`\tl_set_rescan:Nno`

One difficulty when rescanning is that `\scantokens` treats the argument as a file, and without the correct settings a T_EX error occurs:

`\tl_set_rescan:Nnx`

```
! File ended while scanning definition of ...
```

`\tl_set_rescan:cnn`

A related minor issue is a warning due to opening a group before the `\scantokens` and closing it inside that temporary file; we avoid that by setting `\tracingnesting`. The standard solution to the “File ended” error is to grab the rescanned tokens as a delimited argument of an auxiliary, here `__tl_rescan:NNw`, that performs the assignment, then let T_EX “execute” the end of file marker. As usual in delimited arguments we use `\prg_do_nothing:` to avoid stripping an outer set braces: this is removed by using `o`-expanding assignments. The delimiter cannot appear within the rescanned token list because it contains twice the same character, with different catcodes.

`\tl_set_rescan:cno`

`\tl_set_rescan:cnx`

`\tl_gset_rescan:Nnn`

`\tl_gset_rescan:Nno`

`\tl_gset_rescan:Nnx`

`\tl_gset_rescan:cnn`

`\tl_gset_rescan:cno`

`\tl_gset_rescan:cnx`

`\tl_rescan:nn`

`__tl_set_rescan:NNnn`

`__tl_set_rescan_multi:nNN`

`__tl_rescan:NNw`

For `\tl_rescan:nn` we cannot simply call `__tl_set_rescan:NNnn \prg_do_nothing: \use:n` because that would leave the end-of-file marker *after* the result of rescanning. If that rescanned result is code that looks further in the input stream for arguments, it would break.

For multi-line files the only subtlety is that `\newlinechar` should be equal to `\endlinechar` because `\newlinechar` characters become new lines and then become `\endlinechar` characters when writing to an abstract file and reading back. This equality is ensured by setting `\newlinechar` equal to `\endlinechar`. Prior to this, `\endlinechar` is set to `-1` if it was `32` (in particular true after `\ExplSyntaxOn`) to avoid unreasonable line-breaks at every space for instance in error messages triggered by the user setup. Another side effect of reading back from the file is that spaces (catcode 10) are ignored at the beginning of lines, and spaces and tabs (character code 32 and 9) are ignored at the end of lines.

The two `\if_false: ... \fi:` are there to prevent alignment tabs to cause a change of tabular cell while rescanning. We put the “opening” one after `\group_begin:` so that if one accidentally `f`-expands `\tl_set_rescan:Nnn` braces remain balanced. This is essential in `e`-type arguments when `\expanded` is not available.

```
3646 \cs_new_protected:Npn \tl_rescan:nn #1#2
3647 {
3648   \tl_set_rescan:Nnn \l__tl_internal_a_tl {#1} {#2}
3649   \exp_after:wN \tl_clear:N \exp_after:wN \l__tl_internal_a_tl
3650   \l__tl_internal_a_tl
3651 }
3652 \cs_new_protected:Npn \tl_set_rescan:Nnn
```

```

3653 { \_tl\_set\_rescan:NNnn \tl\_set:No }
3654 \cs\_new\_protected:Npn \tl\_gset\_rescan:Nnn
3655 { \_tl\_set\_rescan:NNnn \tl\_gset:No }
3656 \cs\_new\_protected:Npn \_tl\_set\_rescan:NNnn #1#2#3#4
3657 {
3658   \group\_begin:
3659   \if\_false: { \fi:
3660     \int\_set\_eq:NN \tex\_tracingnesting:D \c\_zero\_int
3661     \int\_compare:nNnT \tex\_endlinechar:D = { 32 }
3662     { \int\_set:Nn \tex\_endlinechar:D { -1 } }
3663     \int\_set\_eq:NN \tex\_newlinechar:D \tex\_endlinechar:D
3664     #3 \scan\_stop:
3665     \exp\_args:No \_tl\_set\_rescan:nNN { \tl\_to\_str:n {#4} } #1 #2
3666   \if\_false: } \fi:
3667 }
3668 \cs\_new\_protected:Npn \_tl\_set\_rescan\_multi:nNN #1#2#3
3669 {
3670   \exp\_args:No \tex\_everyeof:D { \c\_tl\_rescan\_marker\_tl }
3671   \exp\_after:wN \_tl\_rescan:NNw
3672   \exp\_after:wN #2
3673   \exp\_after:wN #3
3674   \exp\_after:wN \prg\_do\_nothing:
3675   \tex\_scantokens:D {#1}
3676 }
3677 \exp\_args:Nno \use:n
3678 { \cs\_new:Npn \_tl\_rescan:NNw #1#2#3 } \c\_tl\_rescan\_marker\_tl
3679 {
3680   \group\_end:
3681   #1 #2 {#3}
3682 }
3683 \cs\_generate\_variant:Nn \tl\_set\_rescan:Nnn { Nno , Nnx }
3684 \cs\_generate\_variant:Nn \tl\_set\_rescan:Nnn { c , cno , cnx }
3685 \cs\_generate\_variant:Nn \tl\_gset\_rescan:Nnn { Nno , Nnx }
3686 \cs\_generate\_variant:Nn \tl\_gset\_rescan:Nnn { c , cno }

```

(End definition for `\tl_set_rescan:Nnn` and others. These functions are documented on page 46.)

<pre> _tl_set_rescan:nNN _tl_set_rescan_single:nnNN _tl_set_rescan_single_aux:nnnNN _tl_set_rescan_single_aux:w </pre>	<p>The function <code>_tl_set_rescan:nNN</code> calls <code>_tl_set_rescan_multi:nNN</code> or <code>_tl_set_rescan_single:nnNN { ' }</code> depending on whether its argument is a single-line fragment of code/data or is made of multiple lines by testing for the presence of a <code>\newlinechar</code> character. If <code>\newlinechar</code> is out of range, the argument is assumed to be a single line.</p>
---	--

For a single line, no `\endlinechar` should be added, so it is set to `-1`, and spaces should not be removed. Trailing spaces and tabs are a difficult matter, as `TEX` removes these at a very low level. The only way to preserve them is to rescan not the argument but the argument followed by a character with a reasonable category code. Here, 11 (letter) and 12 (other) are accepted, as these are convenient, suitable for delimiting an argument, and it is very unlikely that none of the ASCII characters are in one of these categories. To avoid selecting one particular character to put at the end, whose category code may have been modified, there is a loop through characters from `'` (ASCII 39) to `~` (ASCII 127). The choice of starting point was made because this is the start of a very long range of characters whose standard category is letter or other, thus minimizing the

number of steps needed by the loop (most often just a single one). If no valid character is found (very rare), fall-back on `__tl_set_rescan_multi:nnn`.

Otherwise, once a valid character is found (let us use `'` in this explanation) run some code very similar to `__tl_set_rescan_multi:nnn` but with `'` added at both ends of the input. Of course, we need to define the auxiliary `__tl_set_rescan_single:NNww` on the fly to remove the additional `'` that is just before `::` (by which we mean `\c__tl_rescan_marker_tl`). Note that the argument must be delimited by `'` with the current catcode; this is done thanks to `\char_generate:nn`. Yet another issue is that the rescanned token list may contain a comment character, in which case the `'` we expected is not there. We fix this as follows: rather than just `::` we set `\everyeof` to `::{<code1>}` `'::<code2>}` `\s__tl_stop`. The auxiliary `__tl_set_rescan_single:NNww` runs the o-expanding assignment, expanding either `<code1>` or `<code2>` before its the main argument `#3`. In the typical case without comment character, `<code1>` is expanded, removing the leading `'`. In the rarer case with comment character, `<code2>` is expanded, calling `__tl_set_rescan_single_aux:w`, which removes the trailing `::{<code1>}` and the leading `'`.

```

3687 \cs_new_protected:Npn \__tl_set_rescan:nnn #1
3688 {
3689   \int_compare:nNnTF \tex_newlinechar:D < 0
3690   { \use_i:nn }
3691   {
3692     \exp_args:Nnf \tl_if_in:nnTF {#1}
3693     { \char_generate:nn { \tex_newlinechar:D } { 12 } }
3694   }
3695   { \__tl_set_rescan_multi:nnn }
3696   {
3697     \int_set:Nn \tex_endlinechar:D { -1 }
3698     \__tl_set_rescan_single:nnNN { ' ' }
3699   }
3700   {#1}
3701 }
3702 \cs_new_protected:Npn \__tl_set_rescan_single:nnNN #1
3703 {
3704   \int_compare:nNnTF
3705   { \char_value_catcode:n {#1} / 2 } = 6
3706   {
3707     \exp_args:Nof \__tl_set_rescan_single_aux:nnnNN
3708     \c__tl_rescan_marker_tl
3709     { \char_generate:nn {#1} { \char_value_catcode:n {#1} } }
3710   }
3711   {
3712     \int_compare:nNnTF {#1} < { '\~ }
3713     {
3714       \exp_args:Nf \__tl_set_rescan_single:nnNN
3715       { \int_eval:n { #1 + 1 } }
3716     }
3717     { \__tl_set_rescan_multi:nnn }
3718   }
3719 }
3720 \cs_new_protected:Npn \__tl_set_rescan_single_aux:nnnNN #1#2#3#4#5
3721 {
3722   \tex_everyeof:D

```



```

3723 {
3724     #1 \use_none:n
3725     #2 #1 { \exp:w \__tl_set_rescan_single_aux:w }
3726     \s__tl_stop
3727 }
3728 \cs_set:Npn \__tl_rescan:NNw ##1##2##3 #2 #1 ##4 ##5 \s__tl_stop
3729 {
3730     \group_end:
3731     ##1 ##2 { ##4 ##3 }
3732 }
3733 \exp_after:wN \__tl_rescan:NNw
3734 \exp_after:wN #4
3735 \exp_after:wN #5
3736 \tex_scantokens:D { #2 #3 #2 }
3737 }
3738 \exp_args:Nno \use:nn
3739 { \cs_new:Npn \__tl_set_rescan_single_aux:w #1 }
3740 \c_tl_rescan_marker_tl #2
3741 { \use_i:nn \exp_end: #1 }

```

8.6 Modifying token list variables

<pre> \tl_replace_all:Nnn \tl_replace_all:cnn \tl_greplace_all:Nnn \tl_greplace_all:cnn \tl_replace_once:Nnn \tl_replace_once:cnn \tl_greplace_once:Nnn \tl_greplace_once:cnn </pre>	<p>All of the <code>replace</code> functions call <code>__tl_replace:NnNNNnn</code> with appropriate arguments. The first two arguments are explained later. The next controls whether the replacement function calls itself (<code>__tl_replace_next:w</code>) or stops (<code>__tl_replace_wrap:w</code>) after the first replacement. Next comes an <code>x</code>-type assignment function <code>\tl_set:Nx</code> or <code>\tl_gset:Nx</code> for local or global replacements. Finally, the three arguments $\langle tl\ var \rangle \{ \langle pattern \rangle \} \{ \langle replacement \rangle \}$ provided by the user. When describing the auxiliary functions below, we denote the contents of the $\langle tl\ var \rangle$ by $\langle token\ list \rangle$.</p>
--	--

(End definition for \tl_replace_all:Nnn and others. These functions are documented on page 45.)

`__tl_replace:NnnNNnn` To implement the actual replacement auxiliary `__tl_replace_auxii:nnNNnn` we need
`__tl_replace_auxi:NnnNNnn` a $\langle \textit{delimiter} \rangle$ with the following properties:
`__tl_replace_auxii:nnNNnn`
`__tl_replace_next:w`

- all occurrences of the $\langle \textit{pattern} \rangle$ #6 in “ $\langle \textit{token list} \rangle \langle \textit{delimiter} \rangle$ ” belong to the $\langle \textit{token list} \rangle$ and have no overlap with the $\langle \textit{delimiter} \rangle$,
- the first occurrence of the $\langle \textit{delimiter} \rangle$ in “ $\langle \textit{token list} \rangle \langle \textit{delimiter} \rangle$ ” is the trailing $\langle \textit{delimiter} \rangle$.

`__tl_replace_wrap:w`

We first find the building blocks for the $\langle \text{delimiter} \rangle$, namely two tokens $\langle A \rangle$ and $\langle B \rangle$ such that $\langle A \rangle$ does not appear in $\#6$ and $\#6$ is not $\langle B \rangle$ (this condition is trivial if $\#6$ has more than one token). Then we consider the delimiters “ $\langle A \rangle$ ” and “ $\langle A \rangle \langle A \rangle^n \langle B \rangle \langle A \rangle^n \langle B \rangle$ ”, for $n \geq 1$, where $\langle A \rangle^n$ denotes n copies of $\langle A \rangle$, and we choose as our $\langle \text{delimiter} \rangle$ the first one which is not in the $\langle \text{token list} \rangle$.

Every delimiter in the set obeys the first condition: $\#6$ does not contain $\langle A \rangle$ hence cannot be overlapping with the $\langle \text{token list} \rangle$ and the $\langle \text{delimiter} \rangle$, and it cannot be within the $\langle \text{delimiter} \rangle$ since it would have to be in one of the two $\langle B \rangle$ hence be equal to this single token (or empty, but this is an error case filtered separately). Given the particular form of these delimiters, for which no prefix is also a suffix, the second condition is actually a consequence of the weaker condition that the $\langle \text{delimiter} \rangle$ we choose does not appear in the $\langle \text{token list} \rangle$. Additionally, the set of delimiters is such that a $\langle \text{token list} \rangle$ of n tokens can contain at most $O(n^{1/2})$ of them, hence we find a $\langle \text{delimiter} \rangle$ with at most $O(n^{1/2})$ tokens in a time at most $O(n^{3/2})$. Bear in mind that these upper bounds are reached only in very contrived scenarios: we include the case “ $\langle A \rangle$ ” in the list of delimiters to try, so that the $\langle \text{delimiter} \rangle$ is simply $\backslash\text{q_tl_mark}$ in the most common situation where neither the $\langle \text{token list} \rangle$ nor the $\langle \text{pattern} \rangle$ contains $\backslash\text{q_tl_mark}$.

Let us now ahead, optimizing for this most common case. First, two special cases: an empty $\langle \text{pattern} \rangle$ $\#6$ is an error, and if $\#1$ is absent from both the $\langle \text{token list} \rangle$ $\#5$ and the $\langle \text{pattern} \rangle$ $\#6$ then we can use it as the $\langle \text{delimiter} \rangle$ through $\backslash\text{__tl_replace_auxii:nNNNnn} \{ \#1 \}$. Otherwise, we end up calling $\backslash\text{__tl_replace:NnNNNnn}$ repeatedly with the first two arguments $\backslash\text{q_tl_mark} \{ ? \}$, $\backslash ? \{ ?? \}$, $\backslash ?? \{ ??? \}$, and so on, until $\#6$ does not contain the control sequence $\#1$, which we take as our $\langle A \rangle$. The argument $\#2$ only serves to collect $?$ characters for $\#1$. Note that the order of the tests means that the first two are done every time, which is wasteful (for instance, we repeatedly test for the emptiness of $\#6$). However, this is rare enough not to matter. Finally, choose $\langle B \rangle$ to be $\backslash\text{q_tl_nil}$ or $\backslash\text{q_tl_stop}$ such that it is not equal to $\#6$.

The $\backslash\text{__tl_replace_auxi:NnnNNNnn}$ auxiliary receives $\{ \langle A \rangle \}$ and $\{ \langle A \rangle^n \langle B \rangle \}$ as its arguments, initially with $n = 1$. If “ $\langle A \rangle \langle A \rangle^n \langle B \rangle \langle A \rangle^n \langle B \rangle$ ” is in the $\langle \text{token list} \rangle$ then increase n and try again. Once it is not anymore in the $\langle \text{token list} \rangle$ we take it as our $\langle \text{delimiter} \rangle$ and pass this to the auxii auxiliary.

```

3754 \cs_new_protected:Npn \__tl_replace:NnNNNnn #1#2#3#4#5#6#7
3755 {
3756   \tl_if_empty:nTF {#6}
3757   {
3758     \__kernel_msg_error:nxx { kernel } { empty-search-pattern }
3759     { \tl_to_str:n {#7} }
3760   }
3761   {
3762     \tl_if_in:onTF { #5 #6 } {#1}
3763     {
3764       \tl_if_in:nnTF {#6} {#1}
3765       { \exp_args:Nc \__tl_replace:NnNNNnn {#2} {#2?} }
3766       {
3767         \__tl_quark_if_nil:nTF {#6}
3768         { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_tl_stop } }
3769         { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_tl_nil } }
3770       }
3771     }
3772     { \__tl_replace_auxii:nNNNnn {#1} }
3773     #3#4#5 {#6} {#7}

```

```

3774     }
3775   }
3776   \cs_new_protected:Npn \__tl_replace_auxi:NnnNNNnn #1#2#3
3777   {
3778     \tl_if_in:NnTF #1 { #2 #3 #3 }
3779     { \__tl_replace_auxi:NnnNNNnn #1 { #2 #3 } {#2} }
3780     { \__tl_replace_auxii:nNNNnn { #2 #3 #3 } }
3781   }

```

The auxiliary `__tl_replace_auxii:nNNNnn` receives the following arguments:

```

{<delimiter>} <function> <assignment>
<tl var> {<pattern>} {<replacement>}

```

All of its work is done between `\group_align_safe_begin:` and `\group_align_safe_end:` to avoid issues in alignments. It does the actual replacement within `#3 #4 {...}`, an x-expanding `<assignment> #3` to the `<tl var> #4`. The auxiliary `__tl_replace_next:w` is called, followed by the `<token list>`, some tokens including the `<delimiter> #1`, followed by the `<pattern> #5`. This auxiliary finds an argument delimited by `#5` (the presence of a trailing `#5` avoids runaway arguments) and calls `__tl_replace_wrap:w` to test whether this `#5` is found within the `<token list>` or is the trailing one.

If on the one hand it is found within the `<token list>`, then `##1` cannot contain the `<delimiter> #1` that we worked so hard to obtain, thus `__tl_replace_wrap:w` gets `##1` as its own argument `##1`, and protects it against the x-expanding assignment. It also finds `\exp_not:n` as `##2` and does nothing to it, thus letting through `\exp_not:n {<replacement>}` into the assignment. Note that `__tl_replace_next:w` and `__tl_replace_wrap:w` are always called followed by two empty brace groups. These are safe because no delimiter can match them. They prevent losing braces when grabbing delimited arguments, but require the use of `\exp_not:o` and `\use_none:nn`, rather than simply `\exp_not:n`. Afterwards, `__tl_replace_next:w` is called to repeat the replacement, or `__tl_replace_wrap:w` if we only want a single replacement. In this second case, `##1` is the `<remaining tokens>` in the `<token list>` and `##2` is some `<ending code>` which ends the assignment and removes the trailing tokens `#5` using some `\if_false: { \fi: }` trickery because `#5` may contain any delimiter.

If on the other hand the argument `##1` of `__tl_replace_next:w` is delimited by the trailing `<pattern> #5`, then `##1` is “`{ } { } <token list> <delimiter> {<ending code>}`”, hence `__tl_replace_wrap:w` finds “`{ } { } <token list>`” as `##1` and the `<ending code>` as `##2`. It leaves the `<token list>` into the assignment and unbraces the `<ending code>` which removes what remains (essentially the `<delimiter>` and `<replacement>`).

```

3782   \cs_new_protected:Npn \__tl_replace_auxii:nNNNnn #1#2#3#4#5#6
3783   {
3784     \group_align_safe_begin:
3785     \cs_set:Npn \__tl_replace_wrap:w ##1 #1 ##2
3786     { \exp_not:o { \use_none:nn ##1 } ##2 }
3787     \cs_set:Npx \__tl_replace_next:w ##1 #5
3788     {
3789       \exp_not:N \__tl_replace_wrap:w ##1
3790       \exp_not:n { #1 }
3791       \exp_not:n { \exp_not:n {#6} }
3792       \exp_not:n { #2 { } { } }
3793     }
3794     #3 #4
3795     {

```

```

3796     \exp_after:wN \_tl_replace_next:w
3797     \exp_after:wN { \exp_after:wN }
3798     \exp_after:wN { \exp_after:wN }
3799     #4
3800     #1
3801     {
3802         \if_false: { \fi: }
3803         \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
3804     }
3805     #5
3806 }
3807 \group_align_safe_end:
3808 }
3809 \cs_new_eq:NN \_tl_replace_wrap:w ?
3810 \cs_new_eq:NN \_tl_replace_next:w ?

```

(End definition for `_tl_replace:NnNNNnn` and others.)

```

\tl_remove_once:Nn Removal is just a special case of replacement.
\tl_remove_once:cn 3811 \cs_new_protected:Npn \tl_remove_once:Nn #1#2
\tl_gremove_once:Nn 3812 { \tl_replace_once:Nnn #1 {#2} { } }
\tl_gremove_once:cn 3813 \cs_new_protected:Npn \tl_gremove_once:Nn #1#2
3814 { \tl_greplace_once:Nnn #1 {#2} { } }
3815 \cs_generate_variant:Nn \tl_remove_once:Nn { c }
3816 \cs_generate_variant:Nn \tl_gremove_once:Nn { c }

```

(End definition for `\tl_remove_once:Nn` and `\tl_gremove_once:Nn`. These functions are documented on page 45.)

```

\tl_remove_all:Nn Removal is just a special case of replacement.
\tl_remove_all:cn 3817 \cs_new_protected:Npn \tl_remove_all:Nn #1#2
\tl_gremove_all:Nn 3818 { \tl_replace_all:Nnn #1 {#2} { } }
\tl_gremove_all:cn 3819 \cs_new_protected:Npn \tl_gremove_all:Nn #1#2
3820 { \tl_greplace_all:Nnn #1 {#2} { } }
3821 \cs_generate_variant:Nn \tl_remove_all:Nn { c }
3822 \cs_generate_variant:Nn \tl_gremove_all:Nn { c }

```

(End definition for `\tl_remove_all:Nn` and `\tl_gremove_all:Nn`. These functions are documented on page 45.)

8.7 Token list conditionals

```

\tl_if_blank_p:n TeX skips spaces when reading a non-delimited arguments. Thus, a <token list> is blank
\tl_if_blank_p:V if and only if \use_none:n <token list> ? is empty after one expansion. The auxiliary
\tl_if_blank_p:o \_tl_if_empty_if:o is a fast emptyness test, converting its argument to a string (after
\tl_if_blank:nTF one expansion) and using the test \if_meaning:w \q__tl_nil ... \q__tl_nil.
\tl_if_blank:VTF 3823 \prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF }
\tl_if_blank:oTF 3824 {
\_tl_if_blank_p:NNw 3825 \_tl_if_empty_if:o { \use_none:n #1 ? }
3826 \prg_return_true:
3827 \else:
3828 \prg_return_false:
3829 \fi:
3830 }

```

```

3831 \prg_generate_conditional_variant:Nnn \tl_if_blank:n
3832 { e , V , o } { p , T , F , TF }

```

(End definition for `\tl_if_blank:nTF` and `__tl_if_blank_p:NNw`. This function is documented on page 46.)

`\tl_if_empty_p:N` These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).

```

\__tl_if_empty_p:c
\__tl_if_empty:NTF
\__tl_if_empty:cTF
3833 \prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
3834 {
3835   \if_meaning:w #1 \c_empty_tl
3836   \prg_return_true:
3837   \else:
3838     \prg_return_false:
3839   \fi:
3840 }
3841 \prg_generate_conditional_variant:Nnn \tl_if_empty:N
3842 { c } { p , T , F , TF }

```

(End definition for `\tl_if_empty:NTF`. This function is documented on page 47.)

`\tl_if_empty_p:n` Convert the argument to a string: this is empty if and only if the argument is. Then
`\tl_if_empty_p:V` `\if_meaning:w \q_@@_nil ... \q_@@_nil` is true if and only if the string ... is empty.
`\tl_if_empty:nTF` It could be tempting to use `\if_meaning:w \q_@@_nil #1 \q_@@_nil` directly. This
`\tl_if_empty:VTF` fails on a token list starting with `\q__tl_nil` of course but more troubling is the case where argument is a complete conditional such as `\if_true: a \else: b \fi:` because then `\if_true:` is used by `\if_meaning:w`, the test turns out false, the `\else:` executes the false branch, the `\fi:` ends it and the `\q__tl_nil` at the end starts executing...

```

3843 \prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
3844 {
3845   \exp_after:wN \if_meaning:w \exp_after:wN \q__tl_nil
3846   \tl_to_str:n {#1} \q__tl_nil
3847   \prg_return_true:
3848   \else:
3849     \prg_return_false:
3850   \fi:
3851 }
3852 \prg_generate_conditional_variant:Nnn \tl_if_empty:n
3853 { V } { p , TF , T , F }

```

(End definition for `\tl_if_empty:nTF`. This function is documented on page 47.)

`\tl_if_empty_p:o` The auxiliary function `__tl_if_empty_if:o` is for use in various token list conditionals which reduce to testing if a given token list is empty after applying a simple function to it.
`\tl_if_empty:oTF` The test for emptiness is based on `\tl_if_empty:nTF`, but the expansion is hard-coded for efficiency, as this auxiliary function is used in several places. We don't put `\prg_return_true:` and so on in the definition of the auxiliary, because that would prevent an optimization applied to conditionals that end with this code.

```

3854 \cs_new:Npn \__tl_if_empty_if:o #1
3855 {
3856   \exp_after:wN \if_meaning:w \exp_after:wN \q__tl_nil
3857   \__kernel_tl_to_str:w \exp_after:wN {#1} \q__tl_nil
3858 }
3859 \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F }

```

```

3860 {
3861   \__tl_if_empty_if:o {#1}
3862   \prg_return_true:
3863   \else:
3864     \prg_return_false:
3865   \fi:
3866 }

```

(End definition for \tl_if_empty:nTF and __tl_if_empty_if:o. This function is documented on page 47.)

\tl_if_eq_p:NN Returns \c_true_bool if and only if the two token list variables are equal.

```

\tl_if_eq_p:Nc 3867 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF }
\tl_if_eq_p:cN 3868 {
\tl_if_eq_p:cc 3869   \if_meaning:w #1 #2
\tl_if_eq:NNTF 3870   \prg_return_true:
\tl_if_eq:NcTF 3871   \else:
\tl_if_eq:cNTF 3872   \prg_return_false:
\tl_if_eq:ccTF 3873   \fi:
3874 }
3875 \prg_generate_conditional_variant:Nnn \tl_if_eq:NN
3876 { Nc , c , cc } { p , TF , T , F }

```

(End definition for \tl_if_eq:NNTF. This function is documented on page 47.)

\l__tl_internal_a_tl Temporary storage.

```

\l__tl_internal_b_tl 3877 \tl_new:N \l__tl_internal_a_tl
3878 \tl_new:N \l__tl_internal_b_tl

```

(End definition for \l__tl_internal_a_tl and \l__tl_internal_b_tl.)

\tl_if_eq:NnTF A simple store and compare routine.

```

3879 \prg_new_protected_conditional:Npnn \tl_if_eq:Nn #1#2 { T , F , TF }
3880 {
3881   \group_begin:
3882     \tl_set:Nn \l__tl_internal_b_tl {#2}
3883     \exp_after:wN
3884   \group_end:
3885   \if_meaning:w #1 \l__tl_internal_b_tl
3886     \prg_return_true:
3887   \else:
3888     \prg_return_false:
3889   \fi:
3890 }
3891 \prg_generate_conditional_variant:Nnn \tl_if_eq:Nn { c } { TF , T , F }

```

(End definition for \tl_if_eq:NnTF. This function is documented on page 47.)

\tl_if_eq:nnTF A simple store and compare routine.

```

3892 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF }
3893 {
3894   \group_begin:
3895     \tl_set:Nn \l__tl_internal_a_tl {#1}
3896     \tl_set:Nn \l__tl_internal_b_tl {#2}
3897     \exp_after:wN

```

```

3898 \group_end:
3899 \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
3900 \prg_return_true:
3901 \else:
3902 \prg_return_false:
3903 \fi:
3904 }

```

(End definition for `\tl_if_eq:nnTF`. This function is documented on page 47.)

`\tl_if_in:NnTF` See `\tl_if_in:nnTF` for further comments. Here we simply expand the token list variable `\tl_if_in:cnTF` and pass it to `\tl_if_in:nnTF`.

```

3905 \cs_new_protected:Npn \tl_if_in:NnT { \exp_args:No \tl_if_in:nnT }
3906 \cs_new_protected:Npn \tl_if_in:NnF { \exp_args:No \tl_if_in:nnF }
3907 \cs_new_protected:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
3908 \prg_generate_conditional_variant:Nnn \tl_if_in:Nn
3909 { c } { T , F , TF }

```

(End definition for `\tl_if_in:NnTF`. This function is documented on page 47.)

`\tl_if_in:nnTF` Once more, the test relies on the emptiness test for robustness. The function `__tl_tmp:w` removes tokens until the first occurrence of #2. If this does not appear in #1, then the final #2 is removed, leaving an empty token list. Otherwise some tokens remain, and the test is false. See `\tl_if_empty:nTF` for details on the emptiness test.

Treating correctly cases like `\tl_if_in:nnTF {a state}{states}`, where #1#2 contains #2 before the end, requires special care. To cater for this case, we insert `{ }{ }` between the two token lists. This marker may not appear in #2 because of T_EX limitations on what can delimit a parameter, hence we are safe. Using two brace groups makes the test work also for empty arguments. The `\if_false:` constructions are a faster way to do `\group_align_safe_begin:` and `\group_align_safe_end:`. The `\scan_stop:` ensures that f-expanding `\tl_if_in:nn` does not lead to unbalanced braces.

```

3910 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
3911 {
3912 \scan_stop:
3913 \if_false: { \fi:
3914 \cs_set:Npn \__tl_tmp:w ##1 #2 { }
3915 \tl_if_empty:oTF { \__tl_tmp:w #1 {} {} #2 }
3916 { \prg_return_false: } { \prg_return_true: }
3917 \if_false: } \fi:
3918 }
3919 \prg_generate_conditional_variant:Nnn \tl_if_in:nn
3920 { V , o , no } { T , F , TF }

```

(End definition for `\tl_if_in:nnTF`. This function is documented on page 47.)

`\tl_if_novalue:p:n` Tests for `-NoValue-`: this is similar to `\tl_if_in:nn` but set up to be expandable and to check the value exactly. The question mark prevents the auxiliary from losing braces.

```

\__tl_if_novalue:w
3921 \cs_set_protected:Npn \__tl_tmp:w #1
3922 {
3923 \prg_new_conditional:Npnn \tl_if_novalue:n ##1
3924 { p , T , F , TF }
3925 {
3926 \str_if_eq:onTF
3927 { \__tl_if_novalue:w ? ##1 { } #1 }

```

```

3928         { ? { } #1 }
3929         { \prg_return_true: }
3930         { \prg_return_false: }
3931     }
3932     \cs_new:Npn \__tl_if_novalue:w ##1 #1 {##1}
3933 }
3934 \exp_args:No \__tl_tmp:w { \c_novalue_tl }

```

(End definition for `\tl_if_novalue:nTF` and `__tl_if_novalue:w`. This function is documented on page 48.)

`\tl_if_single_p:N` Expand the token list and feed it to `\tl_if_single:n`.

```

\tl_if_single:nTF
3935 \cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
3936 \cs_new:Npn \tl_if_single:NT { \exp_args:No \tl_if_single:nT }
3937 \cs_new:Npn \tl_if_single:NF { \exp_args:No \tl_if_single:nF }
3938 \cs_new:Npn \tl_if_single:NTF { \exp_args:No \tl_if_single:nTF }

```

(End definition for `\tl_if_single:NTF`. This function is documented on page 48.)

`\tl_if_single_p:n` This test is similar to `\tl_if_empty:nTF`. Expanding `\use_none:nn #1 ??` once yields an empty result if `#1` is blank, a single `?` if `#1` has a single item, and otherwise yields some tokens ending with `??`. Then, `\tl_to_str:n` makes sure there are no odd category codes. `__tl_if_single_p:n` An earlier version would compare the result to a single `?` using string comparison, but `__tl_if_single:nw` picks the second token in front of it. If `#1` is empty, this token is the trailing `?` and the catcode test yields `false`. If `#1` has a single item, the token is `^` and the catcode test yields `true`. Otherwise, it is one of the characters resulting from `\tl_to_str:n`, and the catcode test yields `false`. Note that `\if_catcode:w` and `__kernel_tl_to_str:w` are primitives that take care of expansion.

```

3939 \prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
3940 {
3941     \if_catcode:w ^ \exp_after:wN \__tl_if_single:nw
3942         \__kernel_tl_to_str:w
3943         \exp_after:wN { \use_none:nn #1 ?? } ^ ? \s__tl_stop
3944     \prg_return_true:
3945 }else:
3946     \prg_return_false:
3947 \fi:
3948 }
3949 \cs_new:Npn \__tl_if_single:nw #1#2#3 \s__tl_stop {#2}

```

(End definition for `\tl_if_single:nTF` and `__tl_if_single:nTF`. This function is documented on page 48.)

`\tl_if_single_token_p:n` There are four cases: empty token list, token list starting with a normal token, with a brace group, or with a space token. If the token list starts with a normal token, remove it and check for emptiness. For the next case, an empty token list is not a single token. Finally, we have a non-empty token list starting with a space or a brace group. Applying f-expansion yields an empty result if and only if the token list is a single space.

```

3950 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
3951 {
3952     \tl_if_head_is_N_type:nTF {#1}
3953     { \__tl_if_empty_if:o { \use_none:n #1 } }
3954     {

```



```

3955     \tl_if_empty:nTF {#1}
3956     { \if_false: }
3957     { \__tl_if_empty_if:o { \exp:w \exp_end_continue_f:w #1 } }
3958   }
3959   \prg_return_true:
3960 \else:
3961   \prg_return_false:
3962 \fi:
3963 }

```

(End definition for `\tl_if_single_token:nTF`. This function is documented on page 48.)

<pre> \tl_case:Nn \tl_case:cn \tl_case:NnTF \tl_case:cnTF __tl_case:nnTF __tl_case:Nw __tl_case_end:nw </pre>	<p>The aim here is to allow the case statement to be evaluated using a known number of expansion steps (two), and without needing to use an explicit “end of recursion” marker. That is achieved by using the test input as the final case, as this is always true. The trick is then to tidy up the output such that the appropriate case code plus either the true or false branch code is inserted.</p> <pre> 3964 \cs_new:Npn \tl_case:Nn #1#2 3965 { 3966 \exp:w 3967 __tl_case:NnTF #1 {#2} { } { } 3968 } 3969 \cs_new:Npn \tl_case:NnT #1#2#3 3970 { 3971 \exp:w 3972 __tl_case:NnTF #1 {#2} {#3} { } 3973 } 3974 \cs_new:Npn \tl_case:NnF #1#2#3 3975 { 3976 \exp:w 3977 __tl_case:NnTF #1 {#2} { } {#3} 3978 } 3979 \cs_new:Npn \tl_case:NnTF #1#2 3980 { 3981 \exp:w 3982 __tl_case:NnTF #1 {#2} 3983 } 3984 \cs_new:Npn __tl_case:NnTF #1#2#3#4 3985 { __tl_case:Nw #1 #2 #1 { } \s__tl_mark {#3} \s__tl_mark {#4} \s__tl_stop } 3986 \cs_new:Npn __tl_case:Nw #1#2#3 3987 { 3988 \tl_if_eq:NNTF #1 #2 3989 { __tl_case_end:nw {#3} } 3990 { __tl_case:Nw #1 } 3991 } 3992 \cs_generate_variant:Nn \tl_case:Nn { c } 3993 \prg_generate_conditional_variant:Nnn \tl_case:Nn 3994 { c } { T , F , TF } </pre>
--	---

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then #1 is the code to insert, #2 is the *next* case to check on and #3 is all of the rest of the cases code. That means that #4 is the **true** branch code, and #5 tidies up the spare `\s__tl_mark` and the **false** branch. On the other hand, if none of the cases matched then we arrive here using the “termination” case of comparing the search with

itself. That means that #1 is empty, #2 is the first `\s__tl_mark` and so #4 is the false code (the `true` code is mopped up by #3).

```
3995 \cs_new:Npn \__tl_case_end:nw #1#2#3 \s__tl_mark #4#5 \s__tl_stop
3996 { \exp_end: #1 #4 }
```

(End definition for `\tl_case:NnTF` and others. This function is documented on page 48.)

8.8 Mapping to token lists

`\tl_map_function:nn` Expandable loop macro for token lists. These have the advantage of not needing to test if the argument is empty, because if it is, the stop marker is read immediately and the loop terminated.

```
\tl_map_function:NN
\__tl_map_function:cn
\__tl_map_function:Nn
3997 \cs_new:Npn \tl_map_function:nn #1#2
3998 {
3999   \__tl_map_function:Nn #2 #1
4000   \q__tl_recursion_tail
4001   \prg_break_point:Nn \tl_map_break: { }
4002 }
4003 \cs_new:Npn \tl_map_function:NN
4004 { \exp_args:No \tl_map_function:nn }
4005 \cs_new:Npn \__tl_map_function:Nn #1#2
4006 {
4007   \__tl_if_recursion_tail_break:nN {#2} \tl_map_break:
4008   #1 {#2} \__tl_map_function:Nn #1
4009 }
4010 \cs_generate_variant:Nn \tl_map_function:NN { c }
```

(End definition for `\tl_map_function:nn`, `\tl_map_function:NN`, and `__tl_map_function:Nn`. These functions are documented on page 49.)

`\tl_map_inline:nn` The inline functions are straight forward by now. We use a little trick with the counter `\g__kernel_pr_g_map_int` to make them nestable. We can also make use of `__tl_map_function:Nn` from before.

```
\tl_map_inline:NN
\__tl_map_function:cn
4011 \cs_new_protected:Npn \tl_map_inline:nn #1#2
4012 {
4013   \int_gincr:N \g__kernel_pr_g_map_int
4014   \cs_gset_protected:cpn
4015   { __tl_map_ \int_use:N \g__kernel_pr_g_map_int :w } ##1 {#2}
4016   \exp_args:Nc \__tl_map_function:Nn
4017   { __tl_map_ \int_use:N \g__kernel_pr_g_map_int :w }
4018   #1 \q__tl_recursion_tail
4019   \prg_break_point:Nn \tl_map_break:
4020   { \int_gdecr:N \g__kernel_pr_g_map_int }
4021 }
4022 \cs_new_protected:Npn \tl_map_inline:NN
4023 { \exp_args:No \tl_map_inline:nn }
4024 \cs_generate_variant:Nn \tl_map_inline:NN { c }
```

(End definition for `\tl_map_inline:nn` and `\tl_map_inline:NN`. These functions are documented on page 49.)

`\tl_map_tokens:nn` Much like the function mapping.

```
\tl_map_tokens:NN
\__tl_map_tokens:cn
\__tl_map_tokens:nn
4025 \cs_new:Npn \tl_map_tokens:nn #1#2
4026 {
```

```

4027 \tl_map_tokens:nn {#2} #1
4028 \q__tl_recursion_tail
4029 \prg_break_point:Nn \tl_map_break: { }
4030 }
4031 \cs_new:Npn \tl_map_tokens:Nn
4032 { \exp_args:No \tl_map_tokens:nn }
4033 \cs_generate_variant:Nn \tl_map_tokens:Nn { c }
4034 \cs_new:Npn \__tl_map_tokens:nn #1#2
4035 {
4036 \tl_if_recursion_tail_break:nN {#2} \tl_map_break:
4037 \use:n {#1} {#2}
4038 \__tl_map_tokens:nn {#1}
4039 }

```

(End definition for `\tl_map_tokens:nn`, `\tl_map_tokens:Nn`, and `__tl_map_tokens:nn`. These functions are documented on page 49.)

```

\tl_map_variable:nNn \tl_map_variable:nNn <token list> <tl var> <action> assigns <tl var> to each element and
\tl_map_variable:NNn executes <action>. The assignment to <tl var> is done after the quark test so that this
\tl_map_variable:cNn variable does not get set to a quark.
\__tl_map_variable:Nnn
4040 \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
4041 {
4042 \__tl_map_variable:Nnn #2 {#3} #1
4043 \q__tl_recursion_tail
4044 \prg_break_point:Nn \tl_map_break: { }
4045 }
4046 \cs_new_protected:Npn \tl_map_variable:NNn
4047 { \exp_args:No \tl_map_variable:nNn }
4048 \cs_new_protected:Npn \__tl_map_variable:Nnn #1#2#3
4049 {
4050 \tl_if_recursion_tail_break:nN {#3} \tl_map_break:
4051 \tl_set:Nn #1 {#3}
4052 \use:n {#2}
4053 \__tl_map_variable:Nnn #1 {#2}
4054 }
4055 \cs_generate_variant:Nn \tl_map_variable:NNn { c }

```

(End definition for `\tl_map_variable:nNn`, `\tl_map_variable:NNn`, and `__tl_map_variable:Nnn`. These functions are documented on page 49.)

`\tl_map_break:` The break statements use the general `\prg_map_break:Nn`.

```

\tl_map_break:n
4056 \cs_new:Npn \tl_map_break:
4057 { \prg_map_break:Nn \tl_map_break: { } }
4058 \cs_new:Npn \tl_map_break:n
4059 { \prg_map_break:Nn \tl_map_break: }

```

(End definition for `\tl_map_break:` and `\tl_map_break:n`. These functions are documented on page 50.)

8.9 Using token lists

`\tl_to_str:n` Another name for a primitive: defined in `l3basics`.

```

\tl_to_str:V
4060 \cs_generate_variant:Nn \tl_to_str:n { V }

```

(End definition for `\tl_to_str:n`. This function is documented on page 51.)

\tl_to_str:N These functions return the replacement text of a token list as a string.

\tl_to_str:c

```

4061 \cs_new:Npn \tl_to_str:N #1 { \__kernel_tl_to_str:w \exp_after:wN {#1} }
4062 \cs_generate_variant:Nn \tl_to_str:N { c }

```

(End definition for \tl_to_str:N. This function is documented on page 51.)

\tl_use:N Token lists which are simply not defined give a clear T_EX error here. No such luck for ones equal to \scan_stop: so instead a test is made and if there is an issue an error is forced.

\tl_use:c

```

4063 \cs_new:Npn \tl_use:N #1
4064 {
4065   \tl_if_exist:NTF #1 {#1}
4066   {
4067     \__kernel_msg_expandable_error:nnn
4068     { kernel } { bad-variable } {#1}
4069   }
4070 }
4071 \cs_generate_variant:Nn \tl_use:N { c }

```

(End definition for \tl_use:N. This function is documented on page 51.)

8.10 Working with the contents of token lists

\tl_count:n Count number of elements within a token list or token list variable. Brace groups within the list are read as a single element. Spaces are ignored. __tl_count:n grabs the element and replaces it by +1. The 0 ensures that it works on an empty list.

\tl_count:V

\tl_count:o

\tl_count:N

\tl_count:c

__tl_count:n

```

4072 \cs_new:Npn \tl_count:n #1
4073 {
4074   \int_eval:n
4075   { 0 \tl_map_function:nN {#1} \__tl_count:n }
4076 }
4077 \cs_new:Npn \tl_count:N #1
4078 {
4079   \int_eval:n
4080   { 0 \tl_map_function:NN #1 \__tl_count:n }
4081 }
4082 \cs_new:Npn \__tl_count:n #1 { + 1 }
4083 \cs_generate_variant:Nn \tl_count:n { V , o }
4084 \cs_generate_variant:Nn \tl_count:N { c }

```

(End definition for \tl_count:n, \tl_count:N, and __tl_count:n. These functions are documented on page 51.)

\tl_count_tokens:n The token count is computed through an \int_eval:n construction. Each 1+ is output to the left, into the integer expression, and the sum is ended by the \exp_end: inserted by __tl_act_end:wn (which is technically implemented as \c_zero_int). Somewhat a hack!

__tl_act_count_normal:nN

__tl_act_count_group:nn

__tl_act_count_space:n

```

4085 \cs_new:Npn \tl_count_tokens:n #1
4086 {
4087   \int_eval:n
4088   {
4089     \__tl_act:NNNnn
4090     \__tl_act_count_normal:nN
4091     \__tl_act_count_group:nn

```

```

4092         \tl_act_count_space:n
4093         { }
4094         {#1}
4095     }
4096 }
4097 \cs_new:Npn \tl_act_count_normal:nN #1 #2 { 1 + }
4098 \cs_new:Npn \tl_act_count_space:n #1 { 1 + }
4099 \cs_new:Npn \tl_act_count_group:nn #1 #2
4100 { 2 + \tl_count_tokens:n {#2} + }

```

(End definition for `\tl_count_tokens:n` and others. This function is documented on page 52.)

```

\red{ \tl_reverse_items:n } Reversal of a token list is done by taking one item at a time and putting it after \s__-
\__tl_reverse_items:nwNwn tl_stop.
\__tl_reverse_items:wn
4101 \cs_new:Npn \tl_reverse_items:n #1
4102 {
4103     \__tl_reverse_items:nwNwn #1 ?
4104     \s__tl_mark \__tl_reverse_items:nwNwn
4105     \s__tl_mark \__tl_reverse_items:wn
4106     \s__tl_stop { }
4107 }
4108 \cs_new:Npn \tl_reverse_items:nwNwn #1 #2 \s__tl_mark #3 #4 \s__tl_stop #5
4109 {
4110     #3 #2
4111     \s__tl_mark \__tl_reverse_items:nwNwn
4112     \s__tl_mark \__tl_reverse_items:wn
4113     \s__tl_stop { {#1} #5 }
4114 }
4115 \cs_new:Npn \tl_reverse_items:wn #1 \s__tl_stop #2
4116 { \exp_not:o { \use_none:nn #2 } }

```

(End definition for `\tl_reverse_items:n`, `__tl_reverse_items:nwNwn`, and `__tl_reverse_items:wn`. This function is documented on page 52.)

```

\red{ \tl_trim_spaces:n } Trimming spaces from around the input is deferred to an internal function whose first
\tl_trim_spaces:o argument is the token list to trim, augmented by an initial \s__tl_mark, and whose
\red{ \tl_trim_spaces_apply:nN } second argument is a <continuation>, which receives as a braced argument \use_none:n
\tl_trim_spaces_apply:oN \s__tl_mark <trimmed token list>. In the case at hand, we take \exp_not:o as our
\red{ \tl_trim_spaces:N } continuation, so that space trimming behaves correctly within an x-type expansion.
\tl_trim_spaces:c
\red{ \tl_gtrim_spaces:N }
\tl_gtrim_spaces:c
4117 \cs_new:Npn \tl_trim_spaces:n #1
4118 { \__tl_trim_spaces:nn { \s__tl_mark #1 } \exp_not:o }
4119 \cs_generate_variant:Nn \tl_trim_spaces:n { o }
4120 \cs_new:Npn \tl_trim_spaces_apply:nN #1#2
4121 { \__tl_trim_spaces:nn { \s__tl_mark #1 } { \exp_args:No #2 } }
4122 \cs_generate_variant:Nn \tl_trim_spaces_apply:nN { o }
4123 \cs_new_protected:Npn \tl_trim_spaces:N #1
4124 { \tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
4125 \cs_new_protected:Npn \tl_gtrim_spaces:N #1
4126 { \tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
4127 \cs_generate_variant:Nn \tl_trim_spaces:N { c }
4128 \cs_generate_variant:Nn \tl_gtrim_spaces:N { c }

```

Trimming spaces from around the input is done using delimited arguments and quarks, and to get spaces at odd places in the definitions, we nest those in `__tl_tmp:w`, which

```

\__tl_trim_spaces:nn
\__tl_trim_spaces_auxi:w
\__tl_trim_spaces_auxii:w
\__tl_trim_spaces_auxiii:w
\__tl_trim_spaces_auxiv:w

```

then receives a single space as its argument: #1 is `_`. Removing leading spaces is done with `_tl_trim_spaces_auxi:w`, which loops until `\s_tl_mark_` matches the end of the token list: then ##1 is the token list and ##3 is `_tl_trim_spaces_auxii:w`. This hands the relevant tokens to the loop `_tl_trim_spaces_auxiii:w`, responsible for trimming trailing spaces. The end is reached when `_tl_nil` matches the one present in the definition of `\tl_trim_spaces:n`. Then `_tl_trim_spaces_auxiv:w` puts the token list into a group, with `\use_none:n` placed there to gobble a lingering `\s_tl_mark`, and feeds this to the *continuation*.

```

4129 \cs_set:Npn \_tl\_tmp:w #1
4130 {
4131   \cs_new:Npn \_tl_trim_spaces:nn ##1
4132   {
4133     \_tl_trim_spaces_auxi:w
4134     ##1
4135     \s\_tl\_nil
4136     \s\_tl\_mark #1 { }
4137     \s\_tl\_mark \_tl_trim_spaces_auxii:w
4138     \_tl_trim_spaces_auxiii:w
4139     #1 \s\_tl\_nil
4140     \_tl_trim_spaces_auxiv:w
4141     \s\_tl\_stop
4142   }
4143   \cs_new:Npn \_tl_trim_spaces_auxi:w ##1 \s\_tl\_mark #1 ##2 \s\_tl\_mark ##3
4144   {
4145     ##3
4146     \_tl_trim_spaces_auxi:w
4147     \s\_tl\_mark
4148     ##2
4149     \s\_tl\_mark #1 {##1}
4150   }
4151   \cs_new:Npn \_tl_trim_spaces_auxii:w
4152   \_tl_trim_spaces_auxi:w \s\_tl\_mark \s\_tl\_mark ##1
4153   {
4154     \_tl_trim_spaces_auxiii:w
4155     ##1
4156   }
4157   \cs_new:Npn \_tl_trim_spaces_auxiii:w ##1 #1 \s\_tl\_nil ##2
4158   {
4159     ##2
4160     ##1 \s\_tl\_nil
4161     \_tl_trim_spaces_auxiii:w
4162   }
4163   \cs_new:Npn \_tl_trim_spaces_auxiv:w ##1 \s\_tl\_nil ##2 \s\_tl\_stop ##3
4164   { ##3 { \use\_none:n ##1 } }
4165 }
4166 \_tl\_tmp:w { ~ }

```

(End definition for `\tl_trim_spaces:n` and others. These functions are documented on page 52.)

`\tl_sort:Nn` Implemented in `l3sort`.

`\tl_sort:cn`

`\tl_gsort:Nn` (End definition for `\tl_sort:Nn`, `\tl_gsort:Nn`, and `\tl_sort:nN`. These functions are documented on page 53.)

`\tl_gsort:cn`

`\tl_sort:nN`

8.11 Token by token changes

`\q__tl_act_mark` The `__tl_act...` functions may be applied to any token list. Hence, we use two private quarks, to allow any token, even quarks, in the token list. Only `\q__tl_act_mark` and `\q__tl_act_stop` may not appear in the token lists manipulated by `__tl_act:NNNnn` functions.

```
4167 \quark_new:N \q__tl_act_mark
4168 \quark_new:N \q__tl_act_stop
```

(End definition for `\q__tl_act_mark` and `\q__tl_act_stop`.)

`__tl_act:NNNnn` To help control the expansion, `__tl_act:NNNnn` should always be preceded by `\exp:w` and ends by producing `\exp_end:` once the result has been obtained. Then loop over tokens, groups, and spaces in #5. The marker `\q__tl_act_mark` is used both to avoid losing outer braces and to detect the end of the token list more easily. The result is stored as an argument for the dummy function `__tl_act_result:n`.

```
\__tl_act_output:n
\__tl_act_reverse_output:n
\__tl_act_loop:w
\__tl_act_normal:NwnNNN
\__tl_act_group:nwnNNN
\__tl_act_space:wnnNNN
\__tl_act_end:w
4169 \cs_new:Npn \__tl_act:NNNnn #1#2#3#4#5
4170 {
4171   \group_align_safe_begin:
4172   \__tl_act_loop:w #5 \q__tl_act_mark \q__tl_act_stop
4173   {#4} #1 #2 #3
4174   \__tl_act_result:n { }
4175 }
```

In the loop, we check how the token list begins and act accordingly. In the “normal” case, we may have reached `\q__tl_act_mark`, the end of the list. Then leave `\exp_end:` and the result in the input stream, to terminate the expansion of `\exp:w`. Otherwise, apply the relevant function to the “arguments”, #3 and to the head of the token list. Then repeat the loop. The scheme is the same if the token list starts with a group or with a space. Some extra work is needed to make `__tl_act_space:wnnNNN` gobble the space.

```
4176 \cs_new:Npn \__tl_act_loop:w #1 \q__tl_act_stop
4177 {
4178   \tl_if_head_is_N_type:nTF {#1}
4179   { \__tl_act_normal:NwnNNN }
4180   {
4181     \tl_if_head_is_group:nTF {#1}
4182     { \__tl_act_group:nwnNNN }
4183     { \__tl_act_space:wnnNNN }
4184   }
4185   #1 \q__tl_act_stop
4186 }
4187 \cs_new:Npn \__tl_act_normal:NwnNNN #1 #2 \q__tl_act_stop #3#4
4188 {
4189   \if_meaning:w \q__tl_act_mark #1
4190   \exp_after:wN \__tl_act_end:wn
4191   \fi:
4192   #4 {#3} #1
4193   \__tl_act_loop:w #2 \q__tl_act_stop
4194   {#3} #4
4195 }
4196 \cs_new:Npn \__tl_act_end:wn #1 \__tl_act_result:n #2
4197 { \group_align_safe_end: \exp_end: #2 }
4198 \cs_new:Npn \__tl_act_group:nwnNNN #1 #2 \q__tl_act_stop #3#4#5
4199 {
```

```

4200     #5 {#3} {#1}
4201     \__tl_act_loop:w #2 \q__tl_act_stop
4202     {#3} #4 #5
4203   }
4204 \exp_last_unbraced:NNo
4205 \cs_new:Npn \__tl_act_space:wwNNN \c_space_tl #1 \q__tl_act_stop #2#3#4#5
4206 {
4207   #5 {#2}
4208   \__tl_act_loop:w #1 \q__tl_act_stop
4209   {#2} #3 #4 #5
4210 }

```

Typically, the output is done to the right of what was already output, using `__tl_act_output:n`, but for the `__tl_act_reverse` functions, it should be done to the left.

```

4211 \cs_new:Npn \__tl_act_output:n #1 #2 \__tl_act_result:n #3
4212 { #2 \__tl_act_result:n { #3 #1 } }
4213 \cs_new:Npn \__tl_act_reverse_output:n #1 #2 \__tl_act_result:n #3
4214 { #2 \__tl_act_result:n { #1 #3 } }

```

(End definition for `__tl_act:NNNnn` and others.)

<pre> \tl_reverse:n \tl_reverse:o \tl_reverse:V __tl_reverse_normal:nN __tl_reverse_group_preserve:nn __tl_reverse_space:n </pre>	<p>The goal here is to reverse without losing spaces nor braces. This is done using the general internal function <code>__tl_act:NNNnn</code>. Spaces and “normal” tokens are output on the left of the current output. Grouped tokens are output to the left but without any reversal within the group. All of the internal functions here drop one argument: this is needed by <code>__tl_act:NNNnn</code> when changing case (to record which direction the change is in), but not when reversing the tokens.</p>
--	--

```

4215 \cs_new:Npn \tl_reverse:n #1
4216 {
4217   \__kernel_exp_not:w \exp_after:wN
4218   {
4219     \exp:w
4220     \__tl_act:NNNnn
4221     \__tl_reverse_normal:nN
4222     \__tl_reverse_group_preserve:nn
4223     \__tl_reverse_space:n
4224     { }
4225     {#1}
4226   }
4227 }
4228 \cs_generate_variant:Nn \tl_reverse:n { o , V }
4229 \cs_new:Npn \__tl_reverse_normal:nN #1#2
4230 { \__tl_act_reverse_output:n {#2} }
4231 \cs_new:Npn \__tl_reverse_group_preserve:nn #1#2
4232 { \__tl_act_reverse_output:n { {#2} } }
4233 \cs_new:Npn \__tl_reverse_space:n #1
4234 { \__tl_act_reverse_output:n { ~ } }

```

(End definition for `\tl_reverse:n` and others. This function is documented on page 52.)

<pre> \tl_reverse:N \tl_reverse:c \tl_greverse:N \tl_greverse:c </pre>	<p>This reverses the list, leaving <code>\exp_stop_f:</code> in front, which stops the f-expansion.</p> <pre> 4235 \cs_new_protected:Npn \tl_reverse:N #1 4236 { \tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } } 4237 \cs_new_protected:Npn \tl_greverse:N #1 </pre>
--	--


```

4238 { \tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
4239 \cs_generate_variant:Nn \tl_reverse:N { c }
4240 \cs_generate_variant:Nn \tl_greverse:N { c }

```

(End definition for `\tl_reverse:N` and `\tl_greverse:N`. These functions are documented on page 52.)

8.12 The first token from a token list

```

\tl_head:N Finding the head of a token list expandably always strips braces, which is fine as this
\tl_head:n is consistent with for example mapping to a list. The empty brace groups in \tl_
\tl_head:V head:n ensure that a blank argument gives an empty result. The result is returned
\tl_head:v within the \unexpanded primitive. The approach here is to use \if_false: to allow
\tl_head:f us to use } as the closing delimiter: this is the only safe choice, as any other token
\__tl_head_auxi:nw would not be able to parse it's own code. Using a marker, we can see if what we are
\__tl_head_auxii:n grabbing is exactly the marker, or there is anything else to deal with. Is there is, there
\tl_head:w is a loop. If not, tidy up and leave the item in the output stream. More detail in
\__tl_tl_head:w http://tex.stackexchange.com/a/70168.
\tl_tail:N
\tl_tail:n
\tl_tail:V
\tl_tail:v
\tl_tail:f
4241 \cs_new:Npn \tl_head:n #1
4242 {
4243   \__kernel_exp_not:w
4244   \if_false: { \fi: \__tl_head_auxi:nw #1 { } \s_tl_stop }
4245 }
4246 \cs_new:Npn \__tl_head_auxi:nw #1#2 \s_tl_stop
4247 {
4248   \exp_after:wN \__tl_head_auxii:n \exp_after:wN {
4249     \if_false: } \fi: {#1}
4250 }
4251 \cs_new:Npn \__tl_head_auxii:n #1
4252 {
4253   \exp_after:wN \if_meaning:w \exp_after:wN \q__tl_nil
4254   \__kernel_tl_to_str:w \exp_after:wN { \use_none:n #1 } \q__tl_nil
4255   \exp_after:wN \use_i:nn
4256   \else:
4257     \exp_after:wN \use_ii:nn
4258   \fi:
4259   {#1}
4260   { \if_false: { \fi: \__tl_head_auxi:nw #1 } }
4261 }
4262 \cs_generate_variant:Nn \tl_head:n { V , v , f }
4263 \cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
4264 \cs_new:Npn \__tl_tl_head:w #1#2 \s_tl_stop {#1}
4265 \cs_new:Npn \tl_head:N { \exp_args:No \tl_head:n }

```

To correctly leave the tail of a token list, it's important *not* to absorb any of the tail part as an argument. For example, the simple definition

```

\cs_new:Npn \tl_tail:n #1 { \tl_tail:w #1 \q_stop }
\cs_new:Npn \tl_tail:w #1#2 \q_stop

```

would give the wrong result for `\tl_tail:n { a { bc } }` (the braces would be stripped). Thus the only safe way to proceed is to first check that there is an item to grab (*i.e.* that the argument is not blank) and assuming there is to dispose of the first item. As with `\tl_head:n`, the result is protected from further expansion by `\unexpanded`.

While we could optimise the test here, this would leave some tokens “banned” in the input, which we do not have with this definition.

```

4266 \cs_new:Npn \tl_tail:n #1
4267 {
4268   \__kernel_exp_not:w
4269   \tl_if_blank:nTF {#1}
4270   { { } }
4271   { \exp_after:wN { \use_none:n #1 } }
4272 }
4273 \cs_generate_variant:Nn \tl_tail:n { V , v , f }
4274 \cs_new:Npn \tl_tail:N { \exp_args:No \tl_tail:n }

```

(End definition for `\tl_head:N` and others. These functions are documented on page 54.)

```

\tl_if_head_eq_meaning_p:nN Accessing the first token of a token list is tricky in three cases: when it has category code
\tl_if_head_eq_meaning:nNTF 1 (begin-group token), when it is an explicit space, with category code 10 and character
\tl_if_head_eq_charcode:nN code 32, or when the token list is empty (obviously).
\tl_if_head_eq_charcode:nNTF Forgetting temporarily about this issue we would use the following test in \tl_if_-
\tl_if_head_eq_charcode_p:nN head_eq_charcode:nN. Here, \tl_head:w yields the first token of the token list, then
\tl_if_head_eq_charcode:fNTF passed to \exp_not:N.
\tl_if_head_eq_catcode_p:nN \if_charcode:w
\tl_if_head_eq_catcode:nNTF \exp_after:wN \exp_not:N \tl_head:w #1 \q_nil \q_stop
                          \exp_not:N #2

```

The two first special cases are detected by testing if the token list starts with an N-type token (the extra ? sends empty token lists to the true branch of this test). In those cases, the first token is a character, and since we only care about its character code, we can use `\str_head:n` to access it (this works even if it is a space character). An empty argument results in `\tl_head:w` leaving two tokens: ? which is taken in the `\if_charcode:w` test, and `\use_none:nn`, which ensures that `\prg_return_false:` is returned regardless of whether the charcode test was true or false.

```

4275 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
4276 {
4277   \if_charcode:w
4278   \exp_not:N #2
4279   \tl_if_head_is_N_type:nTF { #1 ? }
4280   {
4281     \exp_after:wN \exp_not:N
4282     \__tl_tl_head:w #1 { ? \use_none:nn } \s_tl_stop
4283   }
4284   { \str_head:n {#1} }
4285   \prg_return_true:
4286   \else:
4287     \prg_return_false:
4288   \fi:
4289 }
4290 \prg_generate_conditional_variant:Nnn \tl_if_head_eq_charcode:nN
4291 { f } { p , TF , T , F }

```

For `\tl_if_head_eq_catcode:nN`, again we detect special cases with a `\tl_if_head_is_N_type:n`. Then we need to test if the first token is a begin-group token or an explicit

space token, and produce the relevant token, either `\c_group_begin_token` or `\c_space_token`. Again, for an empty argument, a hack is used, removing `\prg_return_true:` and `\else:` with `\use_none:nn` in case the catcode test with the (arbitrarily chosen) `?` is true.

```

4292 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
4293 {
4294   \if_catcode:w
4295     \exp_not:N #2
4296     \tl_if_head_is_N_type:nTF { #1 ? }
4297     {
4298       \exp_after:wN \exp_not:N
4299       \__tl_tl_head:w #1 { ? \use_none:nn } \s__tl_stop
4300     }
4301     {
4302       \tl_if_head_is_group:nTF {#1}
4303       { \c_group_begin_token }
4304       { \c_space_token }
4305     }
4306     \prg_return_true:
4307   \else:
4308     \prg_return_false:
4309   \fi:
4310 }
4311 \prg_generate_conditional_variant:Nnn \tl_if_head_eq_catcode:nN
4312 { o } { p , TF , T , F }

```

For `\tl_if_head_eq_meaning:nN`, again, detect special cases. In the normal case, use `\tl_head:w`, with no `\exp_not:N` this time, since `\if_meaning:w` causes no expansion. With an empty argument, the test is true, and `\use_none:nnn` removes #2 and the usual `\prg_return_true:` and `\else:`. In the special cases, we know that the first token is a character, hence `\if_charcode:w` and `\if_catcode:w` together are enough. We combine them in some order, hopefully faster than the reverse. Tests are not nested because the arguments may contain unmatched primitive conditionals.

```

4313 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
4314 {
4315   \tl_if_head_is_N_type:nTF { #1 ? }
4316   { \__tl_if_head_eq_meaning_normal:nN }
4317   { \__tl_if_head_eq_meaning_special:nN }
4318   {#1} #2
4319 }
4320 \cs_new:Npn \__tl_if_head_eq_meaning_normal:nN #1 #2
4321 {
4322   \exp_after:wN \if_meaning:w
4323   \__tl_tl_head:w #1 { ?? \use_none:nnn } \s__tl_stop #2
4324   \prg_return_true:
4325   \else:
4326     \prg_return_false:
4327   \fi:
4328 }
4329 \cs_new:Npn \__tl_if_head_eq_meaning_special:nN #1 #2
4330 {
4331   \if_charcode:w \str_head:n {#1} \exp_not:N #2
4332   \exp_after:wN \use:n

```

```

4333 \else:
4334 \prg_return_false:
4335 \exp_after:wN \use_none:n
4336 \fi:
4337 {
4338 \if_catcode:w \exp_not:N #2
4339 \tl_if_head_is_group:nTF {#1}
4340 { \c_group_begin_token }
4341 { \c_space_token }
4342 \prg_return_true:
4343 \else:
4344 \prg_return_false:
4345 \fi:
4346 }
4347 }

```

(End definition for `\tl_if_head_eq_meaning:nNTF` and others. These functions are documented on page 55.)

`\tl_if_head_is_N_type_p:n`
`\tl_if_head_is_N_type:nTF`
`__tl_if_head_is_N_type:w`

A token list can be empty, can start with an explicit space character (catcode 10 and charcode 32), can start with a begin-group token (catcode 1), or start with an N-type argument. In the first two cases, the line involving `__tl_if_head_is_N_type:w` produces `~` (and otherwise nothing). In the third case (begin-group token), the lines involving `\exp_after:wN` produce a single closing brace. The category code test is thus true exactly in the fourth case, which is what we want. One cannot optimize by moving one of the `*` to the beginning: if `#1` contains primitive conditionals, all of its occurrences must be dealt with before the `\if_catcode:w` tries to skip the true branch of the conditional.

```

4348 \prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
4349 {
4350 \if_catcode:w
4351 \if_false: { \fi: \__tl_if_head_is_N_type:w ? #1 ~ }
4352 \exp_after:wN \use_none:n
4353 \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
4354 * *
4355 \prg_return_true:
4356 \else:
4357 \prg_return_false:
4358 \fi:
4359 }
4360 \cs_new:Npn \__tl_if_head_is_N_type:w #1 ~
4361 {
4362 \tl_if_empty:oTF { \use_none:n #1 } { ^ } { }
4363 \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
4364 }

```

(End definition for `\tl_if_head_is_N_type:nTF` and `__tl_if_head_is_N_type:w`. This function is documented on page 55.)

`\tl_if_head_is_group_p:n`
`\tl_if_head_is_group:nTF`

Pass the first token of `#1` through `\token_to_str:N`, then check for the brace balance. The extra `?` caters for an empty argument. This could be made faster, but we need all brace tricks to happen in one step of expansion, keeping the token list brace balanced at all times.

```

4365 \prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
4366 {

```

```

4367 \if_catcode:w
4368 \exp_after:wN \use_none:n
4369 \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
4370 * *
4371 \prg_return_false:
4372 \else:
4373 \prg_return_true:
4374 \fi:
4375 }

```

(End definition for `\tl_if_head_is_group:nTF`. This function is documented on page 55.)

`\tl_if_head_is_space_p:n` The auxiliary's argument is all that is before the first explicit space in `?#1?~`. If that is a single `?` the test yields `true`. Otherwise, that is more than one token, and the test yields `false`. The work is done within braces (with an `\if_false: { \fi: ... }` construction) both to hide potential alignment tab characters from \TeX in a table, and to allow for removing what remains of the token list after its first space. The `\exp:w` and `\exp_end:` ensure that the result of a single step of expansion directly yields a balanced token list (no trailing closing brace).

`\tl_if_head_is_space:nTF`
`__tl_if_head_is_space:w`

```

4376 \prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
4377 {
4378 \exp:w \if_false: { \fi:
4379 \__tl_if_head_is_space:w ? #1 ? ~ }
4380 }
4381 \cs_new:Npn \__tl_if_head_is_space:w #1 ~
4382 {
4383 \tl_if_empty:oTF { \use_none:n #1 }
4384 { \exp_after:wN \exp_end: \exp_after:wN \prg_return_true: }
4385 { \exp_after:wN \exp_end: \exp_after:wN \prg_return_false: }
4386 \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
4387 }

```

(End definition for `\tl_if_head_is_space:nTF` and `__tl_if_head_is_space:w`. This function is documented on page 55.)

8.13 Using a single item

`\tl_item:nn` The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then `__tl_if_recursion_tail_break:nN` terminates the loop, and returns nothing at all.

`\tl_item:Nn`
`\tl_item:cn`
`__tl_item_aux:nn`
`__tl_item:nn`

```

4388 \cs_new:Npn \tl_item:nn #1#2
4389 {
4390 \exp_args:Nf \__tl_item:nn
4391 { \exp_args:Nf \__tl_item_aux:nn { \int_eval:n {#2} } {#1} }
4392 #1
4393 \q__tl_recursion_tail
4394 \prg_break_point:
4395 }
4396 \cs_new:Npn \__tl_item_aux:nn #1#2
4397 {
4398 \int_compare:nNnTF {#1} < 0
4399 { \int_eval:n { \tl_count:n {#2} + 1 + #1 } }
4400 {#1}
4401 }

```

```

4402 \cs_new:Npn \__tl_item:nn #1#2
4403 {
4404   \__tl_if_recursion_tail_break:nN {#2} \prg_break:
4405   \int_compare:nNnTF {#1} = 1
4406     { \prg_break:n { \exp_not:n {#2} } }
4407     { \exp_args:Nf \__tl_item:nn { \int_eval:n { #1 - 1 } } }
4408 }
4409 \cs_new:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
4410 \cs_generate_variant:Nn \tl_item:Nn { c }

```

(End definition for `\tl_item:nn` and others. These functions are documented on page 56.)

`\tl_rand_item:n` Importantly `\tl_item:nn` only evaluates its argument once.

```

\__tl_rand_item:n 4411 \cs_new:Npn \tl_rand_item:n #1
\__tl_rand_item:N 4412 {
\__tl_rand_item:c 4413   \tl_if_blank:nF {#1}
4414     { \tl_item:nn {#1} { \int_rand:nn { 1 } { \tl_count:n {#1} } } }
4415   }
4416 \cs_new:Npn \tl_rand_item:N { \exp_args:No \tl_rand_item:n }
4417 \cs_generate_variant:Nn \tl_rand_item:N { c }

```

(End definition for `\tl_rand_item:n` and `\tl_rand_item:N`. These functions are documented on page 56.)

`\tl_range:Nnn` To avoid checking for the end of the token list at every step, start by counting the number
`\tl_range:cnn` l of items and “normalizing” the bounds, namely clamping them to the interval $[0, l]$ and
`\tl_range:nnn` dealing with negative indices. More precisely, `__tl_range_items:nnNn` receives the
`__tl_range:Nnnn` number of items to skip at the beginning of the token list, the index of the last item
`__tl_range:nnnNn` to keep, a function which is either `__tl_range:w` or the token list itself. If nothing
`__tl_range:nnNn` should be kept, leave `{}`: this stops the `f`-expansion of `\tl_head:f` and that function
`__tl_range_skip:w` produces an empty result. Otherwise, repeatedly call `__tl_range_skip:w` to delete `#1`
`__tl_range:w` items from the input stream (the extra brace group avoids an off-by-one shift). For the
`__tl_range_skip_spaces:n` braced version `__tl_range_braced:w` sets up `__tl_range_collect_braced:w` which
`__tl_range_collect:nn` stores items one by one in an argument after the semicolon. Depending on the first token
`__tl_range_collect:ff` of the tail, either just move it (if it is a space) or also decrement the number of items left
`__tl_range_collect_space:nw` to find. Eventually, the result is a brace group followed by the rest of the token list, and
`__tl_range_collect_N:nN` `\tl_head:f` cleans up and gives the result in `\exp_not:n`.
`__tl_range_collect_group:nN`

```

4418 \cs_new:Npn \tl_range:Nnn { \exp_args:No \tl_range:nnn }
4419 \cs_generate_variant:Nn \tl_range:Nnn { c }
4420 \cs_new:Npn \tl_range:nnn { \__tl_range:Nnnn \__tl_range:w }
4421 \cs_new:Npn \__tl_range:Nnnn #1#2#3#4
4422 {
4423   \tl_head:f
4424   {
4425     \exp_args:Nf \__tl_range:nnnNn
4426     { \tl_count:n {#2} } {#3} {#4} #1 {#2}
4427   }
4428 }
4429 \cs_new:Npn \__tl_range:nnnNn #1#2#3
4430 {
4431   \exp_args:Nff \__tl_range:nnNn
4432   {
4433     \exp_args:Nf \__tl_range_normalize:nn
4434     { \int_eval:n { #2 - 1 } } {#1}

```

```

4435     }
4436     {
4437         \exp_args:Nf \__tl_range_normalize:nn
4438         { \int_eval:n {#3} } {#1}
4439     }
4440 }
4441 \cs_new:Npn \__tl_range:nnNn #1#2#3#4
4442 {
4443     \if_int_compare:w #2 > #1 \exp_stop_f: \else:
4444         \exp_after:wN { \exp_after:wN }
4445     \fi:
4446     \exp_after:wN #3
4447     \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ;
4448     \exp_after:wN { \exp:w \__tl_range_skip:w #1 ; { } #4 }
4449 }
4450 \cs_new:Npn \__tl_range_skip:w #1 ; #2
4451 {
4452     \if_int_compare:w #1 > 0 \exp_stop_f:
4453         \exp_after:wN \__tl_range_skip:w
4454         \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
4455     \else:
4456         \exp_after:wN \exp_end:
4457     \fi:
4458 }
4459 \cs_new:Npn \__tl_range:w #1 ; #2
4460 {
4461     \exp_args:Nf \__tl_range_collect:nn
4462     { \__tl_range_skip_spaces:n {#2} } {#1}
4463 }
4464 \cs_new:Npn \__tl_range_skip_spaces:n #1
4465 {
4466     \tl_if_head_is_space:nTF {#1}
4467     { \exp_args:Nf \__tl_range_skip_spaces:n {#1} }
4468     { { } #1 }
4469 }
4470 \cs_new:Npn \__tl_range_collect:nn #1#2
4471 {
4472     \int_compare:nNnTF {#2} = 0
4473     {#1}
4474     {
4475         \exp_args:No \tl_if_head_is_space:nTF { \use_none:n #1 }
4476         {
4477             \exp_args:Nf \__tl_range_collect:nn
4478             { \__tl_range_collect_space:nw #1 }
4479             {#2}
4480         }
4481         {
4482             \__tl_range_collect:ff
4483             {
4484                 \exp_args:No \tl_if_head_is_N_type:nTF { \use_none:n #1 }
4485                 { \__tl_range_collect_N:nN }
4486                 { \__tl_range_collect_group:nn }
4487                 #1
4488             }

```

```

4489         { \int_eval:n { #2 - 1 } }
4490     }
4491 }
4492 }
4493 \cs_new:Npn \__tl_range_collect_space:nw #1 ~ { { #1 ~ } }
4494 \cs_new:Npn \__tl_range_collect_N:nN #1#2 { { #1 #2 } }
4495 \cs_new:Npn \__tl_range_collect_group:nn #1#2 { { #1 {#2} } }
4496 \cs_generate_variant:Nn \__tl_range_collect:nn { ff }

```

(End definition for `\tl_range:Nnn` and others. These functions are documented on page 57.)

`__tl_range_normalize:nn` This function converts an $\langle index \rangle$ argument into an explicit position in the token list (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the $\langle index \rangle$ #1 and the string count #2. If #1 is negative, replace it by #1 + #2 + 1, then limit to the range [0, #2].

```

4497 \cs_new:Npn \__tl_range_normalize:nn #1#2
4498 {
4499     \int_eval:n
4500     {
4501         \if_int_compare:w #1 < 0 \exp_stop_f:
4502         \if_int_compare:w #1 < -#2 \exp_stop_f:
4503         0
4504         \else:
4505             #1 + #2 + 1
4506         \fi:
4507     \else:
4508         \if_int_compare:w #1 < #2 \exp_stop_f:
4509         #1
4510         \else:
4511         #2
4512         \fi:
4513     \fi:
4514 }
4515 }

```

(End definition for `__tl_range_normalize:nn`.)

8.14 Viewing token lists

`\tl_show:N` Showing token list variables is done after checking that the variable is defined (see `__tl_show:c` kernel_register_show:N).

`\tl_log:N`

`\tl_log:c`

`__tl_show:NN`

```

4516 \cs_new_protected:Npn \tl_show:N { \__tl_show:NN \tl_show:n }
4517 \cs_generate_variant:Nn \tl_show:N { c }
4518 \cs_new_protected:Npn \tl_log:N { \__tl_show:NN \tl_log:n }
4519 \cs_generate_variant:Nn \tl_log:N { c }
4520 \cs_new_protected:Npn \__tl_show:NN #1#2
4521 {
4522     \__kernel_chk_defined:NT #2
4523     { \exp_args:Nx #1 { \token_to_str:N #2 = \exp_not:o {#2} } }
4524 }

```

(End definition for `\tl_show:N`, `\tl_log:N`, and `__tl_show:NN`. These functions are documented on page 58.)

`\tl_show:n` Many `show` functions are based on `\tl_show:n`. The argument of `\tl_show:n` is line-wrapped using `\iow_wrap:nnnN` but with a leading `>~` and trailing period, both removed before passing the wrapped text to the `\showtokens` primitive. This primitive shows the result with a leading `>~` and trailing period.

The token list `\l__tl_internal_a_tl` containing the result of all these manipulations is displayed to the terminal using `\tex_showtokens:D` and an odd `\exp_after:wN` which expand the closing brace to improve the output slightly. The calls to `__kernel_iow_with:Nnn` ensure that the `\newlinechar` is set to 10 so that the `\iow_newline:` inserted by the line-wrapping code are correctly recognized by T_EX, and that `\errorcontextlines` is `-1` to avoid printing irrelevant context.

```

4525 \cs_new_protected:Npn \tl_show:n #1
4526 { \iow_wrap:nnnN { >~ \tl_to_str:n {#1} . } { } { } \__tl_show:n }
4527 \cs_new_protected:Npn \__tl_show:n #1
4528 {
4529   \tl_set:Nf \l__tl_internal_a_tl { \__tl_show:w #1 \s__tl_stop }
4530   \__kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
4531   {
4532     \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
4533     {
4534       \tex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
4535       { \exp_after:wN \l__tl_internal_a_tl }
4536     }
4537   }
4538 }
4539 \cs_new:Npn \__tl_show:w #1 > #2 . \s__tl_stop {#2}

```

(End definition for `\tl_show:n`, `__tl_show:n`, and `__tl_show:w`. This function is documented on page 58.)

`\tl_log:n` Logging is much easier, simply line-wrap. The `>~` and trailing period is there to match the output of `\tl_show:n`.

```

4540 \cs_new_protected:Npn \tl_log:n #1
4541 { \iow_wrap:nnnN { > ~ \tl_to_str:n {#1} . } { } { } \iow_log:n }

```

(End definition for `\tl_log:n`. This function is documented on page 58.)

8.15 Internal scan marks

`\s__tl_nil` Internal scan marks. These are defined here at the end because the code for `\scan_new:N` depends on some `l3tl` functions.

```

\s__tl_mark
\s__tl_stop
4542 \scan_new:N \s__tl_nil
4543 \scan_new:N \s__tl_mark
4544 \scan_new:N \s__tl_stop

```

(End definition for `\s__tl_nil`, `\s__tl_mark`, and `\s__tl_stop`.)

8.16 Scratch token lists

`\g_tmpa_tl` Global temporary token list variables. They are supposed to be set and used immediately, with no delay between the definition and the use because you can't count on other macros not to redefine them from under you.

```

4545 \tl_new:N \g_tmpa_tl
4546 \tl_new:N \g_tmpb_tl

```

(End definition for `\g_tmpa_tl` and `\g_tmpb_tl`. These variables are documented on page 59.)

`\l_tmpa_tl` These are local temporary token list variables. Be sure not to assume that the value you
`\l_tmpb_tl` put into them will survive for long—see discussion above.

```
4547 \tl_new:N \l_tmpa_tl
4548 \tl_new:N \l_tmpb_tl
```

(End definition for `\l_tmpa_tl` and `\l_tmpb_tl`. These variables are documented on page 59.)

```
4549 \endpackage
```

9 l3str implementation

```
4550 (*package)
```

```
4551 @<=str>
```

9.1 Internal auxiliaries

`\s__str_mark` Internal scan marks.

```
\s__str_stop
4552 \scan_new:N \s__str_mark
4553 \scan_new:N \s__str_stop
```

(End definition for `\s__str_mark` and `\s__str_stop`.)

`_str_use_none_delimit_by_s_stop:w` Functions to gobble up to a scan mark.

```
\_str_use_i_delimit_by_s_stop:nw
4554 \cs_new:Npn \_str_use_none_delimit_by_s_stop:w #1 \s__str_stop { }
4555 \cs_new:Npn \_str_use_i_delimit_by_s_stop:nw #1 #2 \s__str_stop {#1}
```

(End definition for `_str_use_none_delimit_by_s_stop:w` and `_str_use_i_delimit_by_s_stop:nw`.)

`\q__str_recursion_tail` Internal recursion quarks.

```
\q__str_recursion_stop
4556 \quark_new:N \q__str_recursion_tail
4557 \quark_new:N \q__str_recursion_stop
```

(End definition for `\q__str_recursion_tail` and `\q__str_recursion_stop`.)

`_str_if_recursion_tail_break:NN` Functions to query recursion quarks.

```
\_str_if_recursion_tail_stop_do:Nn
4558 \__kernel_quark_new_test:N \_str_if_recursion_tail_break:NN
4559 \__kernel_quark_new_test:N \_str_if_recursion_tail_stop_do:Nn
```

(End definition for `_str_if_recursion_tail_break:NN` and `_str_if_recursion_tail_stop_do:Nn`.)

9.2 Creating and setting string variables

`\str_new:N` A string is simply a token list. The full mapping system isn't set up yet so do things by hand.

```

\str_new:c
\str_use:N
\str_use:c
\str_clear:N
\str_clear:c
\str_gclear:N
\str_gclear:c
\str_clear_new:N
\str_clear_new:c
\str_gclear_new:N
\str_gclear_new:c
\str_set_eq:NN
\str_set_eq:cN
\str_set_eq:Nc
\str_set_eq:cc
\str_gset_eq:NN
\str_gset_eq:cN
\str_gset_eq:Nc
\str_gset_eq:cc
\str_concat:NNN
\str_concat:ccc
\str_gconcat:NNN
\str_gconcat:ccc

```

```

4560 \group_begin:
4561   \cs_set_protected:Npn \__str_tmp:n #1
4562     {
4563       \tl_if_blank:nF {#1}
4564       {
4565         \cs_new_eq:cc { str_ #1 :N } { tl_ #1 :N }
4566         \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :N } { c }
4567         \__str_tmp:n
4568       }
4569     }
4570   \__str_tmp:n
4571   { new }
4572   { use }
4573   { clear }
4574   { gclear }
4575   { clear_new }
4576   { gclear_new }
4577   { }
4578 \group_end:
4579 \cs_new_eq:NN \str_set_eq:NN \tl_set_eq:NN
4580 \cs_new_eq:NN \str_gset_eq:NN \tl_gset_eq:NN
4581 \cs_generate_variant:Nn \str_set_eq:NN { c , Nc , cc }
4582 \cs_generate_variant:Nn \str_gset_eq:NN { c , Nc , cc }
4583 \cs_new_eq:NN \str_concat:NNN \tl_concat:NNN
4584 \cs_new_eq:NN \str_gconcat:NNN \tl_gconcat:NNN
4585 \cs_generate_variant:Nn \str_concat:NNN { ccc }
4586 \cs_generate_variant:Nn \str_gconcat:NNN { ccc }

```

(End definition for `\str_new:N` and others. These functions are documented on page 60.)

`\str_set:Nn` Simply convert the token list inputs to $\langle strings \rangle$.

```

\str_set:NV
\str_set:Nx
\str_set:cn
\str_set:cV
\str_set:cx
\str_gset:Nn
\str_gset:NV
\str_gset:Nx
\str_gset:cn
\str_gset:cV
\str_gset:cx
\str_const:Nn
\str_const:NV
\str_const:Nx
\str_const:cn
\str_const:cV
\str_const:cx
\str_put_left:Nn
\str_put_left:NV
\str_put_left:Nx
\str_put_left:cn
\str_put_left:cV
\str_put_left:cx
\str_gput_left:Nn
\str_gput_left:NV
\str_gput_left:Nx
\str_gput_left:cn
\str_gput_left:cV

```

```

4587 \group_begin:
4588   \cs_set_protected:Npn \__str_tmp:n #1
4589     {
4590       \tl_if_blank:nF {#1}
4591       {
4592         \cs_new_protected:cpx { str_ #1 :Nn } ##1##2
4593         {
4594           \exp_not:c { tl_ #1 :Nx } ##1
4595           { \exp_not:N \tl_to_str:n {##2} }
4596         }
4597         \cs_generate_variant:cn { str_ #1 :Nn } { NV , Nx , cn , cV , cx }
4598         \__str_tmp:n
4599       }
4600     }
4601   \__str_tmp:n
4602   { set }
4603   { gset }
4604   { const }
4605   { put_left }

```

```

4606     { gput_left }
4607     { put_right }
4608     { gput_right }
4609     { }
4610 \group_end:

```

(End definition for `\str_set:Nn` and others. These functions are documented on page 61.)

9.3 Modifying string variables

```

\str_replace_all:Nnn
\str_replace_all:cnn
\str_greplace_all:Nnn
\str_greplace_all:cnn
\str_replace_once:Nnn
\str_replace_once:cnn
\str_greplace_once:Nnn
\str_greplace_once:cnn
\__str_replace:NNNnn
\__str_replace_aux:NNNnnn
\__str_replace_next:w

```

Start by applying `\tl_to_str:n` to convert the old and new token lists to strings, and also apply `\tl_to_str:N` to avoid any issues if we are fed a token list variable. Then the code is a much simplified version of the token list code because neither the delimiter nor the replacement can contain macro parameters or braces. The delimiter `\s__str_` cannot appear in the string to edit so it is used in all cases. Some x-expansion is unnecessary. There is no need to avoid losing braces nor to protect against expansion. The ending code is much simplified and does not need to hide in braces.

```

4611 \cs_new_protected:Npn \str_replace_once:Nnn
4612   { \__str_replace:NNNnn \prg_do_nothing: \tl_set:Nx }
4613 \cs_new_protected:Npn \str_greplace_once:Nnn
4614   { \__str_replace:NNNnn \prg_do_nothing: \tl_gset:Nx }
4615 \cs_new_protected:Npn \str_replace_all:Nnn
4616   { \__str_replace:NNNnn \__str_replace_next:w \tl_set:Nx }
4617 \cs_new_protected:Npn \str_greplace_all:Nnn
4618   { \__str_replace:NNNnn \__str_replace_next:w \tl_gset:Nx }
4619 \cs_generate_variant:Nn \str_replace_once:Nnn { c }
4620 \cs_generate_variant:Nn \str_greplace_once:Nnn { c }
4621 \cs_generate_variant:Nn \str_replace_all:Nnn { c }
4622 \cs_generate_variant:Nn \str_greplace_all:Nnn { c }
4623 \cs_new_protected:Npn \__str_replace:NNNnn #1#2#3#4#5
4624   {
4625     \tl_if_empty:nTF {#4}
4626     {
4627       \__kernel_msg_error:nxx { kernel } { empty-search-pattern } {#5}
4628     }
4629     {
4630       \use:x
4631       {
4632         \exp_not:n { \__str_replace_aux:NNNnnn #1 #2 #3 }
4633         { \tl_to_str:N #3 }
4634         { \tl_to_str:n {#4} } { \tl_to_str:n {#5} }
4635       }
4636     }
4637   }
4638 \cs_new_protected:Npn \__str_replace_aux:NNNnnn #1#2#3#4#5#6
4639   {
4640     \cs_set:Npn \__str_replace_next:w ##1 #5 { ##1 #6 #1 }
4641     #2 #3
4642     {
4643       \__str_replace_next:w
4644       #4
4645       \__str_use_none_delimit_by_s_stop:w
4646       #5

```

```

4647         \s__str_stop
4648     }
4649 }
4650 \cs_new_eq:NN \__str_replace_next:w ?

```

(End definition for `\str_replace_all:Nnn` and others. These functions are documented on page 62.)

```

\str_remove_once:Nn Removal is just a special case of replacement.
\str_remove_once:cn 4651 \cs_new_protected:Npn \str_remove_once:Nn #1#2
\str_gremove_once:Nn 4652 { \str_replace_once:Nnn #1 {#2} { } }
\str_gremove_once:cn 4653 \cs_new_protected:Npn \str_gremove_once:Nn #1#2
4654 { \str_greplace_once:Nnn #1 {#2} { } }
4655 \cs_generate_variant:Nn \str_remove_once:Nn { c }
4656 \cs_generate_variant:Nn \str_gremove_once:Nn { c }

```

(End definition for `\str_remove_once:Nn` and `\str_gremove_once:Nn`. These functions are documented on page 62.)

```

\str_remove_all:Nn Removal is just a special case of replacement.
\str_remove_all:cn 4657 \cs_new_protected:Npn \str_remove_all:Nn #1#2
\str_gremove_all:Nn 4658 { \str_replace_all:Nnn #1 {#2} { } }
\str_gremove_all:cn 4659 \cs_new_protected:Npn \str_gremove_all:Nn #1#2
4660 { \str_greplace_all:Nnn #1 {#2} { } }
4661 \cs_generate_variant:Nn \str_remove_all:Nn { c }
4662 \cs_generate_variant:Nn \str_gremove_all:Nn { c }

```

(End definition for `\str_remove_all:Nn` and `\str_gremove_all:Nn`. These functions are documented on page 62.)

9.4 String comparisons

```

\str_if_empty_p:N More copy-paste!
\str_if_empty_p:c 4663 \prg_new_eq_conditional:NNn \str_if_exist:N \tl_if_exist:N
\str_if_empty:N\TF 4664 { p , T , F , TF }
\str_if_empty:c\TF 4665 \prg_new_eq_conditional:NNn \str_if_exist:c \tl_if_exist:c
\str_if_exist_p:N 4666 { p , T , F , TF }
\str_if_exist_p:c 4667 \prg_new_eq_conditional:NNn \str_if_empty:N \tl_if_empty:N
\str_if_exist:N\TF 4668 { p , T , F , TF }
\str_if_exist:c\TF 4669 \prg_new_eq_conditional:NNn \str_if_empty:c \tl_if_empty:c
4670 { p , T , F , TF }

```

(End definition for `\str_if_empty:N\TF` and `\str_if_exist:N\TF`. These functions are documented on page 63.)

```

\__str_if_eq:nn String comparisons rely on the primitive \pdfstrcmp if available: LuaTeX does not
\__str_escape:n have it, so emulation is required. As the net result is that we do not always use
the primitive, the correct approach is to wrap up in a function with defined behaviour. That's done by providing a wrapper and then redefining in the LuaTeX case. Note that the necessary Lua code is loaded in l3bootstrap. The need to detokenize and force expansion of input arises from the case where a # token is used in the input, e.g. \__str_if_eq:nn {#} { \tl_to_str:n {#} }, which otherwise would fail as \tex_luaescapestring:D does not double such tokens.

```

```

4671 \cs_new:Npn \__str_if_eq:nn #1#2 { \tex_strcmp:D {#1} {#2} }
4672 \cs_if_exist:NT \tex_luatexversion:D

```

```

4673 {
4674   \cs_set_eq:NN \lua_escape:e \tex_luaescapestring:D
4675   \cs_set_eq:NN \lua_now:e \tex_directlua:D
4676   \cs_set:Npn \__str_if_eq:nn #1#2
4677   {
4678     \lua_now:e
4679     {
4680       l3kernel_strcmp
4681       (
4682         " \__str_escape:n {#1} " ,
4683         " \__str_escape:n {#2} "
4684       )
4685     }
4686   }
4687   \cs_new:Npn \__str_escape:n #1
4688   {
4689     \lua_escape:e
4690     { \__kernel_tl_to_str:w \use:e { {#1} } }
4691   }
4692 }

```

(End definition for __str_if_eq:nn and __str_escape:n.)

\str_if_eq_p:nn Modern engines provide a direct way of comparing two token lists, but returning a number. This set of conditionals therefore make life a bit clearer. The nn and xx versions are created directly as this is most efficient.

```

4693 \prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
4694 {
4695   \if_int_compare:w
4696     \__str_if_eq:nn { \exp_not:n {#1} } { \exp_not:n {#2} }
4697     = 0 \exp_stop_f:
4698     \prg_return_true: \else: \prg_return_false: \fi:
4699 }
4700 \prg_generate_conditional_variant:Nnn \str_if_eq:nn
4701 { V , v , o , nV , no , VV , nv } { p , T , F , TF }
4702 \prg_new_conditional:Npnn \str_if_eq:ee #1#2 { p , T , F , TF }
4703 {
4704   \if_int_compare:w \__str_if_eq:nn {#1} {#2} = 0 \exp_stop_f:
4705   \prg_return_true: \else: \prg_return_false: \fi:
4706 }

```

(End definition for \str_if_eq:nnTF. This function is documented on page 63.)

\str_if_eq_p:NN Note that \str_if_eq:NN is different from \tl_if_eq:NN because it needs to ignore category codes.

```

4707 \prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
4708 {
4709   \if_int_compare:w
4710     \__str_if_eq:nn { \tl_to_str:N #1 } { \tl_to_str:N #2 }
4711     = 0 \exp_stop_f: \prg_return_true: \else: \prg_return_false: \fi:
4712 }
4713 \prg_generate_conditional_variant:Nnn \str_if_eq:NN
4714 { c , Nc , cc } { T , F , TF , p }

```

(End definition for \str_if_eq:NNTF. This function is documented on page 63.)

`\str_if_in:NnTF` Everything here needs to be detokenized but beyond that it is a simple token list test.
`\str_if_in:cnTF` It would be faster to fine-tune the T, F, TF variants by calling the appropriate variant of
`\str_if_in:nnTF` `\tl_if_in:nnTF` directly but that takes more code.

```

4715 \prg_new_protected_conditional:Npnn \str_if_in:Nn #1#2 { T , F , TF }
4716 {
4717   \use:x
4718   { \tl_if_in:nnTF { \tl_to_str:N #1 } { \tl_to_str:n {#2} } }
4719   { \prg_return_true: } { \prg_return_false: }
4720 }
4721 \prg_generate_conditional_variant:Nnn \str_if_in:Nn
4722 { c } { T , F , TF }
4723 \prg_new_protected_conditional:Npnn \str_if_in:nn #1#2 { T , F , TF }
4724 {
4725   \use:x
4726   { \tl_if_in:nnTF { \tl_to_str:n {#1} } { \tl_to_str:n {#2} } }
4727   { \prg_return_true: } { \prg_return_false: }
4728 }

```

(End definition for `\str_if_in:NnTF` and `\str_if_in:nnTF`. These functions are documented on page 63.)

`\str_case:nn` Much the same as `\tl_case:nn(TF)` here: just a change in the internal comparison.

```

\str_case:Vn      4729 \cs_new:Npn \str_case:nn #1#2
\str_case:on      4730 {
\str_case:nV      4731   \exp:w
\str_case:nv      4732   \__str_case:nnTF {#1} {#2} { } { }
\str_case:nnTF    4733 }
\str_case:VnTF    4734 \cs_new:Npn \str_case:nnT #1#2#3
\str_case:onTF    4735 {
\str_case:nVTF    4736   \exp:w
\str_case:nvTF    4737   \__str_case:nnTF {#1} {#2} {#3} { }
\str_case:nvTF    4738 }
\str_case_e:nn    4739 \cs_new:Npn \str_case:nnF #1#2
\str_case_e:nnTF  4740 {
\__str_case:nnTF  4741   \exp:w
\__str_case_e:nnTF 4742   \__str_case:nnTF {#1} {#2} { }
\__str_case:nw    4743 }
\__str_case_e:nw  4744 \cs_new:Npn \str_case:nnTF #1#2
\__str_case_end:nw 4745 {
4746   \exp:w
4747   \__str_case:nnTF {#1} {#2}
4748 }
4749 \cs_new:Npn \__str_case:nnTF #1#2#3#4
4750 { \__str_case:nw {#1} #2 {#1} { } \s__str_mark {#3} \s__str_mark {#4} \s__str_stop }
4751 \cs_generate_variant:Nn \str_case:nn { V , o , nV , nv }
4752 \prg_generate_conditional_variant:Nnn \str_case:nn
4753 { V , o , nV , nv } { T , F , TF }
4754 \cs_new:Npn \__str_case:nw #1#2#3
4755 {
4756   \str_if_eq:nnTF {#1} {#2}
4757   { \__str_case_end:nw {#3} }
4758   { \__str_case:nw {#1} }
4759 }
4760 \cs_new:Npn \str_case_e:nn #1#2

```

```

4761 {
4762   \exp:w
4763   \__str_case_e:nnTF {#1} {#2} { } { }
4764 }
4765 \cs_new:Npn \str_case_e:nnT #1#2#3
4766 {
4767   \exp:w
4768   \__str_case_e:nnTF {#1} {#2} {#3} { }
4769 }
4770 \cs_new:Npn \str_case_e:nnF #1#2
4771 {
4772   \exp:w
4773   \__str_case_e:nnTF {#1} {#2} { }
4774 }
4775 \cs_new:Npn \str_case_e:nnTF #1#2
4776 {
4777   \exp:w
4778   \__str_case_e:nnTF {#1} {#2}
4779 }
4780 \cs_new:Npn \__str_case_e:nnTF #1#2#3#4
4781 { \__str_case_e:nw {#1} #2 {#1} { } \s__str_mark {#3} \s__str_mark {#4} \s__str_stop }
4782 \cs_new:Npn \__str_case_e:nw #1#2#3
4783 {
4784   \str_if_eq:eeTF {#1} {#2}
4785   { \__str_case_end:nw {#3} }
4786   { \__str_case_e:nw {#1} }
4787 }
4788 \cs_new:Npn \__str_case_end:nw #1#2#3 \s__str_mark #4#5 \s__str_stop
4789 { \exp_end: #1 #4 }

```

(End definition for `\str_case:nnTF` and others. These functions are documented on page 64.)

9.5 Mapping to strings

`\str_map_function:Nn` The inline and variable mappings are similar to the usual token list mappings but start out by turning the argument to an “other string”. Doing the same for the expandable function mapping would require `__kernel_str_to_other:n`, quadratic in the string length. To deal with spaces in that case, `__str_map_function:w` replaces the following space by a braced space and a further call to itself. These are received by `__str_map_function:Nn`, which passes the space to `#1` and calls `__str_map_function:w` to deal with the next space. The space before the braced space allows to optimize the `\q__str_recursion_tail` test. Of course we need to include a trailing space (the question mark is needed to avoid losing the space when TeX tokenizes the line). At the cost of about three more auxiliaries this code could get a 9 times speed up by testing only every 9-th character for whether it is `\q__str_recursion_tail` (also by converting 9 spaces at a time in the `\str_map_function:nN` case).

For the `map_variable` functions we use a string assignment to store each character because spaces are made catcode 12 before the loop.

```

4790 \cs_new:Npn \str_map_function:nN #1#2
4791 {
4792   \exp_after:wN \__str_map_function:w
4793   \exp_after:wN \__str_map_function:Nn \exp_after:wN #2
4794   \__kernel_tl_to_str:w {#1}

```



```

4795     \q__str_recursion_tail ? ~
4796     \prg_break_point:Nn \str_map_break: { }
4797 }
4798 \cs_new:Npn \str_map_function:NN
4799 { \exp_args:No \str_map_function:nN }
4800 \cs_new:Npn \__str_map_function:w #1 ~
4801 { #1 { ~ { ~ } \__str_map_function:w } }
4802 \cs_new:Npn \__str_map_function:Nn #1#2
4803 {
4804     \if_meaning:w \q__str_recursion_tail #2
4805     \exp_after:wN \str_map_break:
4806     \fi:
4807     #1 #2 \__str_map_function:Nn #1
4808 }
4809 \cs_generate_variant:Nn \str_map_function:NN { c }
4810 \cs_new_protected:Npn \str_map_inline:nn #1#2
4811 {
4812     \int_gincr:N \g__kernel_prg_map_int
4813     \cs_gset_protected:cpn
4814     { __str_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
4815     \use:x
4816     {
4817         \exp_not:N \__str_map_inline:NN
4818         \exp_not:c { __str_map_ \int_use:N \g__kernel_prg_map_int :w }
4819         \__kernel_str_to_other_fast:n {#1}
4820     }
4821     \q__str_recursion_tail
4822     \prg_break_point:Nn \str_map_break:
4823     { \int_gdecr:N \g__kernel_prg_map_int }
4824 }
4825 \cs_new_protected:Npn \str_map_inline:Nn
4826 { \exp_args:No \str_map_inline:nn }
4827 \cs_generate_variant:Nn \str_map_inline:Nn { c }
4828 \cs_new:Npn \__str_map_inline:NN #1#2
4829 {
4830     \__str_if_recursion_tail_break:NN #2 \str_map_break:
4831     \exp_args:No #1 { \token_to_str:N #2 }
4832     \__str_map_inline:NN #1
4833 }
4834 \cs_new_protected:Npn \str_map_variable:nNn #1#2#3
4835 {
4836     \use:x
4837     {
4838         \exp_not:n { \__str_map_variable:NnN #2 {#3} }
4839         \__kernel_str_to_other_fast:n {#1}
4840     }
4841     \q__str_recursion_tail
4842     \prg_break_point:Nn \str_map_break: { }
4843 }
4844 \cs_new_protected:Npn \str_map_variable:NNn
4845 { \exp_args:No \str_map_variable:nNn }
4846 \cs_new_protected:Npn \__str_map_variable:NnN #1#2#3
4847 {
4848     \__str_if_recursion_tail_break:NN #3 \str_map_break:

```

```

4849 \str_set:Nn #1 {#3}
4850 \use:n {#2}
4851 \__str_map_variable:NnN #1 {#2}
4852 }
4853 \cs_generate_variant:Nn \str_map_variable:NNn { c }
4854 \cs_new:Npn \str_map_break:
4855 { \prg_map_break:Nn \str_map_break: { } }
4856 \cs_new:Npn \str_map_break:n
4857 { \prg_map_break:Nn \str_map_break: }

```

(End definition for `\str_map_function:NN` and others. These functions are documented on page 64.)

9.6 Accessing specific characters in a string

`__kernel_str_to_other:n` First apply `\tl_to_str:n`, then replace all spaces by “other” spaces, 8 at a time, storing the converted part of the string between the `\s__str_mark` and `\s__str_stop` markers. The end is detected when `__str_to_other_loop:w` finds one of the trailing A, distinguished from any contents of the initial token list by their category. Then `__str_to_other_end:w` is called, and finds the result between `\s__str_mark` and the first A (well, there is also the need to remove a space).

```

4858 \cs_new:Npn \__kernel_str_to_other:n #1
4859 {
4860   \exp_after:wN \__str_to_other_loop:w
4861   \tl_to_str:n {#1} ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \s__str_mark \s__str_stop
4862 }
4863 \group_begin:
4864 \tex_lccode:D '\* = '\ %
4865 \tex_lccode:D '\A = '\A %
4866 \tex_lowercase:D
4867 {
4868   \group_end:
4869   \cs_new:Npn \__str_to_other_loop:w
4870   #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \s__str_stop
4871   {
4872     \if_meaning:w A #8
4873     \__str_to_other_end:w
4874     \fi:
4875     \__str_to_other_loop:w
4876     #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \s__str_stop
4877   }
4878   \cs_new:Npn \__str_to_other_end:w \fi: #1 \s__str_mark #2 * A #3 \s__str_stop
4879   { \fi: #2 }
4880 }

```

(End definition for `__kernel_str_to_other:n`, `__str_to_other_loop:w`, and `__str_to_other_end:w`.)

`__kernel_str_to_other_fast:n` The difference with `__kernel_str_to_other:n` is that the converted part is left in the input stream, making these commands only restricted-expandable.

```

4881 \cs_new:Npn \__kernel_str_to_other_fast:n #1
4882 {
4883   \exp_after:wN \__str_to_other_fast_loop:w \tl_to_str:n {#1} ~
4884   A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \s__str_stop
4885 }

```

```

4886 \group_begin:
4887 \tex_lccode:D '\* = '\ %
4888 \tex_lccode:D '\A = '\A %
4889 \tex_lowercase:D
4890 {
4891   \group_end:
4892   \cs_new:Npn \__str_to_other_fast_loop:w
4893     #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 ~
4894     {
4895       \if_meaning:w A #9
4896       \__str_to_other_fast_end:w
4897       \fi:
4898       #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * #9
4899       \__str_to_other_fast_loop:w *
4900     }
4901   \cs_new:Npn \__str_to_other_fast_end:w #1 * A #2 \s__str_stop {#1}
4902 }

```

(End definition for `__kernel_str_to_other_fast:n`, `__kernel_str_to_other_fast_loop:w`, and `__str_to_other_fast_end:w`.)

`\str_item:Nn` The `\str_item:nn` hands its argument with spaces escaped to `__str_item:nn`, and `\str_item:cn` makes sure to turn the result back into a proper string (with category code 10 spaces) eventually. The `\str_item_ignore_spaces:nn` function does not escape spaces, which are thus ignored by `__str_item:nn` since everything else is done with undelimited arguments. Evaluate the `<index>` argument #2 and count characters in the string, passing those two numbers to `__str_item:w` for further analysis. If the `<index>` is negative, shift it by the `<count>` to know the how many character to discard, and if that is still negative give an empty result. If the `<index>` is larger than the `<count>`, give an empty result, and otherwise discard `<index> - 1` characters before returning the following one. The shift by `-1` is obtained by inserting an empty brace group before the string in that case: that brace group also covers the case where the `<index>` is zero.

```

4903 \cs_new:Npn \str_item:Nn { \exp_args:No \str_item:nn }
4904 \cs_generate_variant:Nn \str_item:Nn { c }
4905 \cs_new:Npn \str_item:nn #1#2
4906 {
4907   \exp_args:Nf \tl_to_str:n
4908   {
4909     \exp_args:Nf \__str_item:nn
4910     { \__kernel_str_to_other:n {#1} } {#2}
4911   }
4912 }
4913 \cs_new:Npn \str_item_ignore_spaces:nn #1
4914 { \exp_args:No \__str_item:nn { \tl_to_str:n {#1} } }
4915 \cs_new:Npn \__str_item:nn #1#2
4916 {
4917   \exp_after:wN \__str_item:w
4918   \int_value:w \int_eval:n {#2} \exp_after:wN ;
4919   \int_value:w \__str_count:n {#1} ;
4920   #1 \s__str_stop
4921 }
4922 \cs_new:Npn \__str_item:w #1; #2;
4923 {
4924   \int_compare:nNnTF {#1} < 0

```

```

4925     {
4926         \int_compare:nNnTF {#1} < {-#2}
4927         { \__str_use_none_delimit_by_s_stop:w }
4928         {
4929             \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
4930             \exp:w \exp_after:wN \__str_skip_exp_end:w
4931             \int_value:w \int_eval:n { #1 + #2 } ;
4932         }
4933     }
4934     {
4935         \int_compare:nNnTF {#1} > {#2}
4936         { \__str_use_none_delimit_by_s_stop:w }
4937         {
4938             \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
4939             \exp:w \__str_skip_exp_end:w #1 ; { }
4940         }
4941     }
4942 }

```

(End definition for `\str_item:Nn` and others. These functions are documented on page 67.)

```

\__str_skip_exp_end:w
\__str_skip_loop:wNNNNNNNN
\__str_skip_end:w
\__str_skip_end:NNNNNNNN

```

Removes $\max(\#1, 0)$ characters from the input stream, and then leaves `\exp_end:.` This should be expanded using `\exp:w`. We remove characters 8 at a time until there are at most 8 to remove. Then we do a dirty trick: the `\if_case:w` construction leaves between 0 and 8 times the `\or:` control sequence, and those `\or:` become arguments of `__str_skip_end:NNNNNNNN`. If the number of characters to remove is 6, say, then there are two `\or:` left, and the 8 arguments of `__str_skip_end:NNNNNNNN` are the two `\or:`, and 6 characters from the input stream, exactly what we wanted to remove. Then close the `\if_case:w` conditional with `\fi:`, and stop the initial expansion with `\exp_end:` (see places where `__str_skip_exp_end:w` is called).

```

4943 \cs_new:Npn \__str_skip_exp_end:w #1;
4944 {
4945     \if_int_compare:w #1 > 8 \exp_stop_f:
4946     \exp_after:wN \__str_skip_loop:wNNNNNNNN
4947     \else:
4948     \exp_after:wN \__str_skip_end:w
4949     \int_value:w \int_eval:w
4950     \fi:
4951     #1 ;
4952 }
4953 \cs_new:Npn \__str_skip_loop:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
4954 {
4955     \exp_after:wN \__str_skip_exp_end:w
4956     \int_value:w \int_eval:n { #1 - 8 } ;
4957 }
4958 \cs_new:Npn \__str_skip_end:w #1 ;
4959 {
4960     \exp_after:wN \__str_skip_end:NNNNNNNN
4961     \if_case:w #1 \exp_stop_f: \or: \or: \or: \or: \or: \or: \or: \or:
4962 }
4963 \cs_new:Npn \__str_skip_end:NNNNNNNN #1#2#3#4#5#6#7#8 { \fi: \exp_end: }

```

(End definition for `__str_skip_exp_end:w` and others.)

`\str_range:Nnn` Sanitize the string. Then evaluate the arguments. At this stage we also decrement the
`\str_range:nnn` $\langle start\ index \rangle$, since our goal is to know how many characters should be removed. Then
`\str_range_ignore_spaces:nnn` limit the range to be non-negative and at most the length of the string (this avoids
`__str_range:nnn` needing to check for the end of the string when grabbing characters), shifting negative
`__str_range:w` numbers by the appropriate amount. Afterwards, skip characters, then keep some more,
`__str_range:nnw` and finally drop the end of the string.

```

4964 \cs_new:Npn \str_range:Nnn { \exp_args:No \str_range:nnn }
4965 \cs_generate_variant:Nn \str_range:Nnn { c }
4966 \cs_new:Npn \str_range:nnn #1#2#3
4967 {
4968   \exp_args:Nf \tl_to_str:n
4969   {
4970     \exp_args:Nf \__str_range:nnn
4971     { \__kernel_str_to_other:n {#1} } {#2} {#3}
4972   }
4973 }
4974 \cs_new:Npn \str_range_ignore_spaces:nnn #1
4975 { \exp_args:No \__str_range:nnn { \tl_to_str:n {#1} } }
4976 \cs_new:Npn \__str_range:nnn #1#2#3
4977 {
4978   \exp_after:wN \__str_range:w
4979   \int_value:w \__str_count:n {#1} \exp_after:wN ;
4980   \int_value:w \int_eval:n { (#2) - 1 } \exp_after:wN ;
4981   \int_value:w \int_eval:n {#3} ;
4982   #1 \s__str_stop
4983 }
4984 \cs_new:Npn \__str_range:w #1; #2; #3;
4985 {
4986   \exp_args:Nf \__str_range:nnw
4987   { \__str_range_normalize:nn {#2} {#1} }
4988   { \__str_range_normalize:nn {#3} {#1} }
4989 }
4990 \cs_new:Npn \__str_range:nnw #1#2
4991 {
4992   \exp_after:wN \__str_collect_delimit_by_q_stop:w
4993   \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ;
4994   \exp:w \__str_skip_exp_end:w #1 ;
4995 }

```

(End definition for `\str_range:Nnn` and others. These functions are documented on page 68.)

`__str_range_normalize:nn` This function converts an $\langle index \rangle$ argument into an explicit position in the string (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the $\langle index \rangle$ #1 and the string count #2. If #1 is negative, replace it by $\#1 + \#2 + 1$, then limit to the range $[0, \#2]$.

```

4996 \cs_new:Npn \__str_range_normalize:nn #1#2
4997 {
4998   \int_eval:n
4999   {
5000     \if_int_compare:w #1 < 0 \exp_stop_f:
5001     \if_int_compare:w #1 < -#2 \exp_stop_f:
5002     0
5003     \else:

```

```

5004         #1 + #2 + 1
5005         \fi:
5006     \else:
5007         \if_int_compare:w #1 < #2 \exp_stop_f:
5008             #1
5009         \else:
5010             #2
5011         \fi:
5012     \fi:
5013 }
5014 }

```

(End definition for `_str_range_normalize:nn`.)

`_str_collect_delimit_by_q_stop:w` Collects `max(#1,0)` characters, and removes everything else until `\s_str_stop`. This is somewhat similar to `_str_skip_exp_end:w`, but accepts integer expression arguments. `_str_collect_loop:wn` This time we can only grab 7 characters at a time. At the end, we use an `\if_case:w` trick again, so that the 8 first arguments of `_str_collect_end:nnnnnnnnw` are some `\or:`, followed by an `\fi:`, followed by `#1` characters from the input stream. Simply leaving this in the input stream closes the conditional properly and the `\or:` disappear.

```

5015 \cs_new:Npn \_str_collect_delimit_by_q_stop:w #1;
5016 { \_str_collect_loop:wn #1 ; { } }
5017 \cs_new:Npn \_str_collect_loop:wn #1 ;
5018 {
5019     \if_int_compare:w #1 > 7 \exp_stop_f:
5020     \exp_after:wN \_str_collect_loop:wnNNNNNNN
5021     \else:
5022     \exp_after:wN \_str_collect_end:wn
5023     \fi:
5024     #1 ;
5025 }
5026 \cs_new:Npn \_str_collect_loop:wnNNNNNNN #1; #2 #3#4#5#6#7#8#9
5027 {
5028     \exp_after:wN \_str_collect_loop:wn
5029     \int_value:w \int_eval:n { #1 - 7 } ;
5030     { #2 #3#4#5#6#7#8#9 }
5031 }
5032 \cs_new:Npn \_str_collect_end:wn #1 ;
5033 {
5034     \exp_after:wN \_str_collect_end:nnnnnnnnw
5035     \if_case:w \if_int_compare:w #1 > 0 \exp_stop_f:
5036     #1 \else: 0 \fi: \exp_stop_f:
5037     \or: \or: \or: \or: \or: \or: \or: \fi:
5038 }
5039 \cs_new:Npn \_str_collect_end:nnnnnnnnw #1#2#3#4#5#6#7#8 #9 \s\_str_stop
5040 { #1#2#3#4#5#6#7#8 }

```

(End definition for `_str_collect_delimit_by_q_stop:w` and others.)

9.7 Counting characters

`\str_count_spaces:N` To speed up this function, we grab and discard 9 space-delimited arguments in each iteration of the loop. The loop stops when the last argument is one of the trailing

`\str_count_spaces:c`

`\str_count_spaces:n`

`_str_count_spaces_loop:w`

$X\langle number \rangle$, and that $\langle number \rangle$ is added to the sum of 9 that precedes, to adjust the result.

```

5041 \cs_new:Npn \str_count_spaces:N
5042   { \exp_args:No \str_count_spaces:n }
5043 \cs_generate_variant:Nn \str_count_spaces:N { c }
5044 \cs_new:Npn \str_count_spaces:n #1
5045   {
5046     \int_eval:n
5047     {
5048       \exp_after:wN \__str_count_spaces_loop:w
5049       \tl_to_str:n {#1} ~
5050       X 7 ~ X 6 ~ X 5 ~ X 4 ~ X 3 ~ X 2 ~ X 1 ~ X 0 ~ X -1 ~
5051       \s__str_stop
5052     }
5053   }
5054 \cs_new:Npn \__str_count_spaces_loop:w #1~#2~#3~#4~#5~#6~#7~#8~#9~
5055   {
5056     \if_meaning:w X #9
5057       \__str_use_i_delimit_by_s_stop:nw
5058     \fi:
5059     9 + \__str_count_spaces_loop:w
5060   }

```

(End definition for `\str_count_spaces:N`, `\str_count_spaces:n`, and `__str_count_spaces_loop:w`. These functions are documented on page 66.)

`\str_count:N` To count characters in a string we could first escape all spaces using `__kernel_str_to_other:n`, then pass the result to `\tl_count:n`. However, the escaping step would be quadratic in the number of characters in the string, and we can do better. Namely, `\str_count:n` sum the number of spaces (`\str_count_spaces:n`) and the result of `\tl_count:n`, which ignores spaces. Since strings tend to be longer than token lists, we use specialized functions to count characters ignoring spaces. Namely, `__str_count:n`, used in `\str_item:nn` and `\str_range:nnn`, is similar to `\str_count_ignore_spaces:n` but expects its argument to already be a string or a string with spaces escaped.

```

5061 \cs_new:Npn \str_count:N { \exp_args:No \str_count:n }
5062 \cs_generate_variant:Nn \str_count:N { c }
5063 \cs_new:Npn \str_count:n #1
5064   {
5065     \__str_count_aux:n
5066     {
5067       \str_count_spaces:n {#1}
5068       + \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1}
5069     }
5070   }
5071 \cs_new:Npn \__str_count:n #1
5072   {
5073     \__str_count_aux:n
5074     { \__str_count_loop:NNNNNNNNN #1 }
5075   }
5076 \cs_new:Npn \str_count_ignore_spaces:n #1
5077   {

```

```

5078     \__str_count_aux:n
5079     { \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1} }
5080   }
5081 \cs_new:Npn \__str_count_aux:n #1
5082 {
5083   \int_eval:n
5084   {
5085     #1
5086     { X 8 } { X 7 } { X 6 }
5087     { X 5 } { X 4 } { X 3 }
5088     { X 2 } { X 1 } { X 0 }
5089     \s__str_stop
5090   }
5091 }
5092 \cs_new:Npn \__str_count_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
5093 {
5094   \if_meaning:w X #9
5095     \exp_after:wN \__str_use_none_delimit_by_s_stop:w
5096   \fi:
5097   9 + \__str_count_loop:NNNNNNNNN
5098 }

```

(End definition for `\str_count:N` and others. These functions are documented on page 66.)

9.8 The first character in a string

`\str_head:N` The `\ignore_spaces` variant applies `\tl_to_str:n` then grabs the first item, thus skipping spaces. As usual, `\str_head:N` expands its argument and hands it to `\str_head:n`.
`\str_head:c` To circumvent the fact that \TeX skips spaces when grabbing undelimited macro parameters, `__str_head:w` takes an argument delimited by a space. If `#1` starts with a non-space character, `__str_use_i_delimit_by_s_stop:nw` leaves that in the input stream. On the other hand, if `#1` starts with a space, the `__str_head:w` takes an empty argument, and the single (initially braced) space in the definition of `__str_head:w` makes its way to the output. Finally, for an empty argument, the (braced) empty brace group in the definition of `\str_head:n` gives an empty result after passing through `__str_use_i_delimit_by_s_stop:nw`.
`\str_head:n`
`\str_head_ignore_spaces:n`
`__str_head:w`

```

5099 \cs_new:Npn \str_head:N { \exp_args:No \str_head:n }
5100 \cs_generate_variant:Nn \str_head:N { c }
5101 \cs_new:Npn \str_head:n #1
5102 {
5103   \exp_after:wN \__str_head:w
5104   \tl_to_str:n {#1}
5105   { { } } ~ \s__str_stop
5106 }
5107 \cs_new:Npn \__str_head:w #1 ~ %
5108 { \__str_use_i_delimit_by_s_stop:nw #1 { ~ } }
5109 \cs_new:Npn \str_head_ignore_spaces:n #1
5110 {
5111   \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
5112   \tl_to_str:n {#1} { } \s__str_stop
5113 }

```

(End definition for `\str_head:N` and others. These functions are documented on page 67.)

`\str_tail:N` Getting the tail is a little bit more convoluted than the head of a string. We hit the front of the string with `\reverse_if:N \if_charcode:w \scan_stop:.` This removes the first character, and necessarily makes the test true, since the character cannot match `\scan_stop:.` The auxiliary function then inserts the required `\fi:` to close the conditional, and leaves the tail of the string in the input stream. The details are such that an empty string has an empty tail (this requires in particular that the end-marker `X` be unexpandable and not a control sequence). The `_ignore_spaces` is rather simpler: after converting the input to a string, `__str_tail_auxii:w` removes one undelimited argument and leaves everything else until an end-marker `\s__str_mark`. One can check that an empty (or blank) string yields an empty tail.

```

5114 \cs_new:Npn \str_tail:N { \exp_args:No \str_tail:n }
5115 \cs_generate_variant:Nn \str_tail:N { c }
5116 \cs_new:Npn \str_tail:n #1
5117 {
5118   \exp_after:wN \__str_tail_auxi:w
5119   \reverse_if:N \if_charcode:w
5120     \scan_stop: \tl_to_str:n {#1} X X \s__str_stop
5121 }
5122 \cs_new:Npn \__str_tail_auxi:w #1 X #2 \s__str_stop { \fi: #1 }
5123 \cs_new:Npn \str_tail_ignore_spaces:n #1
5124 {
5125   \exp_after:wN \__str_tail_auxii:w
5126   \tl_to_str:n {#1} \s__str_mark \s__str_mark \s__str_stop
5127 }
5128 \cs_new:Npn \__str_tail_auxii:w #1 #2 \s__str_mark #3 \s__str_stop { #2 }

```

(End definition for `\str_tail:N` and others. These functions are documented on page 67.)

9.9 String manipulation

`\str_foldcase:N` Case changing for programmatic reasons is done by first detokenizing input then doing a simple loop that only has to worry about spaces and everything else. The output is detokenized to allow data sharing with text-based case changing.

```

\str_foldcase:V
\str_lowercase:N
\str_uppercase:N
\str_lowercase:f
\str_uppercase:f
\__str_change_case:nn
\__str_change_case_aux:nn
\__str_change_case_result:n
\__str_change_case_output:nw
\__str_change_case_output:fw
\__str_change_case_end:nw
\__str_change_case_loop:nw
\__str_change_case_space:n
\__str_change_case_char:nN
5129 \cs_new:Npn \str_foldcase:n #1 { \__str_change_case:nn {#1} { fold } }
5130 \cs_new:Npn \str_lowercase:n #1 { \__str_change_case:nn {#1} { lower } }
5131 \cs_new:Npn \str_uppercase:n #1 { \__str_change_case:nn {#1} { upper } }
5132 \cs_generate_variant:Nn \str_foldcase:n { V }
5133 \cs_generate_variant:Nn \str_lowercase:n { f }
5134 \cs_generate_variant:Nn \str_uppercase:n { f }
5135 \cs_new:Npn \__str_change_case:nn #1
5136 {
5137   \exp_after:wN \__str_change_case_aux:nn \exp_after:wN
5138   { \tl_to_str:n {#1} }
5139 }
5140 \cs_new:Npn \__str_change_case_aux:nn #1#2
5141 {
5142   \__str_change_case_loop:nw {#2} #1 \q__str_recursion_tail \q__str_recursion_stop
5143   \__str_change_case_result:n { }
5144 }
5145 \cs_new:Npn \__str_change_case_output:nw #1#2 \__str_change_case_result:n #3
5146 { #2 \__str_change_case_result:n { #3 #1 } }
5147 \cs_generate_variant:Nn \__str_change_case_output:nw { f }
5148 \cs_new:Npn \__str_change_case_end:wn #1 \__str_change_case_result:n #2

```

```

5149 { \tl_to_str:n {#2} }
5150 \cs_new:Npn \__str_change_case_loop:nw #1#2 \q__str_recursion_stop
5151 {
5152   \tl_if_head_is_space:nTF {#2}
5153     { \__str_change_case_space:n }
5154     { \__str_change_case_char:nN }
5155   {#1} #2 \q__str_recursion_stop
5156 }
5157 \exp_last_unbraced:NNNNo
5158 \cs_new:Npn \__str_change_case_space:n #1 \c_space_tl
5159 {
5160   \__str_change_case_output:nw { ~ }
5161   \__str_change_case_loop:nw {#1}
5162 }
5163 \cs_new:Npn \__str_change_case_char:nN #1#2
5164 {
5165   \__str_if_recursion_tail_stop_do:Nn #2
5166   { \__str_change_case_end:wn }
5167   \__str_change_case_output:fw
5168   { \use:c { char_str_ #1 case:N } #2 }
5169   \__str_change_case_loop:nw {#1}
5170 }

```

(End definition for `\str_foldcase:n` and others. These functions are documented on page 70.)

<code>\c_ampersand_str</code>	For all of those strings, use <code>\cs_to_str:N</code> to get characters with the correct category
<code>\c_atsign_str</code>	code without worries
<code>\c_backslash_str</code>	5171 <code>\str_const:Nx \c_ampersand_str { \cs_to_str:N \& }</code>
<code>\c_left_brace_str</code>	5172 <code>\str_const:Nx \c_atsign_str { \cs_to_str:N \@ }</code>
<code>\c_right_brace_str</code>	5173 <code>\str_const:Nx \c_backslash_str { \cs_to_str:N \\ }</code>
<code>\c_circumflex_str</code>	5174 <code>\str_const:Nx \c_left_brace_str { \cs_to_str:N \{ }</code>
<code>\c_colon_str</code>	5175 <code>\str_const:Nx \c_right_brace_str { \cs_to_str:N \} }</code>
<code>\c_dollar_str</code>	5176 <code>\str_const:Nx \c_circumflex_str { \cs_to_str:N \^ }</code>
<code>\c_hash_str</code>	5177 <code>\str_const:Nx \c_colon_str { \cs_to_str:N \: }</code>
<code>\c_percent_str</code>	5178 <code>\str_const:Nx \c_dollar_str { \cs_to_str:N \\$ }</code>
<code>\c_tilde_str</code>	5179 <code>\str_const:Nx \c_hash_str { \cs_to_str:N \# }</code>
<code>\c_underscore_str</code>	5180 <code>\str_const:Nx \c_percent_str { \cs_to_str:N \% }</code>
	5181 <code>\str_const:Nx \c_tilde_str { \cs_to_str:N \~ }</code>
	5182 <code>\str_const:Nx \c_underscore_str { \cs_to_str:N _ }</code>

(End definition for `\c_ampersand_str` and others. These variables are documented on page 71.)

<code>\l_tmpa_str</code>	Scratch strings.
<code>\l_tmpb_str</code>	5183 <code>\str_new:N \l_tmpa_str</code>
<code>\g_tmpa_str</code>	5184 <code>\str_new:N \l_tmpb_str</code>
<code>\g_tmpb_str</code>	5185 <code>\str_new:N \g_tmpa_str</code>
	5186 <code>\str_new:N \g_tmpb_str</code>

(End definition for `\l_tmpa_str` and others. These variables are documented on page 71.)

9.10 Viewing strings

<code>\str_show:n</code>	Displays a string on the terminal.
<code>\str_show:N</code>	5187 <code>\cs_new_eq:NN \str_show:n \tl_show:n</code>
<code>\str_show:c</code>	
<code>\str_log:n</code>	
<code>\str_log:N</code>	
<code>\str_log:c</code>	

```

5188 \cs_new_eq:NN \str_show:N \tl_show:N
5189 \cs_generate_variant:Nn \str_show:N { c }
5190 \cs_new_eq:NN \str_log:n \tl_log:n
5191 \cs_new_eq:NN \str_log:N \tl_log:N
5192 \cs_generate_variant:Nn \str_log:N { c }

```

(End definition for `\str_show:n` and others. These functions are documented on page 70.)

```

5193 \>/package>

```

10 l3str-convert implementation

```

5194 \>*package>

```

```

5195 \>@@=str>

```

10.1 Helpers

10.1.1 Variables and constants

```

\__str_tmp:w Internal scratch space for some functions.
\l__str_internal_int 5196 \cs_new_protected:Npn \__str_tmp:w { }
\l__str_internal_tl 5197 \tl_new:N \l__str_internal_tl
5198 \int_new:N \l__str_internal_int

```

(End definition for `__str_tmp:w`, `\l__str_internal_int`, and `\l__str_internal_tl`.)

`\g__str_result_tl` The `\g__str_result_tl` variable is used to hold the result of various internal string operations (mostly conversions) which are typically performed in a group. The variable is global so that it remains defined outside the group, to be assigned to a user-provided variable.

```

5199 \tl_new:N \g__str_result_tl

```

(End definition for `\g__str_result_tl`.)

`\c__str_replacement_char_int` When converting, invalid bytes are replaced by the Unicode replacement character "FFFD.

```

5200 \int_const:Nn \c__str_replacement_char_int { "FFFD }

```

(End definition for `\c__str_replacement_char_int`.)

```

\c__str_max_byte_int The maximal byte number.
5201 \int_const:Nn \c__str_max_byte_int { 255 }

```

(End definition for `\c__str_max_byte_int`.)

```

\s__str Internal scan marks.
5202 \scan_new:N \s__str

```

(End definition for `\s__str`.)

```

\q__str_nil Internal quarks.
5203 \quark_new:N \q__str_nil

```

(End definition for `\q__str_nil`.)

<code>\g__str_alias_prop</code>	To avoid needing one file per encoding/escaping alias, we keep track of those in a property list.
---------------------------------	---

```

5204 \prop_new:N \g__str_alias_prop
5205 \prop_gput:Nnn \g__str_alias_prop { latin1 } { iso88591 }
5206 \prop_gput:Nnn \g__str_alias_prop { latin2 } { iso88592 }
5207 \prop_gput:Nnn \g__str_alias_prop { latin3 } { iso88593 }
5208 \prop_gput:Nnn \g__str_alias_prop { latin4 } { iso88594 }
5209 \prop_gput:Nnn \g__str_alias_prop { latin5 } { iso88599 }
5210 \prop_gput:Nnn \g__str_alias_prop { latin6 } { iso885910 }
5211 \prop_gput:Nnn \g__str_alias_prop { latin7 } { iso885913 }
5212 \prop_gput:Nnn \g__str_alias_prop { latin8 } { iso885914 }
5213 \prop_gput:Nnn \g__str_alias_prop { latin9 } { iso885915 }
5214 \prop_gput:Nnn \g__str_alias_prop { latin10 } { iso885916 }
5215 \prop_gput:Nnn \g__str_alias_prop { utf16le } { utf16 }
5216 \prop_gput:Nnn \g__str_alias_prop { utf16be } { utf16 }
5217 \prop_gput:Nnn \g__str_alias_prop { utf32le } { utf32 }
5218 \prop_gput:Nnn \g__str_alias_prop { utf32be } { utf32 }
5219 \prop_gput:Nnn \g__str_alias_prop { hexadecimal } { hex }

```

(End definition for \g_str_alias prop.)

`\g_str_error_bool` In conversion functions with a built-in conditional, errors are not reported directly to the user, but the information is collected in this boolean, used at the end to decide on which branch of the conditional to take.

```
5220 \bool_new:N \g_str_error_bool
```

(End definition for \g_str_error bool.)

str_byte	Conversions from one <i><encoding>/<escaping></i> pair to another are done within x-expanding
str_error	assignments. Errors are signalled by raising the relevant flag.

```
5221 \flag_new:n { str_byte }
5222 \flag_new:n { str_error }
```

(End definition for `str byte` and `str error`. These variables are documented on page ??.)

10.2 String conditionals

```

\__str_if_contains_char:NnT \__str_if_contains_char:nnTF {⟨token list⟩} ⟨char⟩
\__str_if_contains_char:NnTF      Expects the ⟨token list⟩ to be an ⟨other string⟩: the caller is responsible for ensuring
\__str_if_contains_char:nnTF      that no (too-)special catcodes remain. Loop over the characters of the string, comparing
    \_str_if_contains_char_aux:nn character codes. The loop is broken if character codes match. Otherwise we return
    \_str_if_contains_char_auxi:nN “false”.
    \_str_if_contains_char_true:

5223 \prg_new_conditional:Npnn \__str_if_contains_char:Nn #1#2 { T , TF }
5224 {
5225     \exp_after:wN \__str_if_contains_char_aux:nn \exp_after:wN {#1} {#2}
5226     { \prg_break:n { ? \fi: } }
5227     \prg_break_point:
5228     \prg_return_false:
5229 }
5230 \cs_new:Npn \__str_if_contains_char_aux:nn #1#2
5231 { \__str_if_contains_char_auxi:nN {#2} #1 }
5232 \prg_new_conditional:Npnn \__str_if_contains_char:nn #1#2 { TF }
5233 {

```

```

5234     \_str_if_contains_char_auxi:nN {#2} #1 { \prg_break:n { ? \fi: } }
5235     \prg_break_point:
5236     \prg_return_false:
5237 }
5238 \cs_new:Npn \_str_if_contains_char_auxi:nN #1#2
5239 {
5240     \if_charcode:w #1 #2
5241     \exp_after:wN \_str_if_contains_char_true:
5242     \fi:
5243     \_str_if_contains_char_auxi:nN {#1}
5244 }
5245 \cs_new:Npn \_str_if_contains_char_true:
5246 { \prg_break:n { \prg_return_true: \use_none:n } }

```

(End definition for _str_if_contains_char:NnT and others.)

`_str_octal_use:NTF` `_str_octal_use:NTF <token> {(true code)} {(false code)}`
 If the `<token>` is an octal digit, it is left in the input stream, *followed* by the `<true code>`. Otherwise, the `<false code>` is left in the input stream.

T_EXhackers note: This function will fail if the escape character is an octal digit. We are thus careful to set the escape character to a known value before using it. T_EX dutifully detects

octal digits for us: if #1 is an octal digit, then the right-hand side of the comparison is '1#1, greater than 1. Otherwise, the right-hand side stops as '1, and the conditional takes the false branch.

```

5247 \prg_new_conditional:Npnn \_str_octal_use:N #1 { TF }
5248 {
5249     \if_int_compare:w 1 < '1 \token_to_str:N #1 \exp_stop_f:
5250     #1 \prg_return_true:
5251     \else:
5252     \prg_return_false:
5253     \fi:
5254 }

```

(End definition for _str_octal_use:NTF.)

`_str_hexadecimal_use:NTF` T_EX detects uppercase hexadecimal digits for us (see _str_octal_use:NTF), but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```

5255 \prg_new_conditional:Npnn \_str_hexadecimal_use:N #1 { TF }
5256 {
5257     \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
5258     #1 \prg_return_true:
5259     \else:
5260     \if_case:w \int_eval:n { \exp_after:wN ' \token_to_str:N #1 - 'a }
5261     A
5262     \or: B
5263     \or: C
5264     \or: D
5265     \or: E
5266     \or: F
5267     \else:
5268     \prg_return_false:
5269     \exp_after:wN \use_none:n

```

```

5270     \fi:
5271     \prg_return_true:
5272 \fi:
5273 }

```

(End definition for `_str_hexadecimal_use:NTF`.)

10.3 Conversions

10.3.1 Producing one byte or character

`\c__str_byte_0_tl` For each integer N in the range $[0, 255]$, we create a constant token list which holds three character tokens with category code other: the character with character code N , followed by the representation of N as two hexadecimal digits. The value -1 is given a default token list which ensures that later functions give an empty result for the input -1 .

```

5274 \group_begin:
5275   \tl_set:Nx \l__str_internal_tl { \tl_to_str:n { 0123456789ABCDEF } }
5276   \tl_map_inline:Nn \l__str_internal_tl
5277   {
5278     \tl_map_inline:Nn \l__str_internal_tl
5279     {
5280       \tl_const:cx { c__str_byte_ \int_eval:n {"#1##1"} _tl }
5281       { \char_generate:nn { "#1##1" } { 12 } #1 ##1 }
5282     }
5283   }
5284 \group_end:
5285 \tl_const:cn { c__str_byte_-1_tl } { { } \use_none:n { } }

```

(End definition for `\c__str_byte_0_tl` and others.)

`__str_output_byte:n` Those functions must be used carefully: feeding them a value outside the range $[-1, 255]$ will attempt to use the undefined token list variable `\c__str_byte_⟨number⟩_tl`. Assuming that the argument is in the right range, we expand the corresponding token list, and pick either the byte (first token) or the hexadecimal representations (second and third tokens). The value -1 produces an empty result in both cases.

```

5286 \cs_new:Npn \__str_output_byte:n #1
5287 { \__str_output_byte:w #1 \__str_output_end: }
5288 \cs_new:Npn \__str_output_byte:w
5289 {
5290   \exp_after:wN \exp_after:wN
5291   \exp_after:wN \use_i:nnn
5292   \cs:w c__str_byte_ \int_eval:w
5293 }
5294 \cs_new:Npn \__str_output_hexadecimal:n #1
5295 {
5296   \exp_after:wN \exp_after:wN
5297   \exp_after:wN \use_none:n
5298   \cs:w c__str_byte_ \int_eval:n {#1} _tl \cs_end:
5299 }
5300 \cs_new:Npn \__str_output_end:
5301 { \scan_stop: _tl \cs_end: }

```

(End definition for `__str_output_byte:n` and others.)

`__str_output_byte_pair_be:n` Convert a number in the range [0,65535] to a pair of bytes, either big-endian or little-endian.
`__str_output_byte_pair_le:n`
`__str_output_byte_pair:nnN`

```

5302 \cs_new:Npn \__str_output_byte_pair_be:n #1
5303 {
5304     \exp_args:Nf \__str_output_byte_pair:nnN
5305     { \int_div_truncate:nn { #1 } { "100 } } {#1} \use:nn
5306 }
5307 \cs_new:Npn \__str_output_byte_pair_le:n #1
5308 {
5309     \exp_args:Nf \__str_output_byte_pair:nnN
5310     { \int_div_truncate:nn { #1 } { "100 } } {#1} \use_ii_i:nn
5311 }
5312 \cs_new:Npn \__str_output_byte_pair:nnN #1#2#3
5313 {
5314     #3
5315     { \__str_output_byte:n { #1 } }
5316     { \__str_output_byte:n { #2 - #1 * "100 } }
5317 }

```

(End definition for `__str_output_byte_pair_be:n`, `__str_output_byte_pair_le:n`, and `__str_output_byte_pair:nnN`.)

10.3.2 Mapping functions for conversions

`__str_convert_gmap:N` This maps the function #1 over all characters in `\g__str_result_tl`, which should be a
`__str_convert_gmap_loop:NN` byte string in most cases, sometimes a native string.

```

5318 \cs_new_protected:Npn \__str_convert_gmap:N #1
5319 {
5320     \tl_gset:Nx \g__str_result_tl
5321     {
5322         \exp_after:wN \__str_convert_gmap_loop:NN
5323         \exp_after:wN #1
5324         \g__str_result_tl { ? \prg_break: }
5325         \prg_break_point:
5326     }
5327 }
5328 \cs_new:Npn \__str_convert_gmap_loop:NN #1#2
5329 {
5330     \use_none:n #2
5331     #1#2
5332     \__str_convert_gmap_loop:NN #1
5333 }

```

(End definition for `__str_convert_gmap:N` and `__str_convert_gmap_loop:NN`.)

`__str_convert_gmap_internal:N` This maps the function #1 over all character codes in `\g__str_result_tl`, which must
`__str_convert_gmap_internal_loop:Nw` be in the internal representation.

```

5334 \cs_new_protected:Npn \__str_convert_gmap_internal:N #1
5335 {
5336     \tl_gset:Nx \g__str_result_tl
5337     {
5338         \exp_after:wN \__str_convert_gmap_internal_loop:Nw
5339         \exp_after:wN #1
5340         \g__str_result_tl \s__str \s__str_stop \prg_break: \s__str

```

```

5341         \prg_break_point:
5342     }
5343 }
5344 \cs_new:Npn \__str_convert_gmap_internal_loop:Nww #1 #2 \s__str #3 \s__str
5345 {
5346     \__str_use_none_delimit_by_s_stop:w #3 \s__str_stop
5347     #1 {#3}
5348     \__str_convert_gmap_internal_loop:Nww #1
5349 }

```

(End definition for `__str_convert_gmap_internal:N` and `__str_convert_gmap_internal_loop:Nw`.)

10.3.3 Error-reporting during conversion

`__str_if_flag_error:nnx` When converting using the function `\str_set_convert:Nnnn`, errors should be reported to the user after each step in the conversion. Errors are signalled by raising some flag (typically `@@_error`), so here we test that flag: if it is raised, give the user an error, otherwise remove the arguments. On the other hand, in the conditional functions `\str_set_convert:NnnnTF`, errors should be suppressed. This is done by changing `__str_if_flag_error:nnx` into `__str_if_flag_no_error:nnx` locally.

```

5350 \cs_new_protected:Npn \__str_if_flag_error:nnx #1
5351 {
5352     \flag_if_raised:nTF {#1}
5353     { \__kernel_msg_error:nnx { str } }
5354     { \use_none:nn }
5355 }
5356 \cs_new_protected:Npn \__str_if_flag_no_error:nnx #1#2#3
5357 { \flag_if_raised:nT {#1} { \bool_gset_true:N \g__str_error_bool } }

```

(End definition for `__str_if_flag_error:nnx` and `__str_if_flag_no_error:nnx`.)

`__str_if_flag_times:nT` At the end of each conversion step, we raise all relevant errors as one error message, built on the fly. The height of each flag indicates how many times a given error was encountered. This function prints `#2` followed by the number of occurrences of an error if it occurred, nothing otherwise.

```

5358 \cs_new:Npn \__str_if_flag_times:nT #1#2
5359 { \flag_if_raised:nT {#1} { #2~(x \flag_height:n {#1} ) } }

```

(End definition for `__str_if_flag_times:nT`.)

10.3.4 Framework for conversions

Most functions in this module expect to be working with “native” strings. Strings can also be stored as bytes, in one of many encodings, for instance UTF8. The bytes themselves can be expressed in various ways in terms of T_EX tokens, for instance as pairs of hexadecimal digits. The questions of going from arbitrary Unicode code points to bytes, and from bytes to tokens are mostly independent.

Conversions are done in four steps:

- “unescape” produces a string of bytes;
- “decode” takes in a string of bytes, and converts it to a list of Unicode characters in an internal representation, with items of the form

$\langle bytes \rangle \backslash s_str \langle Unicode\ code\ point \rangle \backslash s_str$

where we have collected the $\langle bytes \rangle$ which combined to form this particular Unicode character, and the $\langle Unicode\ code\ point \rangle$ is in the range [0, "10FFFF].

- “encode” encodes the internal list of code points as a byte string in the new encoding;
- “escape” escapes bytes as requested.

The process is modified in case one of the encoding is empty (or the conversion function has been set equal to the empty encoding because it was not found): then the unescape or escape step is ignored, and the decode or encode steps work on tokens instead of bytes. Otherwise, each step must ensure that it passes a correct byte string or internal string to the next step.

`\str_set_convert:Nnnn` The input string is stored in `\g__str_result_tl`, then we: unescape and decode; encode and escape; exit the group and store the result in the user’s variable. The various conversion functions all act on `\g__str_result_tl`. Errors are silenced for the conditional functions by redefining `__str_if_flag_error:nxx` locally.

`\str_gset_convert:Nnnn`

`\str_set_convert:NnnnTF`

`\str_gset_convert:NnnnTF`

`__str_convert:nNNnnn`

```

5360 \cs_new_protected:Npn \str_set_convert:Nnnn
5361 { __str_convert:nNNnnn { } \tl_set_eq:NN }
5362 \cs_new_protected:Npn \str_gset_convert:Nnnn
5363 { __str_convert:nNNnnn { } \tl_gset_eq:NN }
5364 \prg_new_protected_conditional:Npnn
5365 \str_set_convert:Nnnn #1#2#3#4 { T , F , TF }
5366 {
5367   \bool_gset_false:N \g__str_error_bool
5368   __str_convert:nNNnnn
5369   { \cs_set_eq:NN __str_if_flag_error:nxx __str_if_flag_no_error:nxx }
5370   \tl_set_eq:NN #1 {#2} {#3} {#4}
5371   \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
5372 }
5373 \prg_new_protected_conditional:Npnn
5374 \str_gset_convert:Nnnn #1#2#3#4 { T , F , TF }
5375 {
5376   \bool_gset_false:N \g__str_error_bool
5377   __str_convert:nNNnnn
5378   { \cs_set_eq:NN __str_if_flag_error:nxx __str_if_flag_no_error:nxx }
5379   \tl_gset_eq:NN #1 {#2} {#3} {#4}
5380   \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
5381 }
5382 \cs_new_protected:Npn __str_convert:nNNnnn #1#2#3#4#5#6
5383 {
5384   \group_begin:
5385   #1
5386   \tl_gset:Nx \g__str_result_tl { __kernel_str_to_other_fast:n {#4} }
5387   \exp_after:wN __str_convert:wwwnn
5388   \tl_to_str:n {#5} /// \s__str_stop
5389   { decode } { unescape }
5390   \prg_do_nothing:
5391   __str_convert_decode_:
5392   \exp_after:wN __str_convert:wwwnn
5393   \tl_to_str:n {#6} /// \s__str_stop

```

```

5394         { encode } { escape }
5395         \use_ii_i:nn
5396         \__str_convert_encode_:
5397     \group_end:
5398     #2 #3 \g__str_result_tl
5399 }

```

(End definition for `\str_set_convert:Nnnn` and others. These functions are documented on page 72.)

`__str_convert:wwwnn` The task of `__str_convert:wwwnn` is to split $\langle encoding \rangle / \langle escaping \rangle$ pairs into their components, #1 and #2. Calls to `__str_convert:nnn` ensure that the corresponding conversion functions are defined. The third auxiliary does the main work.

`__str_convert:NNnNN`

- #1 is the encoding conversion function;
- #2 is the escaping function;
- #3 is the escaping name for use in an error message;
- #4 is `\prg_do_nothing:` for unescaping/decoding, and `\use_ii_i:nn` for encoding/escaping;
- #5 is the default encoding function (either “decode” or “encode”), for which there should be no escaping.

Let us ignore the native encoding for a second. In the unescaping/decoding phase, we want to do #2#1 in this order, and in the encoding/escaping phase, the order should be reversed: #4#2#1 does exactly that. If one of the encodings is the default (native), then the escaping should be ignored, with an error if any was given, and only the encoding, #1, should be performed.

```

5400 \cs_new_protected:Npn \__str_convert:wwwnn
5401     #1 / #2 // #3 \s__str_stop #4#5
5402 {
5403     \__str_convert:nnn {enc} {#4} {#1}
5404     \__str_convert:nnn {esc} {#5} {#2}
5405     \exp_args:Ncc \__str_convert:NNnNN
5406     { __str_convert_#4_#1: } { __str_convert_#5_#2: } {#2}
5407 }
5408 \cs_new_protected:Npn \__str_convert:NNnNN #1#2#3#4#5
5409 {
5410     \if_meaning:w #1 #5
5411         \tl_if_empty:nF {#3}
5412         { \__kernel_msg_error:nxx { str } { native-escaping } {#3} }
5413         #1
5414     \else:
5415         #4 #2 #1
5416     \fi:
5417 }

```

(End definition for `__str_convert:wwwnn` and `__str_convert:NNnNN`.)

`__str_convert:nnn` The arguments of `__str_convert:nnn` are: `enc` or `esc`, used to build filenames, the type of the conversion (unescape, decode, encode, escape), and the encoding or escaping name. If the function is already defined, no need to do anything. Otherwise, filter out all non-alphanumerics in the name, and lowercase it. Feed that, and the same three

`__str_convert:nnnn`

arguments, to `__str_convert:nnnn`. The task is then to make sure that the conversion function `#3_#1` corresponding to the type `#3` and filtered name `#1` is defined, then set our initial conversion function `#3_#4` equal to that.

How do we get the `#3_#1` conversion to be defined if it isn't? Two main cases.

First, if `#1` is a key in `\g__str_alias_prop`, then the value `\l__str_internal_tl` tells us what file to load. Loading is skipped if the file was already read, *i.e.*, if the conversion command based on `\l__str_internal_tl` already exists. Otherwise, try to load the file; if that fails, there is an error, use the default empty name instead.

Second, `#1` may be absent from the property list. The `\cs_if_exist:cF` test is automatically false, and we search for a file defining the encoding or escaping `#1` (this should allow third-party `.def` files). If the file is not found, there is an error, use the default empty name instead.

In all cases, the conversion based on `\l__str_internal_tl` is defined, so we can set the `#3_#1` function equal to that. In some cases (*e.g.*, `utf16be`), the `#3_#1` function is actually defined within the file we just loaded, and it is different from the `\l__str_internal_tl`-based function: we mustn't clobber that different definition.

```

5418 \cs_new_protected:Npn \__str_convert:nnn #1#2#3
5419 {
5420   \cs_if_exist:cF { __str_convert_#2_#3: }
5421   {
5422     \exp_args:Nx \__str_convert:nnnn
5423     { \__str_convert_lowercase_alphanum:n {#3} }
5424     {#1} {#2} {#3}
5425   }
5426 }
5427 \cs_new_protected:Npn \__str_convert:nnnn #1#2#3#4
5428 {
5429   \cs_if_exist:cF { __str_convert_#3_#1: }
5430   {
5431     \prop_get:NnNF \g__str_alias_prop {#1} \l__str_internal_tl
5432     { \tl_set:Nn \l__str_internal_tl {#1} }
5433     \cs_if_exist:cF { __str_convert_#3_ \l__str_internal_tl : }
5434     {
5435       \file_if_exist:nTF { l3str-#2- \l__str_internal_tl .def }
5436       {
5437         \group_begin:
5438         \__str_load_catcodes:
5439         \file_input:n { l3str-#2- \l__str_internal_tl .def }
5440         \group_end:
5441       }
5442       {
5443         \tl_clear:N \l__str_internal_tl
5444         \__kernel_msg_error:nxxx { str } { unknown-#2 } {#4} {#1}
5445       }
5446     }
5447     \cs_if_exist:cF { __str_convert_#3_#1: }
5448     {
5449       \cs_gset_eq:cc { __str_convert_#3_#1: }
5450       { __str_convert_#3_ \l__str_internal_tl : }
5451     }
5452   }
5453   \cs_gset_eq:cc { __str_convert_#3_#4: } { __str_convert_#3_#1: }

```

```
5454 }
```

(End definition for `__str_convert:nnn` and `__str_convert:nnnn`.)

`__str_convert_lowercase_alphanum:n` This function keeps only letters and digits, with upper case letters converted to lower case.
`__str_convert_lowercase_alphanum_loop:N`

```
5455 \cs_new:Npn \__str_convert_lowercase_alphanum:n #1
5456 {
5457   \exp_after:wN \__str_convert_lowercase_alphanum_loop:N
5458   \tl_to_str:n {#1} { ? \prg_break: }
5459   \prg_break_point:
5460 }
5461 \cs_new:Npn \__str_convert_lowercase_alphanum_loop:N #1
5462 {
5463   \use_none:n #1
5464   \if_int_compare:w '#1 > 'Z \exp_stop_f:
5465   \if_int_compare:w '#1 > 'z \exp_stop_f: \else:
5466     \if_int_compare:w '#1 < 'a \exp_stop_f: \else:
5467       #1
5468     \fi:
5469   \fi:
5470   \else:
5471     \if_int_compare:w '#1 < 'A \exp_stop_f:
5472     \if_int_compare:w 1 < 1#1 \exp_stop_f:
5473       #1
5474     \fi:
5475     \else:
5476       \__str_output_byte:n { '#1 + 'a - 'A }
5477     \fi:
5478   \fi:
5479   \__str_convert_lowercase_alphanum_loop:N
5480 }
```

(End definition for `__str_convert_lowercase_alphanum:n` and `__str_convert_lowercase_alphanum_loop:N`.)

`__str_load_catcodes:` Since encoding files may be loaded at arbitrary places in a T_EX document, including within verbatim mode, we set the catcodes of all characters appearing in any encoding definition file.

```
5481 \cs_new_protected:Npn \__str_load_catcodes:
5482 {
5483   \char_set_catcode_escape:N \
5484   \char_set_catcode_group_begin:N \{
5485   \char_set_catcode_group_end:N \}
5486   \char_set_catcode_math_toggle:N \$
5487   \char_set_catcode_alignment:N &
5488   \char_set_catcode_parameter:N #
5489   \char_set_catcode_math_superscript:N ^
5490   \char_set_catcode_ignore:N %
5491   \char_set_catcode_space:N ~
5492   \tl_map_function:nN { abcdefghijklmnopqrstuvwxyz_ABCDEFILNPSTUX }
5493   \char_set_catcode_letter:N
5494   \tl_map_function:nN { 0123456789" '*+-.() , ' ! / < > [ ] ; = }
5495   \char_set_catcode_other:N
```

```

5496     \char_set_catcode_comment:N \%
5497     \int_set:Nn \tex_endlinechar:D {32}
5498   }

```

(End definition for `__str_load_catcodes:`.)

10.3.5 Byte unescape and escape

Strings of bytes may need to be stored in auxiliary files in safe “escaping” formats. Each such escaping is only loaded as needed. By default, on input any non-byte is filtered out, while the output simply consists in letting bytes through.

In the case of 8-bit engines, every character is a byte. For Unicode-aware engines, test the character code; non-bytes cause us to raise the flag `str_byte`. Spaces have already been given the correct category code when this function is called.

```

\__str_filter_bytes:n
\__str_filter_bytes_aux:N

5499 \bool_lazy_any:nTF
5500 {
5501   \sys_if_engine luatex_p:
5502   \sys_if_engine xetex_p:
5503 }
5504 {
5505   \cs_new:Npn \__str_filter_bytes:n #1
5506   {
5507     \__str_filter_bytes_aux:N #1
5508     { ? \prg_break: }
5509     \prg_break_point:
5510   }
5511   \cs_new:Npn \__str_filter_bytes_aux:N #1
5512   {
5513     \use_none:n #1
5514     \if_int_compare:w '#1 < 256 \exp_stop_f:
5515       #1
5516     \else:
5517       \flag_raise:n { str_byte }
5518     \fi:
5519     \__str_filter_bytes_aux:N
5520   }
5521 }
5522 { \cs_new_eq:NN \__str_filter_bytes:n \use:n }

```

(End definition for `__str_filter_bytes:n` and `__str_filter_bytes_aux:N`.)

`__str_convert_unescape:` The simplest unescaping method removes non-bytes from `\g__str_result_tl`.

```

\__str_convert_unescape_bytes:
5523 \bool_lazy_any:nTF
5524 {
5525   \sys_if_engine luatex_p:
5526   \sys_if_engine xetex_p:
5527 }
5528 {
5529   \cs_new_protected:Npn \__str_convert_unescape_:
5530   {
5531     \flag_clear:n { str_byte }
5532     \tl_gset:Nx \g__str_result_tl
5533     { \exp_args:No \__str_filter_bytes:n \g__str_result_tl }

```

```

5534         \_str_if_flag_error:nmx { str_byte } { non-byte } { bytes }
5535     }
5536 }
5537 { \cs_new_protected:Npn \_str_convert_unescape_: { } }
5538 \cs_new_eq:NN \_str_convert_unescape_bytes: \_str_convert_unescape_:

```

(End definition for _str_convert_unescape_: and _str_convert_unescape_bytes:.)

_str_convert_escape_: The simplest form of escape leaves the bytes from the previous step of the conversion unchanged.

```

5539 \cs_new_protected:Npn \_str_convert_escape_: { }
5540 \cs_new_eq:NN \_str_convert_escape_bytes: \_str_convert_escape_:

```

(End definition for _str_convert_escape_: and _str_convert_escape_bytes:.)

10.3.6 Native strings

_str_convert_decode_: Convert each character to its character code, one at a time.

```

5541 \cs_new_protected:Npn \_str_convert_decode_:
5542 { \_str_convert_gmap:N \_str_decode_native_char:N }
5543 \cs_new:Npn \_str_decode_native_char:N #1
5544 { #1 \s_str \int_value:w '#1 \s_str }

```

(End definition for _str_convert_decode_: and _str_decode_native_char:N.)

_str_convert_encode_: The conversion from an internal string to native character tokens basically maps \char_generate:nn through the code-points, but in non-Unicode-aware engines we use a fall-back character ? rather than nothing when given a character code outside [0,255]. We detect the presence of bad characters using a flag and only produce a single error after the x-expanding assignment.

```

5545 \bool_lazy_any:nTF
5546 {
5547     \sys_if_engine luatex_p:
5548     \sys_if_engine xetex_p:
5549 }
5550 {
5551     \cs_new_protected:Npn \_str_convert_encode_:
5552     { \_str_convert_gmap_internal:N \_str_encode_native_char:n }
5553     \cs_new:Npn \_str_encode_native_char:n #1
5554     { \char_generate:nn {#1} {12} }
5555 }
5556 {
5557     \cs_new_protected:Npn \_str_convert_encode_:
5558     {
5559         \flag_clear:n { str_error }
5560         \_str_convert_gmap_internal:N \_str_encode_native_char:n
5561         \_str_if_flag_error:nmx { str_error }
5562         { native-overflow } { }
5563     }
5564     \cs_new:Npn \_str_encode_native_char:n #1
5565     {
5566         \if_int_compare:w #1 > \c__str_max_byte_int
5567         \flag_raise:n { str_error }
5568         ?

```

```

5569         \else:
5570             \char_generate:nm {#1} {12}
5571         \fi:
5572     }
5573     \__kernel_msg_new:nmm { str } { native-overflow }
5574     { Character~code~too~large~for~this~engine. }
5575     {
5576         This~engine~only~support~8-bit~characters:~
5577         valid~character~codes~are~in~the~range~[0,255].~
5578         To~manipulate~arbitrary~Unicode,~use~LuaTeX~or~XeTeX.
5579     }
5580 }

```

(End definition for `__str_convert_encode_:` and `__str_encode_native_char:n`.)

10.3.7 clist

`__str_convert_decode_clist:` Convert each integer to the internal form. We first turn `\g__str_result_tl` into a clist variable, as this avoids problems with leading or trailing commas.

```

5581 \cs_new_protected:Npn \__str_convert_decode_clist:
5582 {
5583     \clist_gset:N \g__str_result_tl \g__str_result_tl
5584     \tl_gset:Nx \g__str_result_tl
5585     {
5586         \exp_args:N \clist_map_function:nN
5587         \g__str_result_tl \__str_decode_clist_char:n
5588     }
5589 }
5590 \cs_new:Npn \__str_decode_clist_char:n #1
5591 { #1 \s__str \int_eval:n {#1} \s__str }

```

(End definition for `__str_convert_decode_clist:` and `__str_decode_clist_char:n`.)

`__str_convert_encode_clist:` Convert the internal list of character codes to a comma-list of character codes. The first line produces a comma-list with a leading comma, removed in the next step (this also works in the empty case, since `\tl_tail:N` does not trigger an error in this case).

```

5592 \cs_new_protected:Npn \__str_convert_encode_clist:
5593 {
5594     \__str_convert_gmap_internal:N \__str_encode_clist_char:n
5595     \tl_gset:Nx \g__str_result_tl { \tl_tail:N \g__str_result_tl }
5596 }
5597 \cs_new:Npn \__str_encode_clist_char:n #1 { , #1 }

```

(End definition for `__str_convert_encode_clist:` and `__str_encode_clist_char:n`.)

10.3.8 8-bit encodings

This section will be entirely rewritten: it is not yet clear in what situations 8-bit encodings are used, hence I don't know what exactly should be optimized. The current approach is reasonably efficient to convert long strings, and it scales well when using many different encodings. An approach based on csnames would have a smaller constant load time for each individual conversion, but has a large hash table cost. Using a range of `\count` registers works for decoding, but not for encoding: one possibility there would be to use

a binary tree for the mapping of Unicode characters to bytes, stored as a box, one per encoding.

Since the section is going to be rewritten, documentation lacks.

All the 8-bit encodings which l3str supports rely on the same internal functions.

`\str_declare_eight_bit_encoding:nnn` All the 8-bit encoding definition file start with `\str_declare_eight_bit_encoding:nnn` $\langle encoding\ name \rangle$ $\{ \langle mapping \rangle \}$ $\{ \langle missing\ bytes \rangle \}$. The $\langle mapping \rangle$ argument is a token list of pairs $\{ \langle byte \rangle \}$ $\{ \langle Unicode \rangle \}$ expressed in uppercase hexadecimal notation. The $\langle missing \rangle$ argument is a token list of $\{ \langle byte \rangle \}$. Every $\langle byte \rangle$ which does not appear in the $\langle mapping \rangle$ nor the $\langle missing \rangle$ lists maps to the same code point in Unicode.

```
5598 \cs_new_protected:Npn \str_declare_eight_bit_encoding:nnn #1#2#3
5599 {
5600   \tl_set:Nn \l__str_internal_tl {#1}
5601   \cs_new_protected:cpn { __str_convert_decode_#1: }
5602     { \__str_convert_decode_eight_bit:n {#1} }
5603   \cs_new_protected:cpn { __str_convert_encode_#1: }
5604     { \__str_convert_encode_eight_bit:n {#1} }
5605   \tl_const:cn { c__str_encoding_#1_tl } {#2}
5606   \tl_const:cn { c__str_encoding_#1_missing_tl } {#3}
5607 }
```

(End definition for `\str_declare_eight_bit_encoding:nnn`. This function is documented on page 74.)

```
\__str_convert_decode_eight_bit:n
\__str_decode_eight_bit_load:nn
\__str_decode_eight_bit_load_missing:n
\__str_decode_eight_bit_char:N
5608 \cs_new_protected:Npn \__str_convert_decode_eight_bit:n #1
5609 {
5610   \group_begin:
5611     \int_zero:N \l__str_internal_int
5612     \exp_last_unbraced:Nx \__str_decode_eight_bit_load:nn
5613       { \tl_use:c { c__str_encoding_#1_tl } }
5614     { \s__str_stop \prg_break: } { }
5615     \prg_break_point:
5616     \exp_last_unbraced:Nx \__str_decode_eight_bit_load_missing:n
5617       { \tl_use:c { c__str_encoding_#1_missing_tl } }
5618     { \s__str_stop \prg_break: }
5619     \prg_break_point:
5620     \flag_clear:n { str_error }
5621     \__str_convert_gmap:N \__str_decode_eight_bit_char:N
5622     \__str_if_flag_error:nnx { str_error } { decode-8-bit } {#1}
5623   \group_end:
5624 }
5625 \cs_new_protected:Npn \__str_decode_eight_bit_load:nn #1#2
5626 {
5627   \__str_use_none_delimit_by_s_stop:w #1 \s__str_stop
5628   \tex_dimen:D "#1 = \l__str_internal_int sp \scan_stop:
5629   \tex_skip:D \l__str_internal_int = "#1 sp \scan_stop:
5630   \tex_toks:D \l__str_internal_int \exp_after:wN { \int_value:w "#2 }
5631   \int_incr:N \l__str_internal_int
5632   \__str_decode_eight_bit_load:nn
5633 }
5634 \cs_new_protected:Npn \__str_decode_eight_bit_load_missing:n #1
5635 {
5636   \__str_use_none_delimit_by_s_stop:w #1 \s__str_stop
5637   \tex_dimen:D "#1 = \l__str_internal_int sp \scan_stop:
```



```

5638 \tex_skip:D \l__str_internal_int = "#1 sp \scan_stop:
5639 \tex_toks:D \l__str_internal_int \exp_after:wN
5640 { \int_use:N \c__str_replacement_char_int }
5641 \int_incr:N \l__str_internal_int
5642 \__str_decode_eight_bit_load_missing:n
5643 }
5644 \cs_new:Npn \__str_decode_eight_bit_char:N #1
5645 {
5646   #1 \s__str
5647   \if_int_compare:w \tex_dimen:D '#1 < \l__str_internal_int
5648     \if_int_compare:w \tex_skip:D \tex_dimen:D '#1 = '#1 \exp_stop_f:
5649     \tex_the:D \tex_toks:D \tex_dimen:D
5650     \fi:
5651   \fi:
5652   \int_value:w '#1 \s__str
5653 }

```

(End definition for __str_convert_decode_eight_bit:n and others.)

```

\__str_convert_encode_eight_bit:n
\__str_encode_eight_bit_load:nn
\__str_encode_eight_bit_char:n
\__str_encode_eight_bit_char_aux:n
5654 \cs_new_protected:Npn \__str_convert_encode_eight_bit:n #1
5655 {
5656   \group_begin:
5657     \int_zero:N \l__str_internal_int
5658     \exp_last_unbraced:Nx \__str_encode_eight_bit_load:nn
5659     { \tl_use:c { c__str_encoding_#1_tl } }
5660     { \s__str_stop \prg_break: } { }
5661     \prg_break_point:
5662     \flag_clear:n { str_error }
5663     \__str_convert_gmap_internal:N \__str_encode_eight_bit_char:n
5664     \__str_if_flag_error:nnx { str_error } { encode-8-bit } {#1}
5665   \group_end:
5666 }
5667 \cs_new_protected:Npn \__str_encode_eight_bit_load:nn #1#2
5668 {
5669   \__str_use_none_delimit_by_s_stop:w #1 \s__str_stop
5670   \tex_dimen:D "#2 = \l__str_internal_int sp \scan_stop:
5671   \tex_skip:D \l__str_internal_int = "#2 sp \scan_stop:
5672   \exp_args:NNf \tex_toks:D \l__str_internal_int
5673   { \__str_output_byte:n { "#1 } }
5674   \int_incr:N \l__str_internal_int
5675   \__str_encode_eight_bit_load:nn
5676 }
5677 \cs_new:Npn \__str_encode_eight_bit_char:n #1
5678 {
5679   \if_int_compare:w #1 > \c_max_register_int
5680     \flag_raise:n { str_error }
5681   \else:
5682     \if_int_compare:w \tex_dimen:D #1 < \l__str_internal_int
5683       \if_int_compare:w \tex_skip:D \tex_dimen:D #1 = #1 \exp_stop_f:
5684       \tex_the:D \tex_toks:D \tex_dimen:D #1 \exp_stop_f:
5685       \exp_after:wN \exp_after:wN \exp_after:wN \use_none:nn
5686     \fi:
5687   \fi:

```

```

5688     \__str_encode_eight_bit_char_aux:n {#1}
5689     \fi:
5690   }
5691 \cs_new:Npn \__str_encode_eight_bit_char_aux:n #1
5692 {
5693   \if_int_compare:w #1 > \c__str_max_byte_int
5694     \flag_raise:n { str_error }
5695   \else:
5696     \__str_output_byte:n {#1}
5697   \fi:
5698 }

```

(End definition for __str_convert_encode_eight_bit:n and others.)

10.4 Messages

General messages, and messages for the encodings and escapings loaded by default (“native”, and “bytes”).

```

5699 \__kernel_msg_new:nnn { str } { unknown-esc }
5700 { Escaping-scheme~'#1'~(filtered:~'#2')~unknown. }
5701 \__kernel_msg_new:nnn { str } { unknown-enc }
5702 { Encoding-scheme~'#1'~(filtered:~'#2')~unknown. }
5703 \__kernel_msg_new:nnnn { str } { native-escaping }
5704 { The~'native'~encoding-scheme~does~not~support~any~escaping. }
5705 {
5706   Since~native~strings~do~not~consist~in~bytes,~
5707   none~of~the~escaping~methods~make~sense.~
5708   The~specified~escaping,~'#1',~will be ignored.
5709 }
5710 \__kernel_msg_new:nnn { str } { file-not-found }
5711 { File~'l3str-#1.def'~not~found. }

```

Message used when the “bytes” unescaping fails because the string given to `\str_set_convert:Nnnn` contains a non-byte. This cannot happen for the -8-bit engines. Messages used for other escapings and encodings are defined in each definition file.

```

5712 \bool_lazy_any:nT
5713 {
5714   \sys_if_engine luatex_p:
5715   \sys_if_engine xetex_p:
5716 }
5717 {
5718   \__kernel_msg_new:nnnn { str } { non-byte }
5719   { String~invalid~in~escaping~'#1':~it~may~only~contain~bytes. }
5720   {
5721     Some~characters~in~the~string~you~asked~to~convert~are~not~
5722     8-bit~characters.~Perhaps~the~string~is~a~'native'~Unicode~string?~
5723     If~it~is,~try~using~\\
5724     \\
5725     \iow_indent:n
5726     {
5727       \iow_char:N\\str_set_convert:Nnnn \\
5728       \ \ <str-var>~\{~<string>~\}~\{~native~\}~\{~<target-encoding>~\}
5729     }
5730   }

```

```
5731 }
```

Those messages are used when converting to and from 8-bit encodings.

```
5732 \__kernel_msg_new:nnnn { str } { decode-8-bit }
5733 { Invalid~string~in~encoding~'#1'. }
5734 {
5735     LaTeX~came~across~a~byte~which~is~not~defined~to~represent~
5736     any~character~in~the~encoding~'#1'.
5737 }
5738 \__kernel_msg_new:nnnn { str } { encode-8-bit }
5739 { Unicode~string~cannot~be~converted~to~encoding~'#1'. }
5740 {
5741     The~encoding~'#1'~only~contains~a~subset~of~all~Unicode~characters.~
5742     LaTeX~was~asked~to~convert~a~string~to~that~encoding,~but~that~
5743     string~contains~a~character~that~'#1'~does~not~support.
5744 }
```

10.5 Escaping definitions

Several of those encodings are defined by the pdf file format. The following byte storage methods are defined:

- **bytes** (default), non-bytes are filtered out, and bytes are left untouched (this is defined by default);
- **hex** or **hexadecimal**, as per the pdfTeX primitive `\pdfescapehex`
- **name**, as per the pdfTeX primitive `\pdfescapename`
- **string**, as per the pdfTeX primitive `\pdfescapestring`
- **url**, as per the percent encoding of urls.

10.5.1 Unescape methods

`__str_convert_unescape_hex:` Take chars two by two, and interpret each pair as the hexadecimal code for a byte.
`__str_unescape_hex_auxi:N` Anything else than hexadecimal digits is ignored, raising the flag. A string which contains
`__str_unescape_hex_auxii:N` an odd number of hexadecimal digits gets 0 appended to it: this is equivalent to appending a 0 in all cases, and dropping it if it is alone.

```
5745 \cs_new_protected:Npn \__str_convert_unescape_hex:
5746 {
5747     \group_begin:
5748     \flag_clear:n { str_error }
5749     \int_set:Nn \tex_escapechar:D { 92 }
5750     \tl_gset:Nx \g__str_result_tl
5751     {
5752         \__str_output_byte:w "
5753         \exp_last_unbraced:Nf \__str_unescape_hex_auxi:N
5754         { \tl_to_str:N \g__str_result_tl }
5755         0 { ? 0 - 1 \prg_break: }
5756         \prg_break_point:
5757         \__str_output_end:
5758     }
5759     \__str_if_flag_error:nnx { str_error } { unescape-hex } { }
```

```

5760     \group_end:
5761   }
5762   \cs_new:Npn \__str_unescape_hex_auxi:N #1
5763   {
5764     \use_none:n #1
5765     \__str_hexadecimal_use:NTF #1
5766     { \__str_unescape_hex_auxii:N }
5767     {
5768       \flag_raise:n { str_error }
5769       \__str_unescape_hex_auxi:N
5770     }
5771   }
5772   \cs_new:Npn \__str_unescape_hex_auxii:N #1
5773   {
5774     \use_none:n #1
5775     \__str_hexadecimal_use:NTF #1
5776     {
5777       \__str_output_end:
5778       \__str_output_byte:w " \__str_unescape_hex_auxi:N
5779     }
5780     {
5781       \flag_raise:n { str_error }
5782       \__str_unescape_hex_auxii:N
5783     }
5784   }
5785   \__kernel_msg_new:nnnn { str } { unescape-hex }
5786   { String~invalid~in~escaping~'hex':~only~hexadecimal~digits~allowed. }
5787   {
5788     Some~characters~in~the~string~you~asked~to~convert~are~not~
5789     hexadecimal~digits~(0-9,~A-F,~a-f)~nor~spaces.
5790   }

```

(End definition for __str_convert_unescape_hex:, __str_unescape_hex_auxi:N, and __str_unescape_hex_auxii:N.)

__str_convert_unescape_name: The __str_convert_unescape_name: function replaces each occurrence of # followed by two hexadecimal digits in \g__str_result_tl by the corresponding byte. The url function is identical, with escape character % instead of #. Thus we define the two together. The arguments of __str_tmp:w are the character code of # or % in hexadecimal, the name of the main function to define, and the name of the auxiliary which performs the loop.

The looping auxiliary #3 finds the next escape character, reads the following two characters, and tests them. The test __str_hexadecimal_use:NTF leaves the upper-case digit in the input stream, hence we surround the test with __str_output_byte:w " and __str_output_end:. If both characters are hexadecimal digits, they should be removed before looping: this is done by \use_i:nnn. If one of the characters is not a hexadecimal digit, then feed "#1 to __str_output_byte:w to produce the escape character, raise the flag, and call the looping function followed by the two characters (remove \use_i:nnn).

```

5791   \cs_set_protected:Npn \__str_tmp:w #1#2#3
5792   {
5793     \cs_new_protected:cpn { __str_convert_unescape_#2: }
5794     {

```

```

5795 \group_begin:
5796 \flag_clear:n { str_byte }
5797 \flag_clear:n { str_error }
5798 \int_set:Nn \tex_escapechar:D { 92 }
5799 \tl_gset:Nx \g__str_result_tl
5800 {
5801 \exp_after:wN #3 \g__str_result_tl
5802 #1 ? { ? \prg_break: }
5803 \prg_break_point:
5804 }
5805 \__str_if_flag_error:nmx { str_byte } { non-byte } { #2 }
5806 \__str_if_flag_error:nmx { str_error } { unescape-#2 } { }
5807 \group_end:
5808 }
5809 \cs_new:Npn #3 ##1##2##3
5810 {
5811 \__str_filter_bytes:n {##1}
5812 \use_none:n ##3
5813 \__str_output_byte:w "
5814 \__str_hexadecimal_use:NTF ##2
5815 {
5816 \__str_hexadecimal_use:NTF ##3
5817 { }
5818 {
5819 \flag_raise:n { str_error }
5820 * 0 + '#1 \use_i:nn
5821 }
5822 }
5823 {
5824 \flag_raise:n { str_error }
5825 0 + '#1 \use_i:nn
5826 }
5827 \__str_output_end:
5828 \use_i:nnn #3 ##2##3
5829 }
5830 \__kernel_msg_new:nnnn { str } { unescape-#2 }
5831 { String~invalid~in~escaping~'#2'. }
5832 {
5833 LaTeX~came~across~the~escape~character~'#1'~not~followed~by~
5834 two~hexadecimal~digits.~This~is~invalid~in~the~escaping~'#2'.
5835 }
5836 }
5837 \exp_after:wN \__str_tmp:w \c_hash_str { name }
5838 \__str_unescape_name_loop:wNN
5839 \exp_after:wN \__str_tmp:w \c_percent_str { url }
5840 \__str_unescape_url_loop:wNN

```

(End definition for `__str_convert_unescape_name:` and others.)

`__str_convert_unescape_string:` The **string** escaping is somewhat similar to the **name** and **url** escapings, with escape character `\`. The first step is to convert all three line endings, `^^J`, `^^M`, and `^^M^^J` to the common `^^J`, as per the PDF specification. This step cannot raise the flag.

`__str_unescape_string_newlines:wN`

`__str_unescape_string_loop:wNNN`

`__str_unescape_string_repeat:NNNNNN`

Then the following escape sequences are decoded.

`\n` Line feed (10)

`\r` Carriage return (13)

`\t` Horizontal tab (9)

`\b` Backspace (8)

`\f` Form feed (12)

`\(` Left parenthesis

`\)` Right parenthesis

`\\` Backslash

`\ddd` (backslash followed by 1 to 3 octal digits) Byte `ddd` (octal), subtracting 256 in case of overflow.

If followed by an end-of-line character, the backslash and the end-of-line are ignored. If followed by anything else, the backslash is ignored, raising the error flag.

```
5841 \group_begin:
5842   \char_set_catcode_other:N \^^J
5843   \char_set_catcode_other:N \^^M
5844   \cs_set_protected:Npn \__str_tmp:w #1
5845   {
5846     \cs_new_protected:Npn \__str_convert_unescape_string:
5847     {
5848       \group_begin:
5849       \flag_clear:n { str_byte }
5850       \flag_clear:n { str_error }
5851       \int_set:Nn \tex_escapechar:D { 92 }
5852       \tl_gset:Nx \g__str_result_tl
5853       {
5854         \exp_after:wN \__str_unescape_string_newlines:wN
5855         \g__str_result_tl \prg_break: ^^M ?
5856         \prg_break_point:
5857       }
5858       \tl_gset:Nx \g__str_result_tl
5859       {
5860         \exp_after:wN \__str_unescape_string_loop:wNNN
5861         \g__str_result_tl #1 ?? { ? \prg_break: }
5862         \prg_break_point:
5863       }
5864       \__str_if_flag_error:nxx { str_byte } { non-byte } { string }
5865       \__str_if_flag_error:nxx { str_error } { unescape-string } { }
5866     } \group_end:
5867   }
5868 }
5869 \exp_args:No \__str_tmp:w { \c_backslash_str }
5870 \exp_last_unbraced:NNNNo
5871   \cs_new:Npn \__str_unescape_string_loop:wNNN #1 \c_backslash_str #2#3#4
5872   {
5873     \__str_filter_bytes:n {#1}
5874     \use_none:n #4
5875     \__str_output_byte:w '
5876     \__str_octal_use:NTF #2
```

```

5877 {
5878   \__str_octal_use:NTF #3
5879   {
5880     \__str_octal_use:NTF #4
5881     {
5882       \if_int_compare:w #2 > 3 \exp_stop_f:
5883         - 256
5884       \fi:
5885       \__str_unescape_string_repeat:NNNNNN
5886     }
5887     { \__str_unescape_string_repeat:NNNNNN ? }
5888   }
5889   { \__str_unescape_string_repeat:NNNNNN ?? }
5890 }
5891 {
5892   \str_case_e:nnF {#2}
5893   {
5894     { \c_backslash_str } { 134 }
5895     { ( } { 50 }
5896     { ) } { 51 }
5897     { r } { 15 }
5898     { f } { 14 }
5899     { n } { 12 }
5900     { t } { 11 }
5901     { b } { 10 }
5902     { ^^J } { 0 - 1 }
5903   }
5904   {
5905     \flag_raise:n { str_error }
5906     0 - 1 \use_i:nn
5907   }
5908 }
5909 \__str_output_end:
5910 \use_i:nn \__str_unescape_string_loop:wNNN #2#3#4
5911 }
5912 \cs_new:Npn \__str_unescape_string_repeat:NNNNNN #1#2#3#4#5#6
5913 { \__str_output_end: \__str_unescape_string_loop:wNNN }
5914 \cs_new:Npn \__str_unescape_string_newlines:wN #1 ^^M #2
5915 {
5916   #1
5917   \if_charcode:w ^^J #2 \else: ^^J \fi:
5918   \__str_unescape_string_newlines:wN #2
5919 }
5920 \__kernel_msg_new:nnnn { str } { unescape-string }
5921 { String~invalid~in~escaping~'string'. }
5922 {
5923   LaTeX~came~across~an~escape~character~'\c_backslash_str'~
5924   not~followed~by~any~of:~'n',~'r',~'t',~'b',~'f',~'(',~')',~
5925   '\c_backslash_str',~one~to~three~octal~digits,~or~the~end~
5926   of~a~line.
5927 }
5928 \group_end:

```

(End definition for __str_convert_unescape_string: and others.)

10.5.2 Escape methods

Currently, none of the escape methods can lead to errors, assuming that their input is made out of bytes.

`__str_convert_escape_hex:` Loop and convert each byte to hexadecimal.

```
\__str_escape_hex_char:N
5929 \cs_new_protected:Npn \__str_convert_escape_hex:
5930 { \__str_convert_gmap:N \__str_escape_hex_char:N }
5931 \cs_new:Npn \__str_escape_hex_char:N #1
5932 { \__str_output_hexadecimal:n { '#1 } }
```

(End definition for `__str_convert_escape_hex:` and `__str_escape_hex_char:N`.)

`__str_convert_escape_name:` For each byte, test whether it should be output as is, or be “hash-encoded”. Roughly, bytes outside the range [“2A”, “7E”] are hash-encoded. We keep two lists of exceptions: `__str_escape_name_char:n` characters in `\c__str_escape_name_not_str` are not hash-encoded, and characters in the `\c__str_escape_name_str` are encoded.

```
\__str_escape_name_char:n
\__str_if_escape_name:nTF
\c__str_escape_name_str
\c__str_escape_name_not_str
5933 \str_const:Nn \c__str_escape_name_not_str { ! " $ & ' } %$
5934 \str_const:Nn \c__str_escape_name_str { { } / < > [ ] }
5935 \cs_new_protected:Npn \__str_convert_escape_name:
5936 { \__str_convert_gmap:N \__str_escape_name_char:n }
5937 \cs_new:Npn \__str_escape_name_char:n #1
5938 {
5939   \__str_if_escape_name:nTF {#1} {#1}
5940   { \c_hash_str \__str_output_hexadecimal:n { '#1 } }
5941 }
5942 \prg_new_conditional:Npnn \__str_if_escape_name:n #1 { TF }
5943 {
5944   \if_int_compare:w '#1 < "2A \exp_stop_f:
5945   \__str_if_contains_char:NnTF \c__str_escape_name_not_str {#1}
5946   \prg_return_true: \prg_return_false:
5947   \else:
5948   \if_int_compare:w '#1 > "7E \exp_stop_f:
5949   \prg_return_false:
5950   \else:
5951   \__str_if_contains_char:NnTF \c__str_escape_name_str {#1}
5952   \prg_return_false: \prg_return_true:
5953   \fi:
5954 \fi:
5955 }
```

(End definition for `__str_convert_escape_name:` and others.)

`__str_convert_escape_string:` Any character below (and including) space, and any character above (and including) `del`, are converted to octal. One backslash is added before each parenthesis and backslash.

```
\__str_escape_string_char:N
\__str_if_escape_string:NnTF
\c__str_escape_string_str
5956 \str_const:Nx \c__str_escape_string_str
5957 { \c_backslash_str ( ) }
5958 \cs_new_protected:Npn \__str_convert_escape_string:
5959 { \__str_convert_gmap:N \__str_escape_string_char:N }
5960 \cs_new:Npn \__str_escape_string_char:N #1
5961 {
5962   \__str_if_escape_string:NnTF #1
5963   {
5964     \__str_if_contains_char:NnT
```



```

5965         \c__str_escape_string_str {#1}
5966         { \c_backslash_str }
5967     #1
5968 }
5969 {
5970     \c_backslash_str
5971     \int_div_truncate:nn {'#1} {64}
5972     \int_mod:nn { \int_div_truncate:nn {'#1} { 8 } } { 8 }
5973     \int_mod:nn {'#1} { 8 }
5974 }
5975 }
5976 \prg_new_conditional:Npnn \__str_if_escape_string:N #1 { TF }
5977 {
5978     \if_int_compare:w '#1 < "21 \exp_stop_f:
5979     \prg_return_false:
5980 }else:
5981     \if_int_compare:w '#1 > "7E \exp_stop_f:
5982     \prg_return_false:
5983 }else:
5984     \prg_return_true:
5985 }fi:
5986 }fi:
5987 }

```

(End definition for __str_convert_escape_string: and others.)

__str_convert_escape_url: This function is similar to __str_convert_escape_name:, escaping different characters.
 __str_escape_url_char:n
 __str_if_escape_url:nTF

```

5988 \cs_new_protected:Npn \__str_convert_escape_url:
5989 { \__str_convert_gmap:N \__str_escape_url_char:n }
5990 \cs_new:Npn \__str_escape_url_char:n #1
5991 {
5992     \__str_if_escape_url:nTF {#1} {#1}
5993     { \c_percent_str \__str_output_hexadecimal:n { '#1 } }
5994 }
5995 \prg_new_conditional:Npnn \__str_if_escape_url:n #1 { TF }
5996 {
5997     \if_int_compare:w '#1 < "41 \exp_stop_f:
5998     \__str_if_contains_char:nnTF { "-.<> } {#1}
5999     \prg_return_true: \prg_return_false:
6000 }else:
6001     \if_int_compare:w '#1 > "7E \exp_stop_f:
6002     \prg_return_false:
6003 }else:
6004     \__str_if_contains_char:nnTF { [ ] } {#1}
6005     \prg_return_false: \prg_return_true:
6006 }fi:
6007 }fi:
6008 }

```

(End definition for __str_convert_escape_url:, __str_escape_url_char:n, and __str_if_escape_url:nTF.)

10.6 Encoding definitions

The **native** encoding is automatically defined. Other encodings are loaded as needed. The following encodings are supported:

- UTF-8;
- UTF-16, big-, little-endian, or with byte order mark;
- UTF-32, big-, little-endian, or with byte order mark;
- the ISO 8859 code pages, numbered from 1 to 16, skipping the inexistent ISO 8859-12.

10.6.1 utf-8 support

`__str_convert_encode_utf8:` Loop through the internal string, and convert each character to its UTF-8 representation. The representation is built from the right-most (least significant) byte to the left-most (most significant) byte. Continuation bytes are in the range [128, 191], taking 64 different values, hence we roughly want to express the character code in base 64, shifting the first digit in the representation by some number depending on how many continuation bytes there are. In the range [0, 127], output the corresponding byte directly. In the range [128, 2047], output the remainder modulo 64, plus 128 as a continuation byte, then output the quotient (which is in the range [0, 31]), shifted by 192. In the next range, [2048, 65535], split the character code into residue and quotient modulo 64, output the residue as a first continuation byte, then repeat; this leaves us with a quotient in the range [0, 15], which we output shifted by 224. The last range, [65536, 1114111], follows the same pattern: once we realize that dividing twice by 64 leaves us with a number larger than 15, we repeat, producing a last continuation byte, and offset the quotient by 240 for the leading byte.

How is that implemented? `__str_encode_utf_vii_loop:wnnw` takes successive quotients as its first argument, the quotient from the previous step as its second argument (except in step 1), the bound for quotients that trigger one more step or not, and finally the offset used if this step should produce the leading byte. Leading bytes can be in the ranges [0, 127], [192, 223], [224, 239], and [240, 247] (really, that last limit should be 244 because Unicode stops at the code point 1114111). At each step, if the quotient `#1` is less than the limit `#3` for that range, output the leading byte (`#1` shifted by `#4`) and stop. Otherwise, we need one more step: use the quotient of `#1` by 64, and `#1` as arguments for the looping auxiliary, and output the continuation byte corresponding to the remainder `#2 - 64#1 + 128`. The bizarre construction `- 1 + 0 *` removes the spurious initial continuation byte (better methods welcome).

```
6009 \cs_new_protected:cpn { __str_convert_encode_utf8: }
6010 { __str_convert_gmap_internal:N __str_encode_utf_viii_char:n }
6011 \cs_new:Npn __str_encode_utf_viii_char:n #1
6012 {
6013   __str_encode_utf_viii_loop:wnnw #1 ; - 1 + 0 * ;
6014   { 128 } { 0 }
6015   { 32 } { 192 }
6016   { 16 } { 224 }
6017   { 8 } { 240 }
6018   \s__str_stop
6019 }
6020 \cs_new:Npn __str_encode_utf_viii_loop:wnnw #1; #2; #3#4 #5 \s__str_stop
```

```

6021 {
6022   \if_int_compare:w #1 < #3 \exp_stop_f:
6023     \__str_output_byte:n { #1 + #4 }
6024     \exp_after:wN \__str_use_none_delimit_by_s_stop:w
6025   \fi:
6026   \exp_after:wN \__str_encode_utf_viii_loop:wnnw
6027     \int_value:w \int_div_truncate:nn {#1} {64} ; #1 ;
6028     #5 \s__str_stop
6029   \__str_output_byte:n { #2 - 64 * ( #1 - 2 ) }
6030 }

```

(End definition for __str_convert_encode_utf8:, __str_encode_utf_viii_char:n, and __str_encode_utf_viii_loop:wnnw.)

\l__str_missing_flag \l__str_extra_flag \l__str_overlong_flag \l__str_overflow_flag

When decoding a string that is purportedly in the UTF-8 encoding, four different errors can occur, signalled by a specific flag for each (we define those flags using \flag_clear_new:n rather than \flag_new:n, because they are shared with other encoding definition files).

- “Missing continuation byte”: a leading byte is not followed by the right number of continuation bytes.
- “Extra continuation byte”: a continuation byte appears where it was not expected, *i.e.*, not after an appropriate leading byte.
- “Overlong”: a Unicode character is expressed using more bytes than necessary, for instance, "C0"80 for the code point 0, instead of a single null byte.
- “Overflow”: this occurs when decoding produces Unicode code points greater than 1114111.

We only raise one L^AT_EX3 error message, combining all the errors which occurred. In the short message, the leading comma must be removed to get a grammatically correct sentence. In the long text, first remind the user what a correct UTF-8 string should look like, then add error-specific information.

```

6031 \flag_clear_new:n { str_missing }
6032 \flag_clear_new:n { str_extra }
6033 \flag_clear_new:n { str_overlong }
6034 \flag_clear_new:n { str_overflow }
6035 \__kernel_msg_new:nnnn { str } { utf8-decode }
6036 {
6037   Invalid~UTF-8~string:
6038   \exp_last_unbraced:Nf \use_none:n
6039   {
6040     \__str_if_flag_times:nT { str_missing } { ,~missing~continuation~byte }
6041     \__str_if_flag_times:nT { str_extra } { ,~extra~continuation~byte }
6042     \__str_if_flag_times:nT { str_overlong } { ,~overlong~form }
6043     \__str_if_flag_times:nT { str_overflow } { ,~code~point~too~large }
6044   }
6045   .
6046 }
6047 {
6048   In~the~UTF-8~encoding,~each~Unicode~character~consists~in~
6049   1~to~4~bytes,~with~the~following~bit~pattern: \\\
6050   \iow_indent:n

```

```

6051     {
6052         Code-point~\ \ \ <~128:~0xxxxxxx \
6053         Code-point~\ \ \ <~2048:~110xxxxx~10xxxxxx \
6054         Code-point~\ \ \ <~65536:~1110xxxx~10xxxxxx~10xxxxxx \
6055         Code-point~ \ \ \ <~1114112:~11110xxx~10xxxxxx~10xxxxxx~10xxxxxx \
6056     }
6057 Bytes-of-the-form-10xxxxxx-are-called-continuation-bytes.
6058 \flag_if_raised:nT { str_missing }
6059 {
6060     \\\
6061     A-leading-byte~(in-the-range~[192,255])~was~not~followed-by~
6062     the-appropriate-number-of~continuation-bytes.
6063 }
6064 \flag_if_raised:nT { str_extra }
6065 {
6066     \\\
6067     LaTeX-came-across-a-continuation-byte-when-it-was-not-expected.
6068 }
6069 \flag_if_raised:nT { str_overlong }
6070 {
6071     \\\
6072     Every~Unicode-code-point~must~be~expressed~in~the~shortest~
6073     possible~form.~For~instance,~'0xC0'~'0x83'~is~not~a~valid~
6074     representation~for~the~code~point~3.
6075 }
6076 \flag_if_raised:nT { str_overflow }
6077 {
6078     \\\
6079     Unicode-limits-code-points-to-the-range~[0,1114111].
6080 }
6081 }

```

(End definition for `\l__str_missing_flag` and others.)

`__str_convert_decode_utf8:` Decoding is significantly harder than encoding. As before, lower some flags, which are tested at the end (in bulk, to trigger at most one L^AT_EX3 error, as explained above). We expect successive multi-byte sequences of the form $\langle start\ byte \rangle \langle continuation\ bytes \rangle$. The `_start` auxiliary tests the first byte:

- [0, "7F]: the byte stands alone, and is converted to its own character code;
- ["80, "BF]: unexpected continuation byte, raise the appropriate flag, and convert that byte to the replacement character "FFFD;
- ["C0, "FF]: this byte should be followed by some continuation byte(s).

In the first two cases, `\use_none_delimit_by_q_stop:w` removes data that only the third case requires, namely the limits of ranges of Unicode characters which can be expressed with 1, 2, 3, or 4 bytes.

We can now concentrate on the multi-byte case and the `_continuation` auxiliary. We expect `#3` to be in the range ["80, "BF]. The test for this goes as follows: if the character code is less than "80, we compare it to `—"C0`, yielding `false`; otherwise to `"C0`, yielding `true` in the range ["80, "BF] and `false` otherwise. If we find that the byte is not a continuation range, stop the current slew of bytes, output the replacement

character, and continue parsing with the `_start` auxiliary, starting at the byte we just tested. Once we know that the byte is a continuation byte, leave it behind us in the input stream, compute what code point the bytes read so far would produce, and feed that number to the `_aux` function.

The `_aux` function tests whether we should look for more continuation bytes or not. If the number it receives as `#1` is less than the maximum `#4` for the current range, then we are done: check for an overlong representation by comparing `#1` with the maximum `#3` for the previous range. Otherwise, we call the `_continuation` auxiliary again, after shifting the “current code point” by `#4` (maximum from the range we just checked).

Two additional tests are needed: if we reach the end of the list of range maxima and we are still not done, then we are faced with an overflow. Clean up, and again insert the code point "FFFD" for the replacement character. Also, every time we read a byte, we need to check whether we reached the end of the string. In a correct UTF-8 string, this happens automatically when the `_start` auxiliary leaves its first argument in the input stream: the end-marker begins with `\prg_break:`, which ends the loop. On the other hand, if the end is reached when looking for a continuation byte, the `\use_none:n #3` construction removes the first token from the end-marker, and leaves the `_end` auxiliary, which raises the appropriate error flag before ending the mapping.

```

6082 \cs_new_protected:cpn { __str_convert_decode_utf8: }
6083 {
6084   \flag_clear:n { str_error }
6085   \flag_clear:n { str_missing }
6086   \flag_clear:n { str_extra }
6087   \flag_clear:n { str_overlong }
6088   \flag_clear:n { str_overflow }
6089   \tl_gset:Nx \g__str_result_tl
6090   {
6091     \exp_after:wN \__str_decode_utf_viii_start:N \g__str_result_tl
6092     { \prg_break: \__str_decode_utf_viii_end: }
6093     \prg_break_point:
6094   }
6095   \__str_if_flag_error:nxx { str_error } { utf8-decode } { }
6096 }
6097 \cs_new:Npn \__str_decode_utf_viii_start:N #1
6098 {
6099   #1
6100   \if_int_compare:w '#1 < "C0 \exp_stop_f:
6101     \s__str
6102     \if_int_compare:w '#1 < "80 \exp_stop_f:
6103       \int_value:w '#1
6104     \else:
6105       \flag_raise:n { str_extra }
6106       \flag_raise:n { str_error }
6107       \int_use:N \c__str_replacement_char_int
6108     \fi:
6109   \else:
6110     \exp_after:wN \__str_decode_utf_viii_continuation:wwN
6111     \int_value:w \int_eval:n { '#1 - "C0 } \exp_after:wN
6112   \fi:
6113   \s__str
6114   \__str_use_none_delimit_by_s_stop:w {"80} {"800} {"10000} {"110000} \s__str_stop
6115   \__str_decode_utf_viii_start:N

```

```

6116     }
6117 \cs_new:Npn \__str_decode_utf_viii_continuation:wwN
6118     #1 \s__str #2 \__str_decode_utf_viii_start:N #3
6119     {
6120     \use_none:n #3
6121     \if_int_compare:w '#3 <
6122         \if_int_compare:w '#3 < "80 \exp_stop_f: - \fi:
6123         "C0 \exp_stop_f:
6124         #3
6125         \exp_after:wN \__str_decode_utf_viii_aux:wNnnwN
6126         \int_value:w \int_eval:n { #1 * "40 + '#3 - "80 } \exp_after:wN
6127     \else:
6128         \s__str
6129         \flag_raise:n { str_missing }
6130         \flag_raise:n { str_error }
6131         \int_use:N \c__str_replacement_char_int
6132     \fi:
6133     \s__str
6134     #2
6135     \__str_decode_utf_viii_start:N #3
6136     }
6137 \cs_new:Npn \__str_decode_utf_viii_aux:wNnnwN
6138     #1 \s__str #2#3#4 #5 \__str_decode_utf_viii_start:N #6
6139     {
6140     \if_int_compare:w #1 < #4 \exp_stop_f:
6141         \s__str
6142         \if_int_compare:w #1 < #3 \exp_stop_f:
6143             \flag_raise:n { str_overlong }
6144             \flag_raise:n { str_error }
6145             \int_use:N \c__str_replacement_char_int
6146         \else:
6147             #1
6148         \fi:
6149     \else:
6150         \if_meaning:w \s__str_stop #5
6151             \__str_decode_utf_viii_overflow:w #1
6152         \fi:
6153         \exp_after:wN \__str_decode_utf_viii_continuation:wwN
6154         \int_value:w \int_eval:n { #1 - #4 } \exp_after:wN
6155     \fi:
6156     \s__str
6157     #2 {#4} #5
6158     \__str_decode_utf_viii_start:N
6159     }
6160 \cs_new:Npn \__str_decode_utf_viii_overflow:w #1 \fi: #2 \fi:
6161     {
6162     \fi: \fi:
6163     \flag_raise:n { str_overflow }
6164     \flag_raise:n { str_error }
6165     \int_use:N \c__str_replacement_char_int
6166     }
6167 \cs_new:Npn \__str_decode_utf_viii_end:
6168     {
6169     \s__str

```

```

6170     \flag_raise:n { str_missing }
6171     \flag_raise:n { str_error }
6172     \int_use:N \c__str_replacement_char_int \s__str
6173     \prg_break:
6174 }

```

(End definition for `__str_convert_decode_utf8:` and others.)

10.6.2 utf-16 support

The definitions are done in a category code regime where the bytes 254 and 255 used by the byte order mark have catcode 12.

```

6175 \group_begin:
6176   \char_set_catcode_other:N ^^fe
6177   \char_set_catcode_other:N ^^ff

```

`__str_convert_encode_utf16:` When the endianness is not specified, it is big-endian by default, and we add a byte-order mark. Convert characters one by one in a loop, with different behaviours depending on the character code.

```

\__str_convert_encode_utf16be:
\__str_convert_encode_utf16le:
\__str_encode_utf_xvi_aux:N
\__str_encode_utf_xvi_char:n

```

- [0, "D7FF]: converted to two bytes;
- ["D800, "DFFF] are used as surrogates: they cannot be converted and are replaced by the replacement character;
- ["E000, "FFFF]: converted to two bytes;
- ["10000, "10FFFF]: converted to a pair of surrogates, each two bytes. The magic "D7C0 is "D800 – "10000/"400.

For the duration of this operation, `__str_tmp:w` is defined as a function to convert a number in the range [0, "FFFF] to a pair of bytes (either big endian or little endian), by feeding the quotient of the division of #1 by "100, followed by #1 to `__str_encode_utf_xvi_be:nn` or its `le` analog: those compute the remainder, and output two bytes for the quotient and remainder.

```

6178   \cs_new_protected:cpn { __str_convert_encode_utf16: }
6179   {
6180     \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n
6181     \tl_gput_left:Nx \g__str_result_tl { ^^fe ^^ff }
6182   }
6183   \cs_new_protected:cpn { __str_convert_encode_utf16be: }
6184   { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n }
6185   \cs_new_protected:cpn { __str_convert_encode_utf16le: }
6186   { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_le:n }
6187   \cs_new_protected:Npn \__str_encode_utf_xvi_aux:N #1
6188   {
6189     \flag_clear:n { str_error }
6190     \cs_set_eq:NN \__str_tmp:w #1
6191     \__str_convert_gmap_internal:N \__str_encode_utf_xvi_char:n
6192     \__str_if_flag_error:nxx { str_error } { utf16-encode } { }
6193   }
6194   \cs_new:Npn \__str_encode_utf_xvi_char:n #1
6195   {
6196     \if_int_compare:w #1 < "D800 \exp_stop_f:

```

```

6197     \__str_tmp:w {#1}
6198 \else:
6199     \if_int_compare:w #1 < "10000 \exp_stop_f:
6200     \if_int_compare:w #1 < "E000 \exp_stop_f:
6201     \flag_raise:n { str_error }
6202     \__str_tmp:w { \c__str_replacement_char_int }
6203 \else:
6204     \__str_tmp:w {#1}
6205 \fi:
6206 \else:
6207     \exp_args:Nf \__str_tmp:w { \int_div_truncate:nn {#1} {"400} + "D7C0 }
6208     \exp_args:Nf \__str_tmp:w { \int_mod:nn {#1} {"400} + "DC00 }
6209 \fi:
6210 \fi:
6211 }

```

(End definition for __str_convert_encode_utf16: and others.)

\l__str_missing_flag When encoding a Unicode string to UTF-16, only one error can occur: code points in the range ["D800, "DFFF], corresponding to surrogates, cannot be encoded. We use the
 \l__str_extra_flag all-purpose flag @@_error to signal that error.
 \l__str_end_flag

When decoding a Unicode string which is purportedly in UTF-16, three errors can occur: a missing trail surrogate, an unexpected trail surrogate, and a string containing an odd number of bytes.

```

6212 \flag_clear_new:n { str_missing }
6213 \flag_clear_new:n { str_extra }
6214 \flag_clear_new:n { str_end }
6215 \__kernel_msg_new:nnnn { str } { utf16-encode }
6216 { Unicode-string-cannot-be-expressed-in-UTF-16:-surrogate. }
6217 {
6218   Surrogate-code-points~(in-the-range~[U+D800,~U+DFFF])~
6219   can-be-expressed-in-the~UTF-8-and~UTF-32~encodings,~
6220   but-not-in-the~UTF-16~encoding.
6221 }
6222 \__kernel_msg_new:nnnn { str } { utf16-decode }
6223 {
6224   Invalid-UTF-16-string:
6225   \exp_last_unbraced:Nf \use_none:n
6226   {
6227     \__str_if_flag_times:nT { str_missing } { ,~missing~trail~surrogate }
6228     \__str_if_flag_times:nT { str_extra } { ,~extra~trail~surrogate }
6229     \__str_if_flag_times:nT { str_end } { ,~odd~number~of~bytes }
6230   }
6231   .
6232 }
6233 {
6234   In~the~UTF-16~encoding,~each~Unicode~character~is~encoded~as~
6235   2-or~4~bytes: \\\
6236   \iow_indent:n
6237   {
6238     Code-point-in~[U+0000,~U+D7FF]:~two~bytes \\\
6239     Code-point-in~[U+D800,~U+DFFF]:~illegal \\\
6240     Code-point-in~[U+E000,~U+FFFF]:~two~bytes \\\
6241     Code-point-in~[U+10000,~U+10FFFF]:~

```



```

6242         a~lead-surrogate-and-a-trail-surrogate \\
6243     }
6244     Lead-surrogates-are-pairs-of-bytes-in-the-range-[0xD800,~0xDBFF],~
6245     and-trail-surrogates-are-in-the-range-[0xDC00,~0xDFFF].
6246     \flag_if_raised:nT { str_missing }
6247     {
6248         \\
6249         A~lead-surrogate-was-not-followed-by-a-trail-surrogate.
6250     }
6251     \flag_if_raised:nT { str_extra }
6252     {
6253         \\
6254         LaTeX-came-across-a-trail-surrogate-when-it-was-not-expected.
6255     }
6256     \flag_if_raised:nT { str_end }
6257     {
6258         \\
6259         The-string-contained-an-odd-number-of-bytes.-This-is-invalid:~
6260         the-basic-code-unit-for-UTF-16-is-16-bits-(2-bytes).
6261     }
6262 }

```

(End definition for `\l__str_missing_flag`, `\l__str_extra_flag`, and `\l__str_end_flag`.)

`__str_convert_decode_utf16:` As for UTF-8, decoding UTF-16 is harder than encoding it. If the endianness is unknown, check the first two bytes: if those are "FE and "FF in either order, remove them and use the corresponding endianness, otherwise assume big-endianness. The three endianness cases are based on a common auxiliary whose first argument is 1 for big-endian and 2 for little-endian, and whose second argument, delimited by the scan mark `\s__str_stop`, is expanded once (the string may be long; passing `\g__str_result_tl` as an argument before expansion is cheaper).

The `__str_decode_utf_xvi:Nw` function defines `__str_tmp:w` to take two arguments and return the character code of the first one if the string is big-endian, and the second one if the string is little-endian, then loops over the string using `__str_decode_utf_xvi_pair:NN` described below.

```

6263     \cs_new_protected:cpn { __str_convert_decode_utf16be: }
6264     { \__str_decode_utf_xvi:Nw 1 \g__str_result_tl \s__str_stop }
6265     \cs_new_protected:cpn { __str_convert_decode_utf16le: }
6266     { \__str_decode_utf_xvi:Nw 2 \g__str_result_tl \s__str_stop }
6267     \cs_new_protected:cpn { __str_convert_decode_utf16: }
6268     {
6269         \exp_after:wN \__str_decode_utf_xvi_bom:NN
6270         \g__str_result_tl \s__str_stop \s__str_stop \s__str_stop
6271     }
6272     \cs_new_protected:Npn \__str_decode_utf_xvi_bom:NN #1#2
6273     {
6274         \str_if_eq:nnTF { #1#2 } { ^^ff ^^fe }
6275         { \__str_decode_utf_xvi:Nw 2 }
6276         {
6277             \str_if_eq:nnTF { #1#2 } { ^^fe ^^ff }
6278             { \__str_decode_utf_xvi:Nw 1 }
6279             { \__str_decode_utf_xvi:Nw 1 #1#2 }
6280         }

```

```

6281     }
6282     \cs_new_protected:Npn \__str_decode_utf_xvi:Nw #1#2 \s__str_stop
6283     {
6284         \flag_clear:n { str_error }
6285         \flag_clear:n { str_missing }
6286         \flag_clear:n { str_extra }
6287         \flag_clear:n { str_end }
6288         \cs_set:Npn \__str_tmp:w ##1 ##2 { ' ## #1 }
6289         \tl_gset:Nx \g__str_result_tl
6290         {
6291             \exp_after:wN \__str_decode_utf_xvi_pair:NN
6292             #2 \q__str_nil \q__str_nil
6293             \prg_break_point:
6294         }
6295         \__str_if_flag_error:nmx { str_error } { utf16-decode } { }
6296     }

```

(End definition for `__str_convert_decode_utf16:` and others.)

```

\__str_decode_utf_xvi_pair:NN
\__str_decode_utf_xvi_quad:NNwNN
\__str_decode_utf_xvi_pair_end:Nw
\__str_decode_utf_xvi_error:nNN
\__str_decode_utf_xvi_extra:NNw

```

Bytes are read two at a time. At this stage, `\@@_tmp:w #1#2` expands to the character code of the most significant byte, and we distinguish cases depending on which range it lies in:

- ["D8, "DB] signals a lead surrogate, and the integer expression yields 1 (ε -TeX rounds ties away from zero);
- ["DC, "DF] signals a trail surrogate, unexpected here, and the integer expression yields 2;
- any other value signals a code point in the Basic Multilingual Plane, which stands for itself, and the `\if_case:w` construction expands to nothing (cases other than 1 or 2), leaving the relevant material in the input stream, followed by another call to the `_pair` auxiliary.

The case of a lead surrogate is treated by the `_quad` auxiliary, whose arguments `#1`, `#2`, `#4` and `#5` are the four bytes. We expect the most significant byte of `#4#5` to be in the range ["DC, "DF] (trail surrogate). The test is similar to the test used for continuation bytes in the UTF-8 decoding functions. In the case where `#4#5` is indeed a trail surrogate, leave `#1#2#4#5 \s__str <code point> \s__str`, and remove the pair `#4#5` before looping with `__str_decode_utf_xvi_pair:NN`. Otherwise, of course, complain about the missing surrogate.

The magic number "D7F7 is such that $"D7F7*"400 = "D800*"400 + "DC00 - "10000$.

Every time we read a pair of bytes, we test for the end-marker `\q__str_nil`. When reaching the end, we additionally check that the string had an even length. Also, if the end is reached when expecting a trail surrogate, we treat that as a missing surrogate.

```

6297     \cs_new:Npn \__str_decode_utf_xvi_pair:NN #1#2
6298     {
6299         \if_meaning:w \q__str_nil #2
6300         \__str_decode_utf_xvi_pair_end:Nw #1
6301         \fi:
6302         \if_case:w
6303             \int_eval:n { ( \__str_tmp:w #1#2 - "D6 ) / 4 } \scan_stop:
6304         \or: \exp_after:wN \__str_decode_utf_xvi_quad:NNwNN
6305         \or: \exp_after:wN \__str_decode_utf_xvi_extra:NNw

```

```

6306     \fi:
6307     #1#2 \s__str
6308     \int_eval:n { "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 } \s__str
6309     \__str_decode_utf_xvi_pair:NN
6310 }
6311 \cs_new:Npn \__str_decode_utf_xvi_quad:NNwNN
6312   #1#2 #3 \__str_decode_utf_xvi_pair:NN #4#5
6313 {
6314   \if_meaning:w \q__str_nil #5
6315     \__str_decode_utf_xvi_error:nNN { missing } #1#2
6316     \__str_decode_utf_xvi_pair_end:Nw #4
6317   \fi:
6318   \if_int_compare:w
6319     \if_int_compare:w \__str_tmp:w #4#5 < "DC \exp_stop_f:
6320       0 = 1
6321     \else:
6322       \__str_tmp:w #4#5 < "E0
6323     \fi:
6324     \exp_stop_f:
6325     #1 #2 #4 #5 \s__str
6326     \int_eval:n
6327     {
6328       ( "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 - "D7F7 ) * "400
6329       + "100 * \__str_tmp:w #4#5 + \__str_tmp:w #5#4
6330     }
6331     \s__str
6332     \exp_after:wN \use_i:nnn
6333   \else:
6334     \__str_decode_utf_xvi_error:nNN { missing } #1#2
6335   \fi:
6336   \__str_decode_utf_xvi_pair:NN #4#5
6337 }
6338 \cs_new:Npn \__str_decode_utf_xvi_pair_end:Nw #1 \fi:
6339 {
6340   \fi:
6341   \if_meaning:w \q__str_nil #1
6342   \else:
6343     \__str_decode_utf_xvi_error:nNN { end } #1 \prg_do_nothing:
6344   \fi:
6345   \prg_break:
6346 }
6347 \cs_new:Npn \__str_decode_utf_xvi_extra:NNw #1#2 \s__str #3 \s__str
6348 { \__str_decode_utf_xvi_error:nNN { extra } #1#2 }
6349 \cs_new:Npn \__str_decode_utf_xvi_error:nNN #1#2#3
6350 {
6351   \flag_raise:n { str_error }
6352   \flag_raise:n { str_#1 }
6353   #2 #3 \s__str
6354   \int_use:N \c__str_replacement_char_int \s__str
6355 }

```

(End definition for __str_decode_utf_xvi_pair:NN and others.)

Restore the original catcodes of bytes 254 and 255.

```

6356 \group_end:

```

10.6.3 utf-32 support

The definitions are done in a category code regime where the bytes 0, 254 and 255 used by the byte order mark have catcode “other”.

```
6357 \group_begin:
6358   \char_set_catcode_other:N \^^00
6359   \char_set_catcode_other:N \^^fe
6360   \char_set_catcode_other:N \^^ff
```

`__str_convert_encode_utf32:` Convert each integer in the comma-list `\g__str_result_tl` to a sequence of four bytes. The functions for big-endian and little-endian encodings are very similar, but the `__str_output_byte:n` instructions are reversed.

```
\__str_convert_encode_utf32be:
  \__str_convert_encode_utf32le:
\__str_encode_utf_xxxii_be:n
  \__str_encode_utf_xxxii_be_aux:nn
\__str_encode_utf_xxxii_le:n
  \__str_encode_utf_xxxii_le_aux:nn
6361   \cs_new_protected:cpn { __str_convert_encode_utf32: }
6362   {
6363     \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_be:n
6364     \tl_gput_left:Nx \g__str_result_tl { ^^00 ^^00 ^^fe ^^ff }
6365   }
6366   \cs_new_protected:cpn { __str_convert_encode_utf32be: }
6367   { \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_be:n }
6368   \cs_new_protected:cpn { __str_convert_encode_utf32le: }
6369   { \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_le:n }
6370   \cs_new:Npn \__str_encode_utf_xxxii_be:n #1
6371   {
6372     \exp_args:Nf \__str_encode_utf_xxxii_be_aux:nn
6373     { \int_div_truncate:nn {#1} { "100 } } {#1}
6374   }
6375   \cs_new:Npn \__str_encode_utf_xxxii_be_aux:nn #1#2
6376   {
6377     ^^00
6378     \__str_output_byte_pair_be:n {#1}
6379     \__str_output_byte:n { #2 - #1 * "100 }
6380   }
6381   \cs_new:Npn \__str_encode_utf_xxxii_le:n #1
6382   {
6383     \exp_args:Nf \__str_encode_utf_xxxii_le_aux:nn
6384     { \int_div_truncate:nn {#1} { "100 } } {#1}
6385   }
6386   \cs_new:Npn \__str_encode_utf_xxxii_le_aux:nn #1#2
6387   {
6388     \__str_output_byte:n { #2 - #1 * "100 }
6389     \__str_output_byte_pair_le:n {#1}
6390     ^^00
6391   }
```

(End definition for `__str_convert_encode_utf32:` and others.)

str_overflow There can be no error when encoding in UTF-32. When decoding, the string may not have length $4n$, or it may contain code points larger than "10FFFF". The latter case often happens if the encoding was in fact not UTF-32, because most arbitrary strings are not valid in UTF-32.

```
6392   \flag_clear_new:n { str_overflow }
6393   \flag_clear_new:n { str_end }
6394   \__kernel_msg_new:nnnn { str } { utf32-decode }
6395   {
```

```

6396 Invalid-UTF-32-string:
6397 \exp_last_unbraced:Nf \use_none:n
6398 {
6399   \__str_if_flag_times:nT { str_overflow } { ,~code-point-too-large }
6400   \__str_if_flag_times:nT { str_end }      { ,~truncated-string }
6401 }
6402 .
6403 }
6404 {
6405   In~the~UTF-32~encoding,~every~Unicode~character~
6406   (in~the~range~[U+0000,~U+10FFFF])~is~encoded~as~4~bytes.
6407   \flag_if_raised:nT { str_overflow }
6408   {
6409     \\\
6410     LaTeX~came~across~a~code~point~larger~than~1114111,~
6411     the~maximum~code~point~defined~by~Unicode.~
6412     Perhaps~the~string~was~not~encoded~in~the~UTF-32~encoding?
6413   }
6414   \flag_if_raised:nT { str_end }
6415   {
6416     \\\
6417     The~length~of~the~string~is~not~a~multiple~of~4.~
6418     Perhaps~the~string~was~truncated?
6419   }
6420 }

```

(End definition for `str_overflow` and `str_end`. These variables are documented on page ??.)

`__str_convert_decode_utf32:` The structure is similar to UTF-16 decoding functions. If the endianness is not given, test the first 4 bytes of the string (possibly `\s__str_stop` if the string is too short) for the presence of a byte-order mark. If there is a byte-order mark, use that endianness, and remove the 4 bytes, otherwise default to big-endian, and leave the 4 bytes in place. The `__str_decode_utf_xxxii:Nw` auxiliary receives 1 or 2 as its first argument indicating endianness, and the string to convert as its second argument (expanded or not). It sets `__str_tmp:w` to expand to the character code of either of its two arguments depending on endianness, then triggers the `_loop` auxiliary inside an x-expanding assignment to `\g__str_result_tl`.

The `_loop` auxiliary first checks for the end-of-string marker `\s__str_stop`, calling the `_end` auxiliary if appropriate. Otherwise, leave the $\langle 4 \text{ bytes} \rangle$ `\s__str` behind, then check that the code point is not overflowing: the leading byte must be 0, and the following byte at most 16.

In the ending code, we check that there remains no byte: there should be nothing left until the first `\s__str_stop`. Break the map.

```

6421 \cs_new_protected:cpn { __str_convert_decode_utf32be: }
6422 { \__str_decode_utf_xxxii:Nw 1 \g__str_result_tl \s__str_stop }
6423 \cs_new_protected:cpn { __str_convert_decode_utf32le: }
6424 { \__str_decode_utf_xxxii:Nw 2 \g__str_result_tl \s__str_stop }
6425 \cs_new_protected:cpn { __str_convert_decode_utf32: }
6426 {
6427   \exp_after:wN \__str_decode_utf_xxxii_bom:NNNN \g__str_result_tl
6428   \s__str_stop \s__str_stop \s__str_stop \s__str_stop \s__str_stop
6429 }
6430 \cs_new_protected:Npn \__str_decode_utf_xxxii_bom:NNNN #1#2#3#4

```

```

6431 {
6432   \str_if_eq:nnTF { #1#2#3#4 } { ^^ff ^^fe ^^00 ^^00 }
6433   { \__str_decode_utf_xxxii:Nw 2 }
6434   {
6435     \str_if_eq:nnTF { #1#2#3#4 } { ^^00 ^^00 ^^fe ^^ff }
6436     { \__str_decode_utf_xxxii:Nw 1 }
6437     { \__str_decode_utf_xxxii:Nw 1 #1#2#3#4 }
6438   }
6439 }
6440 \cs_new_protected:Npn \__str_decode_utf_xxxii:Nw #1#2 \s__str_stop
6441 {
6442   \flag_clear:n { str_overflow }
6443   \flag_clear:n { str_end }
6444   \flag_clear:n { str_error }
6445   \cs_set:Npn \__str_tmp:w ##1 ##2 { ' ## #1 }
6446   \tl_gset:Nx \g__str_result_tl
6447   {
6448     \exp_after:wN \__str_decode_utf_xxxii_loop:NNNN
6449     #2 \s__str_stop \s__str_stop \s__str_stop \s__str_stop
6450     \prg_break_point:
6451   }
6452   \__str_if_flag_error:nnx { str_error } { utf32-decode } { }
6453 }
6454 \cs_new:Npn \__str_decode_utf_xxxii_loop:NNNN #1#2#3#4
6455 {
6456   \if_meaning:w \s__str_stop #4
6457   \exp_after:wN \__str_decode_utf_xxxii_end:w
6458   \fi:
6459   #1#2#3#4 \s__str
6460   \if_int_compare:w \__str_tmp:w #1#4 > 0 \exp_stop_f:
6461     \flag_raise:n { str_overflow }
6462     \flag_raise:n { str_error }
6463     \int_use:N \c__str_replacement_char_int
6464   \else:
6465     \if_int_compare:w \__str_tmp:w #2#3 > 16 \exp_stop_f:
6466       \flag_raise:n { str_overflow }
6467       \flag_raise:n { str_error }
6468       \int_use:N \c__str_replacement_char_int
6469     \else:
6470       \int_eval:n
6471       { \__str_tmp:w #2#3*"10000 + \__str_tmp:w #3#2*"100 + \__str_tmp:w #4#1 }
6472     \fi:
6473     \fi:
6474     \s__str
6475     \__str_decode_utf_xxxii_loop:NNNN
6476   }
6477 \cs_new:Npn \__str_decode_utf_xxxii_end:w #1 \s__str_stop
6478 {
6479   \tl_if_empty:nF {#1}
6480   {
6481     \flag_raise:n { str_end }
6482     \flag_raise:n { str_error }
6483     #1 \s__str
6484     \int_use:N \c__str_replacement_char_int \s__str

```

```

6485     }
6486     \prg_break:
6487 }

```

(End definition for `_str_convert_decode_utf32:` and others.)

Restore the original catcodes of bytes 0, 254 and 255.

```

6488 \group_end:

```

10.7 PDF names and strings by expansion

`\str_convert_pdfname:n` To convert to PDF names by expansion, we work purely on UTF-8 input. The first step is to make a string with “other” spaces, after which we use a simple token-by-token approach. In Unicode engines, we break down everything before one-byte codepoints, but for 8-bit engines there is no need to worry. Actual escaping is covered by the same code as used in the non-expandable route.

```

6489 \cs_new:Npn \str_convert_pdfname:n #1
6490 {
6491     \exp_args:Ne \tl_to_str:n
6492     { \str_map_function:nN {#1} \_str_convert_pdfname:n }
6493 }
6494 \bool_lazy_or:nnTF
6495 { \sys_if_engine_luatex_p: }
6496 { \sys_if_engine_xetex_p: }
6497 {
6498     \cs_new:Npn \_str_convert_pdfname:n #1
6499     {
6500         \int_compare:nNnTF { '#1 } > { "7F }
6501         { \_str_convert_pdfname_bytes:n {#1} }
6502         { \_str_escape_name_char:n {#1} }
6503     }
6504     \cs_new:Npn \_str_convert_pdfname_bytes:n #1
6505     {
6506         \exp_args:Ne \_str_convert_pdfname_bytes_aux:n
6507         { \char_to_utfviii_bytes:n {'#1} }
6508     }
6509     \cs_new:Npn \_str_convert_pdfname_bytes_aux:n #1
6510     { \_str_convert_pdfname_bytes_aux:nnnn #1 }
6511     \cs_new:Npx \_str_convert_pdfname_bytes_aux:nnnn #1#2#3#4
6512     {
6513         \c_hash_str \exp_not:N \_str_output_hexadecimal:n {#1}
6514         \c_hash_str \exp_not:N \_str_output_hexadecimal:n {#2}
6515         \exp_not:N \tl_if_blank:nF {#3}
6516         {
6517             \c_hash_str \exp_not:N \_str_output_hexadecimal:n {#3}
6518             \exp_not:N \tl_if_blank:nF {#4}
6519             {
6520                 \c_hash_str \exp_not:N \_str_output_hexadecimal:n {#4}
6521             }
6522         }
6523     }
6524 }
6525 { \cs_new_eq:NN \_str_convert_pdfname:n \_str_escape_name_char:n }

```

(End definition for `\str_convert_pdfname:n` and others. This function is documented on page 74.)

```
6526 \endpackage
```

10.7.1 iso 8859 support

The ISO-8859-1 encoding exactly matches with the 256 first Unicode characters. For other 8-bit encodings of the ISO-8859 family, we keep track only of differences, and of unassigned bytes.

```
6527 \newcommand{\iso88591}{}
6528 \str_declare_eight_bit_encoding:nnn { iso88591 }
6529 {
6530 }
6531 {
6532 }
6533 \endiso88591

6534 \newcommand{\iso88592}{}
6535 \str_declare_eight_bit_encoding:nnn { iso88592 }
6536 {
6537   { A1 } { 0104 }
6538   { A2 } { 02D8 }
6539   { A3 } { 0141 }
6540   { A5 } { 013D }
6541   { A6 } { 015A }
6542   { A9 } { 0160 }
6543   { AA } { 015E }
6544   { AB } { 0164 }
6545   { AC } { 0179 }
6546   { AE } { 017D }
6547   { AF } { 017B }
6548   { B1 } { 0105 }
6549   { B2 } { 02DB }
6550   { B3 } { 0142 }
6551   { B5 } { 013E }
6552   { B6 } { 015B }
6553   { B7 } { 02C7 }
6554   { B9 } { 0161 }
6555   { BA } { 015F }
6556   { BB } { 0165 }
6557   { BC } { 017A }
6558   { BD } { 02DD }
6559   { BE } { 017E }
6560   { BF } { 017C }
6561   { C0 } { 0154 }
6562   { C3 } { 0102 }
6563   { C5 } { 0139 }
6564   { C6 } { 0106 }
6565   { C8 } { 010C }
6566   { CA } { 0118 }
6567   { CC } { 011A }
6568   { CF } { 010E }
6569   { D0 } { 0110 }
6570   { D1 } { 0143 }
6571   { D2 } { 0147 }
```



```

6572     { D5 } { 0150 }
6573     { D8 } { 0158 }
6574     { D9 } { 016E }
6575     { DB } { 0170 }
6576     { DE } { 0162 }
6577     { E0 } { 0155 }
6578     { E3 } { 0103 }
6579     { E5 } { 013A }
6580     { E6 } { 0107 }
6581     { E8 } { 010D }
6582     { EA } { 0119 }
6583     { EC } { 011B }
6584     { EF } { 010F }
6585     { F0 } { 0111 }
6586     { F1 } { 0144 }
6587     { F2 } { 0148 }
6588     { F5 } { 0151 }
6589     { F8 } { 0159 }
6590     { F9 } { 016F }
6591     { FB } { 0171 }
6592     { FE } { 0163 }
6593     { FF } { 02D9 }
6594 }
6595 {
6596 }
6597 </iso88592>
6598 <iso88593>
6599 \str_declare_eight_bit_encoding:nnn { iso88593 }
6600 {
6601     { A1 } { 0126 }
6602     { A2 } { 02D8 }
6603     { A6 } { 0124 }
6604     { A9 } { 0130 }
6605     { AA } { 015E }
6606     { AB } { 011E }
6607     { AC } { 0134 }
6608     { AF } { 017B }
6609     { B1 } { 0127 }
6610     { B6 } { 0125 }
6611     { B9 } { 0131 }
6612     { BA } { 015F }
6613     { BB } { 011F }
6614     { BC } { 0135 }
6615     { BF } { 017C }
6616     { C5 } { 010A }
6617     { C6 } { 0108 }
6618     { D5 } { 0120 }
6619     { D8 } { 011C }
6620     { DD } { 016C }
6621     { DE } { 015C }
6622     { E5 } { 010B }
6623     { E6 } { 0109 }
6624     { F5 } { 0121 }
6625     { F8 } { 011D }

```

```

6626     { FD } { 016D }
6627     { FE } { 015D }
6628     { FF } { 02D9 }
6629 }
6630 {
6631     { A5 }
6632     { AE }
6633     { BE }
6634     { C3 }
6635     { D0 }
6636     { E3 }
6637     { F0 }
6638 }
6639 </iso88593>
6640 <iso88594>
6641 \str_declare_eight_bit_encoding:nnn { iso88594 }
6642 {
6643     { A1 } { 0104 }
6644     { A2 } { 0138 }
6645     { A3 } { 0156 }
6646     { A5 } { 0128 }
6647     { A6 } { 013B }
6648     { A9 } { 0160 }
6649     { AA } { 0112 }
6650     { AB } { 0122 }
6651     { AC } { 0166 }
6652     { AE } { 017D }
6653     { B1 } { 0105 }
6654     { B2 } { 02DB }
6655     { B3 } { 0157 }
6656     { B5 } { 0129 }
6657     { B6 } { 013C }
6658     { B7 } { 02C7 }
6659     { B9 } { 0161 }
6660     { BA } { 0113 }
6661     { BB } { 0123 }
6662     { BC } { 0167 }
6663     { BD } { 014A }
6664     { BE } { 017E }
6665     { BF } { 014B }
6666     { C0 } { 0100 }
6667     { C7 } { 012E }
6668     { C8 } { 010C }
6669     { CA } { 0118 }
6670     { CC } { 0116 }
6671     { CF } { 012A }
6672     { D0 } { 0110 }
6673     { D1 } { 0145 }
6674     { D2 } { 014C }
6675     { D3 } { 0136 }
6676     { D9 } { 0172 }
6677     { DD } { 0168 }
6678     { DE } { 016A }
6679     { EO } { 0101 }

```

```

6680     { E7 } { 012F }
6681     { E8 } { 010D }
6682     { EA } { 0119 }
6683     { EC } { 0117 }
6684     { EF } { 012B }
6685     { F0 } { 0111 }
6686     { F1 } { 0146 }
6687     { F2 } { 014D }
6688     { F3 } { 0137 }
6689     { F9 } { 0173 }
6690     { FD } { 0169 }
6691     { FE } { 016B }
6692     { FF } { 02D9 }
6693 }
6694 {
6695 }
6696 </iso88594>
6697 <*:iso88595>
6698 \str_declare_eight_bit_encoding:nnn { iso88595 }
6699 {
6700     { A1 } { 0401 }
6701     { A2 } { 0402 }
6702     { A3 } { 0403 }
6703     { A4 } { 0404 }
6704     { A5 } { 0405 }
6705     { A6 } { 0406 }
6706     { A7 } { 0407 }
6707     { A8 } { 0408 }
6708     { A9 } { 0409 }
6709     { AA } { 040A }
6710     { AB } { 040B }
6711     { AC } { 040C }
6712     { AE } { 040E }
6713     { AF } { 040F }
6714     { B0 } { 0410 }
6715     { B1 } { 0411 }
6716     { B2 } { 0412 }
6717     { B3 } { 0413 }
6718     { B4 } { 0414 }
6719     { B5 } { 0415 }
6720     { B6 } { 0416 }
6721     { B7 } { 0417 }
6722     { B8 } { 0418 }
6723     { B9 } { 0419 }
6724     { BA } { 041A }
6725     { BB } { 041B }
6726     { BC } { 041C }
6727     { BD } { 041D }
6728     { BE } { 041E }
6729     { BF } { 041F }
6730     { C0 } { 0420 }
6731     { C1 } { 0421 }
6732     { C2 } { 0422 }
6733     { C3 } { 0423 }

```

6734	{ C4 }	{ 0424 }
6735	{ C5 }	{ 0425 }
6736	{ C6 }	{ 0426 }
6737	{ C7 }	{ 0427 }
6738	{ C8 }	{ 0428 }
6739	{ C9 }	{ 0429 }
6740	{ CA }	{ 042A }
6741	{ CB }	{ 042B }
6742	{ CC }	{ 042C }
6743	{ CD }	{ 042D }
6744	{ CE }	{ 042E }
6745	{ CF }	{ 042F }
6746	{ D0 }	{ 0430 }
6747	{ D1 }	{ 0431 }
6748	{ D2 }	{ 0432 }
6749	{ D3 }	{ 0433 }
6750	{ D4 }	{ 0434 }
6751	{ D5 }	{ 0435 }
6752	{ D6 }	{ 0436 }
6753	{ D7 }	{ 0437 }
6754	{ D8 }	{ 0438 }
6755	{ D9 }	{ 0439 }
6756	{ DA }	{ 043A }
6757	{ DB }	{ 043B }
6758	{ DC }	{ 043C }
6759	{ DD }	{ 043D }
6760	{ DE }	{ 043E }
6761	{ DF }	{ 043F }
6762	{ E0 }	{ 0440 }
6763	{ E1 }	{ 0441 }
6764	{ E2 }	{ 0442 }
6765	{ E3 }	{ 0443 }
6766	{ E4 }	{ 0444 }
6767	{ E5 }	{ 0445 }
6768	{ E6 }	{ 0446 }
6769	{ E7 }	{ 0447 }
6770	{ E8 }	{ 0448 }
6771	{ E9 }	{ 0449 }
6772	{ EA }	{ 044A }
6773	{ EB }	{ 044B }
6774	{ EC }	{ 044C }
6775	{ ED }	{ 044D }
6776	{ EE }	{ 044E }
6777	{ EF }	{ 044F }
6778	{ F0 }	{ 2116 }
6779	{ F1 }	{ 0451 }
6780	{ F2 }	{ 0452 }
6781	{ F3 }	{ 0453 }
6782	{ F4 }	{ 0454 }
6783	{ F5 }	{ 0455 }
6784	{ F6 }	{ 0456 }
6785	{ F7 }	{ 0457 }
6786	{ F8 }	{ 0458 }
6787	{ F9 }	{ 0459 }

```

6788     { FA } { 045A }
6789     { FB } { 045B }
6790     { FC } { 045C }
6791     { FD } { 00A7 }
6792     { FE } { 045E }
6793     { FF } { 045F }
6794 }
6795 {
6796 }
6797 </iso88595>
6798 < *iso88596>
6799 \str_declare_eight_bit_encoding:nmn { iso88596 }
6800 {
6801     { AC } { 060C }
6802     { BB } { 061B }
6803     { BF } { 061F }
6804     { C1 } { 0621 }
6805     { C2 } { 0622 }
6806     { C3 } { 0623 }
6807     { C4 } { 0624 }
6808     { C5 } { 0625 }
6809     { C6 } { 0626 }
6810     { C7 } { 0627 }
6811     { C8 } { 0628 }
6812     { C9 } { 0629 }
6813     { CA } { 062A }
6814     { CB } { 062B }
6815     { CC } { 062C }
6816     { CD } { 062D }
6817     { CE } { 062E }
6818     { CF } { 062F }
6819     { D0 } { 0630 }
6820     { D1 } { 0631 }
6821     { D2 } { 0632 }
6822     { D3 } { 0633 }
6823     { D4 } { 0634 }
6824     { D5 } { 0635 }
6825     { D6 } { 0636 }
6826     { D7 } { 0637 }
6827     { D8 } { 0638 }
6828     { D9 } { 0639 }
6829     { DA } { 063A }
6830     { E0 } { 0640 }
6831     { E1 } { 0641 }
6832     { E2 } { 0642 }
6833     { E3 } { 0643 }
6834     { E4 } { 0644 }
6835     { E5 } { 0645 }
6836     { E6 } { 0646 }
6837     { E7 } { 0647 }
6838     { E8 } { 0648 }
6839     { E9 } { 0649 }
6840     { EA } { 064A }
6841     { EB } { 064B }

```

```

6842     { EC } { 064C }
6843     { ED } { 064D }
6844     { EE } { 064E }
6845     { EF } { 064F }
6846     { F0 } { 0650 }
6847     { F1 } { 0651 }
6848     { F2 } { 0652 }
6849 }
6850 {
6851     { A1 }
6852     { A2 }
6853     { A3 }
6854     { A5 }
6855     { A6 }
6856     { A7 }
6857     { A8 }
6858     { A9 }
6859     { AA }
6860     { AB }
6861     { AE }
6862     { AF }
6863     { B0 }
6864     { B1 }
6865     { B2 }
6866     { B3 }
6867     { B4 }
6868     { B5 }
6869     { B6 }
6870     { B7 }
6871     { B8 }
6872     { B9 }
6873     { BA }
6874     { BC }
6875     { BD }
6876     { BE }
6877     { C0 }
6878     { DB }
6879     { DC }
6880     { DD }
6881     { DE }
6882     { DF }
6883 }
6884 </iso88596>
6885 <iso88597>
6886 \str_declare_eight_bit_encoding:nnn { iso88597 }
6887 {
6888     { A1 } { 2018 }
6889     { A2 } { 2019 }
6890     { A4 } { 20AC }
6891     { A5 } { 20AF }
6892     { AA } { 037A }
6893     { AF } { 2015 }
6894     { B4 } { 0384 }
6895     { B5 } { 0385 }

```

6896	{ B6 }	{ 0386 }
6897	{ B8 }	{ 0388 }
6898	{ B9 }	{ 0389 }
6899	{ BA }	{ 038A }
6900	{ BC }	{ 038C }
6901	{ BE }	{ 038E }
6902	{ BF }	{ 038F }
6903	{ C0 }	{ 0390 }
6904	{ C1 }	{ 0391 }
6905	{ C2 }	{ 0392 }
6906	{ C3 }	{ 0393 }
6907	{ C4 }	{ 0394 }
6908	{ C5 }	{ 0395 }
6909	{ C6 }	{ 0396 }
6910	{ C7 }	{ 0397 }
6911	{ C8 }	{ 0398 }
6912	{ C9 }	{ 0399 }
6913	{ CA }	{ 039A }
6914	{ CB }	{ 039B }
6915	{ CC }	{ 039C }
6916	{ CD }	{ 039D }
6917	{ CE }	{ 039E }
6918	{ CF }	{ 039F }
6919	{ D0 }	{ 03A0 }
6920	{ D1 }	{ 03A1 }
6921	{ D3 }	{ 03A3 }
6922	{ D4 }	{ 03A4 }
6923	{ D5 }	{ 03A5 }
6924	{ D6 }	{ 03A6 }
6925	{ D7 }	{ 03A7 }
6926	{ D8 }	{ 03A8 }
6927	{ D9 }	{ 03A9 }
6928	{ DA }	{ 03AA }
6929	{ DB }	{ 03AB }
6930	{ DC }	{ 03AC }
6931	{ DD }	{ 03AD }
6932	{ DE }	{ 03AE }
6933	{ DF }	{ 03AF }
6934	{ E0 }	{ 03B0 }
6935	{ E1 }	{ 03B1 }
6936	{ E2 }	{ 03B2 }
6937	{ E3 }	{ 03B3 }
6938	{ E4 }	{ 03B4 }
6939	{ E5 }	{ 03B5 }
6940	{ E6 }	{ 03B6 }
6941	{ E7 }	{ 03B7 }
6942	{ E8 }	{ 03B8 }
6943	{ E9 }	{ 03B9 }
6944	{ EA }	{ 03BA }
6945	{ EB }	{ 03BB }
6946	{ EC }	{ 03BC }
6947	{ ED }	{ 03BD }
6948	{ EE }	{ 03BE }
6949	{ EF }	{ 03BF }

```

6950     { F0 } { 03C0 }
6951     { F1 } { 03C1 }
6952     { F2 } { 03C2 }
6953     { F3 } { 03C3 }
6954     { F4 } { 03C4 }
6955     { F5 } { 03C5 }
6956     { F6 } { 03C6 }
6957     { F7 } { 03C7 }
6958     { F8 } { 03C8 }
6959     { F9 } { 03C9 }
6960     { FA } { 03CA }
6961     { FB } { 03CB }
6962     { FC } { 03CC }
6963     { FD } { 03CD }
6964     { FE } { 03CE }
6965 }
6966 {
6967     { AE }
6968     { D2 }
6969 }
6970 </iso88597>
6971 < *iso88598>
6972 \str_declare_eight_bit_encoding:nnn { iso88598 }
6973 {
6974     { AA } { 00D7 }
6975     { BA } { 00F7 }
6976     { DF } { 2017 }
6977     { E0 } { 05D0 }
6978     { E1 } { 05D1 }
6979     { E2 } { 05D2 }
6980     { E3 } { 05D3 }
6981     { E4 } { 05D4 }
6982     { E5 } { 05D5 }
6983     { E6 } { 05D6 }
6984     { E7 } { 05D7 }
6985     { E8 } { 05D8 }
6986     { E9 } { 05D9 }
6987     { EA } { 05DA }
6988     { EB } { 05DB }
6989     { EC } { 05DC }
6990     { ED } { 05DD }
6991     { EE } { 05DE }
6992     { EF } { 05DF }
6993     { F0 } { 05E0 }
6994     { F1 } { 05E1 }
6995     { F2 } { 05E2 }
6996     { F3 } { 05E3 }
6997     { F4 } { 05E4 }
6998     { F5 } { 05E5 }
6999     { F6 } { 05E6 }
7000     { F7 } { 05E7 }
7001     { F8 } { 05E8 }
7002     { F9 } { 05E9 }
7003     { FA } { 05EA }

```



```

7004     { FD } { 200E }
7005     { FE } { 200F }
7006   }
7007   {
7008     { A1 }
7009     { BF }
7010     { C0 }
7011     { C1 }
7012     { C2 }
7013     { C3 }
7014     { C4 }
7015     { C5 }
7016     { C6 }
7017     { C7 }
7018     { C8 }
7019     { C9 }
7020     { CA }
7021     { CB }
7022     { CC }
7023     { CD }
7024     { CE }
7025     { CF }
7026     { D0 }
7027     { D1 }
7028     { D2 }
7029     { D3 }
7030     { D4 }
7031     { D5 }
7032     { D6 }
7033     { D7 }
7034     { D8 }
7035     { D9 }
7036     { DA }
7037     { DB }
7038     { DC }
7039     { DD }
7040     { DE }
7041     { FB }
7042     { FC }
7043   }
7044   </iso88598>
7045   <*iso88599>
7046   \str_declare_eight_bit_encoding:nnn { iso88599 }
7047   {
7048     { D0 } { 011E }
7049     { DD } { 0130 }
7050     { DE } { 015E }
7051     { FO } { 011F }
7052     { FD } { 0131 }
7053     { FE } { 015F }
7054   }
7055   {
7056   }
7057   </iso88599>

```

```

7058 <*iso885910>
7059 \str_declare_eight_bit_encoding:nmn { iso885910 }
7060 {
7061   { A1 } { 0104 }
7062   { A2 } { 0112 }
7063   { A3 } { 0122 }
7064   { A4 } { 012A }
7065   { A5 } { 0128 }
7066   { A6 } { 0136 }
7067   { A8 } { 013B }
7068   { A9 } { 0110 }
7069   { AA } { 0160 }
7070   { AB } { 0166 }
7071   { AC } { 017D }
7072   { AE } { 016A }
7073   { AF } { 014A }
7074   { B1 } { 0105 }
7075   { B2 } { 0113 }
7076   { B3 } { 0123 }
7077   { B4 } { 012B }
7078   { B5 } { 0129 }
7079   { B6 } { 0137 }
7080   { B8 } { 013C }
7081   { B9 } { 0111 }
7082   { BA } { 0161 }
7083   { BB } { 0167 }
7084   { BC } { 017E }
7085   { BD } { 2015 }
7086   { BE } { 016B }
7087   { BF } { 014B }
7088   { C0 } { 0100 }
7089   { C7 } { 012E }
7090   { C8 } { 010C }
7091   { CA } { 0118 }
7092   { CC } { 0116 }
7093   { D1 } { 0145 }
7094   { D2 } { 014C }
7095   { D7 } { 0168 }
7096   { D9 } { 0172 }
7097   { E0 } { 0101 }
7098   { E7 } { 012F }
7099   { E8 } { 010D }
7100   { EA } { 0119 }
7101   { EC } { 0117 }
7102   { F1 } { 0146 }
7103   { F2 } { 014D }
7104   { F7 } { 0169 }
7105   { F9 } { 0173 }
7106   { FF } { 0138 }
7107 }
7108 {
7109 }
7110 </iso885910>
7111 <*iso885911>

```

```

7112 \str_declare_eight_bit_encoding:nmn { iso885911 }
7113 {
7114     { A1 } { OE01 }
7115     { A2 } { OE02 }
7116     { A3 } { OE03 }
7117     { A4 } { OE04 }
7118     { A5 } { OE05 }
7119     { A6 } { OE06 }
7120     { A7 } { OE07 }
7121     { A8 } { OE08 }
7122     { A9 } { OE09 }
7123     { AA } { OE0A }
7124     { AB } { OE0B }
7125     { AC } { OE0C }
7126     { AD } { OE0D }
7127     { AE } { OE0E }
7128     { AF } { OE0F }
7129     { B0 } { OE10 }
7130     { B1 } { OE11 }
7131     { B2 } { OE12 }
7132     { B3 } { OE13 }
7133     { B4 } { OE14 }
7134     { B5 } { OE15 }
7135     { B6 } { OE16 }
7136     { B7 } { OE17 }
7137     { B8 } { OE18 }
7138     { B9 } { OE19 }
7139     { BA } { OE1A }
7140     { BB } { OE1B }
7141     { BC } { OE1C }
7142     { BD } { OE1D }
7143     { BE } { OE1E }
7144     { BF } { OE1F }
7145     { C0 } { OE20 }
7146     { C1 } { OE21 }
7147     { C2 } { OE22 }
7148     { C3 } { OE23 }
7149     { C4 } { OE24 }
7150     { C5 } { OE25 }
7151     { C6 } { OE26 }
7152     { C7 } { OE27 }
7153     { C8 } { OE28 }
7154     { C9 } { OE29 }
7155     { CA } { OE2A }
7156     { CB } { OE2B }
7157     { CC } { OE2C }
7158     { CD } { OE2D }
7159     { CE } { OE2E }
7160     { CF } { OE2F }
7161     { D0 } { OE30 }
7162     { D1 } { OE31 }
7163     { D2 } { OE32 }
7164     { D3 } { OE33 }
7165     { D4 } { OE34 }

```

```

7166     { D5 } { 0E35 }
7167     { D6 } { 0E36 }
7168     { D7 } { 0E37 }
7169     { D8 } { 0E38 }
7170     { D9 } { 0E39 }
7171     { DA } { 0E3A }
7172     { DF } { 0E3F }
7173     { E0 } { 0E40 }
7174     { E1 } { 0E41 }
7175     { E2 } { 0E42 }
7176     { E3 } { 0E43 }
7177     { E4 } { 0E44 }
7178     { E5 } { 0E45 }
7179     { E6 } { 0E46 }
7180     { E7 } { 0E47 }
7181     { E8 } { 0E48 }
7182     { E9 } { 0E49 }
7183     { EA } { 0E4A }
7184     { EB } { 0E4B }
7185     { EC } { 0E4C }
7186     { ED } { 0E4D }
7187     { EE } { 0E4E }
7188     { EF } { 0E4F }
7189     { F0 } { 0E50 }
7190     { F1 } { 0E51 }
7191     { F2 } { 0E52 }
7192     { F3 } { 0E53 }
7193     { F4 } { 0E54 }
7194     { F5 } { 0E55 }
7195     { F6 } { 0E56 }
7196     { F7 } { 0E57 }
7197     { F8 } { 0E58 }
7198     { F9 } { 0E59 }
7199     { FA } { 0E5A }
7200     { FB } { 0E5B }
7201   }
7202   {
7203     { DB }
7204     { DC }
7205     { DD }
7206     { DE }
7207   }
7208   </iso885911>
7209   <iso885913>
7210   \str_declare_eight_bit_encoding:nnn { iso885913 }
7211   {
7212     { A1 } { 201D }
7213     { A5 } { 201E }
7214     { A8 } { 00D8 }
7215     { AA } { 0156 }
7216     { AF } { 00C6 }
7217     { B4 } { 201C }
7218     { B8 } { 00F8 }
7219     { BA } { 0157 }

```

```

7220     { BF } { 00E6 }
7221     { C0 } { 0104 }
7222     { C1 } { 012E }
7223     { C2 } { 0100 }
7224     { C3 } { 0106 }
7225     { C6 } { 0118 }
7226     { C7 } { 0112 }
7227     { C8 } { 010C }
7228     { CA } { 0179 }
7229     { CB } { 0116 }
7230     { CC } { 0122 }
7231     { CD } { 0136 }
7232     { CE } { 012A }
7233     { CF } { 013B }
7234     { D0 } { 0160 }
7235     { D1 } { 0143 }
7236     { D2 } { 0145 }
7237     { D4 } { 014C }
7238     { D8 } { 0172 }
7239     { D9 } { 0141 }
7240     { DA } { 015A }
7241     { DB } { 016A }
7242     { DD } { 017B }
7243     { DE } { 017D }
7244     { E0 } { 0105 }
7245     { E1 } { 012F }
7246     { E2 } { 0101 }
7247     { E3 } { 0107 }
7248     { E6 } { 0119 }
7249     { E7 } { 0113 }
7250     { E8 } { 010D }
7251     { EA } { 017A }
7252     { EB } { 0117 }
7253     { EC } { 0123 }
7254     { ED } { 0137 }
7255     { EE } { 012B }
7256     { EF } { 013C }
7257     { F0 } { 0161 }
7258     { F1 } { 0144 }
7259     { F2 } { 0146 }
7260     { F4 } { 014D }
7261     { F8 } { 0173 }
7262     { F9 } { 0142 }
7263     { FA } { 015B }
7264     { FB } { 016B }
7265     { FD } { 017C }
7266     { FE } { 017E }
7267     { FF } { 2019 }
7268 }
7269 {
7270 }
7271 </iso885913>
7272 <*iso885914>
7273 \str_declare_eight_bit_encoding:nmn { iso885914 }

```

```

7274 {
7275     { A1 } { 1E02 }
7276     { A2 } { 1E03 }
7277     { A4 } { 010A }
7278     { A5 } { 010B }
7279     { A6 } { 1E0A }
7280     { A8 } { 1E80 }
7281     { AA } { 1E82 }
7282     { AB } { 1E0B }
7283     { AC } { 1EF2 }
7284     { AF } { 0178 }
7285     { B0 } { 1E1E }
7286     { B1 } { 1E1F }
7287     { B2 } { 0120 }
7288     { B3 } { 0121 }
7289     { B4 } { 1E40 }
7290     { B5 } { 1E41 }
7291     { B7 } { 1E56 }
7292     { B8 } { 1E81 }
7293     { B9 } { 1E57 }
7294     { BA } { 1E83 }
7295     { BB } { 1E60 }
7296     { BC } { 1EF3 }
7297     { BD } { 1E84 }
7298     { BE } { 1E85 }
7299     { BF } { 1E61 }
7300     { D0 } { 0174 }
7301     { D7 } { 1E6A }
7302     { DE } { 0176 }
7303     { F0 } { 0175 }
7304     { F7 } { 1E6B }
7305     { FE } { 0177 }
7306 }
7307 {
7308 }
7309 </iso885914>
7310 <*iso885915>
7311 \str_declare_eight_bit_encoding:nmn { iso885915 }
7312 {
7313     { A4 } { 20AC }
7314     { A6 } { 0160 }
7315     { A8 } { 0161 }
7316     { B4 } { 017D }
7317     { B8 } { 017E }
7318     { BC } { 0152 }
7319     { BD } { 0153 }
7320     { BE } { 0178 }
7321 }
7322 {
7323 }
7324 </iso885915>
7325 <*iso885916>
7326 \str_declare_eight_bit_encoding:nmn { iso885916 }

```

```

7327 {
7328     { A1 } { 0104 }
7329     { A2 } { 0105 }
7330     { A3 } { 0141 }
7331     { A4 } { 20AC }
7332     { A5 } { 201E }
7333     { A6 } { 0160 }
7334     { A8 } { 0161 }
7335     { AA } { 0218 }
7336     { AC } { 0179 }
7337     { AE } { 017A }
7338     { AF } { 017B }
7339     { B2 } { 010C }
7340     { B3 } { 0142 }
7341     { B4 } { 017D }
7342     { B5 } { 201D }
7343     { B8 } { 017E }
7344     { B9 } { 010D }
7345     { BA } { 0219 }
7346     { BC } { 0152 }
7347     { BD } { 0153 }
7348     { BE } { 0178 }
7349     { BF } { 017C }
7350     { C3 } { 0102 }
7351     { C5 } { 0106 }
7352     { D0 } { 0110 }
7353     { D1 } { 0143 }
7354     { D5 } { 0150 }
7355     { D7 } { 015A }
7356     { D8 } { 0170 }
7357     { DD } { 0118 }
7358     { DE } { 021A }
7359     { E3 } { 0103 }
7360     { E5 } { 0107 }
7361     { F0 } { 0111 }
7362     { F1 } { 0144 }
7363     { F5 } { 0151 }
7364     { F7 } { 015B }
7365     { F8 } { 0171 }
7366     { FD } { 0119 }
7367     { FE } { 021B }
7368 }
7369 {
7370 }
7371 </iso885916>

```

11 l3seq implementation

The following test files are used for this code: m3seq002,m3seq003.

```

7372 <*package>
7373 <@@=seq>

```

A sequence is a control sequence whose top-level expansion is of the form “\s__seq __seq_item:n {⟨item₁⟩} ... __seq_item:n {⟨item_n⟩}”, with a leading scan mark followed by *n* items of the same form. An earlier implementation used the structure “\seq_elt:w ⟨item₁⟩ \seq_elt_end: ... \seq_elt:w ⟨item_n⟩ \seq_elt_end:”. This allowed rapid searching using a delimited function, but was not suitable for items containing {, } and # tokens, and also lead to the loss of surrounding braces around items

__seq_item:n *

__seq_item:n {⟨item⟩}

The internal token used to begin each sequence entry. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.

__seq_push_item_def:n
__seq_push_item_def:x

__seq_push_item_def:n {⟨code⟩}

Saves the definition of __seq_item:n and redefines it to accept one parameter and expand to ⟨code⟩. This function should always be balanced by use of __seq_pop_item_def:.

__seq_pop_item_def:

__seq_pop_item_def:

Restores the definition of __seq_item:n most recently saved by __seq_push_item_def:n. This function should always be used in a balanced pair with __seq_push_item_def:n.

\s__seq

This private scan mark.

7374 \scan_new:N \s__seq

(End definition for \s__seq.)

\s__seq_mark

Private scan marks.

\s__seq_stop

7375 \scan_new:N \s__seq_mark

7376 \scan_new:N \s__seq_stop

(End definition for \s__seq_mark and \s__seq_stop.)

__seq_item:n

The delimiter is always defined, but when used incorrectly simply removes its argument and hits an undefined control sequence to raise an error.

7377 \cs_new:Npn __seq_item:n

7378 {

7379 __kernel_msg_expandable_error:nn { kernel } { misused-sequence }

7380 \use_none:n

7381 }

(End definition for __seq_item:n.)

\l__seq_internal_a_tl

Scratch space for various internal uses.

\l__seq_internal_b_tl

7382 \tl_new:N \l__seq_internal_a_tl

7383 \tl_new:N \l__seq_internal_b_tl

(End definition for \l__seq_internal_a_tl and \l__seq_internal_b_tl.)

__seq_tmp:w

Scratch function for internal use.

7384 \cs_new_eq:NN __seq_tmp:w ?

(End definition for __seq_tmp:w.)

\c_empty_seq A sequence with no item, following the structure mentioned above.

```
7385 \tl_const:Nn \c_empty_seq { \s__seq }
```

(End definition for \c_empty_seq. This variable is documented on page 87.)

11.1 Allocation and initialisation

\seq_new:N Sequences are initialized to \c_empty_seq.

```
\seq_new:c 7386 \cs_new_protected:Npn \seq_new:N #1
7387 {
7388     \__kernel_chk_if_free_cs:N #1
7389     \cs_gset_eq:NN #1 \c_empty_seq
7390 }
7391 \cs_generate_variant:Nn \seq_new:N { c }
```

(End definition for \seq_new:N. This function is documented on page 76.)

\seq_clear:N Clearing a sequence is similar to setting it equal to the empty one.

```
\seq_clear:c 7392 \cs_new_protected:Npn \seq_clear:N #1
\seq_gclear:N 7393 { \seq_set_eq:NN #1 \c_empty_seq }
\seq_gclear:c 7394 \cs_generate_variant:Nn \seq_clear:N { c }
7395 \cs_new_protected:Npn \seq_gclear:N #1
7396 { \seq_gset_eq:NN #1 \c_empty_seq }
7397 \cs_generate_variant:Nn \seq_gclear:N { c }
```

(End definition for \seq_clear:N and \seq_gclear:N. These functions are documented on page 76.)

\seq_clear_new:N Once again we copy code from the token list functions.

```
\seq_clear_new:c 7398 \cs_new_protected:Npn \seq_clear_new:N #1
\seq_gclear_new:N 7399 { \seq_if_exist:NTF #1 { \seq_clear:N #1 } { \seq_new:N #1 } }
\seq_gclear_new:c 7400 \cs_generate_variant:Nn \seq_clear_new:N { c }
7401 \cs_new_protected:Npn \seq_gclear_new:N #1
7402 { \seq_if_exist:NTF #1 { \seq_gclear:N #1 } { \seq_new:N #1 } }
7403 \cs_generate_variant:Nn \seq_gclear_new:N { c }
```

(End definition for \seq_clear_new:N and \seq_gclear_new:N. These functions are documented on page 76.)

\seq_set_eq:NN Copying a sequence is the same as copying the underlying token list.

```
\seq_set_eq:cN 7404 \cs_new_eq:NN \seq_set_eq:NN \tl_set_eq:NN
\seq_set_eq:Nc 7405 \cs_new_eq:NN \seq_set_eq:Nc \tl_set_eq:Nc
\seq_set_eq:cc 7406 \cs_new_eq:NN \seq_set_eq:cN \tl_set_eq:cN
\seq_gset_eq:NN 7407 \cs_new_eq:NN \seq_set_eq:cc \tl_set_eq:cc
\seq_gset_eq:cN 7408 \cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
\seq_gset_eq:Nc 7409 \cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
\seq_gset_eq:cN 7410 \cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN
\seq_gset_eq:cc 7411 \cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc
```

(End definition for \seq_set_eq:NN and \seq_gset_eq:NN. These functions are documented on page 76.)

`\seq_set_from_clist:NN` Setting a sequence from a comma-separated list is done using a simple mapping.

```

\seq_set_from_clist:cN 7412 \cs_new_protected:Npn \seq_set_from_clist:NN #1#2
\seq_set_from_clist:Nc 7413 {
\seq_set_from_clist:cc 7414   \tl_set:Nx #1
\seq_set_from_clist:Nn 7415   { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
\seq_set_from_clist:cn 7416 }
\seq_gset_from_clist:NN 7417 \cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
\seq_gset_from_clist:cN 7418 {
\seq_gset_from_clist:Nc 7419   \tl_set:Nx #1
\seq_gset_from_clist:cc 7420   { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
\seq_gset_from_clist:Nn 7421 }
\seq_gset_from_clist:NN 7422 \cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
\seq_gset_from_clist:cn 7423 {
7424   \tl_gset:Nx #1
7425   { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
7426 }
7427 \cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
7428 {
7429   \tl_gset:Nx #1
7430   { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
7431 }
7432 \cs_generate_variant:Nn \seq_set_from_clist:NN { Nc }
7433 \cs_generate_variant:Nn \seq_set_from_clist:NN { c , cc }
7434 \cs_generate_variant:Nn \seq_set_from_clist:Nn { c }
7435 \cs_generate_variant:Nn \seq_gset_from_clist:NN { Nc }
7436 \cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
7437 \cs_generate_variant:Nn \seq_gset_from_clist:Nn { c }

```

(End definition for `\seq_set_from_clist:NN` and others. These functions are documented on page 76.)

`\seq_const_from_clist:Nn` Almost identical to `\seq_set_from_clist:Nn`.

```

\seq_const_from_clist:cn 7438 \cs_new_protected:Npn \seq_const_from_clist:Nn #1#2
7439 {
7440   \tl_const:Nx #1
7441   { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
7442 }
7443 \cs_generate_variant:Nn \seq_const_from_clist:Nn { c }

```

(End definition for `\seq_const_from_clist:Nn`. This function is documented on page 77.)

`\seq_set_split:Nnn` When the separator is empty, everything is very simple, just map `__seq_wrap_item:n` through the items of the last argument. For non-trivial separators, the goal is to split a given token list at the marker, strip spaces from each item, and remove one set of outer braces if after removing leading and trailing spaces the item is enclosed within braces. After `\tl_replace_all:Nnn`, the token list `\l__seq_internal_a_tl` is a repetition of the pattern `__seq_set_split_auxi:w \prg_do_nothing: <item with spaces>`. Then, x-expansion causes `__seq_set_split_auxi:w` to trim spaces, and leaves its result as `__seq_set_split_auxii:w <trimmed item>`. This is then converted to the `l3seq` internal structure by another x-expansion. In the first step, we insert `\prg_do_nothing:` to avoid losing braces too early; that would cause space trimming to act within those lost braces. The second step is solely there to strip braces which are outermost after space trimming.

```

7444 \cs_new_protected:Npn \seq_set_split:Nnn

```

```

7445 { \__seq_set_split:NNnn \tl_set:Nx }
7446 \cs_new_protected:Npn \seq_gset_split:Nnn
7447 { \__seq_set_split:NNnn \tl_gset:Nx }
7448 \cs_new_protected:Npn \__seq_set_split:NNnn #1#2#3#4
7449 {
7450   \tl_if_empty:nTF {#3}
7451   {
7452     \tl_set:Nn \l__seq_internal_a_tl
7453     { \tl_map_function:nN {#4} \__seq_wrap_item:n }
7454   }
7455   {
7456     \tl_set:Nn \l__seq_internal_a_tl
7457     {
7458       \__seq_set_split_auxi:w \prg_do_nothing:
7459       #4
7460       \__seq_set_split_end:
7461     }
7462     \tl_replace_all:Nnn \l__seq_internal_a_tl { #3 }
7463     {
7464       \__seq_set_split_end:
7465       \__seq_set_split_auxi:w \prg_do_nothing:
7466     }
7467     \tl_set:Nx \l__seq_internal_a_tl { \l__seq_internal_a_tl }
7468   }
7469   #1 #2 { \s__seq \l__seq_internal_a_tl }
7470 }
7471 \cs_new:Npn \__seq_set_split_auxi:w #1 \__seq_set_split_end:
7472 {
7473   \exp_not:N \__seq_set_split_auxii:w
7474   \exp_args:No \tl_trim_spaces:n {#1}
7475   \exp_not:N \__seq_set_split_end:
7476 }
7477 \cs_new:Npn \__seq_set_split_auxii:w #1 \__seq_set_split_end:
7478 { \__seq_wrap_item:n {#1} }
7479 \cs_generate_variant:Nn \seq_set_split:Nnn { NnV }
7480 \cs_generate_variant:Nn \seq_gset_split:Nnn { NnV }

```

(End definition for `\seq_set_split:Nnn` and others. These functions are documented on page 77.)

`\seq_concat:NNN` When concatenating sequences, one must remove the leading `\s__seq` of the second sequence. The result starts with `\s__seq` (of the first sequence), which stops `f`-expansion.

`\seq_concat:ccc`

`\seq_gconcat:NNN`

```

7481 \cs_new_protected:Npn \seq_concat:NNN #1#2#3
7482 { \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
7483 \cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
7484 { \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
7485 \cs_generate_variant:Nn \seq_concat:NNN { ccc }
7486 \cs_generate_variant:Nn \seq_gconcat:NNN { ccc }

```

(End definition for `\seq_concat:NNN` and `\seq_gconcat:NNN`. These functions are documented on page 77.)

`\seq_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

`\seq_if_exist_p:c`

```

7487 \prg_new_eq_conditional:NNn \seq_if_exist:N \cs_if_exist:N
7488 { TF , T , F , p }
7489 \prg_new_eq_conditional:NNn \seq_if_exist:c \cs_if_exist:c

```

`\seq_if_exist:N \underline{TF}`

`\seq_if_exist:c \underline{TF}`

```
7490 { TF , T , F , p }
```

(End definition for `\seq_if_exist:NTF`. This function is documented on page 77.)

11.2 Appending data to either end

`\seq_put_left:Nn` When adding to the left of a sequence, remove `\s__seq`. This is done by `__seq_put_left_aux:w`, which also stops f-expansion.

```

\seq_put_left:Nv 7491 \cs_new_protected:Npn \seq_put_left:Nn #1#2
\seq_put_left:No 7492 {
\seq_put_left:Nx 7493   \tl_set:Nx #1
\seq_put_left:cn 7494   {
\seq_put_left:cV 7495     \exp_not:n { \s__seq \__seq_item:n {#2} }
\seq_put_left:cv 7496     \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
\seq_put_left:co 7497   }
\seq_put_left:cx 7498 }
\seq_gput_left:Nn 7499 \cs_new_protected:Npn \seq_gput_left:Nn #1#2
\seq_gput_left:Nv 7500 {
\seq_gput_left:Nx 7501   \tl_gset:Nx #1
\seq_gput_left:No 7502   {
\seq_gput_left:Nx 7503     \exp_not:n { \s__seq \__seq_item:n {#2} }
\seq_gput_left:cn 7504     \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
\seq_gput_left:cV 7505   }
\seq_gput_left:cv 7506 }
\seq_gput_left:co 7507 \cs_new:Npn \__seq_put_left_aux:w \s__seq { \exp_stop_f: }
\seq_gput_left:cx 7508 \cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , No , Nx }
\__seq_put_left_aux:w 7509 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , co , cx }
7510 \cs_generate_variant:Nn \seq_gput_left:Nn { NV , Nv , No , Nx }
7511 \cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , co , cx }
```

(End definition for `\seq_put_left:Nn`, `\seq_gput_left:Nn`, and `__seq_put_left_aux:w`. These functions are documented on page 77.)

`\seq_put_right:Nn` Since there is no trailing marker, adding an item to the right of a sequence simply means wrapping it in `__seq_item:n`.

```

\seq_put_right:Nv 7512 \cs_new_protected:Npn \seq_put_right:Nn #1#2
\seq_put_right:No 7513 { \tl_put_right:Nn #1 { \__seq_item:n {#2} } }
\seq_put_right:Nx 7514 \cs_new_protected:Npn \seq_gput_right:Nn #1#2
\seq_put_right:cn 7515 { \tl_gput_right:Nn #1 { \__seq_item:n {#2} } }
\seq_put_right:cV 7516 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , No , Nx }
\seq_put_right:cv 7517 \cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , co , cx }
\seq_put_right:co 7518 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , No , Nx }
\seq_put_right:cx 7519 \cs_generate_variant:Nn \seq_put_right:Nn { c , cV , cv , co , cx }
```

`\seq_gput_right:Nn` (End definition for `\seq_put_right:Nn` and `\seq_gput_right:Nn`. These functions are documented on page 77.)

11.3 Modifying sequences

`\seq_wrap_item:n` This function converts its argument to a proper sequence item in an x-expansion context.

```

7520 \cs_new:Npn \__seq_wrap_item:n #1 { \exp_not:n { \__seq_item:n {#1} } }
```

(End definition for `__seq_wrap_item:n`.)

`\l__seq_remove_seq` An internal sequence for the removal routines.

```
7521 \seq_new:N \l__seq_remove_seq
```

(End definition for `\l__seq_remove_seq`.)

`\seq_remove_duplicates:N` Removing duplicates means making a new list then copying it.

```
\seq_remove_duplicates:c 7522 \cs_new_protected:Npn \seq_remove_duplicates:N
\seq_gremove_duplicates:N 7523 { \__seq_remove_duplicates:NN \seq_set_eq:NN }
\seq_gremove_duplicates:c 7524 \cs_new_protected:Npn \seq_gremove_duplicates:N
\__seq_remove_duplicates:NN 7525 { \__seq_remove_duplicates:NN \seq_gset_eq:NN }
7526 \cs_new_protected:Npn \__seq_remove_duplicates:NN #1#2
7527 {
7528   \seq_clear:N \l__seq_remove_seq
7529   \seq_map_inline:Nn #2
7530   {
7531     \seq_if_in:NnF \l__seq_remove_seq {##1}
7532     { \seq_put_right:Nn \l__seq_remove_seq {##1} }
7533   }
7534   #1 #2 \l__seq_remove_seq
7535 }
7536 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
7537 \cs_generate_variant:Nn \seq_gremove_duplicates:N { c }
```

(End definition for `\seq_remove_duplicates:N`, `\seq_gremove_duplicates:N`, and `__seq_remove_duplicates:NN`. These functions are documented on page 80.)

`\seq_remove_all:Nn` The idea of the code here is to avoid a relatively expensive addition of items one at a time
`\seq_remove_all:cn` to an intermediate sequence. The approach taken is therefore similar to that in `__seq_`
`\seq_gremove_all:Nn` `pop_right:NNN`, using a “flexible” x-type expansion to do most of the work. As `\tl_`
`\seq_gremove_all:cn` `if_eq:nnT` is not expandable, a two-part strategy is needed. First, the x-type expansion
`__seq_remove_all_aux:NNn` uses `\str_if_eq:nnT` to find potential matches. If one is found, the expansion is halted
and the necessary set up takes place to use the `\tl_if_eq:NNT` test. The x-type is started
again, including all of the items copied already. This happens repeatedly until the entire
sequence has been scanned. The code is set up to avoid needing and intermediate scratch
list: the lead-off x-type expansion (`#1 #2 {#2}`) ensures that nothing is lost.

```
7538 \cs_new_protected:Npn \seq_remove_all:Nn
7539 { \__seq_remove_all_aux:NNn \tl_set:Nx }
7540 \cs_new_protected:Npn \seq_gremove_all:Nn
7541 { \__seq_remove_all_aux:NNn \tl_gset:Nx }
7542 \cs_new_protected:Npn \__seq_remove_all_aux:NNn #1#2#3
7543 {
7544   \__seq_push_item_def:n
7545   {
7546     \str_if_eq:nnT {##1} {#3}
7547     {
7548       \if_false: { \fi: }
7549       \tl_set:Nn \l__seq_internal_b_tl {##1}
7550       #1 #2
7551       { \if_false: } \fi:
7552       \exp_not:o {#2}
7553       \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
7554       { \use_none:nn }
7555     }
7556     \__seq_wrap_item:n {##1}
```

```

7557     }
7558     \tl_set:Nn \l__seq_internal_a_tl {#3}
7559     #1 #2 {#2}
7560     \__seq_pop_item_def:
7561   }
7562   \cs_generate_variant:Nn \seq_remove_all:Nn { c }
7563   \cs_generate_variant:Nn \seq_gremove_all:Nn { c }

```

(End definition for `\seq_remove_all:Nn`, `\seq_gremove_all:Nn`, and `__seq_remove_all_aux:NNn`. These functions are documented on page 80.)

```

\seq_reverse:N Previously, \seq_reverse:N was coded by collecting the items in reverse order after an
\seq_reverse:c \exp_stop_f: marker.
\seq_greverse:N \cs_new_protected:Npn \seq_reverse:N #1
\seq_greverse:c {
  \cs_set_eq:NN \@@_item:n \@@_reverse_item:nw
  \tl_set:Nf #2 { #2 \exp_stop_f: }
}
\cs_new:Npn \@@_reverse_item:nw #1 #2 \exp_stop_f:
{
  #2 \exp_stop_f:
  \@@_item:n {#1}
}

```

At first, this seems optimal, since we can forget about each item as soon as it is placed after `\exp_stop_f:`. Unfortunately, \TeX 's usual tail recursion does not take place in this case: since the following `__seq_reverse_item:nw` only reads tokens until `\exp_stop_f:`, and never reads the `\@@_item:n {#1}` left by the previous call, \TeX cannot remove that previous call from the stack, and in particular must retain the various macro parameters in memory, until the end of the replacement text is reached. The stack is thus only flushed after all the `__seq_reverse_item:nw` are expanded. Keeping track of the arguments of all those calls uses up a memory quadratic in the length of the sequence. \TeX can then not cope with more than a few thousand items.

Instead, we collect the items in the argument of `\exp_not:n`. The previous calls are cleanly removed from the stack, and the memory consumption becomes linear.

```

7564 \cs_new_protected:Npn \seq_reverse:N
7565 { \__seq_reverse:NN \tl_set:Nx }
7566 \cs_new_protected:Npn \seq_greverse:N
7567 { \__seq_reverse:NN \tl_gset:Nx }
7568 \cs_new_protected:Npn \__seq_reverse:NN #1 #2
7569 {
7570   \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
7571   \cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn
7572   #1 #2 { #2 \exp_not:n { } }
7573   \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
7574 }
7575 \cs_new:Npn \__seq_reverse_item:nwn #1 #2 \exp_not:n #3
7576 {
7577   #2
7578   \exp_not:n { \__seq_item:n {#1} #3 }
7579 }
7580 \cs_generate_variant:Nn \seq_reverse:N { c }
7581 \cs_generate_variant:Nn \seq_greverse:N { c }

```

(End definition for `\seq_reverse:N` and others. These functions are documented on page 80.)

`\seq_sort:Nn` Implemented in `l3sort`.

`\seq_sort:cn`

`\seq_gsort:Nn` (End definition for `\seq_sort:Nn` and `\seq_gsort:Nn`. These functions are documented on page 80.)

`\seq_gsort:cn`

11.4 Sequence conditionals

`\seq_if_empty_p:N` Similar to token lists, we compare with the empty sequence.

```

\seq_if_empty_p:c 7582 \prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }
\seq_if_empty:NTF 7583 {
\seq_if_empty:cTF 7584   \if_meaning:w #1 \c_empty_seq
7585   \prg_return_true:
7586   \else:
7587   \prg_return_false:
7588   \fi:
7589 }
7590 \prg_generate_conditional_variant:Nnn \seq_if_empty:N
7591 { c } { p , T , F , TF }
```

(End definition for `\seq_if_empty:NTF`. This function is documented on page 81.)

`\seq_shuffle:N` We apply the Fisher–Yates shuffle, storing items in `\toks` registers. We use the primitive

`\tex_uniformdeviate:D` for speed reasons. Its non-uniformity is of order its argument

`\seq_gshuffle:N` divided by 2^{28} , not too bad for small lists. For sequences with more than 13 elements

`\seq_gshuffle:c` there are more possible permutations than possible seeds ($13! > 2^{28}$) so the question

`__seq_shuffle:NN` of uniformity is somewhat moot. The integer variables are declared in `l3int`: load-order

`__seq_shuffle_item:n`

issues.

```

\g__seq_internal_seq 7592 \cs_if_exist:NTF \tex_uniformdeviate:D
7593 {
7594   \seq_new:N \g__seq_internal_seq
7595   \cs_new_protected:Npn \seq_shuffle:N { __seq_shuffle:NN \seq_set_eq:NN }
7596   \cs_new_protected:Npn \seq_gshuffle:N { __seq_shuffle:NN \seq_gset_eq:NN }
7597   \cs_new_protected:Npn __seq_shuffle:NN #1#2
7598   {
7599     \int_compare:nNnTF { \seq_count:N #2 } > \c_max_register_int
7600     {
7601       __kernel_msg_error:nxx { kernel } { shuffle-too-large }
7602       { \token_to_str:N #2 }
7603     }
7604     {
7605       \group_begin:
7606       \int_zero:N \l__seq_internal_a_int
7607       __seq_push_item_def:
7608       \cs_gset_eq:NN __seq_item:n __seq_shuffle_item:n
7609       #2
7610       __seq_pop_item_def:
7611       \seq_gset_from_inline_x:Nnn \g__seq_internal_seq
7612       { \int_step_function:nN { \l__seq_internal_a_int } }
7613       { \tex_the:D \tex_toks:D ##1 }
7614       \group_end:
7615       #1 #2 \g__seq_internal_seq
7616       \seq_gclear:N \g__seq_internal_seq
7617     }

```

```

7618     }
7619     \cs_new_protected:Npn \__seq_shuffle_item:n
7620     {
7621         \int_incr:N \l__seq_internal_a_int
7622         \int_set:Nn \l__seq_internal_b_int
7623         { 1 + \tex_uniformdeviate:D \l__seq_internal_a_int }
7624         \tex_toks:D \l__seq_internal_a_int
7625         = \tex_toks:D \l__seq_internal_b_int
7626         \tex_toks:D \l__seq_internal_b_int
7627     }
7628 }
7629 {
7630     \cs_new_protected:Npn \seq_shuffle:N #1
7631     {
7632         \__kernel_msg_error:nnn { kernel } { fp-no-random }
7633         { \seq_shuffle:N #1 }
7634     }
7635     \cs_new_eq:NN \seq_gshuffle:N \seq_shuffle:N
7636 }
7637 \cs_generate_variant:Nn \seq_shuffle:N { c }
7638 \cs_generate_variant:Nn \seq_gshuffle:N { c }

```

(End definition for `\seq_shuffle:N` and others. These functions are documented on page 81.)

`\seq_if_in:NnTF` The approach here is to define `__seq_item:n` to compare its argument with the test sequence. If the two items are equal, the mapping is terminated and `\group_end: \prg_return_true:` is inserted after skipping over the rest of the recursion. On the other hand, if there is no match then the loop breaks, returning `\prg_return_false:`. Everything is inside a group so that `__seq_item:n` is preserved in nested situations.

```

7639 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
7640 { T , F , TF }
7641 {
7642     \group_begin:
7643     \tl_set:Nn \l__seq_internal_a_tl {#2}
7644     \cs_set_protected:Npn \__seq_item:n ##1
7645     {
7646         \tl_set:Nn \l__seq_internal_b_tl {##1}
7647         \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
7648         \exp_after:wN \__seq_if_in:
7649         \fi:
7650     }
7651     #1
7652     \group_end:
7653     \prg_return_false:
7654     \prg_break_point:
7655 }
7656 \cs_new:Npn \__seq_if_in:
7657 { \prg_break:n { \group_end: \prg_return_true: } }
7658 \prg_generate_conditional_variant:Nnn \seq_if_in:Nn
7659 { NV , Nv , No , Nx , c , cV , cv , co , cx } { T , F , TF }

```

(End definition for `\seq_if_in:NnTF` and `__seq_if_in:`. This function is documented on page 81.)

11.5 Recovering data from sequences

`__seq_pop:NNNN` The two `pop` functions share their emptiness tests. We also use a common emptiness test
`__seq_pop_TF:NNNN` for all branching `get` and `pop` functions.

```

7660 \cs_new_protected:Npn \__seq_pop:NNNN #1#2#3#4
7661 {
7662   \if_meaning:w #3 \c_empty_seq
7663     \tl_set:Nn #4 { \q_no_value }
7664   \else:
7665     #1#2#3#4
7666   \fi:
7667 }
7668 \cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
7669 {
7670   \if_meaning:w #3 \c_empty_seq
7671     % \tl_set:Nn #4 { \q_no_value }
7672     \prg_return_false:
7673   \else:
7674     #1#2#3#4
7675     \prg_return_true:
7676   \fi:
7677 }
```

(End definition for `__seq_pop:NNNN` and `__seq_pop_TF:NNNN`.)

`\seq_get_left:NN` Getting an item from the left of a sequence is pretty easy: just trim off the first item
`\seq_get_left:cN` after `__seq_item:n` at the start. We append a `\q_no_value` item to cover the case of
`__seq_get_left:wnw` an empty sequence

```

7678 \cs_new_protected:Npn \seq_get_left:NN #1#2
7679 {
7680   \tl_set:Nx #2
7681   {
7682     \exp_after:wN \__seq_get_left:wnw
7683     #1 \__seq_item:n { \q_no_value } \s__seq_stop
7684   }
7685 }
7686 \cs_new:Npn \__seq_get_left:wnw #1 \__seq_item:n #2#3 \s__seq_stop
7687 { \exp_not:n {#2} }
7688 \cs_generate_variant:Nn \seq_get_left:NN { c }
```

(End definition for `\seq_get_left:NN` and `__seq_get_left:wnw`. This function is documented on page 78.)

`\seq_pop_left:NN` The approach to popping an item is pretty similar to that to get an item, with the only
`\seq_pop_left:cN` difference being that the sequence itself has to be redefined. This makes it more sensible
`\seq_gpop_left:NN` to use an auxiliary function for the local and global cases.

```

7689 \cs_new_protected:Npn \seq_pop_left:NN
7690 { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_set:Nn }
7691 \cs_new_protected:Npn \seq_gpop_left:NN
7692 { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_gset:Nn }
7693 \cs_new_protected:Npn \__seq_pop_left:NNN #1#2#3
7694 { \exp_after:wN \__seq_pop_left:wnwNNN #2 \s__seq_stop #1#2#3 }
7695 \cs_new_protected:Npn \__seq_pop_left:wnwNNN
7696 #1 \__seq_item:n #2#3 \s__seq_stop #4#5#6
```

```

7697 {
7698     #4 #5 { #1 #3 }
7699     \tl_set:Nn #6 {#2}
7700 }
7701 \cs_generate_variant:Nn \seq_pop_left:NN { c }
7702 \cs_generate_variant:Nn \seq_gpop_left:NN { c }

```

(End definition for `\seq_pop_left:NN` and others. These functions are documented on page 78.)

`\seq_get_right:NN` First remove `\s__seq` and prepend `\q_no_value`. The first argument of `__seq_get_right_loop:nw` is the last item found, and the second argument is empty until the end of the loop, where it is code that applies `\exp_not:n` to the last item and ends the loop.

```

\seq_get_right:cN
\__seq_get_right_loop:nw
\__seq_get_right_end:NnN
7703 \cs_new_protected:Npn \seq_get_right:NN #1#2
7704 {
7705     \tl_set:Nx #2
7706     {
7707         \exp_after:wN \use_i_ii:nnn
7708         \exp_after:wN \__seq_get_right_loop:nw
7709         \exp_after:wN \q_no_value
7710         #1
7711         \__seq_get_right_end:NnN \__seq_item:n
7712     }
7713 }
7714 \cs_new:Npn \__seq_get_right_loop:nw #1#2 \__seq_item:n
7715 {
7716     #2 \use_none:n {#1}
7717     \__seq_get_right_loop:nw
7718 }
7719 \cs_new:Npn \__seq_get_right_end:NnN #1#2#3 { \exp_not:n {#2} }
7720 \cs_generate_variant:Nn \seq_get_right:NN { c }

```

(End definition for `\seq_get_right:NN`, `__seq_get_right_loop:nw`, and `__seq_get_right_end:NnN`. This function is documented on page 78.)

`\seq_pop_right:NN` The approach to popping from the right is a bit more involved, but does use some of the same ideas as getting from the right. What is needed is a “flexible length” way to set a token list variable. This is supplied by the `{\if_false:} \fi:` construct. Using an x-type expansion and a “non-expanding” definition for `__seq_item:n`, the left-most $n - 1$ entries in a sequence of n items are stored back in the sequence. That needs a loop of unknown length, hence using the strange `\if_false:` way of including braces. When the last item of the sequence is reached, the closing brace for the assignment is inserted, and `\tl_set:Nn #3` is inserted in front of the final entry. This therefore does the pop assignment. One more iteration is performed, with an empty argument and `\use_none:nn`, which finally stops the loop.

```

7721 \cs_new_protected:Npn \seq_pop_right:NN
7722 { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_set:Nx }
7723 \cs_new_protected:Npn \seq_gpop_right:NN
7724 { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_gset:Nx }
7725 \cs_new_protected:Npn \__seq_pop_right:NNN #1#2#3
7726 {
7727     \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
7728     \cs_set_eq:NN \__seq_item:n \scan_stop:
7729     #1 #2
7730     { \if_false: } \fi: \s__seq

```

```

7731         \exp_after:wN \use_i:nnn
7732         \exp_after:wN \__seq_pop_right_loop:nn
7733         #2
7734         {
7735             \if_false: { \fi: }
7736             \tl_set:Nx #3
7737         }
7738         { } \use_none:nn
7739         \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
7740     }
7741     \cs_new:Npn \__seq_pop_right_loop:nn #1#2
7742     {
7743         #2 { \exp_not:n {#1} }
7744         \__seq_pop_right_loop:nn
7745     }
7746     \cs_generate_variant:Nn \seq_pop_right:NN { c }
7747     \cs_generate_variant:Nn \seq_gpop_right:NN { c }

```

(End definition for `\seq_pop_right:NN` and others. These functions are documented on page 78.)

`\seq_get_left:NNTF` Getting from the left or right with a check on the results. The first argument to `__seq_pop_TF:NNNN` is left unused.

```

7748 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
7749 { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
7750 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
7751 { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
7752 \prg_generate_conditional_variant:Nnn \seq_get_left:NN
7753 { c } { T , F , TF }
7754 \prg_generate_conditional_variant:Nnn \seq_get_right:NN
7755 { c } { T , F , TF }

```

(End definition for `\seq_get_left:NNTF` and `\seq_get_right:NNTF`. These functions are documented on page 79.)

`\seq_pop_left:NNTF` More or less the same for popping.

```

7756 \prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2
7757 { T , F , TF }
7758 { \__seq_pop_TF:NNNN \__seq_pop_left:NN \tl_set:Nn #1 #2 }
7759 \prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2
7760 { T , F , TF }
7761 { \__seq_pop_TF:NNNN \__seq_pop_left:NN \tl_gset:Nn #1 #2 }
7762 \prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2
7763 { T , F , TF }
7764 { \__seq_pop_TF:NNNN \__seq_pop_right:NN \tl_set:Nx #1 #2 }
7765 \prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2
7766 { T , F , TF }
7767 { \__seq_pop_TF:NNNN \__seq_pop_right:NN \tl_gset:Nx #1 #2 }
7768 \prg_generate_conditional_variant:Nnn \seq_pop_left:NN { c }
7769 { T , F , TF }
7770 \prg_generate_conditional_variant:Nnn \seq_gpop_left:NN { c }
7771 { T , F , TF }
7772 \prg_generate_conditional_variant:Nnn \seq_pop_right:NN { c }
7773 { T , F , TF }
7774 \prg_generate_conditional_variant:Nnn \seq_gpop_right:NN { c }
7775 { T , F , TF }

```

(End definition for `\seq_pop_left:NNTF` and others. These functions are documented on page 79.)

`\seq_item:Nn` The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then the argument delimited by `__seq_item:wNn` `\seq_item:n` is `\prg_break:` instead of being empty, terminating the loop and returning nothing at all.

```

7776 \cs_new:Npn \seq_item:Nn #1
7777 { \exp_after:wN \__seq_item:wNn #1 \s__seq_stop #1 }
7778 \cs_new:Npn \__seq_item:wNn \s__seq #1 \s__seq_stop #2#3
7779 {
7780   \exp_args:Nf \__seq_item:nwn
7781   { \exp_args:Nf \__seq_item:nN { \int_eval:n {#3} } #2 }
7782   #1
7783   \prg_break: \__seq_item:n { }
7784   \prg_break_point:
7785 }
7786 \cs_new:Npn \__seq_item:nN #1#2
7787 {
7788   \int_compare:nNnTF {#1} < 0
7789   { \int_eval:n { \seq_count:N #2 + 1 + #1 } }
7790   {#1}
7791 }
7792 \cs_new:Npn \__seq_item:nwn #1#2 \__seq_item:n #3
7793 {
7794   #2
7795   \int_compare:nNnTF {#1} = 1
7796   { \prg_break:n { \exp_not:n {#3} } }
7797   { \exp_args:Nf \__seq_item:nwn { \int_eval:n { #1 - 1 } } }
7798 }
7799 \cs_generate_variant:Nn \seq_item:Nn { c }

```

(End definition for `\seq_item:Nn` and others. This function is documented on page 78.)

`\seq_rand_item:N` Importantly, `\seq_item:Nn` only evaluates its argument once.

```

7800 \cs_new:Npn \seq_rand_item:N #1
7801 {
7802   \seq_if_empty:NF #1
7803   { \seq_item:Nn #1 { \int_rand:nn { 1 } { \seq_count:N #1 } } }
7804 }
7805 \cs_generate_variant:Nn \seq_rand_item:N { c }

```

(End definition for `\seq_rand_item:N`. This function is documented on page 79.)

11.6 Mapping to sequences

`\seq_map_break:` To break a function, the special token `\prg_break_point:Nn` is used to find the end of the code. Any ending code is then inserted before the return value of `\seq_map_break:n` is inserted.

```

7806 \cs_new:Npn \seq_map_break:
7807 { \prg_map_break:Nn \seq_map_break: { } }
7808 \cs_new:Npn \seq_map_break:n
7809 { \prg_map_break:Nn \seq_map_break: }

```

(End definition for `\seq_map_break:` and `\seq_map_break:n`. These functions are documented on page 83.)

`\seq_map_function:NN` The idea here is to apply the code of #2 to each item in the sequence without altering
`\seq_map_function:cN` the definition of `__seq_item:n`. The argument delimited by `__seq_item:n` is almost
`__seq_map_function:NNn` always empty, except at the end of the loop where it is `\prg_break:`. This allows to
break the loop without needing to do a (relatively-expensive) quark test.

```

7810 \cs_new:Npn \seq_map_function:NN #1#2
7811 {
7812   \exp_after:wN \use_i_ii:nnn
7813   \exp_after:wN \__seq_map_function:Nw
7814   \exp_after:wN #2
7815   #1
7816   \prg_break: \__seq_item:n { } \prg_break_point:
7817   \prg_break_point:Nn \seq_map_break: { }
7818 }
7819 \cs_new:Npn \__seq_map_function:Nw #1#2 \__seq_item:n #3
7820 {
7821   #2
7822   #1 {#3}
7823   \__seq_map_function:Nw #1
7824 }
7825 \cs_generate_variant:Nn \seq_map_function:NN { c }

```

(End definition for `\seq_map_function:NN` and `__seq_map_function:NNn`. This function is documented on page 81.)

`__seq_push_item_def:n` The definition of `__seq_item:n` needs to be saved and restored at various points within
`__seq_push_item_def:x` the mapping and manipulation code. That is handled here: as always, this approach uses
`__seq_push_item_def:` global assignments.
`__seq_pop_item_def:`

```

7826 \cs_new_protected:Npn \__seq_push_item_def:n
7827 {
7828   \__seq_push_item_def:
7829   \cs_gset:Npn \__seq_item:n ##1
7830 }
7831 \cs_new_protected:Npn \__seq_push_item_def:x
7832 {
7833   \__seq_push_item_def:
7834   \cs_gset:Npx \__seq_item:n ##1
7835 }
7836 \cs_new_protected:Npn \__seq_push_item_def:
7837 {
7838   \int_gincr:N \g__kernel_prg_map_int
7839   \cs_gset_eq:cN { __seq_map_ \int_use:N \g__kernel_prg_map_int :w }
7840   \__seq_item:n
7841 }
7842 \cs_new_protected:Npn \__seq_pop_item_def:
7843 {
7844   \cs_gset_eq:Nc \__seq_item:n
7845   { __seq_map_ \int_use:N \g__kernel_prg_map_int :w }
7846   \int_gdecr:N \g__kernel_prg_map_int
7847 }

```

(End definition for `__seq_push_item_def:n`, `__seq_push_item_def:`, and `__seq_pop_item_def:`.)

\seq_map_inline:Nn The idea here is that `__seq_item:n` is already “applied” to each item in a sequence, and so an in-line mapping is just a case of redefining `__seq_item:n`.

\seq_map_inline:cn

```

7848 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
7849 {
7850   \__seq_push_item_def:n {#2}
7851   #1
7852   \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
7853 }
7854 \cs_generate_variant:Nn \seq_map_inline:Nn { c }

```

(End definition for `\seq_map_inline:Nn`. This function is documented on page 81.)

\seq_map_tokens:Nn This is based on the function mapping but using the same tricks as described for `\prop_map_tokens:Nn`. The idea is to remove the leading `\s__seq` and apply the tokens such that they are safe with the break points, hence the `\use:n`.

\seq_map_tokens:cn

__seq_map_tokens:nw

```

7855 \cs_new:Npn \seq_map_tokens:Nn #1#2
7856 {
7857   \exp_last_unbraced:Nno
7858   \use_i:nn { \__seq_map_tokens:nw {#2} } #1
7859   \prg_break: \__seq_item:n { } \prg_break_point:
7860   \prg_break_point:Nn \seq_map_break: { }
7861 }
7862 \cs_generate_variant:Nn \seq_map_tokens:Nn { c }
7863 \cs_new:Npn \__seq_map_tokens:nw #1#2 \__seq_item:n #3
7864 {
7865   #2
7866   \use:n {#1} {#3}
7867   \__seq_map_tokens:nw {#1}
7868 }

```

(End definition for `\seq_map_tokens:Nn` and `__seq_map_tokens:nw`. This function is documented on page 82.)

\seq_map_variable:NNn This is just a specialised version of the in-line mapping function, using an `x`-type expansion for the code set up so that the number of `#` tokens required is as expected.

\seq_map_variable:Ncn

\seq_map_variable:cNn

\seq_map_variable:ccn

```

7869 \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
7870 {
7871   \__seq_push_item_def:x
7872   {
7873     \tl_set:Nn \exp_not:N #2 {##1}
7874     \exp_not:n {#3}
7875   }
7876   #1
7877   \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
7878 }
7879 \cs_generate_variant:Nn \seq_map_variable:NNn { Nc }
7880 \cs_generate_variant:Nn \seq_map_variable:NNn { c , cc }

```

(End definition for `\seq_map_variable:NNn`. This function is documented on page 82.)

\seq_map_indexed_function:NN Similar to `\seq_map_function:NN` but we keep track of the item index as a `;`-delimited argument of `__seq_map_indexed:Nw`.

\seq_map_indexed_inline:Nn

__seq_map_indexed:nNn

__seq_map_indexed:Nw

```

7881 \cs_new:Npn \seq_map_indexed_function:NN #1#2
7882 {

```

```

7883     \__seq_map_indexed:NN #1#2
7884     \prg_break_point:Nn \seq_map_break: { }
7885   }
7886 \cs_new_protected:Npn \seq_map_indexed_inline:Nn #1#2
7887 {
7888   \int_gincr:N \g__kernel_prg_map_int
7889   \cs_gset_protected:cpn
7890     { \__seq_map_ \int_use:N \g__kernel_prg_map_int :w } ##1##2 {#2}
7891   \exp_args:NNc \__seq_map_indexed:NN #1
7892     { \__seq_map_ \int_use:N \g__kernel_prg_map_int :w }
7893   \prg_break_point:Nn \seq_map_break:
7894     { \int_gdecr:N \g__kernel_prg_map_int }
7895 }
7896 \cs_new:Npn \__seq_map_indexed:NN #1#2
7897 {
7898   \exp_after:wN \__seq_map_indexed:Nw
7899   \exp_after:wN #2
7900   \int_value:w 1
7901   \exp_after:wN \use_i:nn
7902   \exp_after:wN ;
7903   #1
7904   \prg_break: \__seq_item:n { } \prg_break_point:
7905 }
7906 \cs_new:Npn \__seq_map_indexed:Nw #1#2 ; #3 \__seq_item:n #4
7907 {
7908   #3
7909   #1 {#2} {#4}
7910   \exp_after:wN \__seq_map_indexed:Nw
7911   \exp_after:wN #1
7912   \int_value:w \int_eval:w 1 + #2 ;
7913 }

```

(End definition for `\seq_map_indexed_function:NN` and others. These functions are documented on page 82.)

`\seq_set_map_x:NNn` Very similar to `\seq_set_filter:NNn`. We could actually merge the two within a single function, but it would have weird semantics.

`\seq_gset_map_x:NNn`

`__seq_set_map_x:NNNn`

```

7914 \cs_new_protected:Npn \seq_set_map_x:NNn
7915 { \__seq_set_map_x:NNNn \tl_set:Nx }
7916 \cs_new_protected:Npn \seq_gset_map_x:NNn
7917 { \__seq_set_map_x:NNNn \tl_gset:Nx }
7918 \cs_new_protected:Npn \__seq_set_map_x:NNNn #1#2#3#4
7919 {
7920   \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
7921   #1 #2 { #3 }
7922   \__seq_pop_item_def:
7923 }

```

(End definition for `\seq_set_map_x:NNn`, `\seq_gset_map_x:NNn`, and `__seq_set_map_x:NNNn`. These functions are documented on page 84.)

`\seq_set_map:NNn` Similar to `\seq_set_map_x:NNn`, but prevents expansion of the `<inline function>`.

`\seq_gset_map:NNn`

`__seq_set_map:NNNn`

```

7924 \cs_new_protected:Npn \seq_set_map:NNn
7925 { \__seq_set_map:NNNn \tl_set:Nx }
7926 \cs_new_protected:Npn \seq_gset_map:NNn

```

```

7927 { \__seq_set_map:NNNn \tl_gset:Nx }
7928 \cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
7929 {
7930   \__seq_push_item_def:n { \exp_not:n { \__seq_item:n {#4} } }
7931   #1 #2 { #3 }
7932   \__seq_pop_item_def:
7933 }

```

(End definition for `\seq_set_map:NNn`, `\seq_gset_map:NNn`, and `__seq_set_map:NNNn`. These functions are documented on page 83.)

\seq_count:N Since counting the items in a sequence is quite common, we optimize it by grabbing 8 items at a time and correspondingly adding 8 to an integer expression. At the end of the loop, #9 is `__seq_count_end:w` instead of being empty. It removes 8+ and instead places the number of `__seq_item:n` that `__seq_count:w` grabbed before reaching the end of the sequence.

```

7934 \cs_new:Npn \seq_count:N #1
7935 {
7936   \int_eval:n
7937   {
7938     \exp_after:wN \use_i:nn
7939     \exp_after:wN \__seq_count:w
7940     #1
7941     \__seq_count_end:w \__seq_item:n 7
7942     \__seq_count_end:w \__seq_item:n 6
7943     \__seq_count_end:w \__seq_item:n 5
7944     \__seq_count_end:w \__seq_item:n 4
7945     \__seq_count_end:w \__seq_item:n 3
7946     \__seq_count_end:w \__seq_item:n 2
7947     \__seq_count_end:w \__seq_item:n 1
7948     \__seq_count_end:w \__seq_item:n 0
7949     \prg_break_point:
7950   }
7951 }
7952 \cs_new:Npn \__seq_count:w
7953   #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4 \__seq_item:n
7954   #5 \__seq_item:n #6 \__seq_item:n #7 \__seq_item:n #8 #9 \__seq_item:n
7955   { #9 8 + \__seq_count:w }
7956 \cs_new:Npn \__seq_count_end:w #1#2 \prg_break_point: {#1}
7957 \cs_generate_variant:Nn \seq_count:N { c }

```

(End definition for `\seq_count:N`, `__seq_count:w`, and `__seq_count_end:w`. This function is documented on page 84.)

11.7 Using sequences

\seq_use:Nnnn See `\clist_use:Nnnn` for a general explanation. The main difference is that we use `__seq_item:n` as a delimiter rather than commas. We also need to add `__seq_item:n` at various places, and `\s__seq`.

\seq_use:cnnn

```

\__seq_use:NNnNnn
\__seq_use_setup:w
\__seq_use:nwwwnwn
\__seq_use:nwnn
\seq_use:Nn
\seq_use:cn

```

```

7958 \cs_new:Npn \seq_use:Nnnn #1#2#3#4
7959 {
7960   \seq_if_exist:NTF #1
7961   {
7962     \int_case:nnF { \seq_count:N #1 }

```



```

7963     {
7964         { 0 } { }
7965         { 1 } { \exp_after:wN \__seq_use:NnnNnn #1 ? { } { } }
7966         { 2 } { \exp_after:wN \__seq_use:NnnNnn #1 {#2} }
7967     }
7968     {
7969         \exp_after:wN \__seq_use_setup:w #1 \__seq_item:n
7970         \s__seq_mark { \__seq_use:nwwwnwn {#3} }
7971         \s__seq_mark { \__seq_use:nwn {#4} }
7972         \s__seq_stop { }
7973     }
7974 }
7975 {
7976     \__kernel_msg_expandable_error:nnn
7977     { kernel } { bad-variable } {#1}
7978 }
7979 }
7980 \cs_generate_variant:Nn \seq_use:Nnnn { c }
7981 \cs_new:Npn \__seq_use:NnnNnn #1#2#3#4#5#6 { \exp_not:n { #3 #6 #5 } }
7982 \cs_new:Npn \__seq_use_setup:w \s__seq { \__seq_use:nwwwnwn { } }
7983 \cs_new:Npn \__seq_use:nwwwnwn
7984     #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4#5
7985     \s__seq_mark #6#7 \s__seq_stop #8
7986     {
7987         #6 \__seq_item:n {#3} \__seq_item:n {#4} #5
7988         \s__seq_mark {#6} #7 \s__seq_stop { #8 #1 #2 }
7989     }
7990 \cs_new:Npn \__seq_use:nwn #1 \__seq_item:n #2 #3 \s__seq_stop #4
7991     { \exp_not:n { #4 #1 #2 } }
7992 \cs_new:Npn \seq_use:Nn #1#2
7993     { \seq_use:Nnnn #1 {#2} {#2} {#2} }
7994 \cs_generate_variant:Nn \seq_use:Nn { c }

```

(End definition for `\seq_use:Nnnn` and others. These functions are documented on page 84.)

11.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

\seq_push:Nn Pushing to a sequence is the same as adding on the left.

\seq_push:NV	7995 \cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
\seq_push:Nv	7996 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
\seq_push:No	7997 \cs_new_eq:NN \seq_push:No \seq_put_left:No
\seq_push:Nx	7998 \cs_new_eq:NN \seq_push:Nx \seq_put_left:Nx
\seq_push:cn	7999 \cs_new_eq:NN \seq_push:cn \seq_put_left:cn
\seq_push:cV	8000 \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
\seq_push:cV	8001 \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
\seq_push:co	8002 \cs_new_eq:NN \seq_push:co \seq_put_left:co
\seq_push:co	8003 \cs_new_eq:NN \seq_push:co \seq_put_left:co
\seq_push:cx	8004 \cs_new_eq:NN \seq_push:cx \seq_put_left:cx
\seq_gpush:Nn	8005 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
\seq_gpush:NV	8006 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
\seq_gpush:Nv	8007 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
\seq_gpush:No	8008 \cs_new_eq:NN \seq_gpush:No \seq_gput_left:No
\seq_gpush:Nx	
\seq_gpush:cn	
\seq_gpush:cV	
\seq_gpush:cV	
\seq_gpush:co	
\seq_gpush:co	
\seq_gpush:cx	

```

8009 \cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
8010 \cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
8011 \cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
8012 \cs_new_eq:NN \seq_gpush:cv \seq_gput_left:cv
8013 \cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
8014 \cs_new_eq:NN \seq_gpush:cx \seq_gput_left:cx

```

(End definition for `\seq_push:Nn` and `\seq_gpush:Nn`. These functions are documented on page 86.)

`\seq_get:NN` In most cases, getting items from the stack does not need to specify that this is from the left. So alias are provided.

```

\seq_get:cN
\seq_pop:NN      8015 \cs_new_eq:NN \seq_get:NN \seq_get_left:NN
\seq_pop:cN      8016 \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
\seq_gpop:NN    8017 \cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
\seq_gpop:cN     8018 \cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
                8019 \cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
                8020 \cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN

```

(End definition for `\seq_get:NN`, `\seq_pop:NN`, and `\seq_gpop:NN`. These functions are documented on page 85.)

`\seq_get:NNTF` More copies.

```

\seq_get:cNTF      8021 \prg_new_eq_conditional:NNn \seq_get:NN \seq_get_left:NN { T , F , TF }
\seq_pop:NNTF    8022 \prg_new_eq_conditional:NNn \seq_get:cN \seq_get_left:cN { T , F , TF }
\seq_pop:cNTF      8023 \prg_new_eq_conditional:NNn \seq_pop:NN \seq_pop_left:NN { T , F , TF }
\seq_gpop:NNTF   8024 \prg_new_eq_conditional:NNn \seq_pop:cN \seq_pop_left:cN { T , F , TF }
\seq_gpop:cNTF     8025 \prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
                8026 \prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }

```

(End definition for `\seq_get:NNTF`, `\seq_pop:NNTF`, and `\seq_gpop:NNTF`. These functions are documented on page 85.)

11.9 Viewing sequences

`\seq_show:N` Apply the general `\msg_show:nnnnnn`.

```

\seq_show:c      8027 \cs_new_protected:Npn \seq_show:N { \__seq_show:NN \msg_show:nnxxxx }
\seq_log:N      8028 \cs_generate_variant:Nn \seq_show:N { c }
\seq_log:c      8029 \cs_new_protected:Npn \seq_log:N { \__seq_show:NN \msg_log:nnxxxx }
\__seq_show:NN  8030 \cs_generate_variant:Nn \seq_log:N { c }
                8031 \cs_new_protected:Npn \__seq_show:NN #1#2
                8032 {
                8033   \__kernel_chk_defined:NT #2
                8034   {
                8035     #1 { LaTeX/kernel } { show-seq }
                8036     { \token_to_str:N #2 }
                8037     { \seq_map_function:NN #2 \msg_show_item:n }
                8038     { } { }
                8039   }
                8040 }

```

(End definition for `\seq_show:N`, `\seq_log:N`, and `__seq_show:NN`. These functions are documented on page 88.)

11.10 Scratch sequences

`\l_tmpa_seq` Temporary comma list variables.

`\l_tmpb_seq` 8041 \seq_new:N \l_tmpa_seq

`\g_tmpa_seq` 8042 \seq_new:N \l_tmpb_seq

`\g_tmpb_seq` 8043 \seq_new:N \g_tmpa_seq

8044 \seq_new:N \g_tmpb_seq

(End definition for \l_tmpa_seq and others. These variables are documented on page 88.)

8045 \</package>

12 l3int implementation

8046 *package>

8047 \@@=int)

The following test files are used for this code: m3int001,m3int002,m3int03.

`\c_max_register_int` Done in l3basics.

(End definition for \c_max_register_int. This variable is documented on page 101.)

`__int_to_roman:w` Done in l3basics.

`\if_int_compare:w` *(End definition for __int_to_roman:w and \if_int_compare:w. This function is documented on page 102.)*

`\or:` Done in l3basics.

(End definition for \or:. This function is documented on page 102.)

`\int_value:w` Here are the remaining primitives for number comparisons and expressions.

`__int_eval:w` 8048 \cs_new_eq:NN \int_value:w \tex_number:D

`__int_eval_end:` 8049 \cs_new_eq:NN __int_eval:w \tex_numexpr:D

`\if_int_odd:w` 8050 \cs_new_eq:NN __int_eval_end: \tex_relax:D

`\if_case:w` 8051 \cs_new_eq:NN \if_int_odd:w \tex_ifodd:D

8052 \cs_new_eq:NN \if_case:w \tex_ifcase:D

(End definition for \int_value:w and others. These functions are documented on page 102.)

`\s__int_mark` Scan marks used throughout the module.

`\s__int_stop` 8053 \scan_new:N \s__int_mark

8054 \scan_new:N \s__int_stop

(End definition for \s__int_mark and \s__int_stop.)

`__int_use_none_delimit_by_s_stop:w` Function to gobble until a scan mark.

8055 \cs_new:Npn __int_use_none_delimit_by_s_stop:w #1 \s__int_stop { }

(End definition for __int_use_none_delimit_by_s_stop:w.)

`\q__int_recursion_tail` Quarks for recursion.

`\q__int_recursion_stop` 8056 \quark_new:N \q__int_recursion_tail

8057 \quark_new:N \q__int_recursion_stop

(End definition for \q__int_recursion_tail and \q__int_recursion_stop.)

`_int_if_recursion_tail_stop_do:Nn` Functions to query quarks.
`_int_if_recursion_tail_stop:N`
8058 `_kernel_quark_new_test:N _int_if_recursion_tail_stop_do:Nn`
8059 `_kernel_quark_new_test:N _int_if_recursion_tail_stop:N`
(End definition for `_int_if_recursion_tail_stop_do:Nn` and `_int_if_recursion_tail_stop:N`)

12.1 Integer expressions

`\int_eval:n` Wrapper for `_int_eval:w`: can be used in an integer expression or directly in the input stream. When debugging, use parentheses to catch early termination.

```
8060 \cs_new:Npn \int_eval:n #1
8061 { \int_value:w \_int_eval:w #1 \_int_eval_end: }
8062 \cs_new:Npn \int_eval:w { \int_value:w \_int_eval:w }
```

(End definition for `\int_eval:n` and `\int_eval:w`. These functions are documented on page 90.)

`\int_sign:n` See `\int_abs:n`. Evaluate the expression once (and when debugging is enabled, check that the expression is well-formed), then test the first character to determine the sign. This is wrapped in `\int_value:w ... \exp_stop_f:` to ensure a fixed number of expansions and to avoid dealing with closing the conditionals.

`_int_sign:Nw`

```
8063 \cs_new:Npn \int_sign:n #1
8064 {
8065   \int_value:w \exp_after:wN \_int_sign:Nw
8066   \int_value:w \_int_eval:w #1 \_int_eval_end: ;
8067   \exp_stop_f:
8068 }
8069 \cs_new:Npn \_int_sign:Nw #1#2 ;
8070 {
8071   \if_meaning:w 0 #1
8072   0
8073   \else:
8074     \if_meaning:w - #1 - \fi: 1
8075   \fi:
8076 }
```

(End definition for `\int_sign:n` and `_int_sign:Nw`. This function is documented on page 91.)

`\int_abs:n` Functions for min, max, and absolute value with only one evaluation. The absolute value is obtained by removing a leading sign if any. All three functions expand in two steps.

`_int_abs:N`

`\int_max:nn`

`\int_min:nn`

`_int_maxmin:wwN`

```
8077 \cs_new:Npn \int_abs:n #1
8078 {
8079   \int_value:w \exp_after:wN \_int_abs:N
8080   \int_value:w \_int_eval:w #1 \_int_eval_end:
8081   \exp_stop_f:
8082 }
8083 \cs_new:Npn \_int_abs:N #1
8084 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
8085 \cs_set:Npn \int_max:nn #1#2
8086 {
8087   \int_value:w \exp_after:wN \_int_maxmin:wwN
8088   \int_value:w \_int_eval:w #1 \exp_after:wN ;
8089   \int_value:w \_int_eval:w #2 ;
8090   >
8091   \exp_stop_f:
```

```

8092 }
8093 \cs_set:Npn \int_min:nn #1#2
8094 {
8095   \int_value:w \exp_after:wN \__int_maxmin:wwN
8096   \int_value:w \__int_eval:w #1 \exp_after:wN ;
8097   \int_value:w \__int_eval:w #2 ;
8098   <
8099   \exp_stop_f:
8100 }
8101 \cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
8102 {
8103   \if_int_compare:w #1 #3 #2 ~
8104     #1
8105   \else:
8106     #2
8107   \fi:
8108 }

```

(End definition for `\int_abs:n` and others. These functions are documented on page 91.)

`\int_div_truncate:nn` As `__int_eval:w` rounds the result of a division we also provide a version that truncates the result. We use an auxiliary to make sure numerator and denominator are only evaluated once: this comes in handy when those are more expressions are expensive to evaluate (e.g., `\tl_count:n`). If the numerator `#1#2` is 0, then we divide 0 by the denominator (this ensures that 0/0 is correctly reported as an error). Otherwise, shift the numerator `#1#2` towards 0 by $(| \#3\#4 | - 1)/2$, which we round away from zero. It turns out that this quantity exactly compensates the difference between ε -TeX's rounding and the truncating behaviour that we want. The details are thanks to Heiko Oberdiek: getting things right in all cases is not so easy.

```

8109 \cs_new:Npn \int_div_truncate:nn #1#2
8110 {
8111   \int_value:w \__int_eval:w
8112   \exp_after:wN \__int_div_truncate:NwNw
8113   \int_value:w \__int_eval:w #1 \exp_after:wN ;
8114   \int_value:w \__int_eval:w #2 ;
8115   \__int_eval_end:
8116 }
8117 \cs_new:Npn \__int_div_truncate:NwNw #1#2; #3#4;
8118 {
8119   \if_meaning:w 0 #1
8120     0
8121   \else:
8122     (
8123       #1#2
8124       \if_meaning:w - #1 + \else: - \fi:
8125       ( \if_meaning:w - #3 - \fi: #3#4 - 1 ) / 2
8126     )
8127   \fi:
8128   / #3#4
8129 }

```

For the sake of completeness:

```

8130 \cs_new:Npn \int_div_round:nn #1#2
8131 { \int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }

```

Finally there's the modulus operation.

```

8132 \cs_new:Npn \int_mod:nn #1#2
8133 {
8134   \int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
8135   \int_value:w \__int_eval:w #1 \exp_after:wN ;
8136   \int_value:w \__int_eval:w #2 ;
8137   \__int_eval_end:
8138 }
8139 \cs_new:Npn \__int_mod:ww #1; #2;
8140 { #1 - ( \__int_div_truncate:NwNw #1 ; #2 ; ) * #2 }

```

(End definition for `\int_div_truncate:nn` and others. These functions are documented on page 91.)

`__kernel_int_add:nnn` Equivalent to `\int_eval:n {#1+#2+#3}` except that overflow only occurs if the final result overflows $[-2^{31} + 1, 2^{31} - 1]$. The idea is to choose the order in which the three numbers are added together. If #1 and #2 have opposite signs (one is in $[-2^{31} + 1, -1]$ and the other in $[0, 2^{31} - 1]$) then #1+#2 cannot overflow so we compute the result as #1+#2+#3. If they have the same sign, then either #3 has the same sign and the order does not matter, or #3 has the opposite sign and any order in which #3 is not last will work. We use #1+#3+#2.

```

8141 \cs_new:Npn \__kernel_int_add:nnn #1#2#3
8142 {
8143   \int_value:w \__int_eval:w #1
8144   \if_int_compare:w #2 < \c_zero_int \exp_after:wN \reverse_if:N \fi:
8145   \if_int_compare:w #1 < \c_zero_int + #2 + #3 \else: + #3 + #2 \fi:
8146   \__int_eval_end:
8147 }

```

(End definition for `__kernel_int_add:nnn`.)

12.2 Creating and initialising integers

`\int_new:N` Two ways to do this: one for the format and one for the L^AT_EX 2_ε package. In plain T_EX, `\int_new:c` `\newcount` (and other allocators) are `\outer:` to allow the code here to work in “generic” mode this is therefore accessed by name. (The same applies to `\newbox`, `\newdimen` and so on.)

```

8148 \cs_new_protected:Npn \int_new:N #1
8149 {
8150   \__kernel_chk_if_free_cs:N #1
8151   \cs:w newcount \cs_end: #1
8152 }
8153 \cs_generate_variant:Nn \int_new:N { c }

```

(End definition for `\int_new:N`. This function is documented on page 91.)

`\int_const:Nn` As stated, most constants can be defined as `\chardef` or `\mathchardef` but that's engine dependent. As a result, there is some set up code to determine what can be done. No full engine testing just yet so everything is a little awkward. We cannot use `\int_gset:Nn` because (when `check-declarations` is enabled) this runs some checks that constants would fail.

```

8154 \cs_new_protected:Npn \int_const:Nn #1#2
8155 {
8156   \int_compare:nNnTF {#2} < \c_zero_int

```

```

8157     {
8158         \int_new:N #1
8159         \tex_global:D
8160     }
8161     {
8162         \int_compare:nNnTF {#2} > \c__int_max_constdef_int
8163         {
8164             \int_new:N #1
8165             \tex_global:D
8166         }
8167         {
8168             \__kernel_chk_if_free_cs:N #1
8169             \tex_global:D \__int_constdef:Nw
8170         }
8171     }
8172     #1 = \__int_eval:w #2 \__int_eval_end:
8173 }
8174 \cs_generate_variant:Nn \int_const:Nn { c }
8175 \if_int_odd:w 0
8176     \cs_if_exist:NT \tex_luatexversion:D { 1 }
8177     \cs_if_exist:NT \tex_omathchardef:D { 1 }
8178     \cs_if_exist:NT \tex_XeTeXversion:D { 1 } ~
8179     \cs_if_exist:NTF \tex_omathchardef:D
8180         { \cs_new_eq:NN \__int_constdef:Nw \tex_omathchardef:D }
8181         { \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D }
8182     \__int_constdef:Nw \c__int_max_constdef_int 1114111 ~
8183 \else:
8184     \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
8185     \tex_mathchardef:D \c__int_max_constdef_int 32767 ~
8186 \fi:

```

(End definition for `\int_const:Nn`, `__int_constdef:Nw`, and `\c__int_max_constdef_int`. This function is documented on page 92.)

`\int_zero:N` Functions that reset an *integer* register to zero.

`\int_zero:c` 8187 `\cs_new_protected:Npn \int_zero:N #1 { #1 = \c_zero_int }`

`\int_gzero:N` 8188 `\cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero_int }`

`\int_gzero:c` 8189 `\cs_generate_variant:Nn \int_zero:N { c }`
8190 `\cs_generate_variant:Nn \int_gzero:N { c }`

(End definition for `\int_zero:N` and `\int_gzero:N`. These functions are documented on page 92.)

`\int_zero_new:N` Create a register if needed, otherwise clear it.

`\int_zero_new:c` 8191 `\cs_new_protected:Npn \int_zero_new:N #1`

`\int_gzero_new:N` 8192 `{ \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }`

`\int_gzero_new:c` 8193 `\cs_new_protected:Npn \int_gzero_new:N #1`
8194 `{ \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }`
8195 `\cs_generate_variant:Nn \int_zero_new:N { c }`
8196 `\cs_generate_variant:Nn \int_gzero_new:N { c }`

(End definition for `\int_zero_new:N` and `\int_gzero_new:N`. These functions are documented on page 92.)

`\int_set_eq:NN` Setting equal means using one integer inside the set function of another. Check that assigned integer is local/global. No need to check that the other one is defined as `\TeX` does it for us.

```

\int_set_eq:cN
\int_set_eq:Nc
\int_set_eq:cc
\int_gset_eq:NN
\int_gset_eq:cN
\int_gset_eq:Nc
\int_gset_eq:cc

```

8197 `\cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }`
8198 `\cs_generate_variant:Nn \int_set_eq:NN { c , Nc , cc }`
8199 `\cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }`
8200 `\cs_generate_variant:Nn \int_gset_eq:NN { c , Nc , cc }`

(End definition for `\int_set_eq:NN` and `\int_gset_eq:NN`. These functions are documented on page 92.)

`\int_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

\int_if_exist_p:c
\int_if_exist:NTF
\int_if_exist:cTF

```

8201 `\prg_new_eq_conditional:NNn \int_if_exist:N \cs_if_exist:N`
8202 `{ TF , T , F , p }`
8203 `\prg_new_eq_conditional:NNn \int_if_exist:c \cs_if_exist:c`
8204 `{ TF , T , F , p }`

(End definition for `\int_if_exist:NTF`. This function is documented on page 92.)

12.3 Setting and incrementing integers

`\int_add:Nn` Adding and subtracting to and from a counter.

```

\int_add:cn
\int_gadd:Nn
\int_gadd:cn
\int_sub:Nn
\int_sub:cn
\int_gsub:Nn
\int_gsub:cn

```

8205 `\cs_new_protected:Npn \int_add:Nn #1#2`
8206 `{ \tex_advance:D #1 by __int_eval:w #2 __int_eval_end: }`
8207 `\cs_new_protected:Npn \int_sub:Nn #1#2`
8208 `{ \tex_advance:D #1 by - __int_eval:w #2 __int_eval_end: }`
8209 `\cs_new_protected:Npn \int_gadd:Nn #1#2`
8210 `{ \tex_global:D \tex_advance:D #1 by __int_eval:w #2 __int_eval_end: }`
8211 `\cs_new_protected:Npn \int_gsub:Nn #1#2`
8212 `{ \tex_global:D \tex_advance:D #1 by - __int_eval:w #2 __int_eval_end: }`
8213 `\cs_generate_variant:Nn \int_add:Nn { c }`
8214 `\cs_generate_variant:Nn \int_gadd:Nn { c }`
8215 `\cs_generate_variant:Nn \int_sub:Nn { c }`
8216 `\cs_generate_variant:Nn \int_gsub:Nn { c }`

(End definition for `\int_add:Nn` and others. These functions are documented on page 92.)

`\int_incr:N` Incrementing and decrementing of integer registers is done with the following functions.

```

\int_incr:c
\int_gincr:N
\int_gincr:c
\int_decr:N
\int_decr:c
\int_gdecr:N
\int_gdecr:c

```

8217 `\cs_new_protected:Npn \int_incr:N #1`
8218 `{ \tex_advance:D #1 \c_one_int }`
8219 `\cs_new_protected:Npn \int_decr:N #1`
8220 `{ \tex_advance:D #1 - \c_one_int }`
8221 `\cs_new_protected:Npn \int_gincr:N #1`
8222 `{ \tex_global:D \tex_advance:D #1 \c_one_int }`
8223 `\cs_new_protected:Npn \int_gdecr:N #1`
8224 `{ \tex_global:D \tex_advance:D #1 - \c_one_int }`
8225 `\cs_generate_variant:Nn \int_incr:N { c }`
8226 `\cs_generate_variant:Nn \int_decr:N { c }`
8227 `\cs_generate_variant:Nn \int_gincr:N { c }`
8228 `\cs_generate_variant:Nn \int_gdecr:N { c }`

(End definition for `\int_incr:N` and others. These functions are documented on page 92.)

`\int_set:Nn` As integers are register-based T_EX issues an error if they are not defined.

```

\int_set:cn      8229 \cs_new_protected:Npn \int_set:Nn #1#2
\int_gset:Nn      8230 { #1 ~ \__int_eval:w #2 \__int_eval_end: }
\int_gset:cn      8231 \cs_new_protected:Npn \int_gset:Nn #1#2
                  8232 { \tex_global:D #1 ~ \__int_eval:w #2 \__int_eval_end: }
                  8233 \cs_generate_variant:Nn \int_set:Nn { c }
                  8234 \cs_generate_variant:Nn \int_gset:Nn { c }

```

(End definition for `\int_set:Nn` and `\int_gset:Nn`. These functions are documented on page 93.)

12.4 Using integers

`\int_use:N` Here is how counters are accessed:

```

\int_use:c      8235 \cs_new_eq:NN \int_use:N \tex_the:D
We hand-code this for some speed gain:
                8236 %\cs_generate_variant:Nn \int_use:N { c }
                8237 \cs_new:Npn \int_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }

```

(End definition for `\int_use:N`. This function is documented on page 93.)

12.5 Integer expression conditionals

`__int_compare_error:` Those functions are used for comparison tests which use a simple syntax where only one set of braces is required and additional operators such as `!=` and `>=` are supported. `__int_compare_error:Nw` The tests first evaluate their left-hand side, with a trailing `__int_compare_error:.` This marker is normally not expanded, but if the relation symbol is missing from the test's argument, then the marker inserts `=` (and itself) after triggering the relevant T_EX error. If the first token which appears after evaluating and removing the left-hand side is not a known relation symbol, then a judiciously placed `__int_compare_error:Nw` gets expanded, cleaning up the end of the test and telling the user what the problem was.

```

8238 \cs_new_protected:Npn \__int_compare_error:
8239 {
8240   \if_int_compare:w \c_zero_int \c_zero_int \fi:
8241   =
8242   \__int_compare_error:
8243 }
8244 \cs_new:Npn \__int_compare_error:Nw
8245 #1#2 \s__int_stop
8246 {
8247   { }
8248   \c_zero_int \fi:
8249   \__kernel_msg_expandable_error:nnn
8250   { kernel } { unknown-comparison } {#1}
8251   \prg_return_false:
8252 }

```

(End definition for `__int_compare_error:` and `__int_compare_error:Nw`.)

`\int_compare_p:n` Comparison tests using a simple syntax where only one set of braces is required, additional
`\int_compare:nTF` operators such as `!=` and `>=` are supported, and multiple comparisons can be performed
`__int_compare:w` at once, for instance `0 < 5 <= 1`. The idea is to loop through the argument, finding one
`__int_compare:Nw` operand at a time, and comparing it to the previous one. The looping auxiliary `__-`
`__int_compare:NNw` `int_compare:Nw` reads one *operand* and one *comparison* symbol, and leaves roughly
`__int_compare:nnN`

```

\__int_compare_end=:NNw
\__int_compare=:NNw
\__int_compare<:NNw
\__int_compare>:NNw
\__int_compare==:NNw
\__int_compare!=:NNw
\__int_compare<=:NNw
\__int_compare>=:NNw

```

```

\operand\prg_return_false:\fi:
\reverse_if:N\if_int_compare:w\operand\comparison
\__int_compare:Nw

```

in the input stream. Each call to this auxiliary provides the second operand of the last call's `\if_int_compare:w`. If one of the *comparisons* is `false`, the `true` branch of the `TEX` conditional is taken (because of `\reverse_if:N`), immediately returning `false` as the result of the test. There is no `TEX` conditional waiting the first operand, so we add an `\if_false:` and expand by hand with `\int_value:w`, thus skipping `\prg_return_false:` on the first iteration.

Before starting the loop, the first step is to make sure that there is at least one relation symbol. We first let `TEX` evaluate this left hand side of the (in)equality using `__int_eval:w`. Since the relation symbols `<`, `>`, `=` and `!` are not allowed in integer expressions, they would terminate the expression. If the argument contains no relation symbol, `__int_compare_error:` is expanded, inserting `=` and itself after an error. In all cases, `__int_compare:w` receives as its argument an integer, a relation symbol, and some more tokens. We then setup the loop, which is ended by the two odd-looking items `e` and `{=nd_}`, with a trailing `\s__int_stop` used to grab the entire argument when necessary.

```

8253 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
8254 {
8255     \exp_after:wN \__int_compare:w
8256     \int_value:w \__int_eval:w #1 \__int_compare_error:
8257 }
8258 \cs_new:Npn \__int_compare:w #1 \__int_compare_error:
8259 {
8260     \exp_after:wN \if_false: \int_value:w
8261     \__int_compare:Nw #1 e { = nd_ } \s__int_stop
8262 }

```

The goal here is to find an *operand* and a *comparison*. The *operand* is already evaluated, but we cannot yet grab it as an argument. To access the following relation symbol, we remove the number by applying `__int_to_roman:w`, after making sure that the argument becomes non-positive: its roman numeral representation is then empty. Then probe the first two tokens with `__int_compare:NNw` to determine the relation symbol, building a control sequence from it (`\token_to_str:N` gives better errors if `#1` is not a character). All the extended forms have an extra `=` hence the test for that as a second token. If the relation symbol is unknown, then the control sequence is turned by `TEX` into `\scan_stop:`, ignored thanks to `\unexpanded`, and `__int_compare_error:Nw` raises an error.

```

8263 \cs_new:Npn \__int_compare:Nw #1#2 \s__int_stop
8264 {
8265     \exp_after:wN \__int_compare:NNw
8266     \__int_to_roman:w - 0 #2 \s__int_mark
8267     #1#2 \s__int_stop
8268 }
8269 \cs_new:Npn \__int_compare:NNw #1#2#3 \s__int_mark
8270 {
8271     \__kernel_exp_not:w
8272     \use:c
8273     {
8274         __int_compare_ \token_to_str:N #1

```

```

8275         \if_meaning:w = #2 = \fi:
8276         :NNw
8277     }
8278     \__int_compare_error:Nw #1
8279 }

```

When the last *<operand>* is seen, `__int_compare:NNw` receives `e` and `=nd_` as arguments, hence calling `__int_compare_end_=:NNw` to end the loop: return the result of the last comparison (involving the operand that we just found). When a normal relation is found, the appropriate auxiliary calls `__int_compare:nnN` where `#1` is `\if_int_compare:w` or `\reverse_if:N \if_int_compare:w`, `#2` is the *<operand>*, and `#3` is one of `<`, `=`, or `>`. As announced earlier, we leave the *<operand>* for the previous conditional. If this conditional is true the result of the test is known, so we remove all tokens and return **false**. Otherwise, we apply the conditional `#1` to the *<operand>* `#2` and the comparison `#3`, and call `__int_compare:Nw` to look for additional operands, after evaluating the following expression.

```

8280 \cs_new:cpn { __int_compare_end_=:NNw } #1#2#3 e #4 \s__int_stop
8281 {
8282     {#3} \exp_stop_f:
8283     \prg_return_false: \else: \prg_return_true: \fi:
8284 }
8285 \cs_new:Npn \__int_compare:nnN #1#2#3
8286 {
8287     {#2} \exp_stop_f:
8288     \prg_return_false: \exp_after:wN \__int_use_none_delimit_by_s_stop:w
8289     \fi:
8290     #1 #2 #3 \exp_after:wN \__int_compare:Nw \int_value:w \__int_eval:w
8291 }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding `__int_compare_error:Nw` *<token>* responsible for error detection.

```

8292 \cs_new:cpn { __int_compare=:NNw } #1#2#3 =
8293 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
8294 \cs_new:cpn { __int_compare:<:NNw } #1#2#3 <
8295 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
8296 \cs_new:cpn { __int_compare:>:NNw } #1#2#3 >
8297 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }
8298 \cs_new:cpn { __int_compare=:NNw } #1#2#3 ==
8299 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
8300 \cs_new:cpn { __int_compare!=:NNw } #1#2#3 !=
8301 { \__int_compare:nnN { \if_int_compare:w } {#3} = }
8302 \cs_new:cpn { __int_compare<=:NNw } #1#2#3 <=
8303 { \__int_compare:nnN { \if_int_compare:w } {#3} > }
8304 \cs_new:cpn { __int_compare_>=:NNw } #1#2#3 >=
8305 { \__int_compare:nnN { \if_int_compare:w } {#3} < }

```

(End definition for `\int_compare:nTF` and others. This function is documented on page 94.)

`\int_compare_p:nNn`
`\int_compare:nNnTF`

More efficient but less natural in typing.

```

8306 \prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
8307 {
8308     \if_int_compare:w \__int_eval:w #1 #2 \__int_eval:w #3 \__int_eval_end:
8309     \prg_return_true:

```

```

8310     \else:
8311         \prg_return_false:
8312     \fi:
8313 }

```

(End definition for \int_compare:nNnTF. This function is documented on page 93.)

```

\int_case:nn For integer cases, the first task to fully expand the check condition. The over all idea is
\int_case:nnTF then much the same as for \tl_case:nn(TF) as described in l3tl.
\__int_case:nnTF
\__int_case:nw
\__int_case_end:nw
8314 \cs_new:Npn \int_case:nnTF #1
8315 {
8316     \exp:w
8317     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
8318 }
8319 \cs_new:Npn \int_case:nnT #1#2#3
8320 {
8321     \exp:w
8322     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
8323 }
8324 \cs_new:Npn \int_case:nnF #1#2
8325 {
8326     \exp:w
8327     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
8328 }
8329 \cs_new:Npn \int_case:nn #1#2
8330 {
8331     \exp:w
8332     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
8333 }
8334 \cs_new:Npn \__int_case:nnTF #1#2#3#4
8335 { \__int_case:nw {#1} #2 {#1} { } \s__int_mark {#3} \s__int_mark {#4} \s__int_stop }
8336 \cs_new:Npn \__int_case:nw #1#2#3
8337 {
8338     \int_compare:nNnTF {#1} = {#2}
8339     { \__int_case_end:nw {#3} }
8340     { \__int_case:nw {#1} }
8341 }
8342 \cs_new:Npn \__int_case_end:nw #1#2#3 \s__int_mark #4#5 \s__int_stop
8343 { \exp_end: #1 #4 }

```

(End definition for \int_case:nnTF and others. This function is documented on page 95.)

```

\int_if_odd_p:n A predicate function.
\int_if_odd:nTF
\int_if_even_p:n
\int_if_even:nTF
8344 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
8345 {
8346     \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
8347     \prg_return_true:
8348     \else:
8349     \prg_return_false:
8350     \fi:
8351 }
8352 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
8353 {
8354     \reverse_if:N \if_int_odd:w \__int_eval:w #1 \__int_eval_end:

```

```

8355     \prg_return_true:
8356 \else:
8357     \prg_return_false:
8358 \fi:
8359 }

```

(End definition for `\int_if_odd:nTF` and `\int_if_even:nTF`. These functions are documented on page 95.)

12.6 Integer expression loops

`\int_while_do:nn` These are quite easy given the above functions. The `while` versions test first and then execute the body. The `do_while` does it the other way round.

```

\int_until_do:nn
\int_do_while:nn
\int_do_until:nn
8360 \cs_new:Npn \int_while_do:nn #1#2
8361 {
8362     \int_compare:nT {#1}
8363     {
8364         #2
8365         \int_while_do:nn {#1} {#2}
8366     }
8367 }
8368 \cs_new:Npn \int_until_do:nn #1#2
8369 {
8370     \int_compare:nF {#1}
8371     {
8372         #2
8373         \int_until_do:nn {#1} {#2}
8374     }
8375 }
8376 \cs_new:Npn \int_do_while:nn #1#2
8377 {
8378     #2
8379     \int_compare:nT {#1}
8380     { \int_do_while:nn {#1} {#2} }
8381 }
8382 \cs_new:Npn \int_do_until:nn #1#2
8383 {
8384     #2
8385     \int_compare:nF {#1}
8386     { \int_do_until:nn {#1} {#2} }
8387 }

```

(End definition for `\int_while_do:nn` and others. These functions are documented on page 96.)

`\int_while_do:nNnn` As above but not using the more natural syntax.

```

\int_until_do:nNnn
\int_do_while:nNnn
\int_do_until:nNnn
8388 \cs_new:Npn \int_while_do:nNnn #1#2#3#4
8389 {
8390     \int_compare:nNnT {#1} #2 {#3}
8391     {
8392         #4
8393         \int_while_do:nNnn {#1} #2 {#3} {#4}
8394     }
8395 }
8396 \cs_new:Npn \int_until_do:nNnn #1#2#3#4

```

```

8397 {
8398     \int_compare:nNnF {#1} #2 {#3}
8399     {
8400         #4
8401         \int_until_do:nNnn {#1} #2 {#3} {#4}
8402     }
8403 }
8404 \cs_new:Npn \int_do_while:nNnn #1#2#3#4
8405 {
8406     #4
8407     \int_compare:nNnT {#1} #2 {#3}
8408     { \int_do_while:nNnn {#1} #2 {#3} {#4} }
8409 }
8410 \cs_new:Npn \int_do_until:nNnn #1#2#3#4
8411 {
8412     #4
8413     \int_compare:nNnF {#1} #2 {#3}
8414     { \int_do_until:nNnn {#1} #2 {#3} {#4} }
8415 }

```

(End definition for `\int_while_do:nNnn` and others. These functions are documented on page 96.)

12.7 Integer step functions

`\int_step_function:nnnN` Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```

8416 \cs_new:Npn \int_step_function:nnnN #1#2#3
8417 {
8418     \exp_after:wN \__int_step:wwwN
8419     \int_value:w \__int_eval:w #1 \exp_after:wN ;
8420     \int_value:w \__int_eval:w #2 \exp_after:wN ;
8421     \int_value:w \__int_eval:w #3 ;
8422 }
8423 \cs_new:Npn \__int_step:wwwN #1; #2; #3; #4
8424 {
8425     \int_compare:nNnTF {#2} > \c_zero_int
8426     { \__int_step:NwnnnN > }
8427     {
8428         \int_compare:nNnTF {#2} = \c_zero_int
8429         {
8430             \__kernel_msg_expandable_error:nnn
8431             { kernel } { zero-step } {#4}
8432             \prg_break:
8433         }
8434         { \__int_step:NwnnnN < }
8435     }
8436     #1 ; {#2} {#3} #4
8437     \prg_break_point:
8438 }
8439 \cs_new:Npn \__int_step:NwnnnN #1#2 ; #3#4#5
8440 {

```

```

8441 \if_int_compare:w #2 #1 #4 \exp_stop_f:
8442 \prg_break:n
8443 \fi:
8444 #5 {#2}
8445 \exp_after:wN \__int_step:NwnnN
8446 \exp_after:wN #1
8447 \int_value:w \__int_eval:w #2 + #3 ; {#3} {#4} #5
8448 }
8449 \cs_new:Npn \int_step_function:nN
8450 { \int_step_function:nnnN { 1 } { 1 } }
8451 \cs_new:Npn \int_step_function:nnN #1
8452 { \int_step_function:nnnN {#1} { 1 } }

```

(End definition for `\int_step_function:nnnN` and others. These functions are documented on page 97.)

`\int_step_inline:nn` The approach here is to build a function, with a global integer required to make the
`\int_step_inline:nnn` nesting safe (as seen in other in line functions), and map that function using `\int_-`
`\int_step_inline:nnnn` `step_function:nnnN`. We put a `\prg_break_point:Nn` so that `map_break` functions
`\int_step_variable:nNn` from other modules correctly decrement `\g__kernel_pr_g_map_int` before looking for
`\int_step_variable:nnNn` their own break point. The first argument is `\scan_stop:`, so that no breaking function
`\int_step_variable:nnnnNn` recognizes this break point as its own.

```

\__int_step:NNnnnn
8453 \cs_new_protected:Npn \int_step_inline:nn
8454 { \int_step_inline:nnnn { 1 } { 1 } }
8455 \cs_new_protected:Npn \int_step_inline:nnn #1
8456 { \int_step_inline:nnnn {#1} { 1 } }
8457 \cs_new_protected:Npn \int_step_inline:nnnn
8458 {
8459 \int_gincr:N \g__kernel_pr_g_map_int
8460 \exp_args:NNc \__int_step:NNnnnn
8461 \cs_gset_protected:Npn
8462 { \__int_map_ \int_use:N \g__kernel_pr_g_map_int :w }
8463 }
8464 \cs_new_protected:Npn \int_step_variable:nNn
8465 { \int_step_variable:nnnNn { 1 } { 1 } }
8466 \cs_new_protected:Npn \int_step_variable:nnNn #1
8467 { \int_step_variable:nnnNn {#1} { 1 } }
8468 \cs_new_protected:Npn \int_step_variable:nnnnNn #1#2#3#4#5
8469 {
8470 \int_gincr:N \g__kernel_pr_g_map_int
8471 \exp_args:NNc \__int_step:NNnnnn
8472 \cs_gset_protected:Npx
8473 { \__int_map_ \int_use:N \g__kernel_pr_g_map_int :w }
8474 {#1}{#2}{#3}
8475 {
8476 \tl_set:Nn \exp_not:N #4 {##1}
8477 \exp_not:n {#5}
8478 }
8479 }
8480 \cs_new_protected:Npn \__int_step:NNnnnn #1#2#3#4#5#6
8481 {
8482 #1 #2 ##1 {#6}
8483 \int_step_function:nnnN {#3} {#4} {#5} #2
8484 \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_pr_g_map_int }
8485 }

```

(End definition for `\int_step_inline:nn` and others. These functions are documented on page 97.)

12.8 Formatting integers

`\int_to_arabic:n` Nothing exciting here.

```
8486 \cs_new_eq:NN \int_to_arabic:n \int_eval:n
```

(End definition for `\int_to_arabic:n`. This function is documented on page 98.)

`\int_to_symbols:nnn` For conversion of integers to arbitrary symbols the method is in general as follows. The input number (#1) is compared to the total number of symbols available at each place (#2). If the input is larger than the total number of symbols available then the modulus is needed, with one added so that the positions don't have to number from zero. Using an `f`-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a 'nice' number at each stage. Of course, if the initial input was small enough then there is no problem and everything is easy.

```
8487 \cs_new:Npn \int_to_symbols:nnn #1#2#3
8488 {
8489   \int_compare:nNnTF {#1} > {#2}
8490   {
8491     \exp_args:NNo \exp_args:No \__int_to_symbols:nnnn
8492     {
8493       \int_case:nn
8494       { 1 + \int_mod:nn { #1 - 1 } {#2} }
8495       {#3}
8496     }
8497     {#1} {#2} {#3}
8498   }
8499   { \int_case:nn {#1} {#3} }
8500 }
8501 \cs_new:Npn \__int_to_symbols:nnnn #1#2#3#4
8502 {
8503   \exp_args:Nf \int_to_symbols:nnn
8504   { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}
8505   #1
8506 }
```

(End definition for `\int_to_symbols:nnn` and `__int_to_symbols:nnnn`. This function is documented on page 98.)

`\int_to_alph:n` These both use the above function with input functions that make sense for the alphabet
`\int_to_Alph:n` in English.

```
8507 \cs_new:Npn \int_to_alph:n #1
8508 {
8509   \int_to_symbols:nnn {#1} { 26 }
8510   {
8511     { 1 } { a }
8512     { 2 } { b }
8513     { 3 } { c }
8514     { 4 } { d }
8515     { 5 } { e }
8516     { 6 } { f }
8517     { 7 } { g }
```



```

8518         { 8 } { h }
8519         { 9 } { i }
8520         { 10 } { j }
8521         { 11 } { k }
8522         { 12 } { l }
8523         { 13 } { m }
8524         { 14 } { n }
8525         { 15 } { o }
8526         { 16 } { p }
8527         { 17 } { q }
8528         { 18 } { r }
8529         { 19 } { s }
8530         { 20 } { t }
8531         { 21 } { u }
8532         { 22 } { v }
8533         { 23 } { w }
8534         { 24 } { x }
8535         { 25 } { y }
8536         { 26 } { z }
8537     }
8538 }
8539 \cs_new:Npn \int_to_Alph:n #1
8540 {
8541     \int_to_symbols:nnn {#1} { 26 }
8542     {
8543         { 1 } { A }
8544         { 2 } { B }
8545         { 3 } { C }
8546         { 4 } { D }
8547         { 5 } { E }
8548         { 6 } { F }
8549         { 7 } { G }
8550         { 8 } { H }
8551         { 9 } { I }
8552         { 10 } { J }
8553         { 11 } { K }
8554         { 12 } { L }
8555         { 13 } { M }
8556         { 14 } { N }
8557         { 15 } { O }
8558         { 16 } { P }
8559         { 17 } { Q }
8560         { 18 } { R }
8561         { 19 } { S }
8562         { 20 } { T }
8563         { 21 } { U }
8564         { 22 } { V }
8565         { 23 } { W }
8566         { 24 } { X }
8567         { 25 } { Y }
8568         { 26 } { Z }
8569     }
8570 }

```

(End definition for `\int_to_alpha:n` and `\int_to_Alph:n`. These functions are documented on page 98.)

`\int_to_base:nn` Converting from base ten (#1) to a second base (#2) starts with computing #1: if it is
`\int_to_Base:nn` a complicated calculation, we shouldn't perform it twice. Then check the sign, store it,
`__int_to_base:nn` either - or `\c_empty_tl`, and feed the absolute value to the next auxiliary function.
`__int_to_Base:nn` 8571 `\cs_new:Npn \int_to_base:nn #1`
`__int_to_base:nnN` 8572 `{ \exp_args:Nf __int_to_base:nn { \int_eval:n {#1} } }`
`__int_to_Base:nnN` 8573 `\cs_new:Npn \int_to_Base:nn #1`
`__int_to_base:nnnN` 8574 `{ \exp_args:Nf __int_to_Base:nn { \int_eval:n {#1} } }`
`__int_to_Base:nnnN` 8575 `\cs_new:Npn __int_to_base:nn #1#2`
`__int_to_letter:n` 8576 `{`
`__int_to_Letter:n` 8577 `\int_compare:nNnTF {#1} < 0`
8578 `{ \exp_args:No __int_to_base:nnN { \use_none:n #1 } {#2} - }`
8579 `{ __int_to_base:nnN {#1} {#2} \c_empty_tl }`
8580 `}`
8581 `\cs_new:Npn __int_to_Base:nn #1#2`
8582 `{`
8583 `\int_compare:nNnTF {#1} < 0`
8584 `{ \exp_args:No __int_to_Base:nnN { \use_none:n #1 } {#2} - }`
8585 `{ __int_to_Base:nnN {#1} {#2} \c_empty_tl }`
8586 `}`

Here, the idea is to provide a recursive system to deal with the input. The output is built up after the end of the function. At each pass, the value in #1 is checked to see if it is less than the new base (#2). If it is, then it is converted directly, putting the sign back in front. On the other hand, if the value to convert is greater than or equal to the new base then the modulus and remainder values are found. The modulus is converted to a symbol and put on the right, and the remainder is carried forward to the next round.

8587 `\cs_new:Npn __int_to_base:nnN #1#2#3`
8588 `{`
8589 `\int_compare:nNnTF {#1} < {#2}`
8590 `{ \exp_last_unbraced:Nf #3 { __int_to_letter:n {#1} } }`
8591 `{`
8592 `\exp_args:Nf __int_to_base:nnnN`
8593 `{ __int_to_letter:n { \int_mod:nn {#1} {#2} } }`
8594 `{#1}`
8595 `{#2}`
8596 `#3`
8597 `}`
8598 `}`
8599 `\cs_new:Npn __int_to_base:nnnN #1#2#3#4`
8600 `{`
8601 `\exp_args:Nf __int_to_base:nnN`
8602 `{ \int_div_truncate:nn {#2} {#3} }`
8603 `{#3}`
8604 `#4`
8605 `#1`
8606 `}`
8607 `\cs_new:Npn __int_to_Base:nnN #1#2#3`
8608 `{`
8609 `\int_compare:nNnTF {#1} < {#2}`
8610 `{ \exp_last_unbraced:Nf #3 { __int_to_Letter:n {#1} } }`
8611 `{`
8612 `\exp_args:Nf __int_to_Base:nnnN`
8613 `{ __int_to_Letter:n { \int_mod:nn {#1} {#2} } }`
8614 `{#1}`

```

8615         {#2}
8616         #3
8617     }
8618 }
8619 \cs_new:Npn \__int_to_Base:nnnN #1#2#3#4
8620 {
8621     \exp_args:Nf \__int_to_Base:nnN
8622     { \int_div_truncate:nn {#2} {#3} }
8623     {#3}
8624     #4
8625     #1
8626 }

```

Convert to a letter only if necessary, otherwise simply return the value unchanged. It would be cleaner to use `\int_case:nn`, but in our case, the cases are contiguous, so it is forty times faster to use the `\if_case:w` primitive. The first `\exp_after:wN` expands the conditional, jumping to the correct case, the second one expands after the resulting character to close the conditional. Since `#1` might be an expression, and not directly a single digit, we need to evaluate it properly, and expand the trailing `\fi:`.

```

8627 \cs_new:Npn \__int_to_letter:n #1
8628 {
8629     \exp_after:wN \exp_after:wN
8630     \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
8631         a
8632     \or: b
8633     \or: c
8634     \or: d
8635     \or: e
8636     \or: f
8637     \or: g
8638     \or: h
8639     \or: i
8640     \or: j
8641     \or: k
8642     \or: l
8643     \or: m
8644     \or: n
8645     \or: o
8646     \or: p
8647     \or: q
8648     \or: r
8649     \or: s
8650     \or: t
8651     \or: u
8652     \or: v
8653     \or: w
8654     \or: x
8655     \or: y
8656     \or: z
8657     \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
8658     \fi:
8659 }
8660 \cs_new:Npn \__int_to_Letter:n #1
8661 {

```

```

8662     \exp_after:wN \exp_after:wN
8663     \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
8664         A
8665     \or: B
8666     \or: C
8667     \or: D
8668     \or: E
8669     \or: F
8670     \or: G
8671     \or: H
8672     \or: I
8673     \or: J
8674     \or: K
8675     \or: L
8676     \or: M
8677     \or: N
8678     \or: O
8679     \or: P
8680     \or: Q
8681     \or: R
8682     \or: S
8683     \or: T
8684     \or: U
8685     \or: V
8686     \or: W
8687     \or: X
8688     \or: Y
8689     \or: Z
8690     \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
8691     \fi:
8692 }

```

(End definition for `\int_to_base:nn` and others. These functions are documented on page 99.)

```

\int_to_bin:n  Wrappers around the generic function.
\int_to_hex:n  8693 \cs_new:Npn \int_to_bin:n #1
\int_to_Hex:n  8694 { \int_to_base:nn {#1} { 2 } }
\int_to_oct:n  8695 \cs_new:Npn \int_to_hex:n #1
                8696 { \int_to_base:nn {#1} { 16 } }
                8697 \cs_new:Npn \int_to_Hex:n #1
                8698 { \int_to_Base:nn {#1} { 16 } }
                8699 \cs_new:Npn \int_to_oct:n #1
                8700 { \int_to_base:nn {#1} { 8 } }

```

(End definition for `\int_to_bin:n` and others. These functions are documented on page 99.)

```

\int_to_roman:n The \__int_to_roman:w primitive creates tokens of category code 12 (other). Usually,
\int_to_Roman:n what is actually wanted is letters. The approach here is to convert the output of the
                  primitive into letters using appropriate control sequence names. That keeps everything
\__int_to_roman:N expandable. The loop is terminated by the conversion of the Q.
\__int_to_roman:N
\__int_to_roman_i:w 8701 \cs_new:Npn \int_to_roman:n #1
\__int_to_roman_v:w 8702 {
\__int_to_roman_x:w 8703     \exp_after:wN \__int_to_roman:N
\__int_to_roman_l:w 8704     \__int_to_roman:w \int_eval:n {#1} Q
\__int_to_roman_c:w
\__int_to_roman_d:w
\__int_to_roman_m:w
\__int_to_roman_Q:w
\__int_to_Roman_i:w
\__int_to_Roman_v:w
\__int_to_Roman_x:w
\__int_to_Roman_l:w
\__int_to_Roman_c:w
\__int_to_Roman_d:w

```

```

8705 }
8706 \cs_new:Npn \__int_to_roman:N #1
8707 {
8708   \use:c { __int_to_roman_ #1 :w }
8709   \__int_to_roman:N
8710 }
8711 \cs_new:Npn \int_to_Roman:n #1
8712 {
8713   \exp_after:wN \__int_to_Roman_aux:N
8714   \__int_to_roman:w \int_eval:n {#1} Q
8715 }
8716 \cs_new:Npn \__int_to_Roman_aux:N #1
8717 {
8718   \use:c { __int_to_Roman_ #1 :w }
8719   \__int_to_Roman_aux:N
8720 }
8721 \cs_new:Npn \__int_to_roman_i:w { i }
8722 \cs_new:Npn \__int_to_roman_v:w { v }
8723 \cs_new:Npn \__int_to_roman_x:w { x }
8724 \cs_new:Npn \__int_to_roman_l:w { l }
8725 \cs_new:Npn \__int_to_roman_c:w { c }
8726 \cs_new:Npn \__int_to_roman_d:w { d }
8727 \cs_new:Npn \__int_to_roman_m:w { m }
8728 \cs_new:Npn \__int_to_roman_Q:w #1 { }
8729 \cs_new:Npn \__int_to_Roman_i:w { I }
8730 \cs_new:Npn \__int_to_Roman_v:w { V }
8731 \cs_new:Npn \__int_to_Roman_x:w { X }
8732 \cs_new:Npn \__int_to_Roman_l:w { L }
8733 \cs_new:Npn \__int_to_Roman_c:w { C }
8734 \cs_new:Npn \__int_to_Roman_d:w { D }
8735 \cs_new:Npn \__int_to_Roman_m:w { M }
8736 \cs_new:Npn \__int_to_Roman_Q:w #1 { }

```

(End definition for `\int_to_roman:n` and others. These functions are documented on page 99.)

12.9 Converting from other formats to integers

`__int_pass_signs:wn` Called as `__int_pass_signs:wn <signs and digits> \s__int_stop {<code>}`, this function leaves in the input stream any sign it finds, then inserts the `<code>` before the first non-sign token (and removes `\s__int_stop`). More precisely, it deletes any + and passes any - to the input stream, hence should be called in an integer expression.

```

8737 \cs_new:Npn \__int_pass_signs:wn #1
8738 {
8739   \if:w + \if:w - \exp_not:N #1 + \fi: \exp_not:N #1
8740   \exp_after:wN \__int_pass_signs:wn
8741   \else:
8742     \exp_after:wN \__int_pass_signs_end:wn
8743     \exp_after:wN #1
8744   \fi:
8745 }
8746 \cs_new:Npn \__int_pass_signs_end:wn #1 \s__int_stop #2 { #2 #1 }

```

(End definition for `__int_pass_signs:wn` and `__int_pass_signs_end:wn`.)

`\int_from_alph:n` First take care of signs then loop through the input using the `recursion` quarks. The `__int_from_alph:nN` auxiliary collects in its first argument the value obtained so far, and the auxiliary `__int_from_alph:N` converts one letter to an expression which evaluates to the correct number.

```

8747 \cs_new:Npn \int_from_alph:n #1
8748 {
8749   \int_eval:n
8750   {
8751     \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
8752     \s__int_stop { \__int_from_alph:nN { 0 } }
8753     \q__int_recursion_tail \q__int_recursion_stop
8754   }
8755 }
8756 \cs_new:Npn \__int_from_alph:nN #1#2
8757 {
8758   \__int_if_recursion_tail_stop_do:Nn #2 {#1}
8759   \exp_args:Nf \__int_from_alph:nN
8760   { \int_eval:n { #1 * 26 + \__int_from_alph:N #2 } }
8761 }
8762 \cs_new:Npn \__int_from_alph:N #1
8763 { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } }

```

(End definition for `\int_from_alph:n`, `__int_from_alph:nN`, and `__int_from_alph:N`. This function is documented on page 99.)

`\int_from_base:nn` Leave the signs into the integer expression, then loop through characters, collecting the value found so far in the first argument of `__int_from_base:nnN`. To convert a single character, `__int_from_base:N` checks first for digits, then distinguishes lower from upper case letters, turning them into the appropriate number. Note that this auxiliary does not use `\int_eval:n`, hence is not safe for general use.

```

8764 \cs_new:Npn \int_from_base:nn #1#2
8765 {
8766   \int_eval:n
8767   {
8768     \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
8769     \s__int_stop { \__int_from_base:nnN { 0 } {#2} }
8770     \q__int_recursion_tail \q__int_recursion_stop
8771   }
8772 }
8773 \cs_new:Npn \__int_from_base:nnN #1#2#3
8774 {
8775   \__int_if_recursion_tail_stop_do:Nn #3 {#1}
8776   \exp_args:Nf \__int_from_base:nnN
8777   { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
8778   {#2}
8779 }
8780 \cs_new:Npn \__int_from_base:N #1
8781 {
8782   \int_compare:nNnTF { '#1 } < { 58 }
8783   {#1}
8784   { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
8785 }

```

(End definition for `\int_from_base:nn`, `__int_from_base:nnN`, and `__int_from_base:N`. This function is documented on page 100.)

`\int_from_bin:n` Wrappers around the generic function.

```

\int_from_hex:n 8786 \cs_new:Npn \int_from_bin:n #1
\int_from_oct:n 8787 { \int_from_base:nn {#1} { 2 } }
8788 \cs_new:Npn \int_from_hex:n #1
8789 { \int_from_base:nn {#1} { 16 } }
8790 \cs_new:Npn \int_from_oct:n #1
8791 { \int_from_base:nn {#1} { 8 } }

```

(End definition for `\int_from_bin:n`, `\int_from_hex:n`, and `\int_from_oct:n`. These functions are documented on page 100.)

`\c_int_from_roman_i_int` Constants used to convert from Roman numerals to integers.

```

\c_int_from_roman_v_int 8792 \int_const:cn { c__int_from_roman_i_int } { 1 }
\c_int_from_roman_x_int 8793 \int_const:cn { c__int_from_roman_v_int } { 5 }
\c_int_from_roman_l_int 8794 \int_const:cn { c__int_from_roman_x_int } { 10 }
\c_int_from_roman_c_int 8795 \int_const:cn { c__int_from_roman_l_int } { 50 }
\c_int_from_roman_d_int 8796 \int_const:cn { c__int_from_roman_c_int } { 100 }
\c_int_from_roman_m_int 8797 \int_const:cn { c__int_from_roman_d_int } { 500 }
\c_int_from_roman_I_int 8798 \int_const:cn { c__int_from_roman_m_int } { 1000 }
\c_int_from_roman_V_int 8799 \int_const:cn { c__int_from_roman_I_int } { 1 }
\c_int_from_roman_X_int 8800 \int_const:cn { c__int_from_roman_V_int } { 5 }
\c_int_from_roman_L_int 8801 \int_const:cn { c__int_from_roman_X_int } { 10 }
\c_int_from_roman_C_int 8802 \int_const:cn { c__int_from_roman_L_int } { 50 }
\c_int_from_roman_D_int 8803 \int_const:cn { c__int_from_roman_C_int } { 100 }
\c_int_from_roman_M_int 8804 \int_const:cn { c__int_from_roman_D_int } { 500 }
8805 \int_const:cn { c__int_from_roman_M_int } { 1000 }

```

(End definition for `\c_int_from_roman_i_int` and others.)

`\int_from_roman:n` The method here is to iterate through the input, finding the appropriate value for each letter and building up a sum. This is then evaluated by \TeX . If any unknown letter is found, skip to the closing parenthesis and insert `*0-1` afterwards, to replace the value by `-1`.

```

\__int_from_roman:NN
\__int_from_roman_error:w
8806 \cs_new:Npn \int_from_roman:n #1
8807 {
8808   \int_eval:n
8809   {
8810     (
8811       0
8812       \exp_after:wN \__int_from_roman:NN \tl_to_str:n {#1}
8813       \q__int_recursion_tail \q__int_recursion_tail \q__int_recursion_stop
8814     )
8815   }
8816 }
8817 \cs_new:Npn \__int_from_roman:NN #1#2
8818 {
8819   \__int_if_recursion_tail_stop:N #1
8820   \int_if_exist:cF { c__int_from_roman_ #1 _int }
8821   { \__int_from_roman_error:w }
8822   \__int_if_recursion_tail_stop_do:Nn #2
8823   { + \use:c { c__int_from_roman_ #1 _int } }
8824   \int_if_exist:cF { c__int_from_roman_ #2 _int }
8825   { \__int_from_roman_error:w }
8826   \int_compare:nNnTF

```

```

8827     { \use:c { c__int_from_roman_ #1 _int } }
8828     <
8829     { \use:c { c__int_from_roman_ #2 _int } }
8830     {
8831       + \use:c { c__int_from_roman_ #2 _int }
8832       - \use:c { c__int_from_roman_ #1 _int }
8833       \__int_from_roman:NN
8834     }
8835     {
8836       + \use:c { c__int_from_roman_ #1 _int }
8837       \__int_from_roman:NN #2
8838     }
8839   }
8840   \cs_new:Npn \__int_from_roman_error:w #1 \q__int_recursion_stop #2
8841     { #2 * 0 - 1 }

```

(End definition for `\int_from_roman:n`, `__int_from_roman:NN`, and `__int_from_roman_error:w`. This function is documented on page 100.)

12.10 Viewing integer

`\int_show:N` Diagnostics.
`\int_show:c` 8842 `\cs_new_eq:NN \int_show:N __kernel_register_show:N`
`__int_show:nN` 8843 `\cs_generate_variant:Nn \int_show:N { c }`

(End definition for `\int_show:N` and `__int_show:nN`. This function is documented on page 101.)

`\int_show:n` We don't use the TeX primitive `\showthe` to show integer expressions: this gives a more unified output.

```

8844 \cs_new_protected:Npn \int_show:n
8845   { \msg_show_eval:Nn \int_eval:n }

```

(End definition for `\int_show:n`. This function is documented on page 101.)

`\int_log:N` Diagnostics.
`\int_log:c` 8846 `\cs_new_eq:NN \int_log:N __kernel_register_log:N`
8847 `\cs_generate_variant:Nn \int_log:N { c }`

(End definition for `\int_log:N`. This function is documented on page 101.)

`\int_log:n` Similar to `\int_show:n`.

```

8848 \cs_new_protected:Npn \int_log:n
8849   { \msg_log_eval:Nn \int_eval:n }

```

(End definition for `\int_log:n`. This function is documented on page 101.)

12.11 Random integers

`\int_rand:nn` Defined in `l3fp-random`.

(End definition for `\int_rand:nn`. This function is documented on page 100.)

12.12 Constant integers

`\c_zero_int` The zero is defined in `l3basics`.

```
\c_one_int 8850 \int_const:Nn \c_one_int { 1 }
```

(End definition for `\c_zero_int` and `\c_one_int`. These variables are documented on page 101.)

`\c_max_int` The largest number allowed is $2^{31} - 1$

```
8851 \int_const:Nn \c_max_int { 2 147 483 647 }
```

(End definition for `\c_max_int`. This variable is documented on page 101.)

`\c_max_char_int` The largest character code is 1114111 (hexadecimal 10FFFF) in XeTeX and LuaTeX and 255 in other engines. In many places pTeX and upTeX support larger character codes but for instance the values of `\lccode` are restricted to $[0, 255]$.

```
8852 \int_const:Nn \c_max_char_int
8853 {
8854   \if_int_odd:w 0
8855     \cs_if_exist:NT \tex luatexversion:D { 1 }
8856     \cs_if_exist:NT \tex XeTeXversion:D { 1 } ~
8857     "10FFFF
8858   \else:
8859     "FF
8860   \fi:
8861 }
```

(End definition for `\c_max_char_int`. This variable is documented on page 101.)

12.13 Scratch integers

`\l_tmpa_int` We provide two local and two global scratch counters, maybe we need more or less.

```
\l_tmpb_int 8862 \int_new:N \l_tmpa_int
```

```
\g_tmpa_int 8863 \int_new:N \l_tmpb_int
```

```
\g_tmpb_int 8864 \int_new:N \g_tmpa_int
```

```
8865 \int_new:N \g_tmpb_int
```

(End definition for `\l_tmpa_int` and others. These variables are documented on page 101.)

12.14 Integers for earlier modules

`<@@=seq>`

```
\l__int_internal_a_int
```

```
\l__int_internal_b_int
```

```
8866 \int_new:N \l__int_internal_a_int
```

```
8867 \int_new:N \l__int_internal_b_int
```

(End definition for `\l__int_internal_a_int` and `\l__int_internal_b_int`.)

```
8868 </package>
```

13 l3flag implementation

```
8869 <*package>
```

```
8870 <@@=flag>
```

The following test files are used for this code: *m3flag001*.

13.1 Non-expandable flag commands

The height h of a flag (initially zero) is stored by setting control sequences of the form `\flag <name> <integer>` to `\relax` for $0 \leq \langle integer \rangle < h$. When a flag is raised, a “trap” function `\flag <name>` is called. The existence of this function is also used to test for the existence of a flag.

\flag_new:n For each flag, we define a “trap” function, which by default simply increases the flag by 1 by letting the appropriate control sequence to `\relax`. This can be done expandably!

```
8871 \cs_new_protected:Npn \flag_new:n #1
8872 {
8873   \cs_new:cpn { flag~#1 } ##1 ;
8874   { \exp_after:wN \use_none:n \cs:w flag~#1~##1 \cs_end: }
8875 }
```

(End definition for `\flag_new:n`. This function is documented on page 104.)

\flag_clear:n **__flag_clear:wn** Undefine control sequences, starting from the 0 flag, upwards, until reaching an undefined control sequence. We don’t use `\cs_undefine:c` because that would act globally. When the option `check-declarations` is used, check for the function defined by `\flag_new:n`.

```
8876 \cs_new_protected:Npn \flag_clear:n #1 { \__flag_clear:wn 0 ; {#1} }
8877 \cs_new_protected:Npn \__flag_clear:wn #1 ; #2
8878 {
8879   \if_cs_exist:w flag~#2~#1 \cs_end:
8880   \cs_set_eq:cN { flag~#2~#1 } \tex_undefined:D
8881   \exp_after:wN \__flag_clear:wn
8882   \int_value:w \int_eval:w 1 + #1
8883   \else:
8884     \use_i:nnn
8885   \fi:
8886   ; {#2}
8887 }
```

(End definition for `\flag_clear:n` and `__flag_clear:wn`. This function is documented on page 104.)

\flag_clear_new:n As for other datatypes, clear the *<flag>* or create a new one, as appropriate.

```
8888 \cs_new_protected:Npn \flag_clear_new:n #1
8889 { \flag_if_exist:nTF {#1} { \flag_clear:n } { \flag_new:n } {#1} }
```

(End definition for `\flag_clear_new:n`. This function is documented on page 104.)

\flag_show:n **\flag_log:n** **__flag_show:Nn** Show the height (terminal or log file) using appropriate `l3msg` auxiliaries.

```
8890 \cs_new_protected:Npn \flag_show:n { \__flag_show:Nn \tl_show:n }
8891 \cs_new_protected:Npn \flag_log:n { \__flag_show:Nn \tl_log:n }
8892 \cs_new_protected:Npn \__flag_show:Nn #1#2
8893 {
8894   \exp_args:Nc \__kernel_chk_defined:NT { flag~#2 }
8895   {
```

```

8896         \exp_args:Nx #1
8897         { \tl_to_str:n { flag~#2~height } = \flag_height:n {#2} }
8898     }
8899 }

```

(End definition for `\flag_show:n`, `\flag_log:n`, and `__flag_show:Nn`. These functions are documented on page 104.)

13.2 Expandable flag commands

`\flag_if_exist:p:n` A flag exist if the corresponding trap `\flag <flag name>:n` is defined.

```

\flag_if_exist:nTF
8900 \prg_new_conditional:Npnn \flag_if_exist:n #1 { p , T , F , TF }
8901 {
8902     \cs_if_exist:cTF { flag~#1 }
8903     { \prg_return_true: } { \prg_return_false: }
8904 }

```

(End definition for `\flag_if_exist:nTF`. This function is documented on page 105.)

`\flag_if_raised:p:n` Test if the flag has a non-zero height, by checking the 0 control sequence.

```

\flag_if_raised:nTF
8905 \prg_new_conditional:Npnn \flag_if_raised:n #1 { p , T , F , TF }
8906 {
8907     \if_cs_exist:w flag~#1~0 \cs_end:
8908     \prg_return_true:
8909     \else:
8910     \prg_return_false:
8911     \fi:
8912 }

```

(End definition for `\flag_if_raised:nTF`. This function is documented on page 105.)

`\flag_height:n` Extract the value of the flag by going through all of the control sequences starting from 0.

```

\__flag_height_loop:wn
\__flag_height_end:wn
8913 \cs_new:Npn \flag_height:n #1 { \__flag_height_loop:wn 0; {#1} }
8914 \cs_new:Npn \__flag_height_loop:wn #1 ; #2
8915 {
8916     \if_cs_exist:w flag~#2~#1 \cs_end:
8917     \exp_after:wN \__flag_height_loop:wn \int_value:w \int_eval:w 1 +
8918     \else:
8919     \exp_after:wN \__flag_height_end:wn
8920     \fi:
8921     #1 ; {#2}
8922 }
8923 \cs_new:Npn \__flag_height_end:wn #1 ; #2 {#1}

```

(End definition for `\flag_height:n`, `__flag_height_loop:wn`, and `__flag_height_end:wn`. This function is documented on page 105.)

`\flag_raise:n` Simply apply the trap to the height, after expanding the latter.

```

8924 \cs_new:Npn \flag_raise:n #1
8925 {
8926     \cs:w flag~#1 \exp_after:wN \cs_end:
8927     \int_value:w \flag_height:n {#1} ;
8928 }

```

(End definition for `\flag_raise:n`. This function is documented on page 105.)

```

8929 </package>

```

14 l3prg implementation

The following test files are used for this code: *m3prg001.lvt, m3prg002.lvt, m3prg003.lvt.*

```
8930 <*package>
```

14.1 Primitive conditionals

`\if_bool:N` Those two primitive T_EX conditionals are synonyms. `\if_bool:N` is defined in *l3basics*, as it's needed earlier to define quark test functions.

```
8931 \cs_new_eq:NN \if_predicate:w \tex_ifodd:D
```

(End definition for `\if_bool:N` and `\if_predicate:w`. These functions are documented on page 114.)

14.2 Defining a set of conditional functions

These are all defined in *l3basics*, as they are needed “early”. This is just a reminder!

(End definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 106.)

14.3 The boolean data type

```
8932 <@@=bool>
```

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

```
8933 \cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
8934 \cs_generate_variant:Nn \bool_new:N { c }
```

(End definition for `\bool_new:N`. This function is documented on page 108.)

`\bool_const:Nn` A merger between `\tl_const:Nn` and `\bool_set:Nn`.

```
\bool_const:cn
8935 \cs_new_protected:Npn \bool_const:Nn #1#2
8936 {
8937   \__kernel_chk_if_free_cs:N #1
8938   \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}
8939 }
8940 \cs_generate_variant:Nn \bool_const:Nn { c }
```

(End definition for `\bool_const:Nn`. This function is documented on page 109.)

`\bool_set_true:N` Setting is already pretty easy. When `check-declarations` is active, the definitions are patched to make sure the boolean exists. This is needed because booleans are not based on token lists nor on T_EX registers.

```
\bool_set_true:c
\bool_gset_true:N
\bool_gset_true:c
\bool_set_false:N
\bool_set_false:c
\bool_gset_false:N
\bool_gset_false:c
8941 \cs_new_protected:Npn \bool_set_true:N #1
8942 { \cs_set_eq:NN #1 \c_true_bool }
8943 \cs_new_protected:Npn \bool_set_false:N #1
8944 { \cs_set_eq:NN #1 \c_false_bool }
8945 \cs_new_protected:Npn \bool_gset_true:N #1
8946 { \cs_gset_eq:NN #1 \c_true_bool }
8947 \cs_new_protected:Npn \bool_gset_false:N #1
8948 { \cs_gset_eq:NN #1 \c_false_bool }
8949 \cs_generate_variant:Nn \bool_set_true:N { c }
8950 \cs_generate_variant:Nn \bool_set_false:N { c }
8951 \cs_generate_variant:Nn \bool_gset_true:N { c }
8952 \cs_generate_variant:Nn \bool_gset_false:N { c }
```

(End definition for `\bool_set_true:N` and others. These functions are documented on page 109.)

`\bool_set_eq:NN` The usual copy code. While it would be cleaner semantically to copy the `\cs_set_eq:NN` family of functions, we copy `\tl_set_eq:NN` because that has the correct checking code.

`\bool_set_eq:cN`

`\bool_set_eq:Nc` 8953 `\cs_new_eq:NN \bool_set_eq:NN \tl_set_eq:NN`

`\bool_set_eq:cc` 8954 `\cs_new_eq:NN \bool_gset_eq:NN \tl_gset_eq:NN`

`\bool_gset_eq:NN` 8955 `\cs_generate_variant:Nn \bool_set_eq:NN { Nc, cN, cc }`

`\bool_gset_eq:cN` 8956 `\cs_generate_variant:Nn \bool_gset_eq:NN { Nc, cN, cc }`

`\bool_gset_eq:Nc` (End definition for `\bool_set_eq:NN` and `\bool_gset_eq:NN`. These functions are documented on page 109.)

`\bool_gset_eq:cc`

`\bool_set:Nn` This function evaluates a boolean expression and assigns the first argument the meaning

`\bool_set:cn` `\c_true_bool` or `\c_false_bool`. Again, we include some checking code. It is important

`\bool_gset:Nn` to evaluate the expression before applying the `\chardef` primitive, because that primitive

`\bool_gset:cn` sets the left-hand side to `\scan_stop:` before looking for the right-hand side.

8957 `\cs_new_protected:Npn \bool_set:Nn #1#2`
8958 `{`
8959 `\exp_last_unbraced:NNNf`
8960 `\tex_chardef:D #1 = { \bool_if_p:n {#2} }`
8961 `}`
8962 `\cs_new_protected:Npn \bool_gset:Nn #1#2`
8963 `{`
8964 `\exp_last_unbraced:NNNNf`
8965 `\tex_global:D \tex_chardef:D #1 = { \bool_if_p:n {#2} }`
8966 `}`
8967 `\cs_generate_variant:Nn \bool_set:Nn { c }`
8968 `\cs_generate_variant:Nn \bool_gset:Nn { c }`

(End definition for `\bool_set:Nn` and `\bool_gset:Nn`. These functions are documented on page 109.)

14.4 Internal auxiliaries

`\q__bool_recursion_tail` Internal recursion quarks.

`\q__bool_recursion_stop` 8969 `\quark_new:N \q__bool_recursion_tail`
8970 `\quark_new:N \q__bool_recursion_stop`

(End definition for `\q__bool_recursion_tail` and `\q__bool_recursion_stop`.)

`__bool_use_i_delimit_by_q_recursion_stop:nw` Functions to gobble up to a quark.

8971 `\cs_new:Npn __bool_use_i_delimit_by_q_recursion_stop:nw`
8972 `#1 #2 \q__bool_recursion_stop {#1}`

(End definition for `__bool_use_i_delimit_by_q_recursion_stop:nw`.)

`__bool_if_recursion_tail_stop_do:nn` Functions to query recursion quarks.

8973 `__kernel_quark_new_test:N __bool_if_recursion_tail_stop_do:nn`

(End definition for `__bool_if_recursion_tail_stop_do:nn`.)

`\bool_if_p:N` Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

```

\bool_if:N $\underline{TF}$  8974 \prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
\bool_if:c $\underline{TF}$  8975 {
8976   \if_bool:N #1
8977   \prg_return_true:
8978   \else:
8979   \prg_return_false:
8980   \fi:
8981 }
8982 \prg_generate_conditional_variant:Nnn \bool_if:N { c } { p , T , F , TF }

```

(End definition for `\bool_if:N \underline{TF}` . This function is documented on page 109.)

`\bool_show:n` Show the truth value of the boolean, as true or false.

```

\bool_log:n 8983 \cs_new_protected:Npn \bool_show:n
\__bool_to_str:n 8984 { \msg_show_eval:Nn \__bool_to_str:n }
8985 \cs_new_protected:Npn \bool_log:n
8986 { \msg_log_eval:Nn \__bool_to_str:n }
8987 \cs_new:Npn \__bool_to_str:n #1
8988 { \bool_if:nTF {#1} { true } { false } }

```

(End definition for `\bool_show:n`, `\bool_log:n`, and `__bool_to_str:n`. These functions are documented on page 109.)

`\bool_show:N` Show the truth value of the boolean, as true or false.

```

\bool_show:c 8989 \cs_new_protected:Npn \bool_show:N { \__bool_show:NN \tl_show:n }
\bool_log:N 8990 \cs_generate_variant:Nn \bool_show:N { c }
\bool_log:c 8991 \cs_new_protected:Npn \bool_log:N { \__bool_show:NN \tl_log:n }
\__bool_show:NN 8992 \cs_generate_variant:Nn \bool_log:N { c }
8993 \cs_new_protected:Npn \__bool_show:NN #1#2
8994 {
8995   \__kernel_chk_defined:NT #2
8996   { \exp_args:Nx #1 { \token_to_str:N #2 = \__bool_to_str:n {#2} } }
8997 }

```

(End definition for `\bool_show:N`, `\bool_log:N`, and `__bool_show:NN`. These functions are documented on page 109.)

`\l_tmpa_bool` A few booleans just if you need them.

```

\l_tmpb_bool 8998 \bool_new:N \l_tmpa_bool
\g_tmpa_bool 8999 \bool_new:N \l_tmpb_bool
\g_tmpb_bool 9000 \bool_new:N \g_tmpa_bool
9001 \bool_new:N \g_tmpb_bool

```

(End definition for `\l_tmpa_bool` and others. These variables are documented on page 110.)

`\bool_if_exist_p:N` Copies of the cs functions defined in l3basics.

```

\bool_if_exist_p:c 9002 \prg_new_eq_conditional:NNn \bool_if_exist:N \cs_if_exist:N
\bool_if_exist:N $\underline{TF}$  9003 { TF , T , F , p }
\bool_if_exist:c $\underline{TF}$  9004 \prg_new_eq_conditional:NNn \bool_if_exist:c \cs_if_exist:c
9005 { TF , T , F , p }

```

(End definition for `\bool_if_exist:N \underline{TF}` . This function is documented on page 110.)

14.5 Boolean expressions

`\bool_if_p:n` Evaluating the truth value of a list of predicates is done using an input syntax somewhat similar to the one found in other programming languages with (and) for grouping, ! for logical “Not”, && for logical “And” and || for logical “Or”. However, they perform eager evaluation. We shall use the terms Not, And, Or, Open and Close for these operations.

`\bool_if:nTF`

Any expression is terminated by a Close operation. Evaluation happens from left to right in the following manner using a GetNext function:

- If an Open is seen, start evaluating a new expression using the Eval function and call GetNext again.
- If a Not is seen, remove the ! and call a GetNext function with the logic reversed.
- If none of the above, reinsert the token found (this is supposed to be a predicate function) in front of an Eval function, which evaluates it to the boolean value $\langle true \rangle$ or $\langle false \rangle$.

The Eval function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

$\langle true \rangle$ **And** Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

$\langle false \rangle$ **And** Current truth value is false, logical And seen, stop using the values of predicates within this sub-expression until the next Close. Then return $\langle false \rangle$.

$\langle true \rangle$ **Or** Current truth value is true, logical Or seen, stop using the values of predicates within this sub-expression until the nearest Close. Then return $\langle true \rangle$.

$\langle false \rangle$ **Or** Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

$\langle true \rangle$ **Close** Current truth value is true, Close seen, return $\langle true \rangle$.

$\langle false \rangle$ **Close** Current truth value is false, Close seen, return $\langle false \rangle$.

```

9006 \prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
9007 {
9008   \if_predicate:w \bool_if_p:n {#1}
9009   \prg_return_true:
9010   \else:
9011   \prg_return_false:
9012   \fi:
9013 }
```

(End definition for `\bool_if:nTF`. This function is documented on page 111.)

`\bool_if_p:n` To speed up the case of a single predicate, f-expand and check whether the result is one token (possibly surrounded by spaces), which must be `\c_true_bool` or `\c_false_bool`. We use a version of `\tl_if_single:nTF` optimized for speed since we know that an empty #1 is an error. The auxiliary `__bool_if_p_aux:w` removes the trailing parenthesis and gets rid of any space. For the general case, first issue a `\group_align_safe_begin:` as we are using && as syntax shorthand for the And operation and we need to hide it for

`__bool_if_p:n`

`__bool_if_p_aux:w`

TeX. This group is closed after `__bool_get_next:NN` returns `\c_true_bool` or `\c_false_bool`. That function requires the trailing parenthesis to know where the expression ends.

```

9014 \cs_new:Npn \bool_if_p:n { \exp_args:Nf \__bool_if_p:n }
9015 \cs_new:Npn \__bool_if_p:n #1
9016 {
9017   \tl_if_empty:oT { \use_none:nn #1 . } { \__bool_if_p_aux:w }
9018   \group_align_safe_begin:
9019   \exp_after:wN
9020   \group_align_safe_end:
9021   \exp:w \exp_end_continue_f:w % (
9022   \__bool_get_next:NN \use_i:nnnn #1 )
9023 }
9024 \cs_new:Npn \__bool_if_p_aux:w #1 \use_i:nnnn #2#3 {#2}

```

(End definition for `\bool_if_p:n`, `__bool_if_p:n`, and `__bool_if_p_aux:w`. This function is documented on page 111.)

`__bool_get_next:NN` The GetNext operation. Its first argument is `\use_i:nnnn`, `\use_ii:nnnn`, `\use_iii:nnnn`, or `\use_iv:nnnn` (we call these “states”). In the first state, this function eventually expand to the truth value `\c_true_bool` or `\c_false_bool` of the expression which follows until the next unmatched closing parenthesis. For instance “`__bool_get_next:NN \use_i:nnnn \c_true_bool && \c_true_bool)`” (including the closing parenthesis) expands to `\c_true_bool`. In the second state (after a `!`) the logic is reversed. We call these two states “normal” and the next two “skipping”. In the third state (after `\c_true_bool||`) it always returns `\c_true_bool`. In the fourth state (after `\c_false_bool&&`) it always returns `\c_false_bool` and also stops when encountering `||`, not only parentheses. This code itself is a switch: if what follows is neither `!` nor `(`, we assume it is a predicate.

```

9025 \cs_new:Npn \__bool_get_next:NN #1#2
9026 {
9027   \use:c
9028   {
9029     __bool_
9030     \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
9031     :Nw
9032   }
9033   #1 #2
9034 }

```

(End definition for `__bool_get_next:NN`.)

`__bool_!:Nw` The Not operation reverses the logic: it discards the `!` token and calls the GetNext operation with the appropriate first argument. Namely the first and second states are interchanged, but after `\c_true_bool||` or `\c_false_bool&&` the `!` is ignored.

```

9035 \cs_new:cpn { __bool_!:Nw } #1#2
9036 {
9037   \exp_after:wN \__bool_get_next:NN
9038   #1 \use_ii:nnnn \use_i:nnnn \use_iii:nnnn \use_iv:nnnn
9039 }

```

(End definition for `__bool_!:Nw`.)

`__bool_(:Nw` The Open operation starts a sub-expression after discarding the open parenthesis. This is done by calling `GetNext` (which eventually discards the corresponding closing parenthesis), with a post-processing step which looks for And, Or or Close after the group.

```

9040 \cs_new:cpn { __bool_(:Nw } #1#2
9041 {
9042   \exp_after:wN \__bool_choose:NNN \exp_after:wN #1
9043   \int_value:w \__bool_get_next:NN \use_i:nnnn
9044 }

```

(End definition for `__bool_(:Nw`.)

`__bool_p:Nw` If what follows `GetNext` is neither `!` nor `(`, evaluate the predicate using the primitive `\int_value:w`. The canonical `true` and `false` values have numerical values 1 and 0 respectively. Look for And, Or or Close afterwards.

```

9045 \cs_new:cpn { __bool_p:Nw } #1
9046 { \exp_after:wN \__bool_choose:NNN \exp_after:wN #1 \int_value:w }

```

(End definition for `__bool_p:Nw`.)

`__bool_choose:NNN` The arguments are `#1`: a function such as `\use_i:nnnn`, `#2`: 0 or 1 encoding the current truth value, `#3`: the next operation, And, Or or Close. We distinguish three cases according to a combination of `#1` and `#2`. Case 2 is when `#1` is `\use_iii:nnnn` (state 3), namely after `\c_true_bool ||`. Case 1 is when `#1` is `\use_i:nnnn` and `#2` is `true` or when `#1` is `\use_ii:nnnn` and `#2` is `false`, for instance for `!\c_false_bool`. Case 0 includes the same with `true/false` interchanged and the case where `#1` is `\use_iv:nnnn` namely after `\c_false_bool &&`.

`__bool_|_0:` When seeing `)` the current subexpression is done, leave the appropriate boolean.
`__bool_|_1:` When seeing `&` in case 0 go into state 4, equivalent to having seen `\c_false_bool &&`.
`__bool_|_2:` In case 1, namely when the argument is `true` and we are in a normal state continue in the normal state 1. In case 2, namely when skipping alternatives in an Or, continue in the same state. When seeing `|` in case 0, continue in a normal state; in particular stop skipping for `\c_false_bool &&` because that binds more tightly than `||`. In the other two cases start skipping for `\c_true_bool ||`.

```

9047 \cs_new:Npn \__bool_choose:NNN #1#2#3
9048 {
9049   \use:c
9050   {
9051     __bool_ \token_to_str:N #3 _
9052     #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: } 2 0 :
9053   }
9054 }
9055 \cs_new:cpn { __bool_)_0: } { \c_false_bool }
9056 \cs_new:cpn { __bool_)_1: } { \c_true_bool }
9057 \cs_new:cpn { __bool_)_2: } { \c_true_bool }
9058 \cs_new:cpn { __bool_&_0: } & { \__bool_get_next:NN \use_iv:nnnn }
9059 \cs_new:cpn { __bool_&_1: } & { \__bool_get_next:NN \use_i:nnnn }
9060 \cs_new:cpn { __bool_&_2: } & { \__bool_get_next:NN \use_iii:nnnn }
9061 \cs_new:cpn { __bool_|_0: } | { \__bool_get_next:NN \use_i:nnnn }
9062 \cs_new:cpn { __bool_|_1: } | { \__bool_get_next:NN \use_iii:nnnn }
9063 \cs_new:cpn { __bool_|_2: } | { \__bool_get_next:NN \use_iii:nnnn }

```

(End definition for `__bool_choose:NNN` and others.)

\bool_lazy_all_p:n Go through the list of expressions, stopping whenever an expression is **false**. If the end is reached without finding any **false** expression, then the result is **true**.

\bool_lazy_all:nTF

```

9064 \cs_new:Npn \bool_lazy_all_p:n #1
9065 { \__bool_lazy_all:n #1 \q_bool_recursion_tail \q_bool_recursion_stop }
9066 \prg_new_conditional:Npnn \bool_lazy_all:n #1 { T , F , TF }
9067 {
9068   \if_predicate:w \bool_lazy_all_p:n {#1}
9069   \prg_return_true:
9070   \else:
9071   \prg_return_false:
9072   \fi:
9073 }
9074 \cs_new:Npn \__bool_lazy_all:n #1
9075 {
9076   \__bool_if_recursion_tail_stop_do:nn {#1} { \c_true_bool }
9077   \bool_if:nF {#1}
9078   { \__bool_use_i_delimit_by_q_recursion_stop:nw { \c_false_bool } }
9079   \__bool_lazy_all:n
9080 }

```

(End definition for **\bool_lazy_all:nTF** and **__bool_lazy_all:n**. This function is documented on page 111.)

\bool_lazy_and_p:nn Only evaluate the second expression if the first is **true**. Note that #2 must be removed as an argument, not just by skipping to the **\else:** branch of the conditional since #2 may contain unbalanced **TeX** conditionals.

\bool_lazy_and:nnTF

```

9081 \prg_new_conditional:Npnn \bool_lazy_and:nn #1#2 { p , T , F , TF }
9082 {
9083   \if_predicate:w
9084   \bool_if:nTF {#1} { \bool_if_p:n {#2} } { \c_false_bool }
9085   \prg_return_true:
9086   \else:
9087   \prg_return_false:
9088   \fi:
9089 }

```

(End definition for **\bool_lazy_and:nnTF**. This function is documented on page 111.)

\bool_lazy_any_p:n Go through the list of expressions, stopping whenever an expression is **true**. If the end is reached without finding any **true** expression, then the result is **false**.

\bool_lazy_any:nTF

```

9090 \cs_new:Npn \bool_lazy_any_p:n #1
9091 { \__bool_lazy_any:n #1 \q_bool_recursion_tail \q_bool_recursion_stop }
9092 \prg_new_conditional:Npnn \bool_lazy_any:n #1 { T , F , TF }
9093 {
9094   \if_predicate:w \bool_lazy_any_p:n {#1}
9095   \prg_return_true:
9096   \else:
9097   \prg_return_false:
9098   \fi:
9099 }
9100 \cs_new:Npn \__bool_lazy_any:n #1
9101 {
9102   \__bool_if_recursion_tail_stop_do:nn {#1} { \c_false_bool }
9103   \bool_if:nT {#1}

```

```

9104     { \_bool_use_i_delimit_by_q_recursion_stop:nw { \c_true_bool } }
9105     \_bool_lazy_any:n
9106   }

```

(End definition for `\bool_lazy_any:nTF` and `_bool_lazy_any:n`. This function is documented on page 112.)

`\bool_lazy_or_p:nn` Only evaluate the second expression if the first is false.

```

\bool_lazy_or:nnTF
9107 \prg_new_conditional:Npnn \bool_lazy_or:nn #1#2 { p , T , F , TF }
9108 {
9109   \if_predicate:w
9110     \bool_if:nTF {#1} { \c_true_bool } { \bool_if_p:n {#2} }
9111     \prg_return_true:
9112   \else:
9113     \prg_return_false:
9114   \fi:
9115 }

```

(End definition for `\bool_lazy_or:nnTF`. This function is documented on page 112.)

`\bool_not_p:n` The Not variant just reverses the outcome of `\bool_if_p:n`. Can be optimized but this is nice and simple and according to the implementation plan. Not even particularly useful to have it when the infix notation is easier to use.

```

9116 \cs_new:Npn \bool_not_p:n #1 { \bool_if_p:n { ! ( #1 ) } }

```

(End definition for `\bool_not_p:n`. This function is documented on page 112.)

`\bool_xor_p:nn` Exclusive or. If the boolean expressions have same truth value, return false, otherwise return true.

```

\bool_xor:nnTF
9117 \prg_new_conditional:Npnn \bool_xor:nn #1#2 { p , T , F , TF }
9118 {
9119   \bool_if:nT {#1} \reverse_if:N
9120   \if_predicate:w \bool_if_p:n {#2}
9121     \prg_return_true:
9122   \else:
9123     \prg_return_false:
9124   \fi:
9125 }

```

(End definition for `\bool_xor:nnTF`. This function is documented on page 112.)

14.6 Logical loops

`\bool_while_do:Nn` A while loop where the boolean is tested before executing the statement. The “while” version executes the code as long as the boolean is true; the “until” version executes the code as long as the boolean is false.

```

\bool_while_do:cn
\bool_until_do:Nn
\bool_until_do:cn
9126 \cs_new:Npn \bool_while_do:Nn #1#2
9127   { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
9128 \cs_new:Npn \bool_until_do:Nn #1#2
9129   { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
9130 \cs_generate_variant:Nn \bool_while_do:Nn { c }
9131 \cs_generate_variant:Nn \bool_until_do:Nn { c }

```

(End definition for `\bool_while_do:Nn` and `\bool_until_do:Nn`. These functions are documented on page 112.)

\bool_do_while:Nn A do-while loop where the body is performed at least once and the boolean is tested after executing the body. Otherwise identical to the above functions.

\bool_do_while:cn

```

9132 \cs_new:Npn \bool_do_while:Nn #1#2
9133 { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
9134 \cs_new:Npn \bool_do_until:Nn #1#2
9135 { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
9136 \cs_generate_variant:Nn \bool_do_while:Nn { c }
9137 \cs_generate_variant:Nn \bool_do_until:Nn { c }

```

(End definition for \bool_do_while:Nn and \bool_do_until:Nn. These functions are documented on page 112.)

\bool_while_do:nn Loop functions with the test either before or after the first body expansion.

\bool_do_while:nn

```

9138 \cs_new:Npn \bool_while_do:nn #1#2
9139 {
9140   \bool_if:nT {#1}
9141   {
9142     #2
9143     \bool_while_do:nn {#1} {#2}
9144   }
9145 }
9146 \cs_new:Npn \bool_do_while:nn #1#2
9147 {
9148   #2
9149   \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
9150 }
9151 \cs_new:Npn \bool_until_do:nn #1#2
9152 {
9153   \bool_if:nF {#1}
9154   {
9155     #2
9156     \bool_until_do:nn {#1} {#2}
9157   }
9158 }
9159 \cs_new:Npn \bool_do_until:nn #1#2
9160 {
9161   #2
9162   \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2} }
9163 }

```

(End definition for \bool_while_do:nn and others. These functions are documented on page 113.)

14.7 Producing multiple copies

9164 <@@=prg>

\prg_replicate:nn This function uses a cascading csname technique by David Kastrup (who else :-)

The idea is to make the input 25 result in first adding five, and then 20 copies of the code to be replicated. The technique uses cascading csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with 25 then the next function is the one that inserts two copies but it sees the ten copies handed

```

\__prg_replicate:N
\__prg_replicate_first:N
\__prg_replicate_
\__prg_replicate_0:n
\__prg_replicate_1:n
\__prg_replicate_2:n
\__prg_replicate_3:n
\__prg_replicate_4:n
\__prg_replicate_5:n
\__prg_replicate_6:n
\__prg_replicate_7:n
\__prg_replicate_8:n
\__prg_replicate_9:n
\__prg_replicate_first_~:n
\__prg_replicate_first_0:n
\__prg_replicate_first_1:n
\__prg_replicate_first_2:n

```

down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. These functions also close the expansion of `\exp:w`, which ensures that `\prg_replicate:nn` only requires two steps of expansion.

This function has one flaw though: Since it constantly passes down ten copies of its previous argument it severely affects the main memory once you start demanding hundreds of thousands of copies. Now I don't think this is a real limitation for any ordinary use, and if necessary, it is possible to write `\prg_replicate:nn {1000} { \prg_replicate:nn {1000} {<code>} }`. An alternative approach is to create a string of `m`'s with `\exp:w` which can be done with just four macros but that method has its own problems since it can exhaust the string pool. Also, it is considerably slower than what we use here so the few extra csnames are well spent I would say.

```

9165 \cs_new:Npn \prg_replicate:nn #1
9166 {
9167   \exp:w
9168   \exp_after:wN \__prg_replicate_first:N
9169   \int_value:w \int_eval:n {#1}
9170   \cs_end:
9171 }
9172 \cs_new:Npn \__prg_replicate:N #1
9173 { \cs:w \__prg_replicate_#1 :n \__prg_replicate:N }
9174 \cs_new:Npn \__prg_replicate_first:N #1
9175 { \cs:w \__prg_replicate_first_#1 :n \__prg_replicate:N }

```

Then comes all the functions that do the hard work of inserting all the copies. The first function takes `:n` as a parameter.

```

9176 \cs_new:Npn \__prg_replicate_ :n #1 { \cs_end: }
9177 \cs_new:cpn { __prg_replicate_0:n } #1
9178 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} }
9179 \cs_new:cpn { __prg_replicate_1:n } #1
9180 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1 }
9181 \cs_new:cpn { __prg_replicate_2:n } #1
9182 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1 }
9183 \cs_new:cpn { __prg_replicate_3:n } #1
9184 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1 }
9185 \cs_new:cpn { __prg_replicate_4:n } #1
9186 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1 }
9187 \cs_new:cpn { __prg_replicate_5:n } #1
9188 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1 }
9189 \cs_new:cpn { __prg_replicate_6:n } #1
9190 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
9191 \cs_new:cpn { __prg_replicate_7:n } #1
9192 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1 }
9193 \cs_new:cpn { __prg_replicate_8:n } #1
9194 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1 }
9195 \cs_new:cpn { __prg_replicate_9:n } #1
9196 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1#1 }

```

Users shouldn't ask for something to be replicated once or even not at all but...

```

9197 \cs_new:cpn { __prg_replicate_first_-:n } #1
9198 {
9199   \exp_end:
9200   \__kernel_msg_expandable_error:nn { kernel } { negative-replication }

```

```

9201 }
9202 \cs_new:cpn { __prg_replicate_first_0:n } #1 { \exp_end: }
9203 \cs_new:cpn { __prg_replicate_first_1:n } #1 { \exp_end: #1 }
9204 \cs_new:cpn { __prg_replicate_first_2:n } #1 { \exp_end: #1#1 }
9205 \cs_new:cpn { __prg_replicate_first_3:n } #1 { \exp_end: #1#1#1 }
9206 \cs_new:cpn { __prg_replicate_first_4:n } #1 { \exp_end: #1#1#1#1 }
9207 \cs_new:cpn { __prg_replicate_first_5:n } #1 { \exp_end: #1#1#1#1#1 }
9208 \cs_new:cpn { __prg_replicate_first_6:n } #1 { \exp_end: #1#1#1#1#1#1 }
9209 \cs_new:cpn { __prg_replicate_first_7:n } #1 { \exp_end: #1#1#1#1#1#1#1 }
9210 \cs_new:cpn { __prg_replicate_first_8:n } #1 { \exp_end: #1#1#1#1#1#1#1#1 }
9211 \cs_new:cpn { __prg_replicate_first_9:n } #1
9212 { \exp_end: #1#1#1#1#1#1#1#1#1 }

```

(End definition for `\prg_replicate:nn` and others. This function is documented on page 113.)

14.8 Detecting TeX's mode

`\mode_if_vertical_p:` For testing vertical mode. Strikes me here on the bus with David, that as long as we are just talking about returning true and false states, we can just use the primitive conditionals for this and gobbling the `\exp_end:` in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

```

9213 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
9214 { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_vertical:TF`. This function is documented on page 114.)

`\mode_if_horizontal_p:` For testing horizontal mode.

```

\mode_if_horizontal:TF
9215 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
9216 { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_horizontal:TF`. This function is documented on page 113.)

`\mode_if_inner_p:` For testing inner mode.

```

\mode_if_inner:TF
9217 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
9218 { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_inner:TF`. This function is documented on page 113.)

`\mode_if_math_p:` For testing math mode. At the beginning of an alignment cell, this should be used only inside a non-expandable function.

```

\mode_if_math:TF
9219 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
9220 { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_math:TF`. This function is documented on page 113.)

14.9 Internal programming functions

`\group_align_safe_begin:` `\group_align_safe_end:` T_EX’s alignment structures present many problems. As Knuth says himself in *T_EX: The Program*: “It’s sort of a miracle whenever `\halign` or `\valign` work, [...]” One problem relates to commands that internally issues a `\cr` but also peek ahead for the next character for use in, say, an optional argument. If the next token happens to be a `&` with category code 4 we get some sort of weird error message because the underlying `\futurelet` stores the token at the end of the alignment template. This could be a `&_4` giving a message like `! Misplaced \cr.` or even worse: it could be the `\endtemplate` token causing even more trouble! To solve this we have to open a special group so that T_EX still thinks it’s on safe ground but at the same time we don’t want to introduce any brace group that may find its way to the output. The following functions help with this by using code documented only in Appendix D of *The T_EXbook*... We place the `\if_false: { \fi:` part at that place so that the successive expansions of `\group_align_safe_begin/end:` are always brace balanced.

```

9221 \cs_new:Npn \group_align_safe_begin:
9222   { \if_int_compare:w \if_false: { \fi: ‘} = \c_zero_int \fi: }
9223 \cs_new:Npn \group_align_safe_end:
9224   { \if_int_compare:w ‘{ = \c_zero_int } \fi: }

```

(End definition for `\group_align_safe_begin:` and `\group_align_safe_end:`. These functions are documented on page 115.)

`\g__kernel_prg_map_int` A nesting counter for mapping.

```

9225 \int_new:N \g__kernel_prg_map_int

```

(End definition for `\g__kernel_prg_map_int:`.)

`\prg_break_point:Nn` `\prg_map_break:Nn` These are defined in `l3basics`, as they are needed “early”. This is just a reminder that is the case!

(End definition for `\prg_break_point:Nn` and `\prg_map_break:Nn`. These functions are documented on page 114.)

`\prg_break_point:` Also done in `l3basics`.

`\prg_break:`

`\prg_break:n`

(End definition for `\prg_break_point:`, `\prg_break:`, and `\prg_break:n`. These functions are documented on page 115.)

```

9226 </package>

```

15 l3sys implementation

```

9227 <@@=sys>

```

15.1 Kernel code

```

9228 <*package>

```

15.1.1 Detecting the engine

`__sys_const:nn` Set the T, F, TF, p forms of #1 to be constants equal to the result of evaluating the boolean expression #2.

```

9229 \cs_new_protected:Npn \__sys_const:nn #1#2
9230   {
9231     \bool_if:nTF {#2}

```

```

9232     {
9233         \cs_new_eq:cN { #1 :T } \use:n
9234         \cs_new_eq:cN { #1 :F } \use_none:n
9235         \cs_new_eq:cN { #1 :TF } \use_i:nn
9236         \cs_new_eq:cN { #1 _p: } \c_true_bool
9237     }
9238     {
9239         \cs_new_eq:cN { #1 :T } \use_none:n
9240         \cs_new_eq:cN { #1 :F } \use:n
9241         \cs_new_eq:cN { #1 :TF } \use_ii:nn
9242         \cs_new_eq:cN { #1 _p: } \c_false_bool
9243     }
9244 }

```

(End definition for `_sys_const:nn`.)

`\sys_if_engine luatex_p:` Set up the engine tests on the basis exactly one test should be true. Mainly a case of looking for the appropriate marker primitive.

```

\sys_if_engine luatex:TF
\sys_if_engine pdftex_p:
\sys_if_engine pdftex:TF
\sys_if_engine ptex_p:
\sys_if_engine ptex:TF
\sys_if_engine uptex_p:
\sys_if_engine uptex:TF
\sys_if_engine xetex_p:
\sys_if_engine xetex:TF
\c_sys_engine_str
9245 \str_const:Nx \c_sys_engine_str
9246 {
9247     \cs_if_exist:NT \tex luatexversion:D { luatex }
9248     \cs_if_exist:NT \tex pdftexversion:D { pdftex }
9249     \cs_if_exist:NT \tex kanjiskip:D
9250     {
9251         \cs_if_exist:NTF \tex enablecjktoken:D
9252         { uptex }
9253         { ptex }
9254     }
9255     \cs_if_exist:NT \tex XeTeXversion:D { xetex }
9256 }
9257 \tl_map_inline:nn { { luatex } { pdftex } { ptex } { uptex } { xetex } }
9258 {
9259     \_sys_const:nn { sys_if_engine_ #1 }
9260     { \str_if_eq_p:Vn \c_sys_engine_str {#1} }
9261 }

```

(End definition for `\sys_if_engine luatex:TF` and others. These functions are documented on page 116.)

15.1.2 Randomness

This candidate function is placed there because `\sys_if_rand_exist:TF` is used in `l3fp-rand`.

`\sys_if_rand_exist_p:` Currently, randomness exists under pdfTeX, LuaTeX, pTeX and upTeX.

```

\sys_if_rand_exist:TF
9262 \_sys_const:nn { sys_if_rand_exist }
9263 { \cs_if_exist_p:N \tex uniformdeviate:D }

```

(End definition for `\sys_if_rand_exist:TF`. This function is documented on page 271.)

15.1.3 Platform

`\sys_if_platform_unix_p:` Setting these up requires the file module (file lookup), so is actually implemented there.

```

\sys_if_platform_unix:TF
\sys_if_platform_windows_p:
\sys_if_platform_windows:TF
\c_sys_platform_str

```

(End definition for `\sys_if_platform_unix:TF`, `\sys_if_platform_windows:TF`, and `\c_sys_platform-str`. These functions are documented on page 117.)

15.1.4 Configurations

`\sys_load_backend:n` Loading the backend code is pretty simply: check that the backend is valid, then load it up.

```

\__sys_load_backend_check:N
\c_sys_backend_str
9264 \cs_new_protected:Npn \sys_load_backend:n #1
9265 {
9266   \sys_finalise:
9267   \str_if_exist:NTF \c_sys_backend_str
9268   {
9269     \str_if_eq:VnF \c_sys_backend_str {#1}
9270     { \__kernel_msg_error:nn { sys } { backend-set } }
9271   }
9272   {
9273     \tl_if_blank:nF {#1}
9274     { \tl_set:Nn \g__sys_backend_tl {#1} }
9275     \__sys_load_backend_check:N \g__sys_backend_tl
9276     \str_const:Nx \c_sys_backend_str { \g__sys_backend_tl }
9277     \__kernel_sys_configuration_load:n
9278     { l3backend- \c_sys_backend_str }
9279   }
9280 }
9281 \cs_new_protected:Npn \__sys_load_backend_check:N #1
9282 {
9283   \sys_if_engine_xetex:TF
9284   {
9285     \str_case:VnF #1
9286     {
9287       { dvisvgm } { }
9288       { xdvipdfmx } { }
9289     }
9290     {
9291       \__kernel_msg_error:nxxx { sys } { wrong-backend }
9292       #1 { xdvipdfmx }
9293       \tl_gset:Nn #1 { xdvipdfmx }
9294     }
9295   }
9296   {
9297     \sys_if_output_pdf:TF
9298     {
9299       \str_if_eq:VnF #1 { pdfmode }
9300       {
9301         \__kernel_msg_error:nxxx { sys } { wrong-backend }
9302         #1 { pdfmode }
9303         \tl_gset:Nn #1 { pdfmode }
9304       }
9305     }
9306     {
9307       \str_case:VnF #1
9308       {
9309         { dvipdfmx } { }
9310         { dvips } { }
9311         { dvisvgm } { }
9312       }
9313       {

```

```

9314         \__kernel_msg_error:nnxx { sys } { wrong-backend }
9315         #1 { dvips }
9316         \tl_gset:Nn #1 { dvips }
9317     }
9318 }
9319 }
9320 }

```

(End definition for `\sys_load_backend:n`, `__sys_load_backend_check:N`, and `\c_sys_backend_str`. These functions are documented on page 119.)

```

\g__sys_debug_bool
\g__sys_deprecation_bool
9321 \bool_new:N \g__sys_debug_bool
9322 \bool_new:N \g__sys_deprecation_bool

```

(End definition for `\g__sys_debug_bool` and `\g__sys_deprecation_bool`.)

`\sys_load_debug:` Simple.
`\sys_load_deprecation:`

```

9323 \cs_new_protected:Npn \sys_load_debug:
9324 {
9325     \bool_if:NF \g__sys_debug_bool
9326     { \__kernel_sys_configuration_load:n { l3debug } }
9327     \bool_gset_true:N \g__sys_debug_bool
9328 }
9329 \cs_new_protected:Npn \sys_load_deprecation:
9330 {
9331     \bool_if:NF \g__sys_deprecation_bool
9332     { \__kernel_sys_configuration_load:n { l3deprecation } }
9333     \bool_gset_true:N \g__sys_deprecation_bool
9334 }

```

(End definition for `\sys_load_debug:` and `\sys_load_deprecation:`. These functions are documented on page 119.)

15.1.5 Access to the shell

```

\l__sys_internal_tl
9335 \tl_new:N \l__sys_internal_tl

```

(End definition for `\l__sys_internal_tl`.)

`\c__sys_marker_tl` The same idea as the marker for rescanning token lists.

```

9336 \tl_const:Nx \c__sys_marker_tl { : \token_to_str:N : }

```

(End definition for `\c__sys_marker_tl`.)

`\sys_get_shell:nnN`**`\TF`** Setting using a shell is at this level just a slightly specialised file operation, with an additional check for quotes, as these are not supported.

```

\sys_get_shell:nnN
\__sys_get:nnN
\__sys_get_do:Nw
9337 \cs_new_protected:Npn \sys_get_shell:nnN #1#2#3
9338 {
9339     \sys_get_shell:nnNF {#1} {#2} #3
9340     { \tl_set:Nn #3 { \q_no_value } }
9341 }
9342 \prg_new_protected_conditional:Npnn \sys_get_shell:nnN #1#2#3 { T , F , TF }
9343 {

```

```

9344 \sys_if_shell:TF
9345 { \exp_args:No \__sys_get:nnN { \tl_to_str:n {#1} } {#2} #3 }
9346 { \prg_return_false: }
9347 }
9348 \cs_new_protected:Npn \__sys_get:nnN #1#2#3
9349 {
9350   \tl_if_in:nnTF {#1} { " }
9351   {
9352     \__kernel_msg_error:nnx
9353     { kernel } { quote-in-shell } {#1}
9354     \prg_return_false:
9355   }
9356   {
9357     \group_begin:
9358     \if_false: { \fi:
9359       \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
9360       \exp_args:No \tex_everyeof:D { \c__sys_marker_tl }
9361       #2 \scan_stop:
9362       \exp_after:wN \__sys_get_do:Nw
9363       \exp_after:wN #3
9364       \exp_after:wN \prg_do_nothing:
9365       \tex_input:D | "#1" \scan_stop:
9366       \if_false: } \fi:
9367       \prg_return_true:
9368     }
9369   }
9370 \exp_args:Nno \use:nn
9371 { \cs_new_protected:Npn \__sys_get_do:Nw #1#2 }
9372 { \c__sys_marker_tl }
9373 {
9374   \group_end:
9375   \tl_set:No #1 {#2}
9376 }

```

(End definition for `\sys_get_shell:nnNTF` and others. These functions are documented on page 118.)

`\c__sys_shell_stream_int` This is not needed for LuaTeX: shell escape there isn't done using a TeX interface.

```

9377 \sys_if_engine luatex:F
9378 { \int_const:Nn \c__sys_shell_stream_int { 18 } }

```

(End definition for `\c__sys_shell_stream_int`.)

`\sys_shell_now:n` Execute commands through shell escape immediately.

```

9379 \sys_if_engine luatex:TF
9380 {
9381   \cs_new_protected:Npn \sys_shell_now:n #1
9382   {
9383     \lua_now:e
9384     { \l3kernel.shellescape(" \lua_escape:e { \tl_to_str:n {#1} } ") }
9385   }
9386 }
9387 {
9388   \cs_new_protected:Npn \sys_shell_now:n #1
9389   { \iow_now:Nn \c__sys_shell_stream_int {#1} }
9390 }

```

```
9391 \cs_generate_variant:Nn \sys_shell_now:n { x }
```

(End definition for `\sys_shell_now:n`. This function is documented on page 118.)

`\sys_shell_shipout:n` Execute commands through shell escape at shipout.

```
9392 \sys_if_engine luatex:TF
9393 {
9394   \cs_new_protected:Npn \sys_shell_shipout:n #1
9395   {
9396     \lua_shipout_e:n
9397     { l3kernel.shellescape(" \lua_escape:e { \tl_to_str:n {#1} } ") }
9398   }
9399 }
9400 {
9401   \cs_new_protected:Npn \sys_shell_shipout:n #1
9402   { \iow_shipout:Nn \c__sys_shell_stream_int {#1} }
9403 }
9404 \cs_generate_variant:Nn \sys_shell_shipout:n { x }
```

(End definition for `\sys_shell_shipout:n`. This function is documented on page 118.)

15.2 Dynamic (every job) code

```
\sys_everyjob:
\__sys_everyjob:n
\g__sys_everyjob_tl
9405 \cs_new_protected:Npn \sys_everyjob:
9406 {
9407   \tl_use:N \g__sys_everyjob_tl
9408   \tl_gclear:N \g__sys_everyjob_tl
9409 }
9410 \cs_new_protected:Npn \__sys_everyjob:n #1
9411 { \tl_gput_right:Nn \g__sys_everyjob_tl {#1} }
9412 \tl_new:N \g__sys_everyjob_tl
```

(End definition for `\sys_everyjob:`, `__sys_everyjob:n`, and `\g__sys_everyjob_tl`. This function is documented on page ??.)

15.2.1 The name of the job

`\c_sys_jobname_str` Inherited from the L^AT_EX3 name for the primitive. This *has* to be the primitive as it's set in `\everyjob`. If the user does

```
pdflatex \input some-file-name
```

then `\everyjob` is inserted *before* `\jobname` is changed from `texput`, and thus we would have the wrong result.

```
9413 \__sys_everyjob:n
9414 { \cs_new_eq:NN \c_sys_jobname_str \tex_jobname:D }
```

(End definition for `\c_sys_jobname_str`. This variable is documented on page 116.)

15.2.2 Time and date

`\c_sys_minute_int` `\c_sys_hour_int` `\c_sys_day_int` `\c_sys_month_int` `\c_sys_year_int` Copies of the information provided by T_EX. There is a lot of defensive code in package mode: someone may have moved the primitives, and they can only be recovered if we have `\primitive` and it is working correctly. For IniT_EX of course that is all redundant but does no harm.

```

9415 \__sys_everyjob:n
9416 {
9417   \group_begin:
9418   \cs_set:Npn \__sys_tmp:w #1
9419     {
9420       \str_if_eq:eeTF { \cs_meaning:N #1 } { \token_to_str:N #1 }
9421       { #1 }
9422       {
9423         \cs_if_exist:NTF \tex_primitive:D
9424         {
9425           \bool_lazy_and:nnTF
9426             { \sys_if_engine_xetex_p: }
9427             {
9428               \int_compare_p:nNn
9429                 { \exp_after:wN \use_none:n \tex_XeTeXrevision:D }
9430                 < { 99999 }
9431             }
9432             { 0 }
9433             { \tex_primitive:D #1 }
9434         }
9435         { 0 }
9436       }
9437     }
9438   \int_const:Nn \c_sys_minute_int
9439     { \int_mod:nn { \__sys_tmp:w \time } { 60 } }
9440   \int_const:Nn \c_sys_hour_int
9441     { \int_div_truncate:nn { \__sys_tmp:w \time } { 60 } }
9442   \int_const:Nn \c_sys_day_int { \__sys_tmp:w \day }
9443   \int_const:Nn \c_sys_month_int { \__sys_tmp:w \month }
9444   \int_const:Nn \c_sys_year_int { \__sys_tmp:w \year }
9445   \group_end:
9446 }

```

(End definition for `\c_sys_minute_int` and others. These variables are documented on page 116.)

15.2.3 Random numbers

`\sys_rand_seed:` Unpack the primitive. When random numbers are not available, we return zero after an error (and incidentally make sure the number of expansions needed is the same as with random numbers available).

```

9447 \__sys_everyjob:n
9448 {
9449   \sys_if_rand_exist:TF
9450     { \cs_new:Npn \sys_rand_seed: { \tex_the:D \tex_randomseed:D } }
9451     {
9452       \cs_new:Npn \sys_rand_seed:
9453         {
9454           \int_value:w

```

```

9455         \__kernel_msg_expandable_error:nnn { kernel } { fp-no-random }
9456         { \sys_rand_seed: }
9457         \c_zero_int
9458     }
9459 }
9460 }

```

(End definition for `\sys_rand_seed`:. This function is documented on page 117.)

`\sys_gset_rand_seed:n` The primitive always assigns the seed globally.

```

9461 \__sys_everyjob:n
9462 {
9463     \sys_if_rand_exist:TF
9464     {
9465         \cs_new_protected:Npn \sys_gset_rand_seed:n #1
9466         { \tex_setrandomseed:D \int_eval:n {#1} \exp_stop_f: }
9467     }
9468     {
9469         \cs_new_protected:Npn \sys_gset_rand_seed:n #1
9470         {
9471             \__kernel_msg_error:nnn { kernel } { fp-no-random }
9472             { \sys_gset_rand_seed:n {#1} }
9473         }
9474     }
9475 }

```

(End definition for `\sys_gset_rand_seed:n`. This function is documented on page 117.)

15.2.4 Access to the shell

`\c_sys_shell_escape_int` Expose the engine's shell escape status to the user.

```

9476 \__sys_everyjob:n
9477 {
9478     \int_const:Nn \c_sys_shell_escape_int
9479     {
9480         \sys_if_engine luatex:TF
9481         {
9482             \tex_directlua:D
9483             { tex.sprint(status.shell_escape~or~os.execute()) }
9484         }
9485         { \tex_shellescape:D }
9486     }
9487 }

```

(End definition for `\c_sys_shell_escape_int`. This variable is documented on page 118.)

`\sys_if_shell_p:` Performs a check for whether shell escape is enabled. The first set of functions returns

`\sys_if_shell:TF` true if either of restricted or unrestricted shell escape is enabled, while the other two sets

`\sys_if_shell_unrestricted_p:` of functions return true in only one of these two cases.

```

\sys_if_shell_unrestricted:TF
\sys_if_shell_restricted_p:
\sys_if_shell_restricted:TF
9488 \__sys_everyjob:n
9489 {
9490     \__sys_const:nn { sys_if_shell }
9491     { \int_compare_p:nNn \c_sys_shell_escape_int > 0 }
9492     \__sys_const:nn { sys_if_shell_unrestricted }

```

```

9493     { \int_compare_p:nNn \c_sys_shell_escape_int = 1 }
9494   \__sys_const:nn { sys_if_shell_restricted }
9495     { \int_compare_p:nNn \c_sys_shell_escape_int = 2 }
9496   }

```

(End definition for `\sys_if_shell:TF`, `\sys_if_shell_unrestricted:TF`, and `\sys_if_shell_restricted:TF`. These functions are documented on page 118.)

15.2.5 Held over from l3file

`\g_file_curr_name_str` See comments about `\c_sys_jobname_str`: here, as soon as there is file input/output, things get “tided up”.

```

9497 \__sys_everyjob:n
9498   { \cs_gset_eq:NN \g_file_curr_name_str \tex_jobname:D }

```

(End definition for `\g_file_curr_name_str`. This variable is documented on page 166.)

15.3 Last-minute code

`\sys_finalise:` A simple hook to finalise the system-dependent layer. This is forced by the backend loader, which is forced by the main loader, so we do not need to include that here.

```

\__sys_finalise:n
\g__sys_finalise_tl
9499 \cs_new_protected:Npn \sys_finalise:
9500   {
9501     \sys_everyjob:
9502     \tl_use:N \g__sys_finalise_tl
9503     \tl_gclear:N \g__sys_finalise_tl
9504   }
9505 \cs_new_protected:Npn \__sys_finalise:n #1
9506   { \tl_gput_right:Nn \g__sys_finalise_tl {#1} }
9507 \tl_new:N \g__sys_finalise_tl

```

(End definition for `\sys_finalise:`, `__sys_finalise:n`, and `\g__sys_finalise_tl`. This function is documented on page 119.)

15.3.1 Detecting the output

`\sys_if_output_dvi_p:` This is a simple enough concept: the two views here are complementary.

```

\sys_if_output_dvi:TF
\sys_if_output_pdf_p:
\sys_if_output_pdf:TF
\c_sys_output_str
9508 \__sys_finalise:n
9509   {
9510     \str_const:Nx \c_sys_output_str
9511     {
9512       \int_compare:nNnTF
9513       { \cs_if_exist_use:NF \tex_pdfoutput:D { 0 } } > { 0 }
9514       { pdf }
9515       { dvi }
9516     }
9517     \__sys_const:nn { sys_if_output_dvi }
9518     { \str_if_eq_p:Vn \c_sys_output_str { dvi } }
9519     \__sys_const:nn { sys_if_output_pdf }
9520     { \str_if_eq_p:Vn \c_sys_output_str { pdf } }
9521   }

```

(End definition for `\sys_if_output_dvi:TF`, `\sys_if_output_pdf:TF`, and `\c_sys_output_str`. These functions are documented on page 117.)

15.3.2 Configurations

`\g__sys_backend_tl` As the backend has to be checked and possibly adjusted, the approach here is to create a variable and use that in a one-shot to set a constant.

```

9522 \tl_new:N \g__sys_backend_tl
9523 \__sys_finalise:n
9524 {
9525   \tl_gset:Nx \g__sys_backend_tl
9526   {
9527     \sys_if_engine_xetex:TF
9528     { xdvipdfmx }
9529     {
9530       \sys_if_output_pdf:TF
9531       { pdfmode }
9532       { dvips }
9533     }
9534   }
9535 }
```

If there is a class option set, and recognised, we pick it up: these will over-ride anything set automatically but will themselves be over-written if there is a package option.

```

9536 \__sys_finalise:n
9537 {
9538   \cs_if_exist:NT \@classoptionslist
9539   {
9540     \cs_if_eq:NNF \@classoptionslist \scan_stop:
9541     {
9542       \clist_map_inline:Nn \@classoptionslist
9543       {
9544         \str_case:nnT {#1}
9545         {
9546           { dvipdfmx }
9547           { \tl_gset:Nn \g__sys_backend_tl { dvipdfmx } }
9548           { dvips }
9549           { \tl_gset:Nn \g__sys_backend_tl { dvips } }
9550           { dvisvgm }
9551           { \tl_gset:Nn \g__sys_backend_tl { dvisvgm } }
9552           { pdftex }
9553           { \tl_gset:Nn \g__sys_backend_tl { pdfmode } }
9554           { xetex }
9555           { \tl_gset:Nn \g__sys_backend_tl { xdvipdfmx } }
9556         }
9557         { \clist_remove_all:Nn \@unusedoptionlist {#1} }
9558       }
9559     }
9560   }
9561 }
```

(End definition for `\g__sys_backend_tl`.)

```

9562 \</package>
```


16 l3clist implementation

The following test files are used for this code: *m3clist002*.

9563 `*package`

9564 `\@@=clist`

`\c_empty_clist` An empty comma list is simply an empty token list.

9565 `\cs_new_eq:NN \c_empty_clist \c_empty_tl`

(End definition for `\c_empty_clist`. This variable is documented on page 129.)

`\l__clist_internal_clist` Scratch space for various internal uses. This comma list variable cannot be declared as such because it comes before `\clist_new:N`

9566 `\tl_new:N \l__clist_internal_clist`

(End definition for `\l__clist_internal_clist`.)

`\s__clist_mark` Internal scan marks.

`\s__clist_stop` 9567 `\scan_new:N \s__clist_mark`

9568 `\scan_new:N \s__clist_stop`

(End definition for `\s__clist_mark` and `\s__clist_stop`.)

`__clist_use_none_delimit_by_s_stop:w` Functions to gobble up to a scan mark.

`__clist_use_i_delimit_by_s_stop:nw` 9569 `\cs_new:Npn __clist_use_none_delimit_by_s_stop:w #1 \s__clist_stop { }`

9570 `\cs_new:Npn __clist_use_i_delimit_by_s_stop:nw #1 #2 \s__clist_stop {#1}`

(End definition for `__clist_use_none_delimit_by_s_stop:w` and `__clist_use_i_delimit_by_s_stop:nw`.)

`\q__clist_recursion_tail` Internal recursion quarks.

`\q__clist_recursion_stop` 9571 `\quark_new:N \q__clist_recursion_tail`

9572 `\quark_new:N \q__clist_recursion_stop`

(End definition for `\q__clist_recursion_tail` and `\q__clist_recursion_stop`.)

`__clist_if_recursion_tail_break:nN` Functions to query recursion quarks.

`__clist_if_recursion_tail_stop:n` 9573 `__kernel_quark_new_test:N __clist_if_recursion_tail_break:nN`

9574 `__kernel_quark_new_test:N __clist_if_recursion_tail_stop:n`

(End definition for `__clist_if_recursion_tail_break:nN` and `__clist_if_recursion_tail_stop:n`.)

`__clist_tmp:w` A temporary function for various purposes.

9575 `\cs_new_protected:Npn __clist_tmp:w { }`

(End definition for `__clist_tmp:w`.)

16.1 Removing spaces around items

`__clist_trim_next:w` Called as `\exp:w __clist_trim_next:w \prg_do_nothing: <comma list> ...` it expands to `{<trimmed item>}` where the `<trimmed item>` is the first non-empty result from removing spaces from both ends of comma-delimited items in the `<comma list>`. The `\prg_do_nothing:` marker avoids losing braces. The test for blank items is a somewhat optimized `\tl_if_empty:oTF` construction; if blank, another item is sought, otherwise trim spaces.

```

9576 \cs_new:Npn \__clist_trim_next:w #1 ,
9577 {
9578   \tl_if_empty:oTF { \use_none:nn #1 ? }
9579   { \__clist_trim_next:w \prg_do_nothing: }
9580   { \tl_trim_spaces_apply:oN {#1} \exp_end: }
9581 }
```

(End definition for `__clist_trim_next:w`.)

`__clist_sanitize:n` The auxiliary `__clist_sanitize:Nn` receives a delimiter (`\c_empty_tl` the first time, afterwards a comma) and that item as arguments. Unless we are done with the loop it calls `__clist_wrap_item:w` to unbrace the item (using a comma delimiter is safe since #2 came from removing spaces from an argument delimited by a comma) and possibly re-brace it if needed.

`__clist_sanitize:Nn`

```

9582 \cs_new:Npn \__clist_sanitize:n #1
9583 {
9584   \exp_after:wN \__clist_sanitize:Nn \exp_after:wN \c_empty_tl
9585   \exp:w \__clist_trim_next:w \prg_do_nothing:
9586   #1 , \q__clist_recursion_tail , \q__clist_recursion_stop
9587 }
9588 \cs_new:Npn \__clist_sanitize:Nn #1#2
9589 {
9590   \__clist_if_recursion_tail_stop:n {#2}
9591   #1 \__clist_wrap_item:w #2 ,
9592   \exp_after:wN \__clist_sanitize:Nn \exp_after:wN ,
9593   \exp:w \__clist_trim_next:w \prg_do_nothing:
9594 }
```

(End definition for `__clist_sanitize:n` and `__clist_sanitize:Nn`.)

`__clist_if_wrap:nTF` True if the argument must be wrapped to avoid getting altered by some clist operations.
`__clist_if_wrap:w` That is the case whenever the argument

- starts or end with a space or contains a comma,
- is empty, or
- consists of a single braced group.

All `l3clist` functions go through the same test when they need to determine whether to brace an item, so it is not a problem that this test has false positives such as “`\s__clist_mark ?`”. If the argument starts or end with a space or contains a comma then one of the three arguments of `__clist_if_wrap:w` will have its end delimiter (partly) in one of the three copies of #1 in `__clist_if_wrap:nTF`; this has a knock-on effect meaning that the result of the expansion is not empty; in that case, wrap. Otherwise,

the argument is safe unless it starts with a brace group (or is empty) and it is empty or consists of a single n-type argument.

```

9595 \prg_new_conditional:Npnn \__clist_if_wrap:n #1 { TF }
9596 {
9597   \tl_if_empty:oTF
9598   {
9599     \__clist_if_wrap:w
9600     \s__clist_mark ? #1 ~ \s__clist_mark ? ~ #1
9601     \s__clist_mark , ~ \s__clist_mark #1 ,
9602   }
9603   {
9604     \tl_if_head_is_group:nTF { #1 { } }
9605     {
9606       \tl_if_empty:nTF {#1}
9607       { \prg_return_true: }
9608       {
9609         \tl_if_empty:oTF { \use_none:n #1}
9610         { \prg_return_true: }
9611         { \prg_return_false: }
9612       }
9613     }
9614     { \prg_return_false: }
9615   }
9616   { \prg_return_true: }
9617 }
9618 \cs_new:Npn \__clist_if_wrap:w #1 \s__clist_mark ? ~ #2 ~ \s__clist_mark #3 , { }

```

(End definition for __clist_if_wrap:nTF and __clist_if_wrap:w.)

__clist_wrap_item:w Safe items are put in \exp_not:n, otherwise we put an extra set of braces.

```

9619 \cs_new:Npn \__clist_wrap_item:w #1 ,
9620 { \__clist_if_wrap:nTF {#1} { \exp_not:n { {#1} } } { \exp_not:n {#1} } }

```

(End definition for __clist_wrap_item:w.)

16.2 Allocation and initialisation

\clist_new:N Internally, comma lists are just token lists.

```

\clist_new:c
9621 \cs_new_eq:NN \clist_new:N \tl_new:N
9622 \cs_new_eq:NN \clist_new:c \tl_new:c

```

(End definition for \clist_new:N. This function is documented on page 120.)

\clist_const:Nn Creating and initializing a constant comma list is done by sanitizing all items (stripping spaces and braces).

```

\clist_const:cn
\clist_const:Nx
\clist_const:cx
9623 \cs_new_protected:Npn \clist_const:Nn #1#2
9624 { \tl_const:Nx #1 { \__clist_sanitize:n {#2} } }
9625 \cs_generate_variant:Nn \clist_const:Nn { c , Nx , cx }

```

(End definition for \clist_const:Nn. This function is documented on page 121.)

\clist_clear:N Clearing comma lists is just the same as clearing token lists.

```
\clist_clear:c      9626 \cs_new_eq:NN \clist_clear:N \tl_clear:N
\clist_gclear:N     9627 \cs_new_eq:NN \clist_clear:c \tl_clear:c
\clist_gclear:c     9628 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
                   9629 \cs_new_eq:NN \clist_gclear:c \tl_gclear:c
```

(End definition for \clist_clear:N and \clist_gclear:N. These functions are documented on page 121.)

\clist_clear_new:N Once again a copy from the token list functions.

```
\clist_clear_new:c  9630 \cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N
\clist_gclear_new:N 9631 \cs_new_eq:NN \clist_clear_new:c \tl_clear_new:c
\clist_gclear_new:c 9632 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
                   9633 \cs_new_eq:NN \clist_gclear_new:c \tl_gclear_new:c
```

(End definition for \clist_clear_new:N and \clist_gclear_new:N. These functions are documented on page 121.)

\clist_set_eq:NN Once again, these are simple copies from the token list functions.

```
\clist_set_eq:cN    9634 \cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN
\clist_set_eq:Nc     9635 \cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc
\clist_set_eq:cc     9636 \cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN
\clist_gset_eq:NN    9637 \cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc
\clist_gset_eq:cN    9638 \cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
\clist_gset_eq:Nc    9639 \cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
\clist_gset_eq:cN    9640 \cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
\clist_gset_eq:cc    9641 \cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc
```

(End definition for \clist_set_eq:NN and \clist_gset_eq:NN. These functions are documented on page 121.)

\clist_set_from_seq:NN Setting a comma list from a comma-separated list is done using a simple mapping. Safe items are put in \exp_not:n, otherwise we put an extra set of braces. The first comma must be removed, except in the case of an empty comma-list.

```
\clist_set_from_seq:cN 9642 \cs_new_protected:Npn \clist_set_from_seq:NN
\clist_set_from_seq:Nc 9643 { \__clist_set_from_seq:NNNN \clist_clear:N \tl_set:Nx }
\clist_set_from_seq:cc 9644 \cs_new_protected:Npn \clist_gset_from_seq:NN
\clist_gset_from_seq:cN 9645 { \__clist_set_from_seq:NNNN \clist_gclear:N \tl_gset:Nx }
\clist_gset_from_seq:Nc 9646 \cs_new_protected:Npn \__clist_set_from_seq:NNNN #1#2#3#4
\clist_gset_from_seq:cc 9647 {
\__clist_set_from_seq:NNNN 9648 \seq_if_empty:NTF #4
\__clist_set_from_seq:n 9649 { #1 #3 }
9650 {
9651 #2 #3
9652 {
9653 \exp_after:wN \use_none:n \exp:w \exp_end_continue_f:w
9654 \seq_map_function:NN #4 \__clist_set_from_seq:n
9655 }
9656 }
9657 }
9658 \cs_new:Npn \__clist_set_from_seq:n #1
9659 {
9660 ,
9661 \__clist_if_wrap:NTF {#1}
9662 { \exp_not:n { {#1} } }
```

```

9663     { \exp_not:n {#1} }
9664   }
9665   \cs_generate_variant:Nn \clist_set_from_seq:NN { Nc }
9666   \cs_generate_variant:Nn \clist_set_from_seq:NN { c , cc }
9667   \cs_generate_variant:Nn \clist_gset_from_seq:NN { Nc }
9668   \cs_generate_variant:Nn \clist_gset_from_seq:NN { c , cc }

```

(End definition for `\clist_set_from_seq:NN` and others. These functions are documented on page 121.)

```

\clist_concat:NNN Concatenating comma lists is not quite as easy as it seems, as there needs to be the
\clist_concat:ccc correct addition of a comma to the output. So a little work to do.
\clist_gconcat:NNN
\clist_gconcat:ccc
\__clist_concat:NNNN
9669   \cs_new_protected:Npn \clist_concat:NNN
9670     { \__clist_concat:NNNN \tl_set:Nx }
9671   \cs_new_protected:Npn \clist_gconcat:NNN
9672     { \__clist_concat:NNNN \tl_gset:Nx }
9673   \cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4
9674     {
9675       #1 #2
9676       {
9677         \exp_not:o #3
9678         \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
9679         \exp_not:o #4
9680       }
9681     }
9682   \cs_generate_variant:Nn \clist_concat:NNN { ccc }
9683   \cs_generate_variant:Nn \clist_gconcat:NNN { ccc }

```

(End definition for `\clist_concat:NNN`, `\clist_gconcat:NNN`, and `__clist_concat:NNNN`. These functions are documented on page 121.)

```

\clist_if_exist_p:N Copies of the cs functions defined in l3basics.
\clist_if_exist_p:c
\clist_if_exist:NTF
\clist_if_exist:cTF
9684   \prg_new_eq_conditional:NNn \clist_if_exist:N \cs_if_exist:N
9685     { TF , T , F , p }
9686   \prg_new_eq_conditional:NNn \clist_if_exist:c \cs_if_exist:c
9687     { TF , T , F , p }

```

(End definition for `\clist_if_exist:NTF`. This function is documented on page 121.)

16.3 Adding data to comma lists

```

\clist_set:Nn
\clist_set:NV
\clist_set:No
\clist_set:Nx
\clist_set:cn
\clist_set:cV
\clist_set:co
\clist_set:cx
9688   \cs_new_protected:Npn \clist_set:Nn #1#2
9689     { \tl_set:Nx #1 { \__clist_sanitiz:n {#2} } }
9690   \cs_new_protected:Npn \clist_gset:Nn #1#2
9691     { \tl_gset:Nx #1 { \__clist_sanitiz:n {#2} } }
9692   \cs_generate_variant:Nn \clist_set:Nn { NV , No , Nx , c , cV , co , cx }
9693   \cs_generate_variant:Nn \clist_gset:Nn { NV , No , Nx , c , cV , co , cx }

```

(End definition for `\clist_set:Nn` and `\clist_gset:Nn`. These functions are documented on page 122.)

```

\clist_gset:Nn
\clist_put_left:NV
\clist_put_left:Nx
\clist_put_left:cn
\clist_put_left:cV
\clist_put_left:co
\clist_put_left:cx
\clist_gput_left:Nn

```

Everything is based on concatenation after storing in `\l__clist_internal_clist`. This avoids having to worry here about space-trimming and so on.

```

9694   \cs_new_protected:Npn \clist_put_left:Nn
9695     { \__clist_put_left:NNNN \clist_concat:NNN \clist_set:Nn }
9696   \cs_new_protected:Npn \clist_gput_left:Nn

```

```

9697 { \_clist_put_left:NNN \clist_gconcat:NNN \clist_set:Nn }
9698 \cs_new_protected:Npn \_clist_put_left:NNNn #1#2#3#4
9699 {
9700   #2 \l__clist_internal_clist {#4}
9701   #1 #3 \l__clist_internal_clist #3
9702 }
9703 \cs_generate_variant:Nn \clist_put_left:Nn { NV , No , Nx }
9704 \cs_generate_variant:Nn \clist_put_left:Nn { c , cV , co , cx }
9705 \cs_generate_variant:Nn \clist_gput_left:Nn { NV , No , Nx }
9706 \cs_generate_variant:Nn \clist_gput_left:Nn { c , cV , co , cx }

```

(End definition for `\clist_put_left:Nn`, `\clist_gput_left:Nn`, and `_clist_put_left:NNNn`. These functions are documented on page 122.)

```

\_clist_put_right:Nn
\clist_put_right:NV
\clist_put_right:No
\clist_put_right:Nx
\clist_put_right:cn
\clist_put_right:cV
\clist_put_right:co
\clist_put_right:cx
\_clist_gput_right:Nn
\clist_gput_right:NV
\clist_gput_right:No
\clist_gput_right:Nx
\clist_gput_right:cn
\clist_gput_right:cV
\clist_gput_right:co
\clist_gput_right:cx
\_clist_put_right:NNNn

```

```

9707 \cs_new_protected:Npn \clist_put_right:Nn
9708 { \_clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
9709 \cs_new_protected:Npn \clist_gput_right:Nn
9710 { \_clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
9711 \cs_new_protected:Npn \_clist_put_right:NNNn #1#2#3#4
9712 {
9713   #2 \l__clist_internal_clist {#4}
9714   #1 #3 #3 \l__clist_internal_clist
9715 }
9716 \cs_generate_variant:Nn \clist_put_right:Nn { NV , No , Nx }
9717 \cs_generate_variant:Nn \clist_put_right:Nn { c , cV , co , cx }
9718 \cs_generate_variant:Nn \clist_gput_right:Nn { NV , No , Nx }
9719 \cs_generate_variant:Nn \clist_gput_right:Nn { c , cV , co , cx }

```

(End definition for `\clist_put_right:Nn`, `\clist_gput_right:Nn`, and `_clist_put_right:NNNn`. These functions are documented on page 122.)

16.4 Comma lists as stacks

`\clist_get:NN` Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma. No need to trim spaces as comma-list *variables* are assumed to have “cleaned-up” items. (Note that grabbing a comma-delimited item removes an outer pair of braces if present, exactly as needed to uncover the underlying item.)

```

9720 \cs_new_protected:Npn \clist_get:NN #1#2
9721 {
9722   \if_meaning:w #1 \c_empty_clist
9723     \tl_set:Nn #2 { \q_no_value }
9724   \else:
9725     \exp_after:wN \_clist_get:wN #1 , \s__clist_stop #2
9726   \fi:
9727 }
9728 \cs_new_protected:Npn \_clist_get:wN #1 , #2 \s__clist_stop #3
9729 { \tl_set:Nn #3 {#1} }
9730 \cs_generate_variant:Nn \clist_get:NN { c }

```

(End definition for `\clist_get:NN` and `_clist_get:wN`. This function is documented on page 127.)

`\clist_pop:NN` An empty clist leads to `\q_no_value`, otherwise grab until the first comma and assign to the variable. The second argument of `_clist_pop:wwNNN` is a comma list ending in a

```

\clist_pop:cn
\_clist_gpop:NN
\clist_gpop:cn
\_clist_pop:NNN
\_clist_pop:wwNNN
\_clist_pop:wN

```

comma and `\s__clist_mark`, unless the original clist contained exactly one item: then the argument is just `\s__clist_mark`. The next auxiliary picks either `\exp_not:n` or `\use_none:n` as #2, ensuring that the result can safely be an empty comma list.

```

9731 \cs_new_protected:Npn \clist_pop:NN
9732 { \__clist_pop:NNN \tl_set:Nx }
9733 \cs_new_protected:Npn \clist_gpop:NN
9734 { \__clist_pop:NNN \tl_gset:Nx }
9735 \cs_new_protected:Npn \__clist_pop:NNN #1#2#3
9736 {
9737   \if_meaning:w #2 \c_empty_clist
9738     \tl_set:Nn #3 { \q_no_value }
9739   \else:
9740     \exp_after:wN \__clist_pop:wwNNN #2 , \s__clist_mark \s__clist_stop #1#2#3
9741   \fi:
9742 }
9743 \cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \s__clist_stop #3#4#5
9744 {
9745   \tl_set:Nn #5 {#1}
9746   #3 #4
9747   {
9748     \__clist_pop:wN \prg_do_nothing:
9749     #2 \exp_not:o
9750     , \s__clist_mark \use_none:n
9751     \s__clist_stop
9752   }
9753 }
9754 \cs_new:Npn \__clist_pop:wN #1 , \s__clist_mark #2 #3 \s__clist_stop { #2 {#1} }
9755 \cs_generate_variant:Nn \clist_pop:NN { c }
9756 \cs_generate_variant:Nn \clist_gpop:NN { c }

```

(End definition for `\clist_pop:NN` and others. These functions are documented on page 127.)

```

\clist_get:NNTF The same, as branching code: very similar to the above.
\clist_get:cNTF 9757 \prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
\clist_pop:NNTF 9758 {
\clist_pop:cNTF 9759   \if_meaning:w #1 \c_empty_clist
\clist_gpop:NNTF 9760   \prg_return_false:
\clist_gpop:cNTF 9761   \else:
\__clist_pop_TF:NNN 9762     \exp_after:wN \__clist_get:wN #1 , \s__clist_stop #2
9763     \prg_return_true:
9764   \fi:
9765 }
9766 \prg_generate_conditional_variant:Nnn \clist_get:NN { c } { T , F , TF }
9767 \prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
9768 { \__clist_pop_TF:NNN \tl_set:Nx #1 #2 }
9769 \prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
9770 { \__clist_pop_TF:NNN \tl_gset:Nx #1 #2 }
9771 \cs_new_protected:Npn \__clist_pop_TF:NNN #1#2#3
9772 {
9773   \if_meaning:w #2 \c_empty_clist
9774     \prg_return_false:
9775   \else:
9776     \exp_after:wN \__clist_pop:wwNNN #2 , \s__clist_mark \s__clist_stop #1#2#3
9777     \prg_return_true:

```

```

9778     \fi:
9779   }
9780 \prg_generate_conditional_variant:Nnn \clist_pop:NN { c } { T , F , TF }
9781 \prg_generate_conditional_variant:Nnn \clist_gpop:NN { c } { T , F , TF }

```

(End definition for `\clist_get:NNTF` and others. These functions are documented on page 127.)

```

\clist_push:Nn Pushing to a comma list is the same as adding on the left.
\clist_push:NV 9782 \cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn
\clist_push:No 9783 \cs_new_eq:NN \clist_push:NV \clist_put_left:NV
\clist_push:Nx 9784 \cs_new_eq:NN \clist_push:No \clist_put_left:No
\clist_push:cn 9785 \cs_new_eq:NN \clist_push:Nx \clist_put_left:Nx
\clist_push:cV 9786 \cs_new_eq:NN \clist_push:cn \clist_put_left:cn
\clist_push:co 9787 \cs_new_eq:NN \clist_push:cV \clist_put_left:cV
\clist_push:cx 9788 \cs_new_eq:NN \clist_push:co \clist_put_left:co
\clist_gpush:Nn 9789 \cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
\clist_gpush:NV 9790 \cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV
\clist_gpush:No 9791 \cs_new_eq:NN \clist_gpush:No \clist_gput_left:No
\clist_gpush:Nx 9792 \cs_new_eq:NN \clist_gpush:Nx \clist_gput_left:Nx
\clist_gpush:cn 9793 \cs_new_eq:NN \clist_gpush:cn \clist_gput_left:cn
\clist_gpush:cV 9794 \cs_new_eq:NN \clist_gpush:cV \clist_gput_left:cV
\clist_gpush:co 9795 \cs_new_eq:NN \clist_gpush:co \clist_gput_left:co
\clist_gpush:cx 9796 \cs_new_eq:NN \clist_gpush:cx \clist_gput_left:cx

```

(End definition for `\clist_push:Nn` and `\clist_gpush:Nn`. These functions are documented on page 128.)

16.5 Modifying comma lists

```

\l__clist_internal_remove_clist An internal comma list and a sequence for the removal routines.
\l__clist_internal_remove_seq

```

```

9798 \clist_new:N \l__clist_internal_remove_clist
9799 \seq_new:N \l__clist_internal_remove_seq

```

(End definition for `\l__clist_internal_remove_clist` and `\l__clist_internal_remove_seq`.)

```

\clist_remove_duplicates:N Removing duplicates means making a new list then copying it.
\clist_remove_duplicates:c 9800 \cs_new_protected:Npn \clist_remove_duplicates:N
\clist_gremove_duplicates:N 9801 { \__clist_remove_duplicates:NN \clist_set_eq:NN }
\clist_gremove_duplicates:c 9802 \cs_new_protected:Npn \clist_gremove_duplicates:N
\__clist_remove_duplicates:NN 9803 { \__clist_remove_duplicates:NN \clist_gset_eq:NN }
9804 \cs_new_protected:Npn \__clist_remove_duplicates:NN #1#2
9805 {
9806   \clist_clear:N \l__clist_internal_remove_clist
9807   \clist_map_inline:Nn #2
9808   {
9809     \clist_if_in:NnF \l__clist_internal_remove_clist {##1}
9810     { \clist_put_right:Nn \l__clist_internal_remove_clist {##1} }
9811   }
9812   #1 #2 \l__clist_internal_remove_clist
9813 }
9814 \cs_generate_variant:Nn \clist_remove_duplicates:N { c }
9815 \cs_generate_variant:Nn \clist_gremove_duplicates:N { c }

```


(End definition for `\clist_remove_duplicates:N`, `\clist_gremove_duplicates:N`, and `__clist_remove_duplicates:NN`. These functions are documented on page 123.)

`\clist_remove_all:Nn` The method used here for safe items is very similar to `\tl_replace_all:Nnn`. However, if the item contains commas or leading/trailing spaces, or is empty, or consists of a single brace group, we know that it can only appear within braces so the code would fail; instead just convert to a sequence and do the removal with `l3seq` code (it involves somewhat elaborate code to do most of the work expandably but the final token list comparisons non-expandably).

`\clist_remove_all:cn`

`\clist_gremove_all:Nn`

`\clist_gremove_all:cn`

`__clist_remove_all:NNNn`

`__clist_remove_all:w`

`__clist_remove_all:`

For “safe” items, build a function delimited by the `<item>` that should be removed, surrounded with commas, and call that function followed by the expanded comma list, and another copy of the `<item>`. The loop is controlled by the argument grabbed by `__clist_remove_all:w`: when the item was found, the `\s__clist_mark` delimiter used is the one inserted by `__clist_tmp:w`, and `__clist_use_none_delimit_by_s_stop:w` is deleted. At the end, the final `<item>` is grabbed, and the argument of `__clist_tmp:w` contains `\s__clist_mark`: in that case, `__clist_remove_all:w` removes the second `\s__clist_mark` (inserted by `__clist_tmp:w`), and lets `__clist_use_none_delimit_by_s_stop:w` act.

No brace is lost because items are always grabbed with a leading comma. The result of the first assignment has an extra leading comma, which we remove in a second assignment. Two exceptions: if the clist lost all of its elements, the result is empty, and we shouldn’t remove anything; if the clist started up empty, the first step happens to turn it into a single comma, and the second step removes it.

```

9816 \cs_new_protected:Npn \clist_remove_all:Nn
9817   { \__clist_remove_all:NNNn \clist_set_from_seq:NN \tl_set:Nx }
9818 \cs_new_protected:Npn \clist_gremove_all:Nn
9819   { \__clist_remove_all:NNNn \clist_gset_from_seq:NN \tl_gset:Nx }
9820 \cs_new_protected:Npn \__clist_remove_all:NNNn #1#2#3#4
9821   {
9822     \__clist_if_wrap:nTF {#4}
9823     {
9824       \seq_set_from_clist:NN \l__clist_internal_remove_seq #3
9825       \seq_remove_all:Nn \l__clist_internal_remove_seq {#4}
9826       #1 #3 \l__clist_internal_remove_seq
9827     }
9828     {
9829       \cs_set:Npn \__clist_tmp:w ##1 , #4 ,
9830       {
9831         ##1
9832         , \s__clist_mark , \__clist_use_none_delimit_by_s_stop:w ,
9833         \__clist_remove_all:
9834       }
9835       #2 #3
9836       {
9837         \exp_after:wN \__clist_remove_all:
9838         #3 , \s__clist_mark , #4 , \s__clist_stop
9839       }
9840       \clist_if_empty:NF #3
9841       {
9842         #2 #3
9843         {
9844           \exp_args:No \exp_not:o

```

```

9845         { \exp_after:wN \use_none:n #3 }
9846     }
9847 }
9848 }
9849 }
9850 \cs_new:Npn \__clist_remove_all:
9851 { \exp_after:wN \__clist_remove_all:w \__clist_tmp:w , }
9852 \cs_new:Npn \__clist_remove_all:w #1 , \s__clist_mark , #2 , { \exp_not:n {#1} }
9853 \cs_generate_variant:Nn \clist_remove_all:Nn { c }
9854 \cs_generate_variant:Nn \clist_gremove_all:Nn { c }

```

(End definition for `\clist_remove_all:Nn` and others. These functions are documented on page 123.)

`\clist_reverse:N` Use `\clist_reverse:n` in an x-expanding assignment. The extra work that `\clist_reverse:n` does to preserve braces and spaces would not be needed for the well-controlled case of N-type comma lists, but the slow-down is not too bad.

```

\clist_reverse:c
\clist_greverse:N
\clist_greverse:c
9855 \cs_new_protected:Npn \clist_reverse:N #1
9856 { \tl_set:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
9857 \cs_new_protected:Npn \clist_greverse:N #1
9858 { \tl_gset:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
9859 \cs_generate_variant:Nn \clist_reverse:N { c }
9860 \cs_generate_variant:Nn \clist_greverse:N { c }

```

(End definition for `\clist_reverse:N` and `\clist_greverse:N`. These functions are documented on page 123.)

`\clist_reverse:n` The reversed token list is built one item at a time, and stored between `\s__clist_stop` and `\s__clist_mark`, in the form of ? followed by zero or more instances of “ $\langle item \rangle$,”. We start from a comma list “ $\langle item_1 \rangle, \dots, \langle item_n \rangle$ ”. During the loop, the auxiliary `__clist_reverse:wwNww` receives “ $\langle item_i \rangle$ ” as #1, “ $\langle item_{i+1} \rangle, \dots, \langle item_n \rangle$ ” as #2, `__clist_reverse:wwNww` as #3, what remains until `\s__clist_stop` as #4, and “ $\langle item_{i-1} \rangle, \dots, \langle item_1 \rangle$,” as #5. The auxiliary moves #1 just before #5, with a comma, and calls itself (#3). After the last item is moved, `__clist_reverse:wwNww` receives “`\s__clist_mark __clist_reverse:wwNww !`” as its argument #1, thus `__clist_reverse_end:ww` as its argument #3. This second auxiliary cleans up until the marker !, removes the trailing comma (introduced when the first item was moved after `\s__clist_stop`), and leaves its argument #1 within `\exp_not:n`. There is also a need to remove a leading comma, hence `\exp_not:o` and `\use_none:n`.

```

9861 \cs_new:Npn \clist_reverse:n #1
9862 {
9863     \__clist_reverse:wwNww ? #1 ,
9864     \s__clist_mark \__clist_reverse:wwNww ! ,
9865     \s__clist_mark \__clist_reverse_end:ww
9866     \s__clist_stop ? \s__clist_mark
9867 }
9868 \cs_new:Npn \__clist_reverse:wwNww
9869 #1 , #2 \s__clist_mark #3 #4 \s__clist_stop ? #5 \s__clist_mark
9870 { #3 ? #2 \s__clist_mark #3 #4 \s__clist_stop #1 , #5 \s__clist_mark }
9871 \cs_new:Npn \__clist_reverse_end:ww #1 ! #2 , \s__clist_mark
9872 { \exp_not:o { \use_none:n #2 } }

```

(End definition for `\clist_reverse:n`, `__clist_reverse:wwNww`, and `__clist_reverse_end:ww`. This function is documented on page 123.)

`\clist_sort:Nn` Implemented in `l3sort`.
`\clist_sort:cn`
`\clist_gsort:Nn` (End definition for `\clist_sort:Nn` and `\clist_gsort:Nn`. These functions are documented on page 123.)
`\clist_gsort:cn`

16.6 Comma list conditionals

`\clist_if_empty_p:N` Simple copies from the token list variable material.
`\clist_if_empty_p:c` 9873 `\prg_new_eq_conditional:Nn \clist_if_empty:N \tl_if_empty:N`
`\clist_if_empty:N \underline{TF}` 9874 `{ p , T , F , TF }`
`\clist_if_empty:c \underline{TF}` 9875 `\prg_new_eq_conditional:Nn \clist_if_empty:c \tl_if_empty:c`
9876 `{ p , T , F , TF }`

(End definition for `\clist_if_empty:N \underline{TF}` . This function is documented on page 124.)

`\clist_if_empty_p:n` As usual, we insert a token (here ?) before grabbing any argument: this avoids losing braces. The argument of `\tl_if_empty:oTF` is empty if #1 is ? followed by blank spaces (besides, this particular variant of the emptiness test is optimized). If the item of the comma list is blank, grab the next one. As soon as one item is non-blank, exit: the second auxiliary grabs `\prg_return_false:` as #2, unless every item in the comma list was blank and the loop actually got broken by the trailing `\s__clist_mark \prg_return_false:` item.

```
9877 \prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
9878 {
9879   \__clist_if_empty_n:w ? #1
9880   , \s__clist_mark \prg_return_false:
9881   , \s__clist_mark \prg_return_true:
9882   \s__clist_stop
9883 }
9884 \cs_new:Npn \__clist_if_empty_n:w #1 ,
9885 {
9886   \tl_if_empty:oTF { \use_none:nn #1 ? }
9887   { \__clist_if_empty_n:w ? }
9888   { \__clist_if_empty_n:wNw }
9889 }
9890 \cs_new:Npn \__clist_if_empty_n:wNw #1 \s__clist_mark #2#3 \s__clist_stop {#2}
```

(End definition for `\clist_if_empty:n \underline{TF}` , `__clist_if_empty_n:w`, and `__clist_if_empty_n:wNw`. This function is documented on page 124.)

`\clist_if_in:Nn \underline{TF}` For “safe” items, we simply surround the comma list, and the item, with commas, then use the same code as for `\tl_if_in:Nn`. For “unsafe” items we follow the same route as `\seq_if_in:Nn`, mapping through the list a comparison function. If found, return `true` and remove `\prg_return_false:`.
`\clist_if_in:NV \underline{TF}`
`\clist_if_in:No \underline{TF}`
`\clist_if_in:cn \underline{TF}`
`\clist_if_in:cV \underline{TF}` 9891 `\prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T , F , TF }`
`\clist_if_in:co \underline{TF}` 9892 `{`
`\clist_if_in:nn \underline{TF}` 9893 `\exp_args:No __clist_if_in_return:nnN #1 {#2} #1`
`\clist_if_in:nV \underline{TF}` 9894 `}`
`\clist_if_in:no \underline{TF}` 9895 `\prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T , F , TF }`
`__clist_if_in_return:nnN` 9896 `{`
9897 `\clist_set:Nn \l__clist_internal_clist {#1}`
9898 `\exp_args:No __clist_if_in_return:nnN \l__clist_internal_clist {#2}`
9899 `\l__clist_internal_clist`
9900 `}`

```

9901 \cs_new_protected:Npn \__clist_if_in_return:nnN #1#2#3
9902 {
9903   \__clist_if_wrap:nTF {#2}
9904   {
9905     \cs_set:Npx \__clist_tmp:w ##1
9906     {
9907       \exp_not:N \tl_if_eq:nnT {##1}
9908       \exp_not:n
9909       {
9910         {#2}
9911         { \clist_map_break:n { \prg_return_true: \use_none:n } }
9912       }
9913     }
9914     \clist_map_function:NN #3 \__clist_tmp:w
9915     \prg_return_false:
9916   }
9917   {
9918     \cs_set:Npn \__clist_tmp:w ##1 ,#2, { }
9919     \tl_if_empty:oTF
9920     { \__clist_tmp:w ,#1, {} {} ,#2, }
9921     { \prg_return_false: } { \prg_return_true: }
9922   }
9923 }
9924 \prg_generate_conditional_variant:Nnn \clist_if_in:Nn
9925 { NV , No , c , cV , co } { T , F , TF }
9926 \prg_generate_conditional_variant:Nnn \clist_if_in:nn
9927 { nV , no } { T , F , TF }

```

(End definition for `\clist_if_in:NnTF`, `\clist_if_in:nnTF`, and `__clist_if_in_return:nnN`. These functions are documented on page 124.)

16.7 Mapping to comma lists

`\clist_map_function:NN` If the variable is empty, the mapping is skipped (otherwise, that comma-list would be seen as consisting of one empty item). Then loop over the comma-list, grabbing one comma-delimited item at a time. The end is marked by `\q__clist_recursion_tail`. The auxiliary function `__clist_map_function:Nw` is also used in `\clist_map_inline:Nn`.

```

9928 \cs_new:Npn \clist_map_function:NN #1#2
9929 {
9930   \clist_if_empty:NF #1
9931   {
9932     \exp_last_unbraced:NNo \__clist_map_function:Nw #2 #1
9933     , \q__clist_recursion_tail ,
9934     \prg_break_point:Nn \clist_map_break: { }
9935   }
9936 }
9937 \cs_new:Npn \__clist_map_function:Nw #1#2 ,
9938 {
9939   \__clist_if_recursion_tail_break:nN {#2} \clist_map_break:
9940   #1 {#2}
9941   \__clist_map_function:Nw #1
9942 }
9943 \cs_generate_variant:Nn \clist_map_function:NN { c }

```

(End definition for `\clist_map_function:Nn` and `__clist_map_function:Nw`. This function is documented on page 124.)

```

\clist_map_function:nN The n-type mapping function is a bit more awkward, since spaces must be trimmed from
\__clist_map_function_n:Nn each item. Space trimming is again based on \__clist_trim_next:w. The auxiliary
\__clist_map_unbrace:Nw \__clist_map_function_n:Nn receives as arguments the function, and the next non-
empty item (after space trimming but before brace removal). One level of braces is
removed by \__clist_map_unbrace:Nw.
9944 \cs_new:Npn \clist_map_function:nN #1#2
9945 {
9946   \exp_after:wN \__clist_map_function_n:Nn \exp_after:wN #2
9947   \exp:w \__clist_trim_next:w \prg_do_nothing: #1 , \q__clist_recursion_tail ,
9948   \prg_break_point:Nn \clist_map_break: { }
9949 }
9950 \cs_new:Npn \__clist_map_function_n:Nn #1 #2
9951 {
9952   \__clist_if_recursion_tail_break:nN {#2} \clist_map_break:
9953   \__clist_map_unbrace:Nw #1 #2,
9954   \exp_after:wN \__clist_map_function_n:Nn \exp_after:wN #1
9955   \exp:w \__clist_trim_next:w \prg_do_nothing:
9956 }
9957 \cs_new:Npn \__clist_map_unbrace:Nw #1 #2, { #1 {#2} }

```

(End definition for `\clist_map_function:nN`, `__clist_map_function_n:Nn`, and `__clist_map_unbrace:Nw`. This function is documented on page 124.)

`\clist_map_inline:Nn` Inline mapping is done by creating a suitable function “on the fly”: this is done globally
`\clist_map_inline:cn` to avoid any issues with \TeX ’s groups. We use a different function for each level of
`\clist_map_inline:nn` nesting.

Since the mapping is non-expandable, we can perform the space-trimming needed by the `n` version simply by storing the comma-list in a variable. We don’t need a different comma-list for each nesting level: the comma-list is expanded before the mapping starts.

```

9958 \cs_new_protected:Npn \clist_map_inline:Nn #1#2
9959 {
9960   \clist_if_empty:NF #1
9961   {
9962     \int_gincr:N \g__kernel_prg_map_int
9963     \cs_gset_protected:cpn
9964     { \__clist_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
9965     \exp_last_unbraced:Nco \__clist_map_function:Nw
9966     { \__clist_map_ \int_use:N \g__kernel_prg_map_int :w }
9967     #1 , \q__clist_recursion_tail ,
9968     \prg_break_point:Nn \clist_map_break:
9969     { \int_gdecr:N \g__kernel_prg_map_int }
9970   }
9971 }
9972 \cs_new_protected:Npn \clist_map_inline:nn #1
9973 {
9974   \clist_set:Nn \l__clist_internal_clist {#1}
9975   \clist_map_inline:Nn \l__clist_internal_clist
9976 }
9977 \cs_generate_variant:Nn \clist_map_inline:Nn { c }

```

(End definition for `\clist_map_inline:Nn` and `\clist_map_inline:nn`. These functions are documented on page 125.)

`\clist_map_variable:NNn` As for other comma-list mappings, filter out the case of an empty list. Same approach as
`\clist_map_variable:cNn` `\clist_map_function:Nn`, additionally we store each item in the given variable. As for
`\clist_map_variable:nNn` inline mappings, space trimming for the `n` variant is done by storing the comma list in
`__clist_map_variable:Nnw` a variable. The quark test is done before assigning the item to the variable: this avoids
storing a quark which the user wouldn't expect. The strange `\use:n` avoids unlikely
problems when `#2` would contain `\q__clist_recursion_stop`.

```

9978 \cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
9979 {
9980   \clist_if_empty:NF #1
9981   {
9982     \exp_args:Nno \use:nn
9983     { \__clist_map_variable:Nnw #2 {#3} }
9984     #1
9985     , \q__clist_recursion_tail , \q__clist_recursion_stop
9986     \prg_break_point:Nn \clist_map_break: { }
9987   }
9988 }
9989 \cs_new_protected:Npn \clist_map_variable:nNn #1
9990 {
9991   \clist_set:Nn \l__clist_internal_clist {#1}
9992   \clist_map_variable:NNn \l__clist_internal_clist
9993 }
9994 \cs_new_protected:Npn \__clist_map_variable:Nnw #1#2#3,
9995 {
9996   \__clist_if_recursion_tail_stop:n {#3}
9997   \tl_set:Nn #1 {#3}
9998   \use:n {#2}
9999   \__clist_map_variable:Nnw #1 {#2}
10000 }
10001 \cs_generate_variant:Nn \clist_map_variable:NNn { c }

```

(End definition for `\clist_map_variable:NNn`, `\clist_map_variable:nNn`, and `__clist_map_variable:Nnw`.
These functions are documented on page 125.)

`\clist_map_break:` The break statements use the general `\prg_map_break:Nn` mechanism.
`\clist_map_break:n`

```

10002 \cs_new:Npn \clist_map_break:
10003 { \prg_map_break:Nn \clist_map_break: { } }
10004 \cs_new:Npn \clist_map_break:n
10005 { \prg_map_break:Nn \clist_map_break: }

```

(End definition for `\clist_map_break:` and `\clist_map_break:n`. These functions are documented on
page 125.)

`\clist_count:N` Counting the items in a comma list is done using the same approach as for other token
`\clist_count:c` count functions: turn each entry into a `+1` then use integer evaluation to actually do the
`\clist_count:n` mathematics. In the case of an `n`-type comma-list, we could of course use `\clist_map_`
`__clist_count:n` `function:nN`, but that is very slow, because it carefully removes spaces. Instead, we loop
`__clist_count:w` manually, and skip blank items (but not `{}`, hence the extra spaces).

```

10006 \cs_new:Npn \clist_count:N #1
10007 {
10008   \int_eval:n
10009   {
10010     0
10011     \clist_map_function:NN #1 \__clist_count:n

```

```

10012     }
10013   }
10014   \cs_generate_variant:Nn \clist_count:N { c }
10015   \cs_new:Npx \clist_count:n #1
10016   {
10017     \exp_not:N \int_eval:n
10018     {
10019       0
10020       \exp_not:N \__clist_count:w \c_space_tl
10021       #1 \exp_not:n { , \q__clist_recursion_tail , \q__clist_recursion_stop }
10022     }
10023   }
10024   \cs_new:Npn \__clist_count:n #1 { + 1 }
10025   \cs_new:Npx \__clist_count:w #1 ,
10026   {
10027     \exp_not:n { \exp_args:Nf \__clist_if_recursion_tail_stop:n } {#1}
10028     \exp_not:N \tl_if_blank:nF {#1} { + 1 }
10029     \exp_not:N \__clist_count:w \c_space_tl
10030   }

```

(End definition for `\clist_count:N` and others. These functions are documented on page 126.)

16.8 Using comma lists

`\clist_use:Nnnn` First check that the variable exists. Then count the items in the comma list. If it has none, output nothing. If it has one item, output that item, brace stripped (note that space-trimming has already been done when the comma list was assigned). If it has two, place the *<separator between two>* in the middle.

Otherwise, `__clist_use:nwwwnwn` takes the following arguments; 1: a *<separator>*, 2, 3, 4: three items from the comma list (or quarks), 5: the rest of the comma list, 6: a *<continuation>* function (`use_ii` or `use_iii` with its *<separator>* argument), 7: junk, and 8: the temporary result, which is built in a brace group following `\q__clist_stop`. The *<separator>* and the first of the three items are placed in the result, then we use the *<continuation>*, placing the remaining two items after it. When we begin this loop, the three items really belong to the comma list, the first `\q__clist_mark` is taken as a delimiter to the `use_ii` function, and the continuation is `use_ii` itself. When we reach the last two items of the original token list, `\q__clist_mark` is taken as a third item, and now the second `\q__clist_mark` serves as a delimiter to `use_ii`, switching to the other *<continuation>*, `use_iii`, which uses the *<separator between final two>*.

```

10031   \cs_new:Npn \clist_use:Nnnn #1#2#3#4
10032   {
10033     \clist_if_exist:NTF #1
10034     {
10035       \int_case:nnF { \clist_count:N #1 }
10036       {
10037         { 0 } { }
10038         { 1 } { \exp_after:wN \__clist_use:wwn #1 , , { } }
10039         { 2 } { \exp_after:wN \__clist_use:wwn #1 , {#2} }
10040       }
10041       {
10042         \exp_after:wN \__clist_use:nwwwnwn
10043         \exp_after:wN { \exp_after:wN } #1 ,
10044         \s__clist_mark , { \__clist_use:nwwwnwn {#3} }

```

```

10045         \s__clist_mark , { \__clist_use:nwn {#4} }
10046         \s__clist_stop { }
10047     }
10048 }
10049 {
10050     \__kernel_msg_expandable_error:nnn
10051     { kernel } { bad-variable } {#1}
10052 }
10053 }
10054 \cs_generate_variant:Nn \clist_use:Nnnn { c }
10055 \cs_new:Npn \__clist_use:wn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
10056 \cs_new:Npn \__clist_use:nwnwn #1#2 , #3 , #4 , #5 \s__clist_mark , #6#7 \s__clist_stop #8
10057     { #6 {#3} , {#4} , #5 \s__clist_mark , {#6} #7 \s__clist_stop { #8 #1 #2 } }
10058 \cs_new:Npn \__clist_use:nwn #1#2 , #3 \s__clist_stop #4
10059     { \exp_not:n { #4 #1 #2 } }
10060 \cs_new:Npn \clist_use:Nn #1#2
10061     { \clist_use:Nnnn #1 {#2} {#2} {#2} }
10062 \cs_generate_variant:Nn \clist_use:Nn { c }
10063

```

(End definition for `\clist_use:Nnnn` and others. These functions are documented on page 126.)

16.9 Using a single item

`\clist_item:Nn` To avoid needing to test the end of the list at each step, we first compute the $\langle length \rangle$ of the list. If the item number is 0, less than $-\langle length \rangle$, or more than $\langle length \rangle$, the result is empty. If it is negative, but not less than $-\langle length \rangle$, add $\langle length \rangle + 1$ to the item number before performing the loop. The loop itself is very simple, return the item if the counter reached 1, otherwise, decrease the counter and repeat.

`\clist_item:cn`

`__clist_item:nnnN`

`__clist_item:ffoN`

`__clist_item:ffnN`

`__clist_item_N_loop:nw`

```

10064 \cs_new:Npn \clist_item:Nn #1#2
10065     {
10066         \__clist_item:ffoN
10067         { \clist_count:N #1 }
10068         { \int_eval:n {#2} }
10069         #1
10070         \__clist_item_N_loop:nw
10071     }
10072 \cs_new:Npn \__clist_item:nnnN #1#2#3#4
10073     {
10074         \int_compare:nNnTF {#2} < 0
10075         {
10076             \int_compare:nNnTF {#2} < { - #1 }
10077             { \__clist_use_none_delimit_by_s_stop:w }
10078             { \exp_args:Nf #4 { \int_eval:n { #2 + 1 + #1 } } }
10079         }
10080         {
10081             \int_compare:nNnTF {#2} > {#1}
10082             { \__clist_use_none_delimit_by_s_stop:w }
10083             { #4 {#2} }
10084         }
10085         { } , #3 , \s__clist_stop
10086     }
10087 \cs_generate_variant:Nn \__clist_item:nnnN { ffo, ff }
10088 \cs_new:Npn \__clist_item_N_loop:nw #1 #2,

```



```

10089 {
10090   \int_compare:nNnTF {#1} = 0
10091     { \__clist_use_i_delimit_by_s_stop:nw { \exp_not:n {#2} } }
10092     { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
10093   }
10094 \cs_generate_variant:Nn \clist_item:Nn { c }

```

(End definition for `\clist_item:Nn`, `__clist_item:nnnN`, and `__clist_item_N_loop:nw`. This function is documented on page 128.)

`\clist_item:nn` This starts in the same way as `\clist_item:Nn` by counting the items of the comma list. The final item should be space-trimmed before being brace-stripped, hence we insert a couple of odd-looking `\prg_do_nothing:` to avoid losing braces. Blank items are ignored.

```

\__clist_item_n:nw
\__clist_item_n_loop:nw
\__clist_item_n_end:n
\__clist_item_n_strip:n
\__clist_item_n_strip:w
10095 \cs_new:Npn \clist_item:nn #1#2
10096 {
10097   \__clist_item:ffnN
10098   { \clist_count:n {#1} }
10099   { \int_eval:n {#2} }
10100   {#1}
10101   \__clist_item_n:nw
10102 }
10103 \cs_new:Npn \__clist_item_n:nw #1
10104 { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
10105 \cs_new:Npn \__clist_item_n_loop:nw #1 #2,
10106 {
10107   \exp_args:No \tl_if_blank:nTF {#2}
10108   { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
10109   {
10110     \int_compare:nNnTF {#1} = 0
10111     { \exp_args:No \__clist_item_n_end:n {#2} }
10112     {
10113       \exp_args:Nf \__clist_item_n_loop:nw
10114       { \int_eval:n { #1 - 1 } }
10115       \prg_do_nothing:
10116     }
10117   }
10118 }
10119 \cs_new:Npn \__clist_item_n_end:n #1 #2 \s_clist_stop
10120 { \tl_trim_spaces_apply:nN {#1} \__clist_item_n_strip:n }
10121 \cs_new:Npn \__clist_item_n_strip:n #1 { \__clist_item_n_strip:w #1 , }
10122 \cs_new:Npn \__clist_item_n_strip:w #1 , { \exp_not:n {#1} }

```

(End definition for `\clist_item:nn` and others. This function is documented on page 128.)

`\clist_rand_item:n` The N-type function is not implemented through the n-type function for efficiency: for instance comma-list variables do not require space-trimming of their items. Even testing for emptiness of an n-type comma-list is slow, so we count items first and use that both for the emptiness test and the pseudo-random integer. Importantly, `\clist_item:Nn` and `\clist_item:nn` only evaluate their argument once.

```

10123 \cs_new:Npn \clist_rand_item:n #1
10124 { \exp_args:Nf \__clist_rand_item:nn { \clist_count:n {#1} } {#1} }
10125 \cs_new:Npn \__clist_rand_item:nn #1#2
10126 {
10127   \int_compare:nNnF {#1} = 0

```

```

10128     { \clist_item:nn {#2} { \int_rand:nn { 1 } {#1} } }
10129   }
10130   \cs_new:Npn \clist_rand_item:N #1
10131   {
10132     \clist_if_empty:NF #1
10133     { \clist_item:Nn #1 { \int_rand:nn { 1 } { \clist_count:N #1 } } }
10134   }
10135   \cs_generate_variant:Nn \clist_rand_item:N { c }

```

(End definition for `\clist_rand_item:n`, `\clist_rand_item:N`, and `__clist_rand_item:nn`. These functions are documented on page 128.)

16.10 Viewing comma lists

`\clist_show:N` Apply the general `__kernel_chk_defined:NT` and `\msg_show:nnnnnn`.

```

\clist_show:c 10136 \cs_new_protected:Npn \clist_show:N { \__clist_show:NN \msg_show:nnxxxx }
\clist_log:N   10137 \cs_generate_variant:Nn \clist_show:N { c }
\clist_log:c   10138 \cs_new_protected:Npn \clist_log:N { \__clist_show:NN \msg_log:nnxxxx }
\__clist_show:NN 10139 \cs_generate_variant:Nn \clist_log:N { c }
10140 \cs_new_protected:Npn \__clist_show:NN #1#2
10141 {
10142   \__kernel_chk_defined:NT #2
10143   {
10144     #1 { LaTeX/kernel } { show-clist }
10145     { \token_to_str:N #2 }
10146     { \clist_map_function:NN #2 \msg_show_item:n }
10147     { } { }
10148   }
10149 }

```

(End definition for `\clist_show:N`, `\clist_log:N`, and `__clist_show:NN`. These functions are documented on page 128.)

`\clist_show:n` A variant of the above: no existence check, empty first argument for the message.

```

\clist_log:n   10150 \cs_new_protected:Npn \clist_show:n { \__clist_show:NN \msg_show:nnxxxx }
\__clist_show:NN 10151 \cs_new_protected:Npn \clist_log:n { \__clist_show:NN \msg_log:nnxxxx }
10152 \cs_new_protected:Npn \__clist_show:NN #1#2
10153 {
10154   #1 { LaTeX/kernel } { show-clist }
10155   { } { \clist_map_function:nN {#2} \msg_show_item:n } { } { }
10156 }

```

(End definition for `\clist_show:n`, `\clist_log:n`, and `__clist_show:NN`. These functions are documented on page 129.)

16.11 Scratch comma lists

`\l_tmpa_clist` Temporary comma list variables.

```

\l_tmpb_clist 10157 \clist_new:N \l_tmpa_clist
\g_tmpa_clist 10158 \clist_new:N \l_tmpb_clist
\g_tmpb_clist 10159 \clist_new:N \g_tmpa_clist
10160 \clist_new:N \g_tmpb_clist

```

(End definition for `\l_tmpa_clist` and others. These variables are documented on page 129.)

```

10161 </package>

```

17 I3token implementation

10162 $\langle *package \rangle$

10163 $\langle @@=char \rangle$

17.1 Internal auxiliaries

`\s__char_stop` Internal scan mark.

10164 `\scan_new:N \s__char_stop`

(End definition for `\s__char_stop`.)

`\q__char_no_value` Internal recursion quarks.

10165 `\quark_new:N \q__char_no_value`

(End definition for `\q__char_no_value`.)

`__char_quark_if_no_value p:N` Functions to query recursion quarks.

`__char_quark_if_no_value:NTF` 10166 `__kernel_quark_new_conditional:Nn __char_quark_if_no_value:N { TF }`

(End definition for `__char_quark_if_no_value:NTF`.)

17.2 Manipulating and interrogating character tokens

`\char_set_catcode:nn` Simple wrappers around the primitives.

`\char_value_catcode:n`

`\char_show_value_catcode:n`

10167 `\cs_new_protected:Npn \char_set_catcode:nn #1#2`

10168 `{ \tex_catcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }`

10169 `\cs_new:Npn \char_value_catcode:n #1`

10170 `{ \tex_the:D \tex_catcode:D \int_eval:n {#1} \exp_stop_f: }`

10171 `\cs_new_protected:Npn \char_show_value_catcode:n #1`

10172 `{ \exp_args:Nf \tl_show:n { \char_value_catcode:n {#1} } }`

(End definition for `\char_set_catcode:nn`, `\char_value_catcode:n`, and `\char_show_value_catcode:n`.
These functions are documented on page 133.)

`\char_set_catcode_escape:N`

`\char_set_catcode_group_begin:N`

`\char_set_catcode_group_end:N`

`\char_set_catcode_math_toggle:N`

`\char_set_catcode_alignment:N`

10173 `\cs_new_protected:Npn \char_set_catcode_escape:N #1`

10174 `{ \char_set_catcode:nn { '#1 } { 0 } }`

10175 `\cs_new_protected:Npn \char_set_catcode_group_begin:N #1`

10176 `{ \char_set_catcode:nn { '#1 } { 1 } }`

10177 `\cs_new_protected:Npn \char_set_catcode_group_end:N #1`

10178 `{ \char_set_catcode:nn { '#1 } { 2 } }`

10179 `\cs_new_protected:Npn \char_set_catcode_math_toggle:N #1`

10180 `{ \char_set_catcode:nn { '#1 } { 3 } }`

10181 `\cs_new_protected:Npn \char_set_catcode_alignment:N #1`

10182 `{ \char_set_catcode:nn { '#1 } { 4 } }`

10183 `\cs_new_protected:Npn \char_set_catcode_end_line:N #1`

10184 `{ \char_set_catcode:nn { '#1 } { 5 } }`

10185 `\cs_new_protected:Npn \char_set_catcode_parameter:N #1`

10186 `{ \char_set_catcode:nn { '#1 } { 6 } }`

10187 `\cs_new_protected:Npn \char_set_catcode_math_superscript:N #1`

10188 `{ \char_set_catcode:nn { '#1 } { 7 } }`

10189 `\cs_new_protected:Npn \char_set_catcode_math_subscript:N #1`

10190 `{ \char_set_catcode:nn { '#1 } { 8 } }`

10191 `\cs_new_protected:Npn \char_set_catcode_ignore:N #1`

```

10192 { \char_set_catcode:nn { '#1 } { 9 } }
10193 \cs_new_protected:Npn \char_set_catcode_space:N #1
10194 { \char_set_catcode:nn { '#1 } { 10 } }
10195 \cs_new_protected:Npn \char_set_catcode_letter:N #1
10196 { \char_set_catcode:nn { '#1 } { 11 } }
10197 \cs_new_protected:Npn \char_set_catcode_other:N #1
10198 { \char_set_catcode:nn { '#1 } { 12 } }
10199 \cs_new_protected:Npn \char_set_catcode_active:N #1
10200 { \char_set_catcode:nn { '#1 } { 13 } }
10201 \cs_new_protected:Npn \char_set_catcode_comment:N #1
10202 { \char_set_catcode:nn { '#1 } { 14 } }
10203 \cs_new_protected:Npn \char_set_catcode_invalid:N #1
10204 { \char_set_catcode:nn { '#1 } { 15 } }

```

(End definition for `\char_set_catcode_escape:N` and others. These functions are documented on page 132.)

```

\char_set_catcode_escape:n
  \char_set_catcode_group_begin:n
  \char_set_catcode_group_end:n
  \char_set_catcode_math_toggle:n
  \char_set_catcode_alignment:n
\char_set_catcode_end_line:n
  \char_set_catcode_parameter:n
  \char_set_catcode_math_superscript:n
  \char_set_catcode_math_subscript:n
\char_set_catcode_ignore:n
\char_set_catcode_space:n
\char_set_catcode_letter:n
\char_set_catcode_other:n
\char_set_catcode_active:n
\char_set_catcode_comment:n
\char_set_catcode_invalid:n
10205 \cs_new_protected:Npn \char_set_catcode_escape:n #1
10206 { \char_set_catcode:nn {#1} { 0 } }
10207 \cs_new_protected:Npn \char_set_catcode_group_begin:n #1
10208 { \char_set_catcode:nn {#1} { 1 } }
10209 \cs_new_protected:Npn \char_set_catcode_group_end:n #1
10210 { \char_set_catcode:nn {#1} { 2 } }
10211 \cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
10212 { \char_set_catcode:nn {#1} { 3 } }
10213 \cs_new_protected:Npn \char_set_catcode_alignment:n #1
10214 { \char_set_catcode:nn {#1} { 4 } }
10215 \cs_new_protected:Npn \char_set_catcode_end_line:n #1
10216 { \char_set_catcode:nn {#1} { 5 } }
10217 \cs_new_protected:Npn \char_set_catcode_parameter:n #1
10218 { \char_set_catcode:nn {#1} { 6 } }
10219 \cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
10220 { \char_set_catcode:nn {#1} { 7 } }
10221 \cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
10222 { \char_set_catcode:nn {#1} { 8 } }
10223 \cs_new_protected:Npn \char_set_catcode_ignore:n #1
10224 { \char_set_catcode:nn {#1} { 9 } }
10225 \cs_new_protected:Npn \char_set_catcode_space:n #1
10226 { \char_set_catcode:nn {#1} { 10 } }
10227 \cs_new_protected:Npn \char_set_catcode_letter:n #1
10228 { \char_set_catcode:nn {#1} { 11 } }
10229 \cs_new_protected:Npn \char_set_catcode_other:n #1
10230 { \char_set_catcode:nn {#1} { 12 } }
10231 \cs_new_protected:Npn \char_set_catcode_active:n #1
10232 { \char_set_catcode:nn {#1} { 13 } }
10233 \cs_new_protected:Npn \char_set_catcode_comment:n #1
10234 { \char_set_catcode:nn {#1} { 14 } }
10235 \cs_new_protected:Npn \char_set_catcode_invalid:n #1
10236 { \char_set_catcode:nn {#1} { 15 } }

```

(End definition for `\char_set_catcode_escape:n` and others. These functions are documented on page 132.)

```

\char_set_mathcode:nn
\char_value_mathcode:n
\char_show_value_mathcode:n
  \char_set_lccode:nn
  \char_value_lccode:n
\char_show_value_lccode:n
  \char_set_uccode:nn
  \char_value_uccode:n
\char_show_value_uccode:n
  \char_set_sfcode:nn
  \char_value_sfcode:n

```

Pretty repetitive, but necessary!

```

10237 \cs_new_protected:Npn \char_set_mathcode:nn #1#2
10238 { \tex_mathcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
10239 \cs_new:Npn \char_value_mathcode:n #1
10240 { \tex_the:D \tex_mathcode:D \int_eval:n {#1} \exp_stop_f: }
10241 \cs_new_protected:Npn \char_show_value_mathcode:n #1
10242 { \exp_args:Nf \tl_show:n { \char_value_mathcode:n {#1} } }
10243 \cs_new_protected:Npn \char_set_lccode:nn #1#2
10244 { \tex_lccode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
10245 \cs_new:Npn \char_value_lccode:n #1
10246 { \tex_the:D \tex_lccode:D \int_eval:n {#1} \exp_stop_f: }
10247 \cs_new_protected:Npn \char_show_value_lccode:n #1
10248 { \exp_args:Nf \tl_show:n { \char_value_lccode:n {#1} } }
10249 \cs_new_protected:Npn \char_set_uccode:nn #1#2
10250 { \tex_uccode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
10251 \cs_new:Npn \char_value_uccode:n #1
10252 { \tex_the:D \tex_uccode:D \int_eval:n {#1} \exp_stop_f: }
10253 \cs_new_protected:Npn \char_show_value_uccode:n #1
10254 { \exp_args:Nf \tl_show:n { \char_value_uccode:n {#1} } }
10255 \cs_new_protected:Npn \char_set_sfcode:nn #1#2
10256 { \tex_sfcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
10257 \cs_new:Npn \char_value_sfcode:n #1
10258 { \tex_the:D \tex_sfcode:D \int_eval:n {#1} \exp_stop_f: }
10259 \cs_new_protected:Npn \char_show_value_sfcode:n #1
10260 { \exp_args:Nf \tl_show:n { \char_value_sfcode:n {#1} } }

```

(End definition for `\char_set_mathcode:nn` and others. These functions are documented on page 134.)

`\l_char_active_seq`
`\l_char_special_seq`

Two sequences for dealing with special characters. The first is characters which may be active, the second longer list is for “special” characters more generally. Both lists are escaped so that for example bulk code assignments can be carried out. In both cases, the order is by ASCII character code (as is done in for example `\ExplSyntaxOn`).

```

10261 \seq_new:N \l_char_special_seq
10262 \seq_set_split:Nnn \l_char_special_seq { }
10263 { \ \ " \# \$ \% \& \ \ ^ \_ \{ \} \~ }
10264 \seq_new:N \l_char_active_seq
10265 \seq_set_split:Nnn \l_char_active_seq { }
10266 { \ " \$ \& \ ^ \_ \~ }

```

(End definition for `\l_char_active_seq` and `\l_char_special_seq`. These variables are documented on page 134.)

17.3 Creating character tokens

`\char_set_active_eq:NN`
`\char_set_active_eq:Nc`
`\char_gset_active_eq:NN`
`\char_gset_active_eq:Nc`
`\char_set_active_eq:nN`
`\char_set_active_eq:nc`
`\char_gset_active_eq:nN`
`\char_gset_active_eq:nc`

Four simple functions with very similar definitions, so set up using an auxiliary. These are similar to LuaTeX’s `\letcharcode` primitive.

```

10267 \group_begin:
10268 \char_set_catcode_active:N \^^@
10269 \cs_set_protected:Npn \__char_tmp:nN #1#2
10270 {
10271   \cs_new_protected:cpn { #1 :nN } ##1
10272   {
10273     \group_begin:
10274     \char_set_lccode:nn { \^^@ } { ##1 }
10275     \tex_lowercase:D { \group_end: #2 ^^@ }

```

```

10276     }
10277     \cs_new_protected:cpx { #1 :NN } ##1
10278     { \exp_not:c { #1 : nN } { '##1 } }
10279   }
10280   \__char_tmp:nN { char_set_active_eq } \cs_set_eq:NN
10281   \__char_tmp:nN { char_gset_active_eq } \cs_gset_eq:NN
10282 \group_end:
10283 \cs_generate_variant:Nn \char_set_active_eq:NN { Nc }
10284 \cs_generate_variant:Nn \char_gset_active_eq:NN { Nc }
10285 \cs_generate_variant:Nn \char_set_active_eq:nN { nc }
10286 \cs_generate_variant:Nn \char_gset_active_eq:nN { nc }

```

(End definition for `\char_set_active_eq:NN` and others. These functions are documented on page 130.)

`__char_int_to_roman:w` For efficiency in 8-bit engines, we use the faster primitive approach to making roman numerals.

```

10287 \cs_new_eq:NN \__char_int_to_roman:w \tex_romannumeral:D

```

(End definition for `__char_int_to_roman:w`.)

`\char_generate:nn` The aim here is to generate characters of (broadly) arbitrary category code. Where possible, that is done using engine support (Xe_{La}TeX, Lua_{TeX}). There are though various issues which are covered below. At the interface layer, turn the two arguments into integers up-front so this is only done once.

```

\__char_generate_aux:nn
\__char_generate_aux:nnw
\__char_generate_auxii:nnw
  \l__char_tmp_tl
  \__char_generate_invalid_catcode:
10288 \cs_new:Npn \char_generate:nn #1#2
10289 {
10290   \exp:w \exp_after:wN \__char_generate_aux:w
10291   \int_value:w \int_eval:n {#1} \exp_after:wN ;
10292   \int_value:w \int_eval:n {#2} ;
10293 }

```

Before doing any actual conversion, first some special case filtering. Spaces are out here as Lua_{TeX} emulation only makes normal (charcode 32 spaces). However, `~@` is filtered out separately as that can't be done with macro emulation either, so is flagged up separately. That done, hand off to the engine-dependent part.

```

10294 \cs_new:Npn \__char_generate_aux:w #1 ; #2 ;
10295 {
10296   \if_int_compare:w #2 = 10 \exp_stop_f:
10297   \if_int_compare:w #1 = 0 \exp_stop_f:
10298     \__kernel_msg_expandable_error:nn { kernel } { char-null-space }
10299   \else:
10300     \__kernel_msg_expandable_error:nn { kernel } { char-space }
10301   \fi:
10302 \else:
10303   \if_int_odd:w 0
10304     \if_int_compare:w #2 < 1 \exp_stop_f: 1 \fi:
10305     \if_int_compare:w #2 = 5 \exp_stop_f: 1 \fi:
10306     \if_int_compare:w #2 = 9 \exp_stop_f: 1 \fi:
10307     \if_int_compare:w #2 > 13 \exp_stop_f: 1 \fi: \exp_stop_f:
10308     \__kernel_msg_expandable_error:nn { kernel }
10309     { char-invalid-catcode }
10310   \else:
10311     \if_int_odd:w 0
10312     \if_int_compare:w #1 < 0 \exp_stop_f: 1 \fi:

```

```

10313         \if_int_compare:w #1 > \c_max_char_int 1 \fi: \exp_stop_f:
10314         \__kernel_msg_expandable_error:nn { kernel }
10315         { char-out-of-range }
10316     \else:
10317         \__char_generate_aux:nnw {#1} {#2}
10318     \fi:
10319 \fi:
10320 \fi:
10321 \exp_end:
10322 }
10323 \tl_new:N \l__char_tmp_tl

```

Engine-dependent definitions are now needed for the implementation. For LuaTeX and XeTeX there is engine-level support. They can do cases that macro emulation can't. All of those are filtered out here using a primitive-based boolean expression to avoid fixing the category code of the null character used in the false branch (for 8-bit engines). The final level is the basic definition at the engine level: the arguments here are integers so there is no need to worry about them too much. Older versions of XeTeX cannot generate active characters so we filter that: at some future stage that may change: the slightly odd ordering of auxiliaries reflects that.

```

10324 \group_begin:
10325   \char_set_catcode_active:N ^^L
10326   \cs_set:Npn ^^L { }
10327   \char_set_catcode_other:n { 0 }
10328   \if_int_odd:w 0
10329     \sys_if_engine luatex:T { 1 }
10330     \sys_if_engine xetex:T { 1 } \exp_stop_f:
10331     \sys_if_engine luatex:TF
10332     {
10333       \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
10334       {
10335         #3
10336         \exp_after:wN \exp_after:wN \exp_after:wN \exp_end:
10337         \lua_now:e { l3kernel.charcat(#1, #2) }
10338       }
10339     }
10340   {
10341     \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
10342     {
10343       #3
10344       \exp_after:wN \exp_end:
10345       \tex_Ucharcat:D #1 \exp_stop_f: #2 \exp_stop_f:
10346     }
10347     \cs_if_exist:NF \tex_expanded:D
10348     {
10349       \cs_new_eq:NN \__char_generate_auxii:nnw \__char_generate_aux:nnw
10350       \cs_gset:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
10351       {
10352         #3
10353         \if_int_compare:w #2 = 13 \exp_stop_f:
10354           \__kernel_msg_expandable_error:nn { kernel } { char-active }
10355         \else:
10356           \__char_generate_auxii:nnw {#1} {#2}
10357         \fi:

```

```

10358         \exp_end:
10359     }
10360 }
10361 }
10362 \else:

```

For engines where `\Ucharcat` isn't available or emulated, we have to work in macros, and cover only the 8-bit range. The first stage is to build up a `tl` containing `^^@` with each category code that can be accessed in this way, with an error set up for the other cases. This is all done such that it can be quickly accessed using a `\if_case:w` low-level conditional. There are a few things to notice here. As `^^L` is `\outer` we need to locally set it to avoid a problem. To get open/close braces into the list, they are set up using `\if_false:` pairing and are then x-type expanded together into the desired form.

```

10363     \tl_set:Nn \l__char_tmp_tl { \exp_not:N \or: }
10364     \char_set_catcode_group_begin:n { 0 } % {
10365     \tl_put_right:Nn \l__char_tmp_tl { ^^@ \if_false: } }
10366     \char_set_catcode_group_end:n { 0 }
10367     \tl_put_right:Nn \l__char_tmp_tl { { \fi: \exp_not:N \or: ^^@ } % }
10368     \tl_set:Nx \l__char_tmp_tl { \l__char_tmp_tl }
10369     \char_set_catcode_math_toggle:n { 0 }
10370     \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
10371     \char_set_catcode_alignment:n { 0 }
10372     \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
10373     \tl_put_right:Nn \l__char_tmp_tl { \or: }
10374     \char_set_catcode_parameter:n { 0 }
10375     \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
10376     \char_set_catcode_math_superscript:n { 0 }
10377     \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
10378     \char_set_catcode_math_subscript:n { 0 }
10379     \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
10380     \tl_put_right:Nn \l__char_tmp_tl { \or: }

```

For making spaces, there needs to be an o-type expansion of a `\use:n` (or some other tokenization) to avoid dropping the space.

```

10381     \char_set_catcode_space:n { 0 }
10382     \tl_put_right:No \l__char_tmp_tl { \use:n { \or: } ^^@ }
10383     \char_set_catcode_letter:n { 0 }
10384     \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
10385     \char_set_catcode_other:n { 0 }
10386     \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
10387     \char_set_catcode_active:n { 0 }
10388     \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }

```

Convert the above temporary list into a series of constant token lists, one for each character code, using `\tex_lowercase:D` to convert `^^@` in each case. The x-type expansion ensures that `\tex_lowercase:D` receives the contents of the token list. `^^L` is awkward hence this is done in three parts: up to `^^L`, `^^L` itself and above `^^L`. Notice that at this stage `^^@` is active.

```

10389     \cs_set_protected:Npn \__char_tmp:n #1
10390     {
10391         \char_set_lccode:nn { 0 } {#1}
10392         \char_set_lccode:nn { 32 } {#1}
10393         \exp_args:Nx \tex_lowercase:D
10394         {

```



```

10395         \tl_const:Nn
10396         \exp_not:c { c__char_ \__char_int_to_roman:w #1 _tl }
10397         { \exp_not:o \l__char_tmp_tl }
10398     }
10399 }
10400 \int_step_function:nnN { 0 } { 11 } \__char_tmp:n
10401 \group_begin:
10402     \tl_replace_once:Nnn \l__char_tmp_tl { ^^@ } { \ERROR }
10403     \__char_tmp:n { 12 }
10404 \group_end:
10405 \int_step_function:nnN { 13 } { 255 } \__char_tmp:n

```

As T_EX is very unhappy if it finds an alignment character inside a primitive `\halign` even when skipping false branches, some precautions are required. T_EX is happy if the token is hidden between braces within `\if_false: ... \fi:`.

```

10406     \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
10407     {
10408         #3
10409         \if_false: { \fi:
10410             \exp_after:wN \exp_after:wN
10411             \exp_after:wN \exp_end:
10412             \exp_after:wN \exp_after:wN
10413             \if_case:w #2
10414                 \exp_last_unbraced:Nv \exp_stop_f:
10415                 { c__char_ \__char_int_to_roman:w #1 _tl }
10416             \or: }
10417             \fi:
10418         }
10419     \fi:
10420 \group_end:

```

(End definition for `\char_generate:nn` and others. This function is documented on page 131.)

`\char_to_utfviii_bytes:n`

```

\__char_to_utfviii_bytes_auxi:n
\__char_to_utfviii_bytes_auxii:Nnn
\__char_to_utfviii_bytes_auxiii:n
\__char_to_utfviii_bytes_outputi:nw
\__char_to_utfviii_bytes_outputii:nw
\__char_to_utfviii_bytes_outputiii:nw
\__char_to_utfviii_bytes_outputiv:nw
\__char_to_utfviii_bytes_output:nnn
\__char_to_utfviii_bytes_output:fnn
\__char_to_utfviii_bytes_end:

```

This code converts a codepoint into the correct UTF-8 representation. In terms of the algorithm itself, see <https://en.wikipedia.org/wiki/UTF-8> for the octet pattern.

```

10421 \cs_new:Npn \char_to_utfviii_bytes:n #1
10422 {
10423     \exp_args:Nf \__char_to_utfviii_bytes_auxi:n
10424     { \int_eval:n {#1} }
10425 }
10426 \cs_new:Npn \__char_to_utfviii_bytes_auxi:n #1
10427 {
10428     \if_int_compare:w #1 > "80 \exp_stop_f:
10429     \if_int_compare:w #1 < "800 \exp_stop_f:
10430         \__char_to_utfviii_bytes_outputi:nw
10431         { \__char_to_utfviii_bytes_auxii:Nnn C {#1} { 64 } }
10432         \__char_to_utfviii_bytes_outputii:nw
10433         { \__char_to_utfviii_bytes_auxiii:n {#1} }
10434     \else:
10435         \if_int_compare:w #1 < "10000 \exp_stop_f:
10436         \__char_to_utfviii_bytes_outputi:nw
10437         { \__char_to_utfviii_bytes_auxii:Nnn E {#1} { 64 * 64 } }
10438         \__char_to_utfviii_bytes_outputii:nw
10439         {

```

```

10440         \__char_to_utfviii_bytes_auxiii:n
10441         { \int_div_truncate:nn {#1} { 64 } }
10442     }
10443     \__char_to_utfviii_bytes_outputiii:nw
10444     { \__char_to_utfviii_bytes_auxiii:n {#1} }
10445 \else:
10446     \__char_to_utfviii_bytes_outputi:nw
10447     {
10448         \__char_to_utfviii_bytes_auxii:Nnn F
10449         {#1} { 64 * 64 * 64 }
10450     }
10451     \__char_to_utfviii_bytes_outputii:nw
10452     {
10453         \__char_to_utfviii_bytes_auxiii:n
10454         { \int_div_truncate:nn {#1} { 64 * 64 } }
10455     }
10456     \__char_to_utfviii_bytes_outputiii:nw
10457     {
10458         \__char_to_utfviii_bytes_auxiii:n
10459         { \int_div_truncate:nn {#1} { 64 } }
10460     }
10461     \__char_to_utfviii_bytes_outputiv:nw
10462     { \__char_to_utfviii_bytes_auxiii:n {#1} }
10463 \fi:
10464 \fi:
10465 \else:
10466     \__char_to_utfviii_bytes_outputi:nw {#1}
10467 \fi:
10468     \__char_to_utfviii_bytes_end: { } { } { } { }
10469 }
10470 \cs_new:Npn \__char_to_utfviii_bytes_auxii:Nnn #1#2#3
10471 { "#10 + \int_div_truncate:nn {#2} {#3} }
10472 \cs_new:Npn \__char_to_utfviii_bytes_auxiii:n #1
10473 { \int_mod:nn {#1} { 64 } + 128 }
10474 \cs_new:Npn \__char_to_utfviii_bytes_outputi:nw
10475 #1 #2 \__char_to_utfviii_bytes_end: #3
10476 { \__char_to_utfviii_bytes_output:fnn { \int_eval:n {#1} } { } {#2} }
10477 \cs_new:Npn \__char_to_utfviii_bytes_outputii:nw
10478 #1 #2 \__char_to_utfviii_bytes_end: #3#4
10479 { \__char_to_utfviii_bytes_output:fnn { \int_eval:n {#1} } { {#3} } {#2} }
10480 \cs_new:Npn \__char_to_utfviii_bytes_outputiii:nw
10481 #1 #2 \__char_to_utfviii_bytes_end: #3#4#5
10482 {
10483     \__char_to_utfviii_bytes_output:fnn
10484     { \int_eval:n {#1} } { {#3} {#4} } {#2}
10485 }
10486 \cs_new:Npn \__char_to_utfviii_bytes_outputiv:nw
10487 #1 #2 \__char_to_utfviii_bytes_end: #3#4#5#6
10488 {
10489     \__char_to_utfviii_bytes_output:fnn
10490     { \int_eval:n {#1} } { {#3} {#4} {#5} } {#2}
10491 }
10492 \cs_new:Npn \__char_to_utfviii_bytes_output:nnn #1#2#3
10493 {

```

```

10494     #3
10495     \__char_to_utfviii_bytes_end: #2 {#1}
10496   }
10497   \cs_generate_variant:Nn \__char_to_utfviii_bytes_output:nnn { f }
10498   \cs_new:Npn \__char_to_utfviii_bytes_end: { }

```

(End definition for `\char_to_utfviii_bytes:n` and others. This function is documented on page 273.)

```

\char_to_nfd:N Look up any NFD and recursively produce the result.
\__char_to_nfd:n
\__char_to_nfd:Nw
10499 \cs_new:Npn \char_to_nfd:N #1
10500 {
10501   \cs_if_exist:cTF { c__char_nfd_ \token_to_str:N #1 _ tl }
10502   {
10503     \exp_after:wN \exp_after:wN \exp_after:wN \__char_to_nfd:Nw
10504     \exp_after:wN \exp_after:wN \exp_after:wN #1
10505     \cs:w c__char_nfd_ \token_to_str:N #1 _ tl \cs_end:
10506     \s__char_stop
10507   }
10508   { \exp_not:n {#1} }
10509 }
10510 \cs_set_eq:NN \__char_to_nfd:n \char_to_nfd:N
10511 \cs_new:Npn \__char_to_nfd:Nw #1#2#3 \s__char_stop
10512 {
10513   \exp_args:Ne \__char_to_nfd:n
10514   { \char_generate:nn { ' #2 } { \__char_change_case_catcode:N #1 } }
10515   \tl_if_blank:nF {#3}
10516   {
10517     \exp_args:Ne \__char_to_nfd:n
10518     { \char_generate:nn { ' #3 } { \char_value_catcode:n { ' #3 } } }
10519   }
10520 }

```

(End definition for `\char_to_nfd:N`, `__char_to_nfd:n`, and `__char_to_nfd:Nw`. This function is documented on page 273.)

```

\char_lowercase:N To ensure that the category codes produced are predictable, every character is re-
\char_uppercase:N generated even if it is otherwise unchanged. This makes life a little interesting when
\char_titlecase:N we might have multiple output characters: we have to grab each of them and case change
\char_foldcase:N them in reverse order to maintain f-type expandability.

```

```

\__char_change_case:nNN 10521 \cs_new:Npn \char_lowercase:N #1
\__char_change_case:nN 10522 { \__char_change_case:nNN { lower } \char_value_lccode:n #1 }
\__char_change_case_multi:nN 10523 \cs_new:Npn \char_uppercase:N #1
\__char_change_case_multi:vN 10524 { \__char_change_case:nNN { upper } \char_value_uccode:n #1 }
\__char_change_case_multi:NNNw 10525 \cs_new:Npn \char_titlecase:N #1
\__char_change_case:NNN 10526 {
\__char_change_case:NNNN 10527   \tl_if_exist:cTF { c__char_titlecase_ \token_to_str:N #1 _ tl }
\__char_change_case:NN 10528   {
\__char_change_case_catcode:N 10529     \__char_change_case_multi:vN
10530     { c__char_titlecase_ \token_to_str:N #1 _ tl } #1
10531   }
10532   { \char_uppercase:N #1 }
10533 }
\char_str_lowercase:N 10534 \cs_new:Npn \char_foldcase:N #1
\char_str_uppercase:N 10535 { \__char_change_case:nNN { fold } \char_value_lccode:n #1 }
\char_str_titlecase:N
\char_str_foldcase:N
\__char_str_change_case:nNN
\__char_str_change_case:nN

```

```

10536 \cs_new:Npn \__char_change_case:nNN #1#2#3
10537 {
10538   \tl_if_exist:cTF { c__char_ #1 case_ \token_to_str:N #3 _tl }
10539   {
10540     \__char_change_case_multi:vN
10541     { c__char_ #1 case_ \token_to_str:N #3 _tl } #3
10542   }
10543   { \exp_args:Nf \__char_change_case:nN { #2 { '#3 } } #3 }
10544 }
10545 \cs_new:Npn \__char_change_case:nN #1#2
10546 {
10547   \int_compare:nNnTF {#1} = 0
10548   { #2 }
10549   { \char_generate:nn {#1} { \__char_change_case_catcode:N #2 } }
10550 }
10551 \cs_new:Npn \__char_change_case_multi:nN #1#2
10552 { \__char_change_case_multi:NNNNw #2 #1 \q__char_no_value \q__char_no_value \s__char_stop
10553 \cs_generate_variant:Nn \__char_change_case_multi:nN { v }
10554 \cs_new:Npn \__char_change_case_multi:NNNNw #1#2#3#4#5 \s__char_stop
10555 {
10556   \__char_quark_if_no_value:NTF #4
10557   {
10558     \__char_quark_if_no_value:NTF #3
10559     { \__char_change_case:NN #1 #2 }
10560     { \__char_change_case:NNN #1 #2#3 }
10561   }
10562   { \__char_change_case:NNNN #1 #2#3#4 }
10563 }
10564 \cs_new:Npn \__char_change_case:NNN #1#2#3
10565 {
10566   \exp_args:Nnf \use:nn
10567   { \__char_change_case:NN #1 #2 }
10568   { \__char_change_case:NN #1 #3 }
10569 }
10570 \cs_new:Npn \__char_change_case:NNNN #1#2#3#4
10571 {
10572   \exp_args:Nnff \use:nnn
10573   { \__char_change_case:NN #1 #2 }
10574   { \__char_change_case:NN #1 #3 }
10575   { \__char_change_case:NN #1 #4 }
10576 }
10577 \cs_new:Npn \__char_change_case:NN #1#2
10578 { \char_generate:nn { '#2 } { \__char_change_case_catcode:N #1 } }
10579 \cs_new:Npn \__char_change_case_catcode:N #1
10580 {
10581   \if_catcode:w \exp_not:N #1 \c_math_toggle_token
10582   3
10583   \else:
10584     \if_catcode:w \exp_not:N #1 \c_alignment_token
10585     4
10586     \else:
10587       \if_catcode:w \exp_not:N #1 \c_math_superscript_token
10588       7
10589       \else:

```

```

10590         \if_catcode:w \exp_not:N #1 \c_math_subscript_token
10591             8
10592         \else:
10593             \if_catcode:w \exp_not:N #1 \c_space_token
10594                 10
10595             \else:
10596                 \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
10597                     11
10598                 \else:
10599                     \if_catcode:w \exp_not:N #1 \c_catcode_other_token
10600                         12
10601                     \else:
10602                         13
10603                     \fi:
10604                 \fi:
10605             \fi:
10606         \fi:
10607     \fi:
10608 \fi:
10609 \fi:
10610 }

```

Same story for the string version, except category code is easier to follow. This of course makes this version significantly faster.

```

10611 \cs_new:Npn \char_str_lowercase:N #1
10612 { \__char_str_change_case:nNN { lower } \char_value_lccode:n #1 }
10613 \cs_new:Npn \char_str_uppercase:N #1
10614 { \__char_str_change_case:nNN { upper } \char_value_uccode:n #1 }
10615 \cs_new:Npn \char_str_titlecase:N #1
10616 {
10617     \tl_if_exist:cTF { c__char_titlecase_ \token_to_str:N #1 _tl }
10618     { \tl_to_str:c { c__char_titlecase_ \token_to_str:N #1 _tl } }
10619     { \char_str_uppercase:N #1 }
10620 }
10621 \cs_new:Npn \char_str_foldcase:N #1
10622 { \__char_str_change_case:nNN { fold } \char_value_lccode:n #1 }
10623 \cs_new:Npn \__char_str_change_case:nNN #1#2#3
10624 {
10625     \tl_if_exist:cTF { c__char_ #1 case_ \token_to_str:N #3 _tl }
10626     { \tl_to_str:c { c__char_ #1 case_ \token_to_str:N #3 _tl } }
10627     { \exp_args:Nf \__char_str_change_case:nN { #2 { ' #3 } } #3 }
10628 }
10629 \cs_new:Npn \__char_str_change_case:nN #1#2
10630 {
10631     \int_compare:nNnTF {#1} = 0
10632     { \tl_to_str:n {#2} }
10633     { \char_generate:nn {#1} { 12 } }
10634 }
10635 \bool_lazy_or:nnF
10636 { \cs_if_exist_p:N \tex luatexversion:D }
10637 { \cs_if_exist_p:N \tex XeTeXversion:D }
10638 {
10639     \cs_set:Npn \__char_str_change_case:nN #1#2
10640     { \tl_to_str:n {#2} }

```

```
10641 }
```

(End definition for `\char_lowercase:N` and others. These functions are documented on page 131.)

`\c_catcode_other_space_tl` Create a space with category code 12: an “other” space.

```
10642 \tl_const:Nx \c_catcode_other_space_tl { \char_generate:nn { '\ } { 12 } }
```

(End definition for `\c_catcode_other_space_tl`. This function is documented on page 131.)

17.4 Generic tokens

```
10643 <@@=token>
```

`\s__token_stop` Internal scan marks.

```
10644 \scan_new:N \s__token_stop
```

(End definition for `\s__token_stop`.)

`\token_to_meaning:N` These are all defined in `l3basics`, as they are needed “early”. This is just a reminder!

`\token_to_meaning:c` (End definition for `\token_to_meaning:N` and `\token_to_str:N`. These functions are documented on page 135.)

`\token_to_str:N`

`\token_to_str:c`

`\c_group_begin_token`

`\c_group_end_token`

`\c_math_toggle_token`

`\c_alignment_token`

`\c_parameter_token`

`\c_math_superscript_token`

`\c_math_subscript_token`

`\c_space_token`

`\c_catcode_letter_token`

`\c_catcode_other_token`

We define these useful tokens. For the brace and space tokens things have to be done by hand: the formal argument spec. for `\cs_new_eq:NN` does not cover them so we do things by hand. (As currently coded it would *work* with `\cs_new_eq:NN` but that’s not really a great idea to show off: we want people to stick to the defined interfaces and that includes us.) So that these few odd names go into the log when appropriate there is a need to hand-apply the `__kernel_chk_if_free_cs:N` check.

```
10645 \group_begin:
10646   \__kernel_chk_if_free_cs:N \c_group_begin_token
10647   \tex_global:D \tex_let:D \c_group_begin_token {
10648     \__kernel_chk_if_free_cs:N \c_group_end_token
10649     \tex_global:D \tex_let:D \c_group_end_token }
10650   \char_set_catcode_math_toggle:N \*
10651   \cs_new_eq:NN \c_math_toggle_token *
10652   \char_set_catcode_alignment:N \*
10653   \cs_new_eq:NN \c_alignment_token *
10654   \cs_new_eq:NN \c_parameter_token #
10655   \cs_new_eq:NN \c_math_superscript_token ^
10656   \char_set_catcode_math_subscript:N \*
10657   \cs_new_eq:NN \c_math_subscript_token *
10658   \__kernel_chk_if_free_cs:N \c_space_token
10659   \use:n { \tex_global:D \tex_let:D \c_space_token = ~ } ~
10660   \cs_new_eq:NN \c_catcode_letter_token a
10661   \cs_new_eq:NN \c_catcode_other_token 1
10662 \group_end:
```

(End definition for `\c_group_begin_token` and others. These functions are documented on page 135.)

`\c_catcode_active_tl` Not an implicit token!

```
10663 \group_begin:
10664   \char_set_catcode_active:N \*
10665   \tl_const:Nn \c_catcode_active_tl { \exp_not:N * }
10666 \group_end:
```

(End definition for `\c_catcode_active_tl`. This variable is documented on page 135.)

17.5 Token conditionals

`\token_if_group_begin_p:N` Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.
`\token_if_group_begin:N \underline{TF}`

```
10667 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
10668 {
10669     \if_catcode:w \exp_not:N #1 \c_group_begin_token
10670     \prg_return_true: \else: \prg_return_false: \fi:
10671 }
```

(End definition for `\token_if_group_begin:N \underline{TF}` . This function is documented on page 136.)

`\token_if_group_end_p:N` Check if token is a end group token. We use the constant `\c_group_end_token` for this.

`\token_if_group_end:N \underline{TF}`

```
10672 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
10673 {
10674     \if_catcode:w \exp_not:N #1 \c_group_end_token
10675     \prg_return_true: \else: \prg_return_false: \fi:
10676 }
```

(End definition for `\token_if_group_end:N \underline{TF}` . This function is documented on page 136.)

`\token_if_math_toggle_p:N` Check if token is a math shift token. We use the constant `\c_math_toggle_token` for this.
`\token_if_math_toggle:N \underline{TF}`

```
10677 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
10678 {
10679     \if_catcode:w \exp_not:N #1 \c_math_toggle_token
10680     \prg_return_true: \else: \prg_return_false: \fi:
10681 }
```

(End definition for `\token_if_math_toggle:N \underline{TF}` . This function is documented on page 136.)

`\token_if_alignment_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_token` for this.
`\token_if_alignment:N \underline{TF}`

```
10682 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
10683 {
10684     \if_catcode:w \exp_not:N #1 \c_alignment_token
10685     \prg_return_true: \else: \prg_return_false: \fi:
10686 }
```

(End definition for `\token_if_alignment:N \underline{TF}` . This function is documented on page 136.)

`\token_if_parameter_p:N` Check if token is a parameter token. We use the constant `\c_parameter_token` for this.

`\token_if_parameter:N \underline{TF}` We have to trick T_EX a bit to avoid an error message: within a group we prevent `\c_parameter_token` from behaving like a macro parameter character. The definitions of `\prg_new_conditional:Npnn` are global, so they remain after the group.

```
10687 \group_begin:
10688 \cs_set_eq:NN \c_parameter_token \scan_stop:
10689 \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
10690 {
10691     \if_catcode:w \exp_not:N #1 \c_parameter_token
10692     \prg_return_true: \else: \prg_return_false: \fi:
10693 }
10694 \group_end:
```

(End definition for `\token_if_parameter:N \underline{TF}` . This function is documented on page 136.)

`\token_if_math_superscript_p:N` Check if token is a math superscript token. We use the constant `\c_math_superscript_token` for this.

`\token_if_math_superscript:N \underline{TF}`

```
10695 \prg_new_conditional:Npnn \token_if_math_superscript:N #1
10696 { p , T , F , TF }
10697 {
10698     \if_catcode:w \exp_not:N #1 \c_math_superscript_token
10699     \prg_return_true: \else: \prg_return_false: \fi:
10700 }
```

(End definition for `\token_if_math_superscript:N \underline{TF}` . This function is documented on page 136.)

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_math_subscript_token` for this.

`\token_if_math_subscript:N \underline{TF}`

```
10701 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T , F , TF }
10702 {
10703     \if_catcode:w \exp_not:N #1 \c_math_subscript_token
10704     \prg_return_true: \else: \prg_return_false: \fi:
10705 }
```

(End definition for `\token_if_math_subscript:N \underline{TF}` . This function is documented on page 136.)

`\token_if_space_p:N` Check if token is a space token. We use the constant `\c_space_token` for this.

`\token_if_space:N \underline{TF}`

```
10706 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T , F , TF }
10707 {
10708     \if_catcode:w \exp_not:N #1 \c_space_token
10709     \prg_return_true: \else: \prg_return_false: \fi:
10710 }
```

(End definition for `\token_if_space:N \underline{TF}` . This function is documented on page 136.)

`\token_if_letter_p:N` Check if token is a letter token. We use the constant `\c_catcode_letter_token` for this.

`\token_if_letter:N \underline{TF}`

```
10711 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T , F , TF }
10712 {
10713     \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
10714     \prg_return_true: \else: \prg_return_false: \fi:
10715 }
```

(End definition for `\token_if_letter:N \underline{TF}` . This function is documented on page 137.)

`\token_if_other_p:N` Check if token is an other char token. We use the constant `\c_catcode_other_token` for this.

`\token_if_other:N \underline{TF}`

```
10716 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T , F , TF }
10717 {
10718     \if_catcode:w \exp_not:N #1 \c_catcode_other_token
10719     \prg_return_true: \else: \prg_return_false: \fi:
10720 }
```

(End definition for `\token_if_other:N \underline{TF}` . This function is documented on page 137.)

`\token_if_active_p:N` Check if token is an active char token. We use the constant `\c_catcode_active_tl` for this. A technical point is that `\c_catcode_active_tl` is in fact a macro expanding to `\exp_not:N *`, where `*` is active.

`\token_if_active:N \underline{TF}`

```
10721 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T , F , TF }
10722 {
10723     \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
10724     \prg_return_true: \else: \prg_return_false: \fi:
10725 }
```


(End definition for `\token_if_active:NTF`. This function is documented on page 137.)

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.

`\token_if_eq_meaning:NNTF`

```

10726 \prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T , F , TF }
10727 {
10728     \if_meaning:w #1 #2
10729     \prg_return_true: \else: \prg_return_false: \fi:
10730 }

```

(End definition for `\token_if_eq_meaning:NNTF`. This function is documented on page 137.)

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.

`\token_if_eq_catcode:NNTF`

```

10731 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }
10732 {
10733     \if_catcode:w \exp_not:N #1 \exp_not:N #2
10734     \prg_return_true: \else: \prg_return_false: \fi:
10735 }

```

(End definition for `\token_if_eq_catcode:NNTF`. This function is documented on page 137.)

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.

`\token_if_eq_charcode:NNTF`

```

10736 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }
10737 {
10738     \if_charcode:w \exp_not:N #1 \exp_not:N #2
10739     \prg_return_true: \else: \prg_return_false: \fi:
10740 }

```

(End definition for `\token_if_eq_charcode:NNTF`. This function is documented on page 137.)

`\token_if_macro_p:N` When a token is a macro, `\token_to_meaning:N` always outputs something like
`\token_if_macro:NTF` `\long macro:#1->#1` so we could naively check to see if the meaning contains `->`.
`__token_if_macro_p:w` However, this can fail the five `\...mark` primitives, whose meaning has the form `\...mark:<user material>`. The problem is that the `<user material>` can contain `->`.

However, only characters, macros, and marks can contain the colon character. The idea is thus to grab until the first `:`, and analyse what is left. However, macros can have any combination of `\long`, `\protected` or `\outer` (not used in L^AT_EX3) before the string `macro:.` We thus only select the part of the meaning between the first `ma` and the first following `:`. If this string is `cro`, then we have a macro. If the string is `rk`, then we have a mark. The string can also be `cro parameter character` for a colon with a weird category code (namely the usual category code of `#`). Otherwise, it is empty.

This relies on the fact that `\long`, `\protected`, `\outer` cannot contain `ma`, regardless of the escape character, even if the escape character is `m...`

Both `ma` and `:` must be of category code 12 (other), so are detokenized.

```

10741 \use:x
10742 {
10743     \prg_new_conditional:Npnn \exp_not:N \token_if_macro:N ##1
10744     { p , T , F , TF }
10745     {
10746         \exp_not:N \exp_after:wN \exp_not:N \__token_if_macro_p:w
10747         \exp_not:N \token_to_meaning:N ##1 \tl_to_str:n { ma : }
10748         \s__token_stop
10749     }
10750     \cs_new:Npn \exp_not:N \__token_if_macro_p:w

```

```

10751     ##1 \tl_to_str:n { ma } ##2 \c_colon_str ##3 \s__token_stop
10752   }
10753   {
10754     \str_if_eq:nnTF { #2 } { cro }
10755     { \prg_return_true: }
10756     { \prg_return_false: }
10757   }

```

(End definition for `\token_if_macro:NTF` and `__token_if_macro_p:w`. This function is documented on page 137.)

`\token_if_cs_p:N` Check if token has same catcode as a control sequence. This follows the same pattern as
`\token_if_cs:NTF` for `\token_if_letter:N` etc. We use `\scan_stop:` for this.

```

10758 \prg_new_conditional:Npnn \token_if_cs:N #1 { p , T , F , TF }
10759 {
10760   \if_catcode:w \exp_not:N #1 \scan_stop:
10761   \prg_return_true: \else: \prg_return_false: \fi:
10762 }

```

(End definition for `\token_if_cs:NTF`. This function is documented on page 137.)

`\token_if_expandable_p:N` Check if token is expandable. We use the fact that T_EX temporarily converts `\exp_not:N`
`\token_if_expandable:NTF` `\token` into `\scan_stop:` if `\token` is expandable. An undefined token is not considered as expandable. No problem nesting the conditionals, since the third #1 is only skipped if it is non-expandable (hence not part of T_EX's conditional apparatus).

```

10763 \prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T , F , TF }
10764 {
10765   \exp_after:wN \if_meaning:w \exp_not:N #1 #1
10766   \prg_return_false:
10767   \else:
10768     \if_cs_exist:N #1
10769     \prg_return_true:
10770     \else:
10771     \prg_return_false:
10772   \fi:
10773   \fi:
10774 }

```

(End definition for `\token_if_expandable:NTF`. This function is documented on page 137.)

`__token_delimit_by_char:w` These auxiliary functions are used below to define some conditionals which detect whether
`__token_delimit_by_count:w` the `\meaning` of their argument begins with a particular string. Each auxiliary takes an
`__token_delimit_by_dimen:w` argument delimited by a string, a second one delimited by `\s__token_stop`, and returns
`__token_delimit_by_macro:w` the first one and its delimiter. This result is eventually compared to another string.

```

10775 \group_begin:
10776 \cs_set_protected:Npn \__token_tmp:w #1
10777 {
10778   \use:x
10779   {
10780     \cs_new:Npn \exp_not:c { __token_delimit_by_ #1 :w }
10781     #####1 \tl_to_str:n {#1} #####2 \s__token_stop
10782     { #####1 \tl_to_str:n {#1} }
10783   }
10784 }

```

```

10785 \__token_tmp:w { char" }
10786 \__token_tmp:w { count }
10787 \__token_tmp:w { dimen }
10788 \__token_tmp:w { macro }
10789 \__token_tmp:w { muskip }
10790 \__token_tmp:w { skip }
10791 \__token_tmp:w { toks }
10792 \group_end:

```

(End definition for `__token_delimit_by_char:w` and others.)

<pre> \token_if_chardef_p:N \token_if_chardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:NTF \token_if_long_macro_p:N \token_if_long_macro:NTF \token_if_protected_macro_p:N \token_if_protected_macro:NTF \token_if_protected_long_macro_p:N \token_if_protected_long_macro:NTF \token_if_dim_register_p:N \token_if_dim_register:NTF \token_if_int_register_p:N \token_if_int_register:NTF \token_if_muskip_register_p:N \token_if_muskip_register:NTF \token_if_skip_register_p:N \token_if_skip_register:NTF \token_if_toks_register_p:N \token_if_toks_register:NTF </pre>	<p>Each of these conditionals tests whether its argument's <code>\meaning</code> starts with a given string. This is essentially done by having an auxiliary grab an argument delimited by the string and testing whether the argument was empty. Of course, a copy of this string must first be added to the end of the <code>\meaning</code> to avoid a runaway argument in case it does not contain the string. Two complications arise. First, the escape character is not fixed, and cannot be included in the delimiter of the auxiliary function (this function cannot be defined on the fly because tests must remain expandable): instead the first argument of the auxiliary (plus the delimiter to avoid complications with trailing spaces) is compared using <code>\str_if_eq:eeTF</code> to the result of applying <code>\token_to_str:N</code> to a control sequence. Second, the <code>\meaning</code> of primitives such as <code>\dimen</code> or <code>\dimendef</code> starts in the same way as registers such as <code>\dimen123</code>, so they must be tested for.</p> <p>Characters used as delimiters must have catcode 12 and are obtained through <code>\tl_to_str:n</code>. This requires doing all definitions within <code>x</code>-expansion. The temporary function <code>__token_tmp:w</code> used to define each conditional receives three arguments: the name of the conditional, the auxiliary's delimiter (also used to name the auxiliary), and the string to which one compares the auxiliary's result. Note that the <code>\meaning</code> of a protected long macro starts with <code>\protected\long macro</code>, with no space after <code>\protected</code> but a space after <code>\long</code>, hence the mixture of <code>\token_to_str:N</code> and <code>\tl_to_str:n</code>.</p> <p>For the first five conditionals, <code>\cs_if_exist:cT</code> turns out to be <code>false</code>, and the code boils down to a string comparison between the result of the auxiliary on the <code>\meaning</code> of the conditional's argument <code>####1</code>, and <code>#3</code>. Both are evaluated at run-time, as this is important to get the correct escape character.</p>
--	--

The other five conditionals have additional code that compares the argument `####1` to two $\text{T}_{\text{E}}\text{X}$ primitives which would wrongly be recognized as registers otherwise. Despite using $\text{T}_{\text{E}}\text{X}$'s primitive conditional construction, this does not break when `####1` is itself a conditional, because branches of the conditionals are only skipped if `####1` is one of the two primitives that are tested for (which are not $\text{T}_{\text{E}}\text{X}$ conditionals).

```

10793 \group_begin:
10794 \cs_set_protected:Npn \__token_tmp:w #1#2#3
10795 {
10796   \use:x
10797   {
10798     \prg_new_conditional:Npnn \exp_not:c { token_if_ #1 :N } ####1
10799     { p , T , F , TF }
10800     {
10801       \cs_if_exist:cT { tex_ #2 :D }
10802       {
10803         \exp_not:N \if_meaning:w ####1 \exp_not:c { tex_ #2 :D }
10804         \exp_not:N \prg_return_false:
10805         \exp_not:N \else:
10806         \exp_not:N \if_meaning:w ####1 \exp_not:c { tex_ #2 def:D }

```

```

10807         \exp_not:N \prg_return_false:
10808         \exp_not:N \else:
10809     }
10810 \exp_not:N \str_if_eq:eeTF
10811 {
10812     \exp_not:N \exp_after:wN
10813     \exp_not:c { __token_delimit_by_ #2 :w }
10814     \exp_not:N \token_to_meaning:N ####1
10815     ? \tl_to_str:n {#2} \s__token_stop
10816 }
10817 { \exp_not:n {#3} }
10818 { \exp_not:N \prg_return_true: }
10819 { \exp_not:N \prg_return_false: }
10820 \cs_if_exist:cT { tex_ #2 :D }
10821 {
10822     \exp_not:N \fi:
10823     \exp_not:N \fi:
10824 }
10825 }
10826 }
10827 }
10828 \__token_tmp:w { chardef } { char" } { \token_to_str:N \char" }
10829 \__token_tmp:w { mathchardef } { char" } { \token_to_str:N \mathchar" }
10830 \__token_tmp:w { long_macro } { macro } { \tl_to_str:n { \long } macro }
10831 \__token_tmp:w { protected_macro } { macro }
10832 { \tl_to_str:n { \protected } macro }
10833 \__token_tmp:w { protected_long_macro } { macro }
10834 { \token_to_str:N \protected \tl_to_str:n { \long } macro }
10835 \__token_tmp:w { dim_register } { dimen } { \token_to_str:N \dimen }
10836 \__token_tmp:w { int_register } { count } { \token_to_str:N \count }
10837 \__token_tmp:w { muskip_register } { muskip } { \token_to_str:N \muskip }
10838 \__token_tmp:w { skip_register } { skip } { \token_to_str:N \skip }
10839 \__token_tmp:w { toks_register } { toks } { \token_to_str:N \toks }
10840 \group_end:

```

(End definition for `\token_if_chardef:NTF` and others. These functions are documented on page 138.)

`\token_if_primitive_p:N` We filter out macros first, because they cause endless trouble later otherwise.

`\token_if_primitive:NTF` Primitives are almost distinguished by the fact that the result of `\token_to_meaning:N` is formed from letters only. Every other token has either a space (e.g., the letter A), a digit (e.g., `\count123`) or a double quote (e.g., `\char"A`).

`__token_if_primitive:NNw` Ten exceptions: on the one hand, `\tex_undefined:D` is not a primitive, but its meaning is undefined, only letters; on the other hand, `\space`, `\italiccorr`, `\hyphen`, `\firstmark`, `\topmark`, `\botmark`, `\splitfirstmark`, `\splitbotmark`, and `\nullfont` are primitives, but have non-letters in their meaning.

`__token_if_primitive:Nw`

`__token_if_primitive_undefined:N`

We start by removing the two first (non-space) characters from the meaning. This removes the escape character (which may be nonexistent depending on `\endlinechar`), and takes care of three of the exceptions: `\space`, `\italiccorr` and `\hyphen`, whose meaning is at most two characters. This leaves a string terminated by some `:`, and `\s__token_stop`.

The meaning of each one of the five `\...mark` primitives has the form `<letters>:<user material>`. In other words, the first non-letter is a colon. We remove everything after the first colon.

We are now left with a string, which we must analyze. For primitives, it contains only letters. For non-primitives, it contains either " , or a space, or a digit. Two exceptions remain: `\tex_undefined:D`, which is not a primitive, and `\nullfont`, which is a primitive.

Spaces cannot be grabbed in an undelimited way, so we check them separately. If there is a space, we test for `\nullfont`. Otherwise, we go through characters one by one, and stop at the first character less than 'A (this is not quite a test for "only letters", but is close enough to work in this context). If this first character is : then we have a primitive, or `\tex_undefined:D`, and if it is " or a digit, then the token is not a primitive.

```

10841 \tex_chardef:D \c__token_A_int = 'A ~ %
10842 \use:x
10843 {
10844   \prg_new_conditional:Npnn \exp_not:N \token_if_primitive:N ##1
10845   { p , T , F , TF }
10846   {
10847     \exp_not:N \token_if_macro:NTF ##1
10848     \exp_not:N \prg_return_false:
10849     {
10850       \exp_not:N \exp_after:wN \exp_not:N \__token_if_primitive:NNw
10851       \exp_not:N \token_to_meaning:N ##1
10852       \tl_to_str:n { : : : } \s__token_stop ##1
10853     }
10854   }
10855   \cs_new:Npn \exp_not:N \__token_if_primitive:NNw
10856   ##1##2 ##3 \c_colon_str ##4 \s__token_stop
10857   {
10858     \exp_not:N \tl_if_empty:oTF
10859     { \exp_not:N \__token_if_primitive_space:w ##3 ~ }
10860     {
10861       \exp_not:N \__token_if_primitive_loop:N ##3
10862       \c_colon_str \s__token_stop
10863     }
10864     { \exp_not:N \__token_if_primitive_nullfont:N }
10865   }
10866 }
10867 \cs_new:Npn \__token_if_primitive_space:w #1 ~ { }
10868 \cs_new:Npn \__token_if_primitive_nullfont:N #1
10869 {
10870   \if_meaning:w \tex_nullfont:D #1
10871   \prg_return_true:
10872   \else:
10873   \prg_return_false:
10874   \fi:
10875 }
10876 \cs_new:Npn \__token_if_primitive_loop:N #1
10877 {
10878   \if_int_compare:w ' #1 < \c__token_A_int %
10879   \exp_after:wN \__token_if_primitive:Nw
10880   \exp_after:wN #1
10881   \else:
10882   \exp_after:wN \__token_if_primitive_loop:N
10883   \fi:
10884 }

```

```

10885 \cs_new:Npn \__token_if_primitive:Nw #1 #2 \s__token_stop
10886 {
10887   \if:w : #1
10888     \exp_after:wN \__token_if_primitive_undefined:N
10889   \else:
10890     \prg_return_false:
10891     \exp_after:wN \use_none:n
10892   \fi:
10893 }
10894 \cs_new:Npn \__token_if_primitive_undefined:N #1
10895 {
10896   \if_cs_exist:N #1
10897     \prg_return_true:
10898   \else:
10899     \prg_return_false:
10900   \fi:
10901 }

```

(End definition for `\token_if_primitive:NTF` and others. This function is documented on page 139.)

17.6 Peeking ahead at the next token

10902 `<@@=peek>`

Peeking ahead is implemented using a two part mechanism. The outer level provides a defined interface to the lower level material. This allows a large amount of code to be shared. There are four cases:

1. peek at the next token;
2. peek at the next non-space token;
3. peek at the next token and remove it;
4. peek at the next non-space token and remove it.

`\l_peek_token` Storage tokens which are publicly documented: the token peeked.

`\g_peek_token` 10903 `\cs_new_eq:NN \l_peek_token ?`
 10904 `\cs_new_eq:NN \g_peek_token ?`

(End definition for `\l_peek_token` and `\g_peek_token`. These variables are documented on page 139.)

`\l__peek_search_token` The token to search for as an implicit token: cf. `\l__peek_search_tl`.

10905 `\cs_new_eq:NN \l__peek_search_token ?`

(End definition for `\l__peek_search_token`.)

`\l__peek_search_tl` The token to search for as an explicit token: cf. `\l__peek_search_token`.

10906 `\tl_new:N \l__peek_search_tl`

(End definition for `\l__peek_search_tl`.)

`__peek_true:w` Functions used by the branching and space-stripping code.

`__peek_true_aux:w` 10907 `\cs_new:Npn __peek_true:w { }`
`__peek_false:w` 10908 `\cs_new:Npn __peek_true_aux:w { }`
`__peek_tmp:w` 10909 `\cs_new:Npn __peek_false:w { }`
 10910 `\cs_new:Npn __peek_tmp:w { }`

(End definition for `_peek_true:w` and others.)

`\s__peek_mark` Internal scan marks.

`\s__peek_stop` 10911 `\scan_new:N \s__peek_mark`
 10912 `\scan_new:N \s__peek_stop`

(End definition for `\s__peek_mark` and `\s__peek_stop`.)

`_peek_use_none_delimit_by_s_stop:w` Functions to gobble up to a scan mark.

10913 `\cs_new:Npn _peek_use_none_delimit_by_s_stop:w #1 \s__peek_stop { }`

(End definition for `_peek_use_none_delimit_by_s_stop:w`.)

`\peek_after:Nw` Simple wrappers for `\futurelet`: no arguments absorbed here.
`\peek_gafter:Nw`

10914 `\cs_new_protected:Npn \peek_after:Nw`
 10915 `{ \tex_futurelet:D \l_peek_token }`
 10916 `\cs_new_protected:Npn \peek_gafter:Nw`
 10917 `{ \tex_global:D \tex_futurelet:D \g_peek_token }`

(End definition for `\peek_after:Nw` and `\peek_gafter:Nw`. These functions are documented on page 139.)

`_peek_true_remove:w` A function to remove the next token and then regain control.

10918 `\cs_new_protected:Npn _peek_true_remove:w`
 10919 `{`
 10920 `\tex_afterassignment:D _peek_true_aux:w`
 10921 `\cs_set_eq:NN _peek_tmp:w`
 10922 `}`

(End definition for `_peek_true_remove:w`.)

`\peek_remove_spaces:n` Repeatedly use `_peek_true_remove:w` to remove a space and call `_peek_true_`
`_peek_remove_spaces:` `aux:w`.

10923 `\cs_new_protected:Npn \peek_remove_spaces:n #1`
 10924 `{`
 10925 `\cs_set:Npx _peek_false:w { \exp_not:n {#1} }`
 10926 `\group_align_safe_begin:`
 10927 `\cs_set:Npn _peek_true_aux:w { \peek_after:Nw _peek_remove_spaces: }`
 10928 `_peek_true_aux:w`
 10929 `}`
 10930 `\cs_new_protected:Npn _peek_remove_spaces:`
 10931 `{`
 10932 `\if_meaning:w \l_peek_token \c_space_token`
 10933 `\exp_after:wN _peek_true_remove:w`
 10934 `\else:`
 10935 `\group_align_safe_end:`
 10936 `\exp_after:wN _peek_false:w`
 10937 `\fi:`
 10938 `}`

(End definition for `\peek_remove_spaces:n` and `_peek_remove_spaces:`. This function is documented on page 274.)

`__peek_token_generic_aux:NNNTF` The generic functions store the test token in both implicit and explicit modes, and the **true** and **false** code as token lists, more or less. The two branches have to be absorbed here as the input stream needs to be cleared for the peek function itself. Here, #1 is `__peek_true_remove:w` when removing the token and `__peek_true_aux:w` otherwise.

```

10939 \cs_new_protected:Npn \__peek_token_generic_aux:NNNTF #1#2#3#4#5
10940 {
10941   \group_align_safe_begin:
10942   \cs_set_eq:NN \l__peek_search_token #3
10943   \tl_set:Nn \l__peek_search_tl {#3}
10944   \cs_set:Npx \__peek_true_aux:w
10945   {
10946     \exp_not:N \group_align_safe_end:
10947     \exp_not:n {#4}
10948   }
10949   \cs_set_eq:NN \__peek_true:w #1
10950   \cs_set:Npx \__peek_false:w
10951   {
10952     \exp_not:N \group_align_safe_end:
10953     \exp_not:n {#5}
10954   }
10955   \peek_after:Nw #2
10956 }

```

(End definition for `__peek_token_generic_aux:NNNTF`.)

`__peek_token_generic:NNTF` For token removal there needs to be a call to the auxiliary function which does the work.

```

\__peek_token_remove_generic:NNTF
10957 \cs_new_protected:Npn \__peek_token_generic:NNTF
10958 { \__peek_token_generic_aux:NNNTF \__peek_true_aux:w }
10959 \cs_new_protected:Npn \__peek_token_generic:NNT #1#2#3
10960 { \__peek_token_generic:NNTF #1 #2 {#3} { } }
10961 \cs_new_protected:Npn \__peek_token_generic:NNF #1#2#3
10962 { \__peek_token_generic:NNTF #1 #2 { } {#3} }
10963 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF
10964 { \__peek_token_generic_aux:NNNTF \__peek_true_remove:w }
10965 \cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
10966 { \__peek_token_remove_generic:NNTF #1 #2 {#3} { } }
10967 \cs_new_protected:Npn \__peek_token_remove_generic:NNF #1#2#3
10968 { \__peek_token_remove_generic:NNTF #1 #2 { } {#3} }

```

(End definition for `__peek_token_generic:NNTF` and `__peek_token_remove_generic:NNTF`.)

`__peek_execute_branches_meaning:` The meaning test is straight forward.

```

10969 \cs_new:Npn \__peek_execute_branches_meaning:
10970 {
10971   \if_meaning:w \l__peek_token \l__peek_search_token
10972   \exp_after:wN \__peek_true:w
10973   \else:
10974   \exp_after:wN \__peek_false:w
10975   \fi:
10976 }

```

(End definition for `__peek_execute_branches_meaning:.`)


```

\__peek_execute_branches_catcode:
\__peek_execute_branches_charcode:
\__peek_execute_branches_catcode_aux:
\__peek_execute_branches_catcode_auxii:N
\__peek_execute_branches_catcode_auxiii:

```

The catcode and charcode tests are very similar, and in order to use the same auxiliaries we do something a little bit odd, firing `\if_catcode:w` and `\if_charcode:w` before finding the operands for those tests, which are only given in the `auxii:N` and `auxiii:` auxiliaries. For our purposes, three kinds of tokens may follow the peeking function:

- control sequences which are not equal to a non-active character token (*e.g.*, macro, primitive);
- active characters which are not equal to a non-active character token (*e.g.*, macro, primitive);
- explicit non-active character tokens, or control sequences or active characters set equal to a non-active character token.

The first two cases are not distinguishable simply using TeX's `\futurelet`, because we can only access the `\meaning` of tokens in that way. In those cases, detected thanks to a comparison with `\scan_stop:`, we grab the following token, and compare it explicitly with the explicit search token stored in `\l__peek_search_tl`. The `\exp_not:N` prevents outer macros (coming from non- \LaTeX 3 code) from blowing up. In the third case, `\l__peek_token` is good enough for the test, and we compare it again with the explicit search token. Just like the peek token, the search token may be of any of the three types above, hence the need to use the explicit token that was given to the peek function.

```

10977 \cs_new:Npn \__peek_execute_branches_catcode:
10978 { \if_catcode:w \__peek_execute_branches_catcode_aux: }
10979 \cs_new:Npn \__peek_execute_branches_charcode:
10980 { \if_charcode:w \__peek_execute_branches_catcode_aux: }
10981 \cs_new:Npn \__peek_execute_branches_catcode_aux:
10982 {
10983     \if_catcode:w \exp_not:N \l__peek_token \scan_stop:
10984     \exp_after:wN \exp_after:wN
10985     \exp_after:wN \__peek_execute_branches_catcode_auxii:N
10986     \exp_after:wN \exp_not:N
10987     \else:
10988     \exp_after:wN \__peek_execute_branches_catcode_auxiii:
10989     \fi:
10990 }
10991 \cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1
10992 {
10993     \exp_not:N #1
10994     \exp_after:wN \exp_not:N \l__peek_search_tl
10995     \exp_after:wN \__peek_true:w
10996     \else:
10997     \exp_after:wN \__peek_false:w
10998     \fi:
10999     #1
11000 }
11001 \cs_new:Npn \__peek_execute_branches_catcode_auxiii:
11002 {
11003     \exp_not:N \l__peek_token
11004     \exp_after:wN \exp_not:N \l__peek_search_tl
11005     \exp_after:wN \__peek_true:w
11006     \else:
11007     \exp_after:wN \__peek_false:w
11008     \fi:

```

```
11009 }
```

(End definition for `_peek_execute_branches_catcode:` and others.)

`\peek_catcode:N \overline{TF}` The public functions themselves cannot be defined using `\prg_new_conditional:Npnn`.
`\peek_catcode_remove:N \overline{TF}` Instead, the TF, T, F variants are defined in terms of corresponding variants of `__-`
`\peek_charcode:N \overline{TF}` `peek_token_generic:NNTF` or `__peek_token_remove_generic:NNTF`, with first ar-
`\peek_charcode_remove:N \overline{TF}` gument one of `__peek_execute_branches_catcode:`, `__peek_execute_branches-`
`\peek_meaning:N \overline{TF}` `charcode:`, or `__peek_execute_branches_meaning:`.
`\peek_meaning_remove:N \overline{TF}`

```
11010 \tl_map_inline:nn { { catcode } { charcode } { meaning } }
11011 {
11012   \tl_map_inline:nn { { } { _remove } }
11013   {
11014     \tl_map_inline:nn { { TF } { T } { F } }
11015     {
11016       \cs_new_protected:cpx { peek_ #1 ##1 :N ####1 }
11017       {
11018         \exp_not:c { __peek_token ##1 _generic:NN ####1 }
11019         \exp_not:c { __peek_execute_branches_ #1 : }
11020       }
11021     }
11022   }
11023 }
```

(End definition for `\peek_catcode:N \overline{TF}` and others. These functions are documented on page 139.)

`\peek_catcode_ignore_spaces:N \overline{TF}` To ignore spaces, remove them using `\peek_remove_spaces:n` before running the tests.
`\peek_catcode_remove_ignore_spaces:N \overline{TF}`
`\peek_charcode_ignore_spaces:N \overline{TF}`
`\peek_charcode_remove_ignore_spaces:N \overline{TF}`
`\peek_meaning_ignore_spaces:N \overline{TF}`
`\peek_meaning_remove_ignore_spaces:N \overline{TF}`

```
11024 \tl_map_inline:nn
11025 {
11026   { catcode } { catcode_remove }
11027   { charcode } { charcode_remove }
11028   { meaning } { meaning_remove }
11029 }
11030 {
11031   \cs_new_protected:cpx { peek_#1_ignore_spaces:N $\overline{TF}$  } ##1##2##3
11032   {
11033     \peek_remove_spaces:n
11034     { \exp_not:c { peek_#1:N $\overline{TF}$  } ##1 {##2} {##3} }
11035   }
11036   \cs_new_protected:cpx { peek_#1_ignore_spaces:NT } ##1##2
11037   {
11038     \peek_remove_spaces:n
11039     { \exp_not:c { peek_#1:NT } ##1 {##2} }
11040   }
11041   \cs_new_protected:cpx { peek_#1_ignore_spaces:N $\overline{F}$  } ##1##2
11042   {
11043     \peek_remove_spaces:n
11044     { \exp_not:c { peek_#1:N $\overline{F}$  } ##1 {##2} }
11045   }
11046 }
```

(End definition for `\peek_catcode_ignore_spaces:N \overline{TF}` and others. These functions are documented on page 140.)

`\peek_N_type:TF` All tokens are N-type tokens, except in four cases: begin-group tokens, end-group tokens, space tokens with character code 32, and outer tokens. Since `\l_peek_token` might be outer, we cannot use the convenient `\bool_if:nTF` function, and must resort to the old trick of using `\ifodd` to expand a set of tests. The `false` branch of this test is taken if the token is one of the first three kinds of non-N-type tokens (explicit or implicit), thus we call `__peek_false:w`. In the `true` branch, we must detect outer tokens, without impacting performance too much for non-outer tokens. The first filter is to search for `outer` in the `\meaning` of `\l_peek_token`. If that is absent, `__peek_use_none_delimit_by_s_stop:w` cleans up, and we call `__peek_true:w`. Otherwise, the token can be a non-outer macro or a primitive mark whose parameter or replacement text contains `outer`, it can be the primitive `\outer`, or it can be an outer token. Macros and marks would have `ma` in the part before the first occurrence of `outer`; the meaning of `\outer` has nothing after `outer`, contrarily to outer macros; and that covers all cases, calling `__peek_true:w` or `__peek_false:w` as appropriate. Here, there is no *search token*, so we feed a dummy `\scan_stop:` to the `__peek_token_generic:NNTF` function.

```

11047 \group_begin:
11048   \cs_set_protected:Npn \__peek_tmp:w #1 \s__peek_stop
11049   {
11050     \cs_new_protected:Npn \__peek_execute_branches_N_type:
11051     {
11052       \if_int_odd:w
11053         \if_catcode:w \exp_not:N \l_peek_token { 0 \exp_stop_f: \fi:
11054         \if_catcode:w \exp_not:N \l_peek_token } 0 \exp_stop_f: \fi:
11055         \if_meaning:w \l_peek_token \c_space_token 0 \exp_stop_f: \fi:
11056         1 \exp_stop_f:
11057         \exp_after:wN \__peek_N_type:w
11058         \token_to_meaning:N \l_peek_token
11059         \s__peek_mark \__peek_N_type_aux:nnw
11060         #1 \s__peek_mark \__peek_use_none_delimit_by_s_stop:w
11061         \s__peek_stop
11062         \exp_after:wN \__peek_true:w
11063       \else:
11064         \exp_after:wN \__peek_false:w
11065       \fi:
11066     }
11067     \cs_new_protected:Npn \__peek_N_type:w ##1 #1 ##2 \s__peek_mark ##3
11068     { ##3 {##1} {##2} }
11069   }
11070   \exp_after:wN \__peek_tmp:w \tl_to_str:n { outer } \s__peek_stop
11071 \group_end:
11072 \cs_new_protected:Npn \__peek_N_type_aux:nnw #1 #2 #3 \fi:
11073 {
11074   \fi:
11075   \tl_if_in:noTF {#1} { \tl_to_str:n {ma} }
11076   { \__peek_true:w }
11077   { \tl_if_empty:nTF {#2} { \__peek_true:w } { \__peek_false:w } }
11078 }
11079 \cs_new_protected:Npn \peek_N_type:TF
11080 {
11081   \__peek_token_generic:NNTF
11082   \__peek_execute_branches_N_type: \scan_stop:
11083 }
11084 \cs_new_protected:Npn \peek_N_type:T

```

```

11085 { \__peek_token_generic:NNT \__peek_execute_branches_N_type: \scan_stop: }
11086 \cs_new_protected:Npn \peek_N_type:F
11087 { \__peek_token_generic:NNF \__peek_execute_branches_N_type: \scan_stop: }
(End definition for \peek_N_type:TF and others. This function is documented on page 142.)
11088 \</package>

```

18 l3prop implementation

The following test files are used for this code: `m3prop001`, `m3prop002`, `m3prop003`, `m3prop004`, `m3show001`.

```

11089 \*package>
11090 \@@=prop>

```

A property list is a macro whose top-level expansion is of the form

```

\__prop \__prop_pair:wn <key1> \__prop {<value1>}
...
\__prop_pair:wn <keyn> \__prop {<valuen>}

```

where `__prop` is a scan mark (equal to `\scan_stop:`), and `__prop_pair:wn` can be used to map through the property list.

`__prop` The internal token used at the beginning of property lists. This is also used after each `<key>` (see `__prop_pair:wn`).

(End definition for `__prop`.)

```

\__prop_pair:wn \__prop_pair:wn <key> \__prop {<item>}

```

The internal token used to begin each key–value pair in the property list. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.

(End definition for `__prop_pair:wn`.)

`\l__prop_internal_tl` Token list used to store new key–value pairs to be inserted by functions of the `\prop_put:Nnn` family.

(End definition for `\l__prop_internal_tl`.)

```

\__prop_split:NnTF \__prop_split:NnTF <property list> {<key>} {<true code>} {<false code>}

```

Updated: 2013-01-08

Splits the `<property list>` at the `<key>`, giving three token lists: the `<extract>` of `<property list>` before the `<key>`, the `<value>` associated with the `<key>` and the `<extract>` of the `<property list>` after the `<value>`. Both `<extracts>` retain the internal structure of a property list, and the concatenation of the two `<extracts>` is a property list. If the `<key>` is present in the `<property list>` then the `<true code>` is left in the input stream, with `#1`, `#2`, and `#3` replaced by the first `<extract>`, the `<value>`, and the second `<extract>`. If the `<key>` is not present in the `<property list>` then the `<false code>` is left in the input stream, with no trailing material. Both `<true code>` and `<false code>` are used in the replacement text of a macro defined internally, hence macro parameter characters should be doubled, except `#1`, `#2`, and `#3` which stand in the `<true code>` for the three extracts from the property list. The `<key>` comparison takes place as described for `\str_if_eq:nn`.

`\s__prop` A private scan mark is used as a marker after each key, and at the very beginning of the property list.

```
11091 \scan_new:N \s__prop
```

(End definition for \s__prop.)

`__prop_pair:wn` The delimiter is always defined, but when misused simply triggers an error and removes its argument.

```
11092 \cs_new:Npn \__prop_pair:wn #1 \s__prop #2
```

```
11093 { \__kernel_msg_expandable_error:nn { kernel } { misused-prop } }
```

(End definition for __prop_pair:wn.)

`\l__prop_internal_tl` Token list used to store the new key–value pair inserted by `\prop_put:Nnn` and friends.

```
11094 \tl_new:N \l__prop_internal_tl
```

(End definition for \l__prop_internal_tl.)

`\c_empty_prop` An empty prop.

```
11095 \tl_const:Nn \c_empty_prop { \s__prop }
```

(End definition for \c_empty_prop. This variable is documented on page 151.)

18.1 Internal auxiliaries

`\s__prop_mark` Internal scan marks.

```
\s__prop_stop 11096 \scan_new:N \s__prop_mark
```

```
11097 \scan_new:N \s__prop_stop
```

(End definition for \s__prop_mark and \s__prop_stop.)

`\q__prop_recursion_tail` Internal recursion quarks.

```
\q__prop_recursion_stop 11098 \quark_new:N \q__prop_recursion_tail
```

```
11099 \quark_new:N \q__prop_recursion_stop
```

(End definition for \q__prop_recursion_tail and \q__prop_recursion_stop.)

`_prop_if_recursion_tail_stop:n` Functions to query recursion quarks.

```
\_prop_if_recursion_tail_stop:o 11100 \__kernel_quark_new_test:N \_prop_if_recursion_tail_stop:n
```

```
11101 \cs_generate_variant:Nn \_prop_if_recursion_tail_stop:n { o }
```

(End definition for _prop_if_recursion_tail_stop:n and _prop_if_recursion_tail_stop:o.)

18.2 Allocation and initialisation

\prop_new:N Property lists are initialized with the value `\c_empty_prop`.

```
\prop_new:c 11102 \cs_new_protected:Npn \prop_new:N #1
11103 {
11104     \__kernel_chk_if_free_cs:N #1
11105     \cs_gset_eq:NN #1 \c_empty_prop
11106 }
11107 \cs_generate_variant:Nn \prop_new:N { c }
```

(End definition for `\prop_new:N`. This function is documented on page 145.)

\prop_clear:N The same idea for clearing.

```
\prop_clear:c 11108 \cs_new_protected:Npn \prop_clear:N #1
\prop_gclear:N 11109 { \prop_set_eq:NN #1 \c_empty_prop }
\prop_gclear:c 11110 \cs_generate_variant:Nn \prop_clear:N { c }
11111 \cs_new_protected:Npn \prop_gclear:N #1
11112 { \prop_gset_eq:NN #1 \c_empty_prop }
11113 \cs_generate_variant:Nn \prop_gclear:N { c }
```

(End definition for `\prop_clear:N` and `\prop_gclear:N`. These functions are documented on page 145.)

\prop_clear_new:N Once again a simple variation of the token list functions.

```
\prop_clear_new:c 11114 \cs_new_protected:Npn \prop_clear_new:N #1
\prop_gclear_new:N 11115 { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
\prop_gclear_new:c 11116 \cs_generate_variant:Nn \prop_clear_new:N { c }
11117 \cs_new_protected:Npn \prop_gclear_new:N #1
11118 { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
11119 \cs_generate_variant:Nn \prop_gclear_new:N { c }
```

(End definition for `\prop_clear_new:N` and `\prop_gclear_new:N`. These functions are documented on page 145.)

\prop_set_eq:NN These are simply copies from the token list functions.

```
\prop_set_eq:cN 11120 \cs_new_eq:NN \prop_set_eq:NN \tl_set_eq:NN
\prop_set_eq:Nc 11121 \cs_new_eq:NN \prop_set_eq:Nc \tl_set_eq:Nc
\prop_set_eq:cc 11122 \cs_new_eq:NN \prop_set_eq:cN \tl_set_eq:cN
\prop_gset_eq:NN 11123 \cs_new_eq:NN \prop_set_eq:cc \tl_set_eq:cc
\prop_gset_eq:cN 11124 \cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
\prop_gset_eq:Nc 11125 \cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
\prop_gset_eq:cN 11126 \cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
\prop_gset_eq:cc 11127 \cs_new_eq:NN \prop_gset_eq:cc \tl_gset_eq:cc
```

(End definition for `\prop_set_eq:NN` and `\prop_gset_eq:NN`. These functions are documented on page 145.)

\l_tmpa_prop We can now initialize the scratch variables.

```
\l_tmpb_prop 11128 \prop_new:N \l_tmpa_prop
\g_tmpa_prop 11129 \prop_new:N \l_tmpb_prop
\g_tmpb_prop 11130 \prop_new:N \g_tmpa_prop
11131 \prop_new:N \g_tmpb_prop
```

(End definition for `\l_tmpa_prop` and others. These variables are documented on page 150.)

\l__prop_internal_prop Property list used by `\prop_set_from_keyval:Nn` and others.

```
11132 \prop_new:N \l__prop_internal_prop
```

(End definition for \l__prop_internal_prop.)

`\prop_set_from_keyval:Nn` To avoid tracking throughout the loop the variable name and whether the assignment
`\prop_set_from_keyval:cn` is local/global, do everything in a scratch variable and empty it afterwards to avoid
`\prop_gset_from_keyval:Nn` wasting memory. Loop through items separated by commas, with `\prg_do_nothing:`
`\prop_gset_from_keyval:cn` to avoid losing braces. After checking for termination, split the item at the first and
`\prop_const_from_keyval:Nn` then at the second = (which ought to be the first of the trailing = that we added). For
`\prop_const_from_keyval:cn` both splits trim spaces and call a function (first `__prop_from_keyval_key:w` then `__-`
`__prop_from_keyval:n` `prop_from_keyval_value:w`), followed by the trimmed material, `\s__prop_mark`, the
`__prop_from_keyval_loop:w` subsequent part of the item, and the trailing =’s and `\s__prop_stop`. After finding
`__prop_from_keyval_split:Nw` the *<key>* just store it after `\s__prop_stop`. After finding the *<value>* ignore completely
`__prop_from_keyval_key:n` empty items (both trailing = were used as delimiters and all parts are empty); if the
`__prop_from_keyval_key:w` remaining part #2 consists exactly of the second trailing = (namely there was exactly one
`__prop_from_keyval_value:n` = in the item) then output one key–value pair for the property list; otherwise complain
`__prop_from_keyval_value:w` about a missing or extra =.

```

11133 \cs_new_protected:Npn \prop_set_from_keyval:Nn #1#2
11134 {
11135   \prop_clear:N \l__prop_internal_prop
11136   \__prop_from_keyval:n {#2}
11137   \prop_set_eq:NN #1 \l__prop_internal_prop
11138   \prop_clear:N \l__prop_internal_prop
11139 }
11140 \cs_generate_variant:Nn \prop_set_from_keyval:Nn { c }
11141 \cs_new_protected:Npn \prop_gset_from_keyval:Nn #1#2
11142 {
11143   \prop_clear:N \l__prop_internal_prop
11144   \__prop_from_keyval:n {#2}
11145   \prop_gset_eq:NN #1 \l__prop_internal_prop
11146   \prop_clear:N \l__prop_internal_prop
11147 }
11148 \cs_generate_variant:Nn \prop_gset_from_keyval:Nn { c }
11149 \cs_new_protected:Npn \prop_const_from_keyval:Nn #1#2
11150 {
11151   \prop_clear:N \l__prop_internal_prop
11152   \__prop_from_keyval:n {#2}
11153   \tl_const:Nx #1 { \exp_not:o \l__prop_internal_prop }
11154   \prop_clear:N \l__prop_internal_prop
11155 }
11156 \cs_generate_variant:Nn \prop_const_from_keyval:Nn { c }
11157 \cs_new_protected:Npn \__prop_from_keyval:n #1
11158 {
11159   \__prop_from_keyval_loop:w \prg_do_nothing: #1 ,
11160   \q__prop_recursion_tail , \q__prop_recursion_stop
11161 }
11162 \cs_new_protected:Npn \__prop_from_keyval_loop:w #1 ,
11163 {
11164   \__prop_if_recursion_tail_stop:o {#1}
11165   \__prop_from_keyval_split:Nw \__prop_from_keyval_key:n
11166   #1 = = \s__prop_stop {#1}
11167   \__prop_from_keyval_loop:w \prg_do_nothing:
11168 }
11169 \cs_new_protected:Npn \__prop_from_keyval_split:Nw #1#2 =
11170 { \tl_trim_spaces_apply:oN {#2} #1 }

```

```

11171 \cs_new_protected:Npn \__prop_from_keyval_key:n #1
11172 { \__prop_from_keyval_key:w #1 \s__prop_mark }
11173 \cs_new_protected:Npn \__prop_from_keyval_key:w #1 \s__prop_mark #2 \s__prop_stop
11174 {
11175   \__prop_from_keyval_split:Nw \__prop_from_keyval_value:n
11176   \prg_do_nothing: #2 \s__prop_stop {#1}
11177 }
11178 \cs_new_protected:Npn \__prop_from_keyval_value:n #1
11179 { \__prop_from_keyval_value:w #1 \s__prop_mark }
11180 \cs_new_protected:Npn \__prop_from_keyval_value:w #1 \s__prop_mark #2 \s__prop_stop #3#4
11181 {
11182   \tl_if_empty:nF { #3 #1 #2 }
11183   {
11184     \str_if_eq:nnTF {#2} { = }
11185     { \prop_put:Nnn \l__prop_internal_prop {#3} {#1} }
11186     {
11187       \__kernel_msg_error:nnx { kernel } { prop-keyval }
11188       { \exp_not:o {#4} }
11189     }
11190   }
11191 }

```

(End definition for `\prop_set_from_keyval:Nn` and others. These functions are documented on page 145.)

18.3 Accessing data in property lists

```

\__prop_split:NnTF
\__prop_split_aux:NnTF
\__prop_split_aux:w

```

This function is used by most of the module, and hence must be fast. It receives a $\langle \textit{property list} \rangle$, a $\langle \textit{key} \rangle$, a $\langle \textit{true code} \rangle$ and a $\langle \textit{false code} \rangle$. The aim is to split the $\langle \textit{property list} \rangle$ at the given $\langle \textit{key} \rangle$ into the $\langle \textit{extract}_1 \rangle$ before the key–value pair, the $\langle \textit{value} \rangle$ associated with the $\langle \textit{key} \rangle$ and the $\langle \textit{extract}_2 \rangle$ after the key–value pair. This is done using a delimited function, whose definition is as follows, where the $\langle \textit{key} \rangle$ is turned into a string.

```

\cs_set:Npn \__prop_split_aux:w #1
\__prop_pair:wn \__prop #2
#3 \s__prop_mark #4 #5 \s__prop_stop
{ #4 { \__prop_split_aux:w #1 } { \__prop_pair:wn #2 } { #3 } }

```

If the $\langle \textit{key} \rangle$ is present in the property list, `__prop_split_aux:w`'s #1 is the part before the $\langle \textit{key} \rangle$, #2 is the $\langle \textit{value} \rangle$, #3 is the part after the $\langle \textit{key} \rangle$, #4 is `\use_i:nn`, and #5 is additional tokens that we do not care about. The $\langle \textit{true code} \rangle$ is left in the input stream, and can use the parameters #1, #2, #3 for the three parts of the property list as desired. Namely, the original property list is in this case #1 `__prop_pair:wn __prop {#2} #3`.

If the $\langle \textit{key} \rangle$ is not there, then the $\langle \textit{function} \rangle$ is `\use_ii:nn`, which keeps the $\langle \textit{false code} \rangle$.

```

11192 \cs_new_protected:Npn \__prop_split:NnTF #1#2
11193 { \exp_args:NNo \__prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }
11194 \cs_new_protected:Npn \__prop_split_aux:NnTF #1#2#3#4
11195 {
11196   \cs_set:Npn \__prop_split_aux:w ##1
11197   \__prop_pair:wn #2 \s__prop ##2 ##3 \s__prop_mark ##4 ##5 \s__prop_stop
11198   { ##4 {#3} {#4} }

```



```

11199     \exp_after:wN \__prop_split_aux:w #1 \s__prop_mark \use_i:nn
11200     \__prop_pair:wn #2 \s__prop { } \s__prop_mark \use_ii:nn \s__prop_stop
11201   }
11202 \cs_new:Npn \__prop_split_aux:w { }

```

(End definition for `__prop_split:NnTF`, `__prop_split_aux:NnTF`, and `__prop_split_aux:w`.)

`\prop_remove:Nn` Deleting from a property starts by splitting the list. If the key is present in the property list, the returned value is ignored. If the key is missing, nothing happens.

```

\prop_remove:NV
\prop_remove:cn 11203 \cs_new_protected:Npn \prop_remove:Nn #1#2
\prop_remove:cV 11204 {
\prop_gremove:Nn 11205   \__prop_split:NnTF #1 {#2}
\prop_gremove:NV 11206   { \tl_set:Nn #1 { ##1 ##3 } }
\prop_gremove:cn 11207   { }
\prop_gremove:cV 11208 }
11209 \cs_new_protected:Npn \prop_gremove:Nn #1#2
11210 {
11211   \__prop_split:NnTF #1 {#2}
11212   { \tl_gset:Nn #1 { ##1 ##3 } }
11213   { }
11214 }
11215 \cs_generate_variant:Nn \prop_remove:Nn { NV }
11216 \cs_generate_variant:Nn \prop_remove:Nn { c , cV }
11217 \cs_generate_variant:Nn \prop_gremove:Nn { NV }
11218 \cs_generate_variant:Nn \prop_gremove:Nn { c , cV }

```

(End definition for `\prop_remove:Nn` and `\prop_gremove:Nn`. These functions are documented on page 147.)

`\prop_get:NnN` Getting an item from a list is very easy: after splitting, if the key is in the property list, just set the token list variable to the return value, otherwise to `\q_no_value`.

```

\prop_get:NVN
\prop_get:NoN 11219 \cs_new_protected:Npn \prop_get:NnN #1#2#3
\prop_get:cnN 11220 {
\prop_get:cVN 11221   \__prop_split:NnTF #1 {#2}
\prop_get:coN 11222   { \tl_set:Nn #3 {##2} }
11223   { \tl_set:Nn #3 { \q_no_value } }
11224 }
11225 \cs_generate_variant:Nn \prop_get:NnN { NV , Nv , No }
11226 \cs_generate_variant:Nn \prop_get:NnN { c , cV , cv , co }

```

(End definition for `\prop_get:NnN`. This function is documented on page 146.)

`\prop_pop:NnN` Popping a value also starts by doing the split. If the key is present, save the value in the token list and update the property list as when deleting. If the key is missing, save `\q_no_value` in the token list.

```

\prop_pop:cnN 11227 \cs_new_protected:Npn \prop_pop:NnN #1#2#3
\prop_pop:coN 11228 {
\prop_gpop:NoN 11229   \__prop_split:NnTF #1 {#2}
\prop_gpop:cnN 11230   {
\prop_gpop:coN 11231     \tl_set:Nn #3 {##2}
11232     \tl_set:Nn #1 { ##1 ##3 }
11233   }
11234   { \tl_set:Nn #3 { \q_no_value } }
11235 }

```

```

11236 \cs_new_protected:Npn \prop_gpop:NnN #1#2#3
11237 {
11238   \__prop_split:NnTF #1 {#2}
11239   {
11240     \tl_set:Nn #3 {##2}
11241     \tl_gset:Nn #1 { ##1 ##3 }
11242   }
11243   { \tl_set:Nn #3 { \q_no_value } }
11244 }
11245 \cs_generate_variant:Nn \prop_pop:NnN { No }
11246 \cs_generate_variant:Nn \prop_pop:NnN { c , co }
11247 \cs_generate_variant:Nn \prop_gpop:NnN { No }
11248 \cs_generate_variant:Nn \prop_gpop:NnN { c , co }

```

(End definition for `\prop_pop:NnN` and `\prop_gpop:NnN`. These functions are documented on page 146.)

\prop_item:Nn Getting the value corresponding to a key in a property list in an expandable fashion is similar to mapping some tokens. Go through the property list one $\langle key \rangle$ – $\langle value \rangle$ pair at a time: the arguments of `__prop_item_Nn:nwn` are the $\langle key \rangle$ we are looking for, a $\langle key \rangle$ of the property list, and its associated value. The $\langle keys \rangle$ are compared (as strings). If they match, the $\langle value \rangle$ is returned, within `\exp_not:n`. The loop terminates even if the $\langle key \rangle$ is missing, and yields an empty value, because we have appended the appropriate $\langle key \rangle$ – $\langle empty\ value \rangle$ pair to the property list.

```

11249 \cs_new:Npn \prop_item:Nn #1#2
11250 {
11251   \exp_last_unbraced:Noo \__prop_item_Nn:nwn { \tl_to_str:n {#2} } #1
11252   \__prop_pair:wn \tl_to_str:n {#2} \s_prop { }
11253   \prg_break_point:
11254 }
11255 \cs_new:Npn \__prop_item_Nn:nwn #1#2 \__prop_pair:wn #3 \s_prop #4
11256 {
11257   \str_if_eq:eeTF {#1} {#3}
11258   { \prg_break:n { \exp_not:n {#4} } }
11259   { \__prop_item_Nn:nwn {#1} }
11260 }
11261 \cs_generate_variant:Nn \prop_item:Nn { c }

```

(End definition for `\prop_item:Nn` and `__prop_item_Nn:nwn`. This function is documented on page 147.)

\prop_count:N Counting the key–value pairs in a property list is done using the same approach as for other count functions: turn each entry into a +1 then use integer evaluation to actually do the mathematics.

```

11262 \cs_new:Npn \prop_count:N #1
11263 {
11264   \int_eval:n
11265   {
11266     0
11267     \prop_map_function:NN #1 \__prop_count:nn
11268   }
11269 }
11270 \cs_new:Npn \__prop_count:nn #1#2 { + 1 }
11271 \cs_generate_variant:Nn \prop_count:N { c }

```

(End definition for `\prop_count:N` and `__prop_count:nn`. This function is documented on page 147.)

`\prop_pop:NnTF` Popping an item from a property list, keeping track of whether the key was present or not, is implemented as a conditional. If the key was missing, neither the property list, nor the token list are altered. Otherwise, `\prg_return_true:` is used after the assignments.

`\prop_gpop:NnTF`

```

11272 \prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
11273 {
11274   \__prop_split:NnTF #1 {#2}
11275   {
11276     \tl_set:Nn #3 {##2}
11277     \tl_set:Nn #1 { ##1 ##3 }
11278     \prg_return_true:
11279   }
11280   { \prg_return_false: }
11281 }
11282 \prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
11283 {
11284   \__prop_split:NnTF #1 {#2}
11285   {
11286     \tl_set:Nn #3 {##2}
11287     \tl_gset:Nn #1 { ##1 ##3 }
11288     \prg_return_true:
11289   }
11290   { \prg_return_false: }
11291 }
11292 \prg_generate_conditional_variant:Nnn \prop_pop:NnN { c } { T , F , TF }
11293 \prg_generate_conditional_variant:Nnn \prop_gpop:NnN { c } { T , F , TF }

```

(End definition for `\prop_pop:NnNTF` and `\prop_gpop:NnNTF`. These functions are documented on page 148.)

`\prop_put:Nnn` Since the branches of `__prop_split:NnTF` are used as the replacement text of an internal macro, and since the `<key>` and new `<value>` may contain arbitrary tokens, it is not safe to include them in the argument of `__prop_split:NnTF`. We thus start by storing in `\l__prop_internal_tl` tokens which (after x-expansion) encode the key–value pair. This variable can safely be used in `__prop_split:NnTF`. If the `<key>` was absent, append the new key–value to the list. Otherwise concatenate the extracts `##1` and `##3` with the new key–value pair `\l__prop_internal_tl`. The updated entry is placed at the same spot as the original `<key>` in the property list, preserving the order of entries.

```

11294 \cs_new_protected:Npn \prop_put:Nnn { \__prop_put:NNnn \tl_set:Nx }
11295 \cs_new_protected:Npn \prop_gput:Nnn { \__prop_put:NNnn \tl_gset:Nx }
11296 \cs_new_protected:Npn \__prop_put:NNnn #1#2#3#4
11297 {
11298   \tl_set:Nn \l__prop_internal_tl
11299   {
11300     \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
11301     \s__prop { \exp_not:n {#4} }
11302   }
11303   \__prop_split:NnTF #2 {#3}
11304   { #1 #2 { \exp_not:n {##1} \l__prop_internal_tl \exp_not:n {##3} } }
11305   { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
11306 }
11307 \cs_generate_variant:Nn \prop_put:Nnn

```

`\prop_gput:Nnn`

`\prop_gput:NnV`
`\prop_gput:Nno`
`\prop_gput:Nnx`
`\prop_gput:NvN`
`\prop_gput:NvV`
`\prop_gput:Nnon`
`\prop_gput:Nnoo`
`\prop_gput:Nnxx`
`\prop_gput:Ncnn`

```

11308 {      NnV , Nno , Nnx , NV , NVV , NVx , Nvx , No , Noo , Nxx }
11309 \cs_generate_variant:Nn \prop_put:Nnn
11310 { c , cnV , cno , cnx , cV , cVV , cVx , cvx , co , coo , cxx }
11311 \cs_generate_variant:Nn \prop_gput:Nnn
11312 {      NnV , Nno , Nnx , NV , NVV , NVx , Nvx , No , Noo , Nxx }
11313 \cs_generate_variant:Nn \prop_gput:Nnn
11314 { c , cnV , cno , cnx , cV , cVV , cVx , cvx , co , coo , cxx }

```

(End definition for `\prop_put:Nnn` and others. These functions are documented on page 146.)

```

\prop_put_if_new:Nnn Adding conditionally also splits. If the key is already present, the three brace groups
\prop_put_if_new:cnn given by \__prop_split:NnTF are removed. If the key is new, then the value is added,
\prop_gput_if_new:Nnn being careful to convert the key to a string using \tl_to_str:n.
\prop_gput_if_new:cnn
\__prop_put_if_new:NNnn
11315 \cs_new_protected:Npn \prop_put_if_new:Nnn
11316 { \__prop_put_if_new:NNnn \tl_set:Nx }
11317 \cs_new_protected:Npn \prop_gput_if_new:Nnn
11318 { \__prop_put_if_new:NNnn \tl_gset:Nx }
11319 \cs_new_protected:Npn \__prop_put_if_new:NNnn #1#2#3#4
11320 {
11321   \tl_set:Nn \l__prop_internal_tl
11322   {
11323     \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
11324     \s__prop \exp_not:n { {#4} }
11325   }
11326   \__prop_split:NnTF #2 {#3}
11327   { }
11328   { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
11329 }
11330 \cs_generate_variant:Nn \prop_put_if_new:Nnn { c }
11331 \cs_generate_variant:Nn \prop_gput_if_new:Nnn { c }

```

(End definition for `\prop_put_if_new:Nnn`, `\prop_gput_if_new:Nnn`, and `__prop_put_if_new:NNnn`. These functions are documented on page 146.)

18.4 Property list conditionals

```

\prop_if_exist_p:N Copies of the cs functions defined in l3basics.
\prop_if_exist_p:c 11332 \prg_new_eq_conditional:NNn \prop_if_exist:N \cs_if_exist:N
\prop_if_exist:NTF 11333 { TF , T , F , p }
\prop_if_exist:cTF 11334 \prg_new_eq_conditional:NNn \prop_if_exist:c \cs_if_exist:c
11335 { TF , T , F , p }

```

(End definition for `\prop_if_exist:NTF`. This function is documented on page 147.)

```

\prop_if_empty_p:N Same test as for token lists.
\prop_if_empty_p:c 11336 \prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
\prop_if_empty:NTF 11337 {
11338   \tl_if_eq:NNTF #1 \c_empty_prop
11339   \prg_return_true: \prg_return_false:
11340 }
11341 \prg_generate_conditional_variant:Nnn \prop_if_empty:N
11342 { c } { p , T , F , TF }

```

(End definition for `\prop_if_empty:NTF`. This function is documented on page 147.)

```

\prop_if_in_p:Nn Testing expandably if a key is in a property list requires to go through the key–value
\prop_if_in_p:NV pairs one by one. This is rather slow, and a faster test would be
\prop_if_in_p:No
\prop_if_in_p:cn \prg_new_protected_conditional:Npnn \prop_if_in:Nn #1 #2
\prop_if_in_p:cV {
\prop_if_in_p:co \@@_split:NnTF #1 {#2}
\prop_if_in:NnTF { \prg_return_true: }
\prop_if_in:NVTF { \prg_return_false: }
\prop_if_in:NoTF }
\prop_if_in:cnTF but \__prop_split:NnTF is non-expandable.
\prop_if_in:cVTF Instead, the key is compared to each key in turn using \str_if_eq:ee, which is
\prop_if_in:coTF expandable. To terminate the mapping, we append to the property list the key that is
\__prop_if_in:nwwn searched for. This second \tl_to_str:n is not expanded at the start, but only when
\__prop_if_in:N included in the \str_if_eq:ee. It cannot make the breaking mechanism choke, because
the arbitrary token list material is enclosed in braces. The second argument of \__prop_
if_in:nwwn is most often empty. When the key is found in the list, \__prop_if_in:N
receives \__prop_pair:wn, and if it is found as the extra item, the function receives
\q__prop_recursion_tail, easily recognizable.

```

Here, \prop_map_function:NN is not sufficient for the mapping, since it can only map a single token, and cannot carry the key that is searched for.

```

11343 \prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
11344 {
11345     \exp_last_unbraced:Noo \__prop_if_in:nwwn { \tl_to_str:n {#2} } #1
11346     \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
11347     \q__prop_recursion_tail
11348     \prg_break_point:
11349 }
11350 \cs_new:Npn \__prop_if_in:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
11351 {
11352     \str_if_eq:eeTF {#1} {#3}
11353     { \__prop_if_in:N }
11354     { \__prop_if_in:nwwn {#1} }
11355 }
11356 \cs_new:Npn \__prop_if_in:N #1
11357 {
11358     \if_meaning:w \q__prop_recursion_tail #1
11359     \prg_return_false:
11360     \else:
11361     \prg_return_true:
11362     \fi:
11363     \prg_break:
11364 }
11365 \prg_generate_conditional_variant:Nnn \prop_if_in:Nn
11366 { NV , No , c , cV , co } { p , T , F , TF }

```

(End definition for \prop_if_in:NnTF, __prop_if_in:nwwn, and __prop_if_in:N. This function is documented on page 148.)

18.5 Recovering values from property lists with branching

```

\prop_get:NnTF Getting the value corresponding to a key, keeping track of whether the key was present
\prop_get:NVNTF or not, is implemented as a conditional (with side effects). If the key was absent, the
\prop_get:NoNTF token list is not altered.
\prop_get:cnNTF
\prop_get:cVNTF
\prop_get:coNTF

```

```

11367 \prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
11368 {
11369   \__prop_split:NnTF #1 {#2}
11370   {
11371     \tl_set:Nn #3 {##2}
11372     \prg_return_true:
11373   }
11374   { \prg_return_false: }
11375 }
11376 \prg_generate_conditional_variant:Nnn \prop_get:NnN
11377 { NV , Nv , No , c , cV , cv , co } { T , F , TF }

```

(End definition for `\prop_get:NnNTF`. This function is documented on page 148.)

18.6 Mapping to property lists

`\prop_map_function:NN` The argument delimited by `__prop_pair:wn` is empty except at the end of the loop where it is `\prg_break:.` No need for any quark test.

```

11378 \cs_new:Npn \prop_map_function:NN #1#2
11379 {
11380   \exp_after:wN \use_i_ii:nnn
11381   \exp_after:wN \__prop_map_function:Nwwn
11382   \exp_after:wN #2
11383   #1
11384   \prg_break: \__prop_pair:wn \s__prop { } \prg_break_point:
11385   \prg_break_point:Nn \prop_map_break: { }
11386 }
11387 \cs_new:Npn \__prop_map_function:Nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
11388 {
11389   #2
11390   #1 {#3} {#4}
11391   \__prop_map_function:Nwwn #1
11392 }
11393 \cs_generate_variant:Nn \prop_map_function:NN { Nc , c , cc }

```

(End definition for `\prop_map_function:NN` and `__prop_map_function:Nwwn`. This function is documented on page 149.)

`\prop_map_inline:Nn` Mapping in line requires a nesting level counter. Store the current definition of `__prop_pair:wn`, and define it anew. At the end of the loop, revert to the earlier definition. Note that besides pairs of the form `__prop_pair:wn <key> \s__prop {<value>}`, there are a leading and a trailing tokens, but both are equal to `\scan_stop:`, hence have no effect in such inline mapping. Such `\scan_stop:` could have affected ligatures if they appeared during the mapping.

```

11394 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
11395 {
11396   \cs_gset_eq:cN
11397   { __prop_map_ \int_use:N \g__kernel_prg_map_int :wn } \__prop_pair:wn
11398   \int_gincr:N \g__kernel_prg_map_int
11399   \cs_gset_protected:Npn \__prop_pair:wn ##1 \s__prop ##2 {#2}
11400   #1
11401   \prg_break_point:Nn \prop_map_break:
11402   {
11403     \int_gdecr:N \g__kernel_prg_map_int

```

```

11404         \cs_gset_eq:Nc \__prop_pair:wn
11405         { \__prop_map_ \int_use:N \g__kernel_prg_map_int :wn }
11406     }
11407 }
11408 \cs_generate_variant:Nn \prop_map_inline:Nn { c }

```

(End definition for `\prop_map_inline:Nn`. This function is documented on page 149.)

`\prop_map_tokens:Nn` The mapping is very similar to `\prop_map_function:NN`. The `\use_i:nn` removes the
`\prop_map_tokens:cn` leading `\s__prop`. The odd construction `\use:n {#1}` allows #1 to contain any token
`__prop_map_tokens:nwn` without interfering with `\prop_map_break:.` The loop stops when the argument delimited by `__prop_pair:wn` is `\prg_break:` instead of being empty.

```

11409 \cs_new:Npn \prop_map_tokens:Nn #1#2
11410 {
11411     \exp_last_unbraced:Nno
11412     \use_i:nn { \__prop_map_tokens:nwn {#2} } #1
11413     \prg_break: \__prop_pair:wn \s__prop { } \prg_break_point:
11414     \prg_break_point:Nn \prop_map_break: { }
11415 }
11416 \cs_new:Npn \__prop_map_tokens:nwn #1#2 \__prop_pair:wn #3 \s__prop #4
11417 {
11418     #2
11419     \use:n {#1} {#3} {#4}
11420     \__prop_map_tokens:nwn {#1}
11421 }
11422 \cs_generate_variant:Nn \prop_map_tokens:Nn { c }

```

(End definition for `\prop_map_tokens:Nn` and `__prop_map_tokens:nwn`. This function is documented on page 149.)

`\prop_map_break:` The break statements are based on the general `\prg_map_break:Nn`.
`\prop_map_break:n`

```

11423 \cs_new:Npn \prop_map_break:
11424 { \prg_map_break:Nn \prop_map_break: { } }
11425 \cs_new:Npn \prop_map_break:n
11426 { \prg_map_break:Nn \prop_map_break: }

```

(End definition for `\prop_map_break:` and `\prop_map_break:n`. These functions are documented on page 149.)

18.7 Viewing property lists

`\prop_show:N` Apply the general `__kernel_chk_defined:NT` and `\msg_show:nnnnnn`. Contrarily to
`\prop_show:c` sequences and comma lists, we use `\msg_show_item:nn` to format both the key and the
`\prop_log:N` value for each pair.

```

\prop_log:c
11427 \cs_new_protected:Npn \prop_show:N { \__prop_show:NN \msg_show:nnxxxx }
11428 \cs_generate_variant:Nn \prop_show:N { c }
11429 \cs_new_protected:Npn \prop_log:N { \__prop_show:NN \msg_log:nnxxxx }
11430 \cs_generate_variant:Nn \prop_log:N { c }
11431 \cs_new_protected:Npn \__prop_show:NN #1#2
11432 {
11433     \__kernel_chk_defined:NT #2
11434     {
11435         #1 { LaTeX/kernel } { show-prop }
11436         { \token_to_str:N #2 }

```

```

11437         { \prop_map_function:NN #2 \msg_show_item:nn }
11438         { } { }
11439     }
11440 }

```

(End definition for `\prop_show:N` and `\prop_log:N`. These functions are documented on page 150.)

```

11441 \</package>

```

19 l3msg implementation

```

11442 \*package>

```

```

11443 \@@=msg

```

`\l__msg_internal_tl` A general scratch for the module.

```

11444 \tl_new:N \l__msg_internal_tl

```

(End definition for `\l__msg_internal_tl`.)

`\l__msg_name_str` Used to save module info when creating messages.

```

\l__msg_text_str 11445 \str_new:N \l__msg_name_str

```

```

11446 \str_new:N \l__msg_text_str

```

(End definition for `\l__msg_name_str` and `\l__msg_text_str`.)

19.1 Internal auxiliaries

`\s__msg_mark` Internal scan marks.

```

\s__msg_stop 11447 \scan_new:N \s__msg_mark

```

```

11448 \scan_new:N \s__msg_stop

```

(End definition for `\s__msg_mark` and `\s__msg_stop`.)

`__msg_use_none_delimit_by_s_stop:w` Functions to gobble up to a scan mark.

```

11449 \cs_new:Npn \__msg_use_none_delimit_by_s_stop:w #1 \s__msg_stop { }

```

(End definition for `__msg_use_none_delimit_by_s_stop:w`.)

19.2 Creating messages

Messages are created and used separately, so there two parts to the code here. First, a mechanism for creating message text. This is pretty simple, as there is not actually a lot to do.

`\c__msg_text_prefix_tl` Locations for the text of messages.

```

\c__msg_more_text_prefix_tl 11450 \tl_const:Nn \c__msg_text_prefix_tl { msg-text~>~ }

```

```

11451 \tl_const:Nn \c__msg_more_text_prefix_tl { msg-extra~text~>~ }

```

(End definition for `\c__msg_text_prefix_tl` and `\c__msg_more_text_prefix_tl`.)

\msg_if_exist_p:nn Test whether the control sequence containing the message text exists or not.

\msg_if_exist:nnTF

```
11452 \prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
11453 {
11454     \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
11455     { \prg_return_true: } { \prg_return_false: }
11456 }
```

(End definition for \msg_if_exist:nnTF. This function is documented on page 153.)

__msg_chk_if_free:nn This auxiliary is similar to __kernel_chk_if_free_cs:N, and is used when defining messages with \msg_new:nnnn.

```
11457 \cs_new_protected:Npn \__msg_chk_free:nn #1#2
11458 {
11459     \msg_if_exist:nnT {#1} {#2}
11460     {
11461         \__kernel_msg_error:nxxx { kernel } { message-already-defined }
11462         {#1} {#2}
11463     }
11464 }
```

(End definition for __msg_chk_if_free:nn.)

\msg_new:nnnn Setting a message simply means saving the appropriate text into two functions. A sanity check first.

\msg_new:nnn

\msg_gset:nnnn

\msg_gset:nnn

\msg_set:nnnn

\msg_set:nnn

```
11465 \cs_new_protected:Npn \msg_new:nnnn #1#2
11466 {
11467     \__msg_chk_free:nn {#1} {#2}
11468     \msg_gset:nnnn {#1} {#2}
11469 }
11470 \cs_new_protected:Npn \msg_new:nnn #1#2#3
11471 { \msg_new:nnnn {#1} {#2} {#3} { } }
11472 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
11473 {
11474     \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
11475     ##1##2##3##4 {#3}
11476     \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
11477     ##1##2##3##4 {#4}
11478 }
11479 \cs_new_protected:Npn \msg_set:nnn #1#2#3
11480 { \msg_set:nnnn {#1} {#2} {#3} { } }
11481 \cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
11482 {
11483     \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
11484     ##1##2##3##4 {#3}
11485     \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
11486     ##1##2##3##4 {#4}
11487 }
11488 \cs_new_protected:Npn \msg_gset:nnn #1#2#3
11489 { \msg_gset:nnnn {#1} {#2} {#3} { } }
```

(End definition for \msg_new:nnnn and others. These functions are documented on page 152.)

19.3 Messages: support functions and text

Simple pieces of text for messages.

```

\c__msg_coding_error_text_tl
\c__msg_continue_text_tl
\c__msg_critical_text_tl
\c__msg_fatal_text_tl
\c__msg_help_text_tl
\c__msg_no_info_text_tl
\c__msg_on_line_text_tl
\c__msg_return_text_tl
\c__msg_trouble_text_tl
11490 \tl_const:Nn \c__msg_coding_error_text_tl
11491 {
11492   This~is~a~coding~error.
11493   \\ \\
11494 }
11495 \tl_const:Nn \c__msg_continue_text_tl
11496 { Type~<return>~to~continue }
11497 \tl_const:Nn \c__msg_critical_text_tl
11498 { Reading~the~current~file~'\g_file_curr_name_str'~will~stop. }
11499 \tl_const:Nn \c__msg_fatal_text_tl
11500 { This~is~a~fatal~error:~LaTeX~will~abort. }
11501 \tl_const:Nn \c__msg_help_text_tl
11502 { For~immediate~help~type~H~<return> }
11503 \tl_const:Nn \c__msg_no_info_text_tl
11504 {
11505   LaTeX~does~not~know~anything~more~about~this~error,~sorry.
11506   \c__msg_return_text_tl
11507 }
11508 \tl_const:Nn \c__msg_on_line_text_tl { on~line }
11509 \tl_const:Nn \c__msg_return_text_tl
11510 {
11511   \\ \\
11512   Try~typing~<return>~to~proceed.
11513   \\
11514   If~that~doesn't~work,~type~X~<return>~to~quit.
11515 }
11516 \tl_const:Nn \c__msg_trouble_text_tl
11517 {
11518   \\ \\
11519   More~errors~will~almost~certainly~follow: \\
11520   the~LaTeX~run~should~be~aborted.
11521 }

```

(End definition for `\c__msg_coding_error_text_tl` and others.)

`\msg_line_number:` For writing the line number nicely. `\msg_line_context:` was set up earlier, so this is not new.

```

11522 \cs_new:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
11523 \cs_gset:Npn \msg_line_context:
11524 {
11525   \c__msg_on_line_text_tl
11526   \c_space_tl
11527   \msg_line_number:
11528 }

```

(End definition for `\msg_line_number:` and `\msg_line_context:`. These functions are documented on page 153.)

19.4 Showing messages: low level mechanism

`__msg_interrupt:Nnnn` The low-level interruption macro is rather opaque, unfortunately. Depending on the availability of more information there is a choice of how to set up the further help. We

feed the extra help text and the message itself to a wrapping auxiliary, in this order because we must first setup TeX's `\errhelp` register before issuing an `\errmessage`. To deal with the various cases of critical or fatal errors with and without help text, there is a bit of argument-passing to do.

```

11529 \cs_new_protected:Npn \__msg_interrupt:NnnnN #1#2#3#4#5
11530 {
11531   \str_set:Nx \l__msg_text_str { #1 {#2} }
11532   \str_set:Nx \l__msg_name_str { \msg_module_name:n {#2} }
11533   \cs_if_eq:cNTF
11534     { \c__msg_more_text_prefix_tl #2 / #3 }
11535     \__msg_no_more_text:nnnn
11536     {
11537       \__msg_interrupt_wrap:nnn
11538       { \use:c { \c__msg_text_prefix_tl #2 / #3 } #4 }
11539       { \c__msg_continue_text_tl }
11540       {
11541         \c__msg_no_info_text_tl
11542         \tl_if_empty:NF #5
11543         { \\ \\ #5 }
11544       }
11545     }
11546     {
11547       \__msg_interrupt_wrap:nnn
11548       { \use:c { \c__msg_text_prefix_tl #2 / #3 } #4 }
11549       { \c__msg_help_text_tl }
11550       {
11551         \use:c { \c__msg_more_text_prefix_tl #2 / #3 } #4
11552         \tl_if_empty:NF #5
11553         { \\ \\ #5 }
11554       }
11555     }
11556   }
11557 \cs_new:Npn \__msg_no_more_text:nnnn #1#2#3#4 { }

```

(End definition for `__msg_interrupt:Nnnn` and `__msg_no_more_text:nnnn`.)

```

\__msg_interrupt_wrap:nnn
\__msg_interrupt_text:n
\__msg_interrupt_more_text:n

```

First setup TeX's `\errhelp` register with the extra help #1, then build a nice-looking error message with #2. Everything is done using x-type expansion as the new line markers are different for the two type of text and need to be correctly set up. The auxiliary `__msg_interrupt_more_text:n` receives its argument as a line-wrapped string, which is thus unaffected by expansion. We ave to split the main text into two parts as only the “message” itself is wrapped with a leader: the generic help is wrapped at full width. We also have to allow for the two characters used by `\errmessage` itself.

```

11558 \cs_new_protected:Npn \__msg_interrupt_wrap:nnn #1#2#3
11559 {
11560   \iow_wrap:nnnN { \\ #3 } { } { } \__msg_interrupt_more_text:n
11561   \group_begin:
11562     \int_sub:Nn \l_iow_line_count_int { 2 }
11563     \iow_wrap:nxnN { \l__msg_text_str : ~ #1 }
11564     {
11565       ( \l__msg_name_str )
11566       \prg_replicate:nn
11567       {

```

```

11568         \str_count:N \l__msg_text_str
11569         - \str_count:N \l__msg_name_str
11570         + 2
11571     }
11572     { ~ }
11573 }
11574 { } \__msg_interrupt_text:n
11575 \iow_wrap:nnn { \l__msg_internal_tl \\ \\ #2 } { } { }
11576 \__msg_interrupt:n
11577 }
11578 \cs_new_protected:Npn \__msg_interrupt_text:n #1
11579 {
11580     \group_end:
11581     \tl_set:Nn \l__msg_internal_tl {#1}
11582 }
11583 \cs_new_protected:Npn \__msg_interrupt_more_text:n #1
11584 { \exp_args:Nx \tex_errhelp:D { #1 \iow_newline: } }

```

(End definition for `__msg_interrupt_wrap:nnn`, `__msg_interrupt_text:n`, and `__msg_interrupt_more_text:n`.)

`__msg_interrupt:n`

The business end of the process starts by producing some visual separation of the message from the main part of the log. The error message needs to be printed with everything made “invisible”: T_EX’s own information involves the macro in which `\errmessage` is called, and the end of the argument of the `\errmessage`, including the closing brace. We use an active `!` to call the `\errmessage` primitive, and end its argument with `\use_none:n {{spaces}}` which fills the output with spaces. Two trailing closing braces are turned into spaces to hide them as well. The group in which we alter the definition of the active `!` is closed before producing the message: this ensures that tokens inserted by typing `I` in the command-line are inserted after the message is entirely cleaned up.

The `__kernel_iow_with:Nnn` auxiliary, defined in `l3file`, expects an *integer variable*, an integer *value*, and some *code*. It runs the *code* after ensuring that the *integer variable* takes the given *value*, then restores the former value of the *integer variable* if needed. We use it to ensure that the `\newlinechar` is 10, as needed for `\iow_newline:` to work, and that `\errorcontextlines` is `-1`, to avoid showing irrelevant context. Note that restoring the former value of these integers requires inserting tokens after the `\errmessage`, which go in the way of tokens which could be inserted by the user. This is unavoidable.

```

11585 \group_begin:
11586   \char_set_lccode:nn { 38 } { 32 } % &
11587   \char_set_lccode:nn { 46 } { 32 } % .
11588   \char_set_lccode:nn { 123 } { 32 } % {
11589   \char_set_lccode:nn { 125 } { 32 } % }
11590   \char_set_catcode_active:N \&
11591   \tex_lowercase:D
11592   {
11593     \group_end:
11594     \cs_new_protected:Npn \__msg_interrupt:n #1
11595     {
11596       \iow_term:n { }
11597       \__kernel_iow_with:Nnn \tex_newlinechar:D { ‘^^J }
11598       {
11599         \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }

```

```

11600         {
11601             \group_begin:
11602             \cs_set_protected:Npn &
11603             {
11604                 \tex_errmessage:D
11605                 {
11606                     #1
11607                     \use_none:n
11608                     { ..... }
11609                 }
11610             }
11611             \exp_after:wN
11612             \group_end:
11613             &
11614         }
11615     }
11616 }
11617 }

```

(End definition for `_msg_interrupt:n`.)

19.5 Displaying messages

L^AT_EX is handling error messages and so the T_EX ones are disabled.

```

11618 \int_gset:Nn \tex_errorcontextlines:D { -1 }

```

`\msg_fatal_text:n` A function for issuing messages: both the text and order could in principle vary. The module name may be empty for kernel messages, hence the slightly contorted code path for a space.

```

\msg_critical_text:n
\msg_error_text:n
\msg_warning_text:n
\msg_info_text:n
\_msg_text:nn
\_msg_text:n
11619 \cs_new:Npn \msg_fatal_text:n #1
11620 {
11621     Fatal ~
11622     \msg_error_text:n {#1}
11623 }
11624 \cs_new:Npn \msg_critical_text:n #1
11625 {
11626     Critical ~
11627     \msg_error_text:n {#1}
11628 }
11629 \cs_new:Npn \msg_error_text:n #1
11630 { \_msg_text:nn {#1} { Error } }
11631 \cs_new:Npn \msg_warning_text:n #1
11632 { \_msg_text:nn {#1} { Warning } }
11633 \cs_new:Npn \msg_info_text:n #1
11634 { \_msg_text:nn {#1} { Info } }
11635 \cs_new:Npn \_msg_text:nn #1#2
11636 {
11637     \exp_args:Nf \_msg_text:n { \msg_module_type:n {#1} }
11638     \msg_module_name:n {#1} ~
11639     #2
11640 }
11641 \cs_new:Npn \_msg_text:n #1
11642 {
11643     \tl_if_blank:nF {#1}

```

```

11644     { #1 ~ }
11645 }

```

(End definition for `\msg_fatal_text:n` and others. These functions are documented on page 153.)

`\g_msg_module_name_prop` For storing public module information: the kernel data is set up in advance.

```

\g_msg_module_type_prop 11646 \prop_new:N \g_msg_module_name_prop
11647 \prop_gput:Nnn \g_msg_module_name_prop { LaTeX } { LaTeX3 }
11648 \prop_new:N \g_msg_module_type_prop
11649 \prop_gput:Nnn \g_msg_module_type_prop { LaTeX } { }

```

(End definition for `\g_msg_module_name_prop` and `\g_msg_module_type_prop`. These variables are documented on page 154.)

`\msg_module_type:n` Contextual footer information, with the potential to give modules an alternative name.

```

11650 \cs_new:Npn \msg_module_type:n #1
11651 {
11652   \prop_if_in:NnTF \g_msg_module_type_prop {#1}
11653   { \prop_item:Nn \g_msg_module_type_prop {#1} }
11654   { Package }
11655 }

```

(End definition for `\msg_module_type:n`. This function is documented on page 154.)

`\msg_module_name:n` Contextual footer information, with the potential to give modules an alternative name.

```

\msg_see_documentation_text:n 11656 \cs_new:Npn \msg_module_name:n #1
11657 {
11658   \prop_if_in:NnTF \g_msg_module_name_prop {#1}
11659   { \prop_item:Nn \g_msg_module_name_prop {#1} }
11660   {#1}
11661 }
11662 \cs_new:Npn \msg_see_documentation_text:n #1
11663 {
11664   See-the~ \msg_module_name:n {#1} ~
11665   documentation-for-further-information.
11666 }

```

(End definition for `\msg_module_name:n` and `\msg_see_documentation_text:n`. These functions are documented on page 154.)

`__msg_class_new:nn`

```

11667 \group_begin:
11668 \cs_set_protected:Npn \__msg_class_new:nn #1#2
11669 {
11670   \prop_new:c { l__msg_redirect_ #1 _prop }
11671   \cs_new_protected:cpn { __msg_ #1 _code:nnnnnn }
11672   ##1##2##3##4##5##6 {#2}
11673   \cs_new_protected:cpn { msg_ #1 :nnnnnn } ##1##2##3##4##5##6
11674   {
11675     \use:x
11676     {
11677       \exp_not:n { \__msg_use:nnnnnn {#1} {##1} {##2} }
11678       { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
11679       { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
11680     }

```

```

11681     }
11682     \cs_new_protected:cpx { msg_ #1 :nnnnn } ##1##2##3##4##5
11683     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
11684     \cs_new_protected:cpx { msg_ #1 :nnnn } ##1##2##3##4
11685     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
11686     \cs_new_protected:cpx { msg_ #1 :nnn } ##1##2##3
11687     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
11688     \cs_new_protected:cpx { msg_ #1 :nn } ##1##2
11689     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
11690     \cs_new_protected:cpx { msg_ #1 :nnxxxx } ##1##2##3##4##5##6
11691     {
11692         \use:x
11693         {
11694             \exp_not:N \exp_not:n
11695             { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} }
11696             {##3} {##4} {##5} {##6}
11697         }
11698     }
11699     \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
11700     { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
11701     \cs_new_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
11702     { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
11703     \cs_new_protected:cpx { msg_ #1 :nnx } ##1##2##3
11704     { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
11705 }

```

(End definition for `_msg_class_new:nn`.)

`\msg_fatal:nnnnnn` For fatal errors, after the error message \TeX bails out. We force a bail out rather than using `\end` as this means it does not matter if we are in a context where normally the run cannot end.

```

\msg_fatal:nnxxx 11706 \_msg\_class\_new:nn { fatal }
\msg_fatal:nnnn 11707 {
\msg_fatal:nnxx 11708     \_msg\_interrupt:NnnnN
\msg_fatal:nnn 11709     \msg_fatal_text:n {#1} {#2}
\msg_fatal:nnx 11710     { {#3} {#4} {#5} {#6} }
\msg_fatal:nn 11711     \c\_msg_fatal_text_tl
\_msg_fatal_exit: 11712     \_msg_fatal_exit:
11713 }
11714 \cs_new_protected:Npn \_msg_fatal_exit:
11715 {
11716     \tex_batchmode:D
11717     \tex_read:D -1 to \l\_msg\_internal_tl
11718 }

```

(End definition for `\msg_fatal:nnnnnn` and others. These functions are documented on page 155.)

`\msg_critical:nnnnnn` Not quite so bad: just end the current file.

```

\msg_critical:nnxxxx 11719 \_msg\_class\_new:nn { critical }
\msg_critical:nnnnn 11720 {
\msg_critical:nnxxx 11721     \_msg\_interrupt:NnnnN
\msg_critical:nnnn 11722     \msg_critical_text:n {#1} {#2}
\msg_critical:nnxx 11723     { {#3} {#4} {#5} {#6} }
\msg_critical:nnn 11724     \c\_msg\_critical\_text\_tl
\msg_critical:nnx
\msg_critical:nn

```

```

11725     \tex_endinput:D
11726   }

```

(End definition for `\msg_critical:nnnnnn` and others. These functions are documented on page 155.)

`\msg_error:nnnnnn` For an error, the interrupt routine is called. We check if there is a “more text” by
`\msg_error:nnxxxx` comparing that control sequence with a permanently empty text.

```

\msg_error:nnnnn 11727   \_msg_class_new:nn { error }
\msg_error:nnxxx 11728   {
\msg_error:nnnn 11729     \_msg_interrupt:NnnnN
\msg_error:nnxx 11730     \msg_error_text:n {#1} {#2}
\msg_error:nnn 11731     { {#3} {#4} {#5} {#6} }
\msg_error:nnx 11732     \c_empty_tl
\msg_error:nn 11733   }

```

(End definition for `\msg_error:nnnnnn` and others. These functions are documented on page 155.)

`\msg_warning:nnnnnn` Warnings are printed to the terminal.

```

\msg_warning:nnxxxx 11734   \_msg_class_new:nn { warning }
\msg_warning:nnnnn 11735   {
\msg_warning:nnxxx 11736     \str_set:Nx \l__msg_text_str { \msg_warning_text:n {#1} }
\msg_warning:nnnn 11737     \str_set:Nx \l__msg_name_str { \msg_module_name:n {#1} }
\msg_warning:nnxx 11738     \iow_term:n { }
\msg_warning:nnn 11739     \iow_wrap:nxnN
\msg_warning:nnx 11740     {
11741       \l__msg_text_str : ~
11742       \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
11743     }
11744     {
11745       ( \l__msg_name_str )
11746       \prg_replicate:nn
11747       {
11748         \str_count:N \l__msg_text_str
11749         - \str_count:N \l__msg_name_str
11750       }
11751       { ~ }
11752     }
11753     { } \iow_term:n
11754     \iow_term:n { }
11755   }

```

(End definition for `\msg_warning:nnnnnn` and others. These functions are documented on page 155.)

`\msg_info:nnnnnn` Information only goes into the log.

```

\msg_info:nnxxxx 11756   \_msg_class_new:nn { info }
\msg_info:nnnnn 11757   {
\msg_info:nnxxx 11758     \str_set:Nx \l__msg_text_str { \msg_info_text:n {#1} }
\msg_info:nnnn 11759     \str_set:Nx \l__msg_name_str { \msg_module_name:n {#1} }
\msg_info:nnxx 11760     \iow_log:n { }
\msg_info:nnn 11761     \iow_wrap:nxnN
\msg_info:nnx 11762     {
11763       \l__msg_text_str : ~
11764       \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
11765     }
11766     {

```



```

11767         ( \l__msg_name_str )
11768         \prg_replicate:nn
11769         {
11770             \str_count:N \l__msg_text_str
11771             - \str_count:N \l__msg_name_str
11772         }
11773         { ~ }
11774     }
11775     { } \iow_log:n
11776     \iow_log:n { }
11777 }

```

(End definition for \msg_info:nnnnnn and others. These functions are documented on page 156.)

\msg_log:nnnnnn “Log” data is very similar to information, but with no extras added.

```

\msg_log:nnxxxx 11778 \__msg_class_new:nn { log }
\msg_log:nnnnnn 11779 {
\msg_log:nnxxx 11780     \iow_wrap:nnnN
\msg_log:nnnn 11781     { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
\msg_log:nnxx 11782     { } { } \iow_log:n
\msg_log:nnn 11783 }
\msg_log:nnx
\msg_log:nn

```

(End definition for \msg_log:nnnnnn and others. These functions are documented on page 156.)

\msg_term:nnnnnn “Term” is used for communicating with the user through the terminal, like diagnostic messages, and debugging. This is similar to “log” messages, but uses the terminal output.

```

\msg_term:nnxxxx 11784 \__msg_class_new:nn { term }
\msg_term:nnxxx 11785 {
\msg_term:nnnn 11786     \iow_wrap:nnnN
\msg_term:nnxx 11787     { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
\msg_term:nnn 11788     { } { } \iow_term:n
\msg_term:nnx 11789 }
\msg_term:nn

```

(End definition for \msg_term:nnnnnn and others. These functions are documented on page 156.)

\msg_none:nnnnnn The none message type is needed so that input can be gobbled.

```

\msg_none:nnxxxx 11790 \__msg_class_new:nn { none } { }
\msg_none:nnnnnn
\msg_none:nnxxx

```

(End definition for \msg_none:nnnnnn and others. These functions are documented on page 156.)

\msg_show:nnnnnn The show message type is used for \seq_show:N and similar complicated data structures. Wrap the given text with a trailing dot (important later) then pass it to __msg_show:n. If there is \\>~ (or if the whole thing starts with >~) we split there, print the first part and show the second part using \showtokens (the \exp_after:wN ensure a nice display). Note that this primitive adds a leading >~ and trailing dot. That is why we included a trailing dot before wrapping and removed it afterwards. If there is no \\>~ do the same but with an empty second part which adds a spurious but inevitable >~.

```

\msg_show:nnxx 11791 \__msg_class_new:nn { show }
\msg_show:nnn 11792 {
\__msg_show:n 11793     \iow_wrap:nnnN
\__msg_show:w 11794     { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
\__msg_show_dot:w 11795     { } { } \__msg_show:n
\__msg_show:nn 11796 }
11797 \cs_new_protected:Npn \__msg_show:n #1

```

```

11798 {
11799   \tl_if_in:nnTF { ^^J #1 } { ^^J > ~ }
11800   {
11801     \tl_if_in:nnTF { #1 \s__msg_mark } { . \s__msg_mark }
11802     { \__msg_show_dot:w } { \__msg_show:w }
11803     ^^J #1 \s__msg_stop
11804   }
11805   { \__msg_show:nn { ? #1 } { } }
11806 }
11807 \cs_new:Npn \__msg_show_dot:w #1 ^^J > ~ #2 . \s__msg_stop
11808 { \__msg_show:nn {#1} {#2} }
11809 \cs_new:Npn \__msg_show:w #1 ^^J > ~ #2 \s__msg_stop
11810 { \__msg_show:nn {#1} {#2} }
11811 \cs_new_protected:Npn \__msg_show:nn #1#2
11812 {
11813   \tl_if_empty:nF {#1}
11814   { \exp_args:No \iow_term:n { \use_none:n #1 } }
11815   \tl_set:Nn \l__msg_internal_tl {#2}
11816   \__kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
11817   {
11818     \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
11819     {
11820       \tex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
11821       { \exp_after:wN \l__msg_internal_tl }
11822     }
11823   }
11824 }

```

(End definition for `\msg_show:nnnnnn` and others. These functions are documented on page 268.)

End the group to eliminate `__msg_class_new:nn`.

```

11825 \group_end:

```

`__msg_class_chk_exist:nT` Checking that a message class exists. We build this from `\cs_if_free:cTF` rather than `\cs_if_exist:cTF` because that avoids reading the second argument earlier than necessary.

```

11826 \cs_new:Npn \__msg_class_chk_exist:nT #1
11827 {
11828   \cs_if_free:cTF { __msg_ #1 _code:nnnnnn }
11829   { \__kernel_msg_error:nx { kernel } { message-class-unknown } {#1} }
11830 }

```

(End definition for `__msg_class_chk_exist:nT`.)

`\l__msg_class_tl` Support variables needed for the redirection system.
`\l__msg_current_class_tl`

```

11831 \tl_new:N \l__msg_class_tl
11832 \tl_new:N \l__msg_current_class_tl

```

(End definition for `\l__msg_class_tl` and `\l__msg_current_class_tl`.)

`\l__msg_redirect_prop` For redirection of individually-named messages

```

11833 \prop_new:N \l__msg_redirect_prop

```

(End definition for `\l__msg_redirect_prop`.)

\l__msg_hierarchy_seq During redirection, split the message name into a sequence: {/module/submodule}, {/module}, and {}.

```
11834 \seq_new:N \l__msg_hierarchy_seq
```

(End definition for \l__msg_hierarchy_seq.)

\l__msg_class_loop_seq Classes encountered when following redirections to check for loops.

```
11835 \seq_new:N \l__msg_class_loop_seq
```

(End definition for \l__msg_class_loop_seq.)

__msg_use:nnnnnnn

__msg_use_redirect_name:n

__msg_use_hierarchy:nwwN

__msg_use_redirect_module:n

__msg_use_code:

Actually using a message is a multi-step process. First, some safety checks on the message and class requested. The code and arguments are then stored to avoid passing them around. The assignment to __msg_use_code: is similar to \tl_set:Nn. The message is eventually produced with whatever \l__msg_class_tl is when __msg_use_code: is called. Here is also a good place to suppress tracing output if the trace package is loaded since all (non-expandable) messages go through this auxiliary.

```
11836 \cs_new_protected:Npn \__msg_use:nnnnnnn #1#2#3#4#5#6#7
11837 {
11838   \cs_if_exist_use:N \conditionally@traceoff
11839   \msg_if_exist:nnTF {#2} {#3}
11840   {
11841     \__msg_class_chk_exist:nT {#1}
11842     {
11843       \tl_set:Nn \l__msg_current_class_tl {#1}
11844       \cs_set_protected:Npx \__msg_use_code:
11845       {
11846         \exp_not:n
11847         {
11848           \use:c { __msg_ \l__msg_class_tl _code:nnnnnnn }
11849           {#2} {#3} {#4} {#5} {#6} {#7}
11850         }
11851       }
11852       \__msg_use_redirect_name:n { #2 / #3 }
11853     }
11854   }
11855   { \__kernel_msg_error:nxxx { kernel } { message-unknown } {#2} {#3} }
11856   \cs_if_exist_use:N \conditionally@traceon
11857 }
11858 \cs_new_protected:Npn \__msg_use_code: { }
```

The first check is for a individual message redirection. If this applies then no further redirection is attempted. Otherwise, split the message name into $\langle module \rangle$, $\langle submodule \rangle$ and $\langle message \rangle$ (with an arbitrary number of slashes), and store {/module/submodule}, {/module} and {} into \l__msg_hierarchy_seq. We then map through this sequence, applying the most specific redirection.

```
11859 \cs_new_protected:Npn \__msg_use_redirect_name:n #1
11860 {
11861   \prop_get:NnNTF \l__msg_redirect_prop { / #1 } \l__msg_class_tl
11862   { \__msg_use_code: }
11863   {
11864     \seq_clear:N \l__msg_hierarchy_seq
11865     \__msg_use_hierarchy:nwwN { }
11866     #1 \s__msg_mark \__msg_use_hierarchy:nwwN
```

```

11867         / \s__msg_mark \__msg_use_none_delimit_by_s_stop:w
11868         \s__msg_stop
11869         \__msg_use_redirect_module:n { }
11870     }
11871 }
11872 \cs_new_protected:Npn \__msg_use_hierarchy:nwwN #1#2 / #3 \s__msg_mark #4
11873 {
11874     \seq_put_left:Nn \l__msg_hierarchy_seq {#1}
11875     #4 { #1 / #2 } #3 \s__msg_mark #4
11876 }

```

At this point, the items of `\l__msg_hierarchy_seq` are the various levels at which we should look for a redirection. Redirections which are less specific than the argument of `__msg_use_redirect_module:n` are not attempted. This argument is empty for a class redirection, `/module` for a module redirection, *etc.* Loop through the sequence to find the most specific redirection, with module `##1`. The loop is interrupted after testing for a redirection for `##1` equal to the argument `#1` (least specific redirection allowed). When a redirection is found, break the mapping, then if the redirection targets the same class, output the code with that class, and otherwise set the target as the new current class, and search for further redirections. Those redirections should be at least as specific as `##1`.

```

11877 \cs_new_protected:Npn \__msg_use_redirect_module:n #1
11878 {
11879     \seq_map_inline:Nn \l__msg_hierarchy_seq
11880     {
11881         \prop_get:cnNTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
11882         {##1} \l__msg_class_tl
11883         {
11884             \seq_map_break:n
11885             {
11886                 \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
11887                 { \__msg_use_code: }
11888                 {
11889                     \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
11890                     \__msg_use_redirect_module:n {##1}
11891                 }
11892             }
11893         }
11894         {
11895             \str_if_eq:nnT {##1} {#1}
11896             {
11897                 \tl_set_eq:NN \l__msg_class_tl \l__msg_current_class_tl
11898                 \seq_map_break:n { \__msg_use_code: }
11899             }
11900         }
11901     }
11902 }

```

(End definition for `__msg_use:nnnnnnn` and others.)

`\msg_redirect_name:nnn` Named message always use the given class even if that class is redirected further. An empty target class cancels any existing redirection for that message.

```

11903 \cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
11904 {

```

```

11905     \tl_if_empty:nTF {#3}
11906     { \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 } }
11907     {
11908         \__msg_class_chk_exist:nT {#3}
11909         { \prop_put:Nnn \l__msg_redirect_prop { / #1 / #2 } {#3} }
11910     }
11911 }

```

(End definition for `\msg_redirect_name:nnn`. This function is documented on page 158.)

```

\msg_redirect_class:nn
\msg_redirect_module:nnn
  \__msg_redirect:nnn
\__msg_redirect_loop_chk:nnn
\__msg_redirect_loop_list:n

```

If the target class is empty, eliminate the corresponding redirection. Otherwise, add the redirection. We must then check for a loop: as an initialization, we start by storing the initial class in `\l__msg_current_class_tl`.

```

11912 \cs_new_protected:Npn \msg_redirect_class:nn
11913   { \__msg_redirect:nnn { } }
11914 \cs_new_protected:Npn \msg_redirect_module:nnn #1
11915   { \__msg_redirect:nnn { / #1 } }
11916 \cs_new_protected:Npn \__msg_redirect:nnn #1#2#3
11917   {
11918     \__msg_class_chk_exist:nT {#2}
11919     {
11920       \tl_if_empty:nTF {#3}
11921       { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
11922       {
11923         \__msg_class_chk_exist:nT {#3}
11924         {
11925           \prop_put:cn { l__msg_redirect_ #2 _prop } {#1} {#3}
11926           \tl_set:Nn \l__msg_current_class_tl {#2}
11927           \seq_clear:N \l__msg_class_loop_seq
11928           \__msg_redirect_loop_chk:nnn {#2} {#3} {#1}
11929         }
11930       }
11931     }
11932   }

```

Since multiple redirections can only happen with increasing specificity, a loop requires that all steps are of the same specificity. The new redirection can thus only create a loop with other redirections for the exact same module, #1, and not submodules. After some initialization above, follow redirections with `\l__msg_class_tl`, and keep track in `\l__msg_class_loop_seq` of the various classes encountered. A redirection from a class to itself, or the absence of redirection both mean that there is no loop. A redirection to the initial class marks a loop. To break it, we must decide which redirection to cancel. The user most likely wants the newly added redirection to hold with no further redirection. We thus remove the redirection starting from #2, target of the new redirection. Note that no message is emitted by any of the underlying functions: otherwise we may get an infinite loop because of a message from the message system itself.

```

11933 \cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
11934   {
11935     \seq_put_right:Nn \l__msg_class_loop_seq {#1}
11936     \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_class_tl
11937     {
11938       \str_if_eq:VnF \l__msg_class_tl {#1}
11939       {
11940         \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl

```

```

11941         {
11942         \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
11943         \__kernel_msg_warning:nnxxxx
11944         { kernel } { message-redirect-loop }
11945         { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
11946         { \seq_item:Nn \l__msg_class_loop_seq { 2 } }
11947         {#3}
11948         {
11949         \seq_map_function:NN \l__msg_class_loop_seq
11950         \__msg_redirect_loop_list:n
11951         { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
11952         }
11953         }
11954         { \__msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} }
11955     }
11956 }
11957 }
11958 \cs_generate_variant:Nn \__msg_redirect_loop_chk:nnn { o }
11959 \cs_new:Npn \__msg_redirect_loop_list:n #1 { {#1} ~ => ~ }

```

(End definition for `\msg_redirect_class:nn` and others. These functions are documented on page 158.)

19.6 Kernel-specific functions

`__kernel_msg_new:nnnn` The kernel needs some messages of its own. These are created using pre-built functions. Two functions are provided: one more general and one which only has the short text part.

```

\__kernel_msg_set:nnnn
\__kernel_msg_set:nnn
11960 \cs_new_protected:Npn \__kernel_msg_new:nnnn #1#2
11961 { \msg_new:nnnn { LaTeX } { #1 / #2 } }
11962 \cs_new_protected:Npn \__kernel_msg_new:nnn #1#2
11963 { \msg_new:nnn { LaTeX } { #1 / #2 } }
11964 \cs_new_protected:Npn \__kernel_msg_set:nnnn #1#2
11965 { \msg_set:nnnn { LaTeX } { #1 / #2 } }
11966 \cs_new_protected:Npn \__kernel_msg_set:nnn #1#2
11967 { \msg_set:nnn { LaTeX } { #1 / #2 } }

```

(End definition for `__kernel_msg_new:nnnn` and others.)

`__msg_kernel_class_new:nN` All the functions for kernel messages come in variants ranging from 0 to 4 arguments. Those with less than 4 arguments are defined in terms of the 4-argument variant, in a way very similar to `__msg_class_new:nn`. This auxiliary is destroyed at the end of the group.

```

11968 \group_begin:
11969 \cs_set_protected:Npn \__msg_kernel_class_new:nN #1
11970 { \__msg_kernel_class_new_aux:nN { __kernel_msg_ #1 } }
11971 \cs_set_protected:Npn \__msg_kernel_class_new_aux:nN #1#2
11972 {
11973     \cs_new_protected:cpn { #1 :nnnnnn } ##1##2##3##4##5##6
11974     {
11975         \use:x
11976         {
11977             \exp_not:n { #2 { LaTeX } { ##1 / ##2 } }
11978             { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
11979             { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }

```

```

11980     }
11981   }
11982   \cs_new_protected:cpx { #1 :nnnnn } ##1##2##3##4##5
11983   { \exp_not:c { #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
11984   \cs_new_protected:cpx { #1 :nnnn } ##1##2##3##4
11985   { \exp_not:c { #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
11986   \cs_new_protected:cpx { #1 :nnn } ##1##2##3
11987   { \exp_not:c { #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
11988   \cs_new_protected:cpx { #1 :nn } ##1##2
11989   { \exp_not:c { #1 :nnnnnn } {##1} {##2} { } { } { } { } }
11990   \cs_new_protected:cpx { #1 :nnxxxx } ##1##2##3##4##5##6
11991   {
11992     \use:x
11993     {
11994       \exp_not:N \exp_not:n
11995       { \exp_not:c { #1 :nnnnnn } {##1} {##2} }
11996       {##3} {##4} {##5} {##6}
11997     }
11998   }
11999   \cs_new_protected:cpx { #1 :nnxxx } ##1##2##3##4##5
12000   { \exp_not:c { #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
12001   \cs_new_protected:cpx { #1 :nnxx } ##1##2##3##4
12002   { \exp_not:c { #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
12003   \cs_new_protected:cpx { #1 :nnx } ##1##2##3
12004   { \exp_not:c { #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
12005 }

```

(End definition for `_msg_kernel_class_new:nN` and `_msg_kernel_class_new_aux:nN`.)

`_kernel_msg_fatal:nnnnnn` Neither fatal kernel errors nor kernel errors can be redirected. We directly use the code for (non-kernel) fatal errors and errors, adding the “`LATEX`” module name. Three functions are already defined by `l3basics`; we need to undefine them to avoid errors.

```

12006   \_msg_kernel_class_new:nN { fatal } \_msg_fatal_code:nnnnnn
12007   \_msg_kernel_class_new:nN { critical } \_msg_critical_code:nnnnnn
12008   \cs_undefine:N \_kernel_msg_error:nnxx
12009   \cs_undefine:N \_kernel_msg_error:nnx
12010   \cs_undefine:N \_kernel_msg_error:nn
12011   \_msg_kernel_class_new:nN { error } \_msg_error_code:nnnnnn

```

(End definition for `_kernel_msg_fatal:nnnnnn` and others.)

`_kernel_msg_warninglannnnnn` Kernel messages which can be redirected simply use the machinery for normal messages, with the module name “`LATEX`”.

```

12012   \_msg_kernel_class_new:nN { warning } \msg_warning:nnxxxx
12013   \_msg_kernel_class_new:nN { info } \msg_info:nnxxxx

```

(End definition for `_kernel_msg_warning:nnnnnn` and others.)

End the group to eliminate `_msg_kernel_class_new:nN`.

```

12014 \group_end:

```

Error messages needed to actually implement the message system itself.

```

12015 \_kernel_msg_new:nnnn { kernel } { message-already-defined }
12016 { Message~'##2'~for~module~'##1'~already-defined. }
12017 {
12018   \c_msg_coding_error_text_tl

```

```

12019 LaTeX~was~asked~to~define~a~new~message~called~'#2'\
12020 by~the~module~'#1':~this~message~already~exists.
12021 \c__msg_return_text_tl
12022 }
12023 \__kernel_msg_new:nnnn { kernel } { message-unknown }
12024 { Unknown~message~'#2'~for~module~'#1'. }
12025 {
12026 \c__msg_coding_error_text_tl
12027 LaTeX~was~asked~to~display~a~message~called~'#2'\
12028 by~the~module~'#1':~this~message~does~not~exist.
12029 \c__msg_return_text_tl
12030 }
12031 \__kernel_msg_new:nnnn { kernel } { message-class-unknown }
12032 { Unknown~message~class~'#1'. }
12033 {
12034 LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\
12035 this~was~never~defined.
12036 \c__msg_return_text_tl
12037 }
12038 \__kernel_msg_new:nnnn { kernel } { message-redirect-loop }
12039 {
12040 Message~redirection~loop~caused~by~ {#1} ~=>~ {#2}
12041 \tl_if_empty:nF {#3} { ~for~module~' \use_none:n #3 ' } .
12042 }
12043 {
12044 Adding~the~message~redirection~ {#1} ~=>~ {#2}
12045 \tl_if_empty:nF {#3} { ~for~the~module~' \use_none:n #3 ' } ~
12046 created~an~infinite~loop\\
12047 \iow_indent:n { #4 \\ }
12048 }

```

Messages for earlier kernel modules plus a few for l3keys which cover coding errors.

```

12049 \__kernel_msg_new:nnnn { kernel } { bad-number-of-arguments }
12050 { Function~'#1'~cannot~be~defined~with~#2~arguments. }
12051 {
12052 \c__msg_coding_error_text_tl
12053 LaTeX~has~been~asked~to~define~a~function~'#1'~with~
12054 #2~arguments.~
12055 TeX~allows~between~0~and~9~arguments~for~a~single~function.
12056 }
12057 \__kernel_msg_new:nnn { kernel } { char-active }
12058 { Cannot~generate~active~chars. }
12059 \__kernel_msg_new:nnn { kernel } { char-invalid-catcode }
12060 { Invalid~catcode~for~char~generation. }
12061 \__kernel_msg_new:nnn { kernel } { char-null-space }
12062 { Cannot~generate~null~char~as~a~space. }
12063 \__kernel_msg_new:nnn { kernel } { char-out-of-range }
12064 { Charcode~requested~out~of~engine~range. }
12065 \__kernel_msg_new:nnn { kernel } { char-space }
12066 { Cannot~generate~space~chars. }
12067 \__kernel_msg_new:nnnn { kernel } { command-already-defined }
12068 { Control~sequence~#1~already~defined. }
12069 {
12070 \c__msg_coding_error_text_tl
12071 LaTeX~has~been~asked~to~create~a~new~control~sequence~'#1'~

```



```

12072     but~this~name~has~already~been~used~elsewhere. \\ \\
12073     The~current~meaning~is:\\
12074     \\ #2
12075   }
12076   \_kernel_msg_new:nnnn { kernel } { command-not-defined }
12077   { Control~sequence~#1~undefined. }
12078   {
12079     \c__msg_coding_error_text_tl
12080     LaTeX~has~been~asked~to~use~a~control~sequence~'#1':\\
12081     this~has~not~been~defined~yet.
12082   }
12083   \_kernel_msg_new:nnnn { kernel } { empty-search-pattern }
12084   { Empty~search~pattern. }
12085   {
12086     \c__msg_coding_error_text_tl
12087     LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'#1':~that~
12088     would~lead~to~an~infinite~loop!
12089   }
12090   \_kernel_msg_new:nnnn { kernel } { out-of-registers }
12091   { No~room~for~a~new~#1. }
12092   {
12093     TeX~only~supports~\int_use:N \c_max_register_int \ %
12094     of~each~type.~All~the~#1~registers~have~been~used.~
12095     This~run~will~be~aborted~now.
12096   }
12097   \_kernel_msg_new:nnnn { kernel } { non-base-function }
12098   { Function~'#1'~is~not~a~base~function }
12099   {
12100     \c__msg_coding_error_text_tl
12101     Functions~defined~through~\iow_char:N\\cs_new:Nn~must~have~
12102     a~signature~consisting~of~only~normal~arguments~'N'~and~'n'.~
12103     To~define~variants~use~\iow_char:N\\cs_generate_variant:Nn~
12104     and~to~define~other~functions~use~\iow_char:N\\cs_new:Npn.
12105   }
12106   \_kernel_msg_new:nnnn { kernel } { missing-colon }
12107   { Function~'#1'~contains~no~':'. }
12108   {
12109     \c__msg_coding_error_text_tl
12110     Code~level~functions~must~contain~':'~to~separate~the~
12111     argument~specification~from~the~function~name.~This~is~
12112     needed~when~defining~conditionals~or~variants,~or~when~building~a~
12113     parameter~text~from~the~number~of~arguments~of~the~function.
12114   }
12115   \_kernel_msg_new:nnnn { kernel } { overflow }
12116   { Integers~larger~than~2^{30}-1~cannot~be~stored~in~arrays. }
12117   {
12118     An~attempt~was~made~to~store~#3~
12119     \tl_if_empty:nF {#2} { at~position~#2~ } in~the~array~'#1'.~
12120     The~largest~allowed~value~#4~will~be~used~instead.
12121   }
12122   \_kernel_msg_new:nnnn { kernel } { out-of-bounds }
12123   { Access~to~an~entry~beyond~an~array's~bounds. }
12124   {
12125     An~attempt~was~made~to~access~or~store~data~at~position~#2~of~the~

```

```

12126     array~'1',~but~this~array~has~entries~at~positions~from~1~to~#3.
12127 }
12128 \__kernel_msg_new:nnnn { kernel } { protected-predicate }
12129 { Predicate~'1'~must~be~expandable. }
12130 {
12131     \c__msg_coding_error_text_tl
12132     LaTeX~has~been~asked~to~define~'1'~as~a~protected~predicate.~
12133     Only~expandable~tests~can~have~a~predicate~version.
12134 }
12135 \__kernel_msg_new:nnn { kernel } { randint-backward-range }
12136 { Bounds~ordered~backwards~in~\iow_char:N\int_rand:nn~{#1}~{#2}. }
12137 \__kernel_msg_new:nnnn { kernel } { conditional-form-unknown }
12138 { Conditional~form~'1'~for~function~'2'~unknown. }
12139 {
12140     \c__msg_coding_error_text_tl
12141     LaTeX~has~been~asked~to~define~the~conditional~form~'1'~of~
12142     the~function~'2',~but~only~'TF',~'T',~'F',~and~'p'~forms~exist.
12143 }
12144 \__kernel_msg_new:nnnn { kernel } { key-no-property }
12145 { No~property~given~in~definition~of~key~'1'. }
12146 {
12147     \c__msg_coding_error_text_tl
12148     Inside~\keys_define:nn  each~key~name~
12149     needs~a~property:  \ \ \
12150     \iow_indent:n { #1 .<property> } \ \ \
12151     LaTeX~did~not~find~a~'.'~to~indicate~the~start~of~a~property.
12152 }
12153 \__kernel_msg_new:nnnn { kernel } { key-property-boolean-values-only }
12154 { The~property~'1'~accepts~boolean~values~only. }
12155 {
12156     \c__msg_coding_error_text_tl
12157     The~property~'1'~only~accepts~the~values~'true'~and~'false'.
12158 }
12159 \__kernel_msg_new:nnnn { kernel } { key-property-requires-value }
12160 { The~property~'1'~requires~a~value. }
12161 {
12162     \c__msg_coding_error_text_tl
12163     LaTeX~was~asked~to~set~property~'1'~for~key~'2'.\ \
12164     No~value~was~given~for~the~property,~and~one~is~required.
12165 }
12166 \__kernel_msg_new:nnnn { kernel } { key-property-unknown }
12167 { The~key~property~'1'~is~unknown. }
12168 {
12169     \c__msg_coding_error_text_tl
12170     LaTeX~has~been~asked~to~set~the~property~'1'~for~key~'2':~
12171     this~property~is~not~defined.
12172 }
12173 \__kernel_msg_new:nnnn { kernel } { quote-in-shell }
12174 { Quotes~in~shell~command~'1'. }
12175 { Shell~commands~cannot~contain~quotes~( "). }
12176 \__kernel_msg_new:nnnn { kernel } { invalid-quark-function }
12177 { Quark~test~function~'1'~is~invalid. }
12178 {
12179     \c__msg_coding_error_text_tl

```

```

12180 LaTeX-has-been-asked-to-create-quark-test-function-~'~#1'~
12181 \tl_if_empty:nTF {#2}
12182 { but-that-name~ }
12183 { with-signature-~'~#2',~but-that-signature~ }
12184 is-not-valid.
12185 }
12186 \__kernel_msg_new:nnn { kernel } { invalid-quark }
12187 { Invalid-quark-variable-~'~#1'. }
12188 \__kernel_msg_new:nnnn { kernel } { scanmark-already-defined }
12189 { Scan-mark-~#1-already-defined. }
12190 {
12191 \c__msg_coding_error_text_tl
12192 LaTeX-has-been-asked-to-create-a-new-scan-mark-~'~#1'~
12193 but-this-name-has-already-been-used-for-a-scan-mark.
12194 }
12195 \__kernel_msg_new:nnnn { kernel } { shuffle-too-large }
12196 { The-sequence-~#1-is-too-long-to-be-shuffled-by-TeX. }
12197 {
12198 TeX-has~ \int_eval:n { \c_max_register_int + 1 } ~
12199 toks-registers:~this-only-allows-to-shuffle-up-to~
12200 \int_use:N \c_max_register_int \ items.~
12201 The-list-will-not-be-shuffled.
12202 }
12203 \__kernel_msg_new:nnnn { kernel } { variable-not-defined }
12204 { Variable-~#1-undefined. }
12205 {
12206 \c__msg_coding_error_text_tl
12207 LaTeX-has-been-asked-to-show-a-variable-~#1,~but-this-has-not~
12208 been-defined-yet.
12209 }
12210 \__kernel_msg_new:nnnn { kernel } { variant-too-long }
12211 { Variant-form-~'~#1'~longer-than-base-signature-of-~'~#2'. }
12212 {
12213 \c__msg_coding_error_text_tl
12214 LaTeX-has-been-asked-to-create-a-variant-of-the-function-~'~#2'~
12215 with-a-signature-starting-with-~'~#1',~but-that-is-longer-than~
12216 the-signature-(part-after-the-colon)-of-~'~#2'.
12217 }
12218 \__kernel_msg_new:nnnn { kernel } { invalid-variant }
12219 { Variant-form-~'~#1'~invalid-for-base-form-~'~#2'. }
12220 {
12221 \c__msg_coding_error_text_tl
12222 LaTeX-has-been-asked-to-create-a-variant-of-the-function-~'~#2'~
12223 with-a-signature-starting-with-~'~#1',~but-cannot-change-an-argument~
12224 from-type-~'~#3'~to-type-~'~#4'.
12225 }
12226 \__kernel_msg_new:nnnn { kernel } { invalid-exp-args }
12227 { Invalid-variant-specifier-~'~#1'~in-~'~#2'. }
12228 {
12229 \c__msg_coding_error_text_tl
12230 LaTeX-has-been-asked-to-create-an-\iow_char:N\exp_args:N...~
12231 function-with-signature-~'~N#2'~but-~'~#1'~is-not-a-valid-argument~
12232 specifier.
12233 }

```

```

12234 \__kernel_msg_new:nnn { kernel } { deprecated-variant }
12235 {
12236   Variant~form~'#1'~deprecated~for~base~form~'#2'~.~
12237   One~should~not~change~an~argument~from~type~'#3'~to~type~'#4'
12238   \str_case:nnF {#3}
12239   {
12240     { n } { :~use~a~'\token_if_eq_charcode:NNTF #4 c v V'~variant? }
12241     { N } { :~base~form~only~accepts~a~single~token~argument. }
12242     {#4} { :~base~form~is~already~a~variant. }
12243   } { . }
12244 }

```

Some errors are only needed in package mode if debugging is enabled by one of the options `enable-debug`, `check-declarations`, `log-functions`, or on the contrary if debugging is turned off. In format mode the error is somewhat different.

```

12245 \__kernel_msg_new:nnnn { kernel } { enable-debug }
12246 { To~use~'#1'~set~the~'enable-debug'~option. }
12247 {
12248   The~function~'#1'~will~be~ignored~because~it~can~only~work~if~
12249   some~internal~functions~in~expl3~have~been~appropriately~
12250   defined.~This~only~happens~if~one~of~the~options~
12251   'enable-debug',~'check-declarations'~or~'log-functions'~was~
12252   given~as~an~option:~see~the~main~expl3~documentation.
12253 }

```

Some errors only appear in expandable settings, hence don't need a “more-text” argument.

```

12254 \__kernel_msg_new:nnn { kernel } { bad-exp-end-f }
12255 { Misused~\exp_end_continue_f:w or~:nw }
12256 \__kernel_msg_new:nnn { kernel } { bad-variable }
12257 { Erroneous~variable~#1 used! }
12258 \__kernel_msg_new:nnn { kernel } { misused-sequence }
12259 { A~sequence~was~misused. }
12260 \__kernel_msg_new:nnn { kernel } { misused-prop }
12261 { A~property~list~was~misused. }
12262 \__kernel_msg_new:nnn { kernel } { negative-replication }
12263 { Negative~argument~for~\iow_char:N\prg_replicate:nn. }
12264 \__kernel_msg_new:nnn { kernel } { prop-keyval }
12265 { Missing/extra~'=~in~'#1'~(in~'..._keyval:Nn') }
12266 \__kernel_msg_new:nnn { kernel } { unknown-comparison }
12267 { Relation~'#1'~unknown:~use~=,~<,~>,~==,~!=,~<=,~>=. }
12268 \__kernel_msg_new:nnn { kernel } { zero-step }
12269 { Zero~step~size~for~step~function~#1. }
12270 \cs_if_exist:NF \tex_expanded:D
12271 {
12272   \__kernel_msg_new:nnn { kernel } { e-type }
12273   { #1 ~ in~e-type~argument }
12274 }

```

Messages used by the “show” functions.

```

12275 \__kernel_msg_new:nnn { kernel } { show-clist }
12276 {
12277   The~comma~list~ \tl_if_empty:nF {#1} { #1 ~ }
12278   \tl_if_empty:nTF {#2}
12279   { is~empty \\>~ . }

```

```

12280         { contains~the~items~(without~outer~braces): #2 . }
12281     }
12282     \__kernel_msg_new:nnn { kernel } { show-intarray }
12283     { The~integer~array~#1~contains~#2~items: \\ #3 . }
12284     \__kernel_msg_new:nnn { kernel } { show-prop }
12285     {
12286         The~property~list~#1~
12287         \tl_if_empty:nTF {#2}
12288         { is~empty \\>~ . }
12289         { contains~the~pairs~(without~outer~braces): #2 . }
12290     }
12291     \__kernel_msg_new:nnn { kernel } { show-seq }
12292     {
12293         The~sequence~#1~
12294         \tl_if_empty:nTF {#2}
12295         { is~empty \\>~ . }
12296         { contains~the~items~(without~outer~braces): #2 . }
12297     }
12298     \__kernel_msg_new:nnn { kernel } { show-streams }
12299     {
12300         \tl_if_empty:nTF {#2} { No~ } { The~following~ }
12301         \str_case:nn {#1}
12302         {
12303             { ior } { input ~ }
12304             { iow } { output ~ }
12305         }
12306         streams~are~
12307         \tl_if_empty:nTF {#2} { open } { in~use: #2 . }
12308     }

```

System layer messages

```

12309     \__kernel_msg_new:nnnn { sys } { backend-set }
12310     { Backend~configuration~already~set. }
12311     {
12312         Run~time~backend~selection~may~only~be~carried~out~once~during~a~run.~
12313         This~second~attempt~to~set~them~will~be~ignored.
12314     }
12315     \__kernel_msg_new:nnnn { sys } { wrong-backend }
12316     { Backend~request~inconsistent~with~engine:~using~'#2'~backend. }
12317     {
12318         You~have~requested~backend~'#1',~but~this~is~not~suitable~for~use~with~the~
12319         active~engine.~LaTeX3~will~use~the~'#2'~backend~instead.
12320     }

```

19.7 Expandable errors

`__msg_expandable_error:n` In expansion only context, we cannot use the normal means of reporting errors. Instead, we feed \TeX an undefined control sequence, `\LaTeX3 error:`. It is thus interrupted, and shows the context, which thanks to the odd-looking `\use:n` is

```

<argument> \LaTeX3 error:
                The error message.

```

In other words, \TeX is processing the argument of `\use:n`, which is `\LaTeX3 error: <error message>`. Then `__msg_expandable_error:w` cleans up. In fact, there is an

extra subtlety: if the user inserts tokens for error recovery, they should be kept. Thus we also use an odd space character (with category code 7) and keep tokens until that space character, dropping everything else until `\s__msg_stop`. The `\exp_end:` prevents losing braces around the user-inserted text if any, and stops the expansion of `\exp:w`. The group is used to prevent `\LaTeX3~error:` from being globally equal to `\scan_stop:`.

```

12321 \group_begin:
12322 \cs_set_protected:Npn \__msg_tmp:w #1#2
12323 {
12324   \cs_new:Npn \__msg_expandable_error:n ##1
12325   {
12326     \exp:w
12327     \exp_after:wN \exp_after:wN
12328     \exp_after:wN \__msg_expandable_error:w
12329     \exp_after:wN \exp_after:wN
12330     \exp_after:wN \exp_end:
12331     \use:n { #1 #2 ##1 } #2
12332   }
12333   \cs_new:Npn \__msg_expandable_error:w ##1 #2 ##2 #2 {##1}
12334 }
12335 \exp_args:Ncx \__msg_tmp:w { LaTeX3~error: }
12336 { \char_generate:nn { ' \ } { 7 } }
12337 \group_end:

```

(End definition for `__msg_expandable_error:n` and `__msg_expandable_error:w`.)

```

\__kernel_msg_expandable_error:nnnnnn
\__kernel_msg_expandable_error:nnffff
\__kernel_msg_expandable_error:nnnnnn
\__kernel_msg_expandable_error:nnffff
\__kernel_msg_expandable_error:nnnn
\__kernel_msg_expandable_error:nnff
\__kernel_msg_expandable_error:nnnn
\__kernel_msg_expandable_error:nnff
\__kernel_msg_expandable_error:nn
\__kernel_msg_expandable_error:nn

```

The command built from the csname `\c__msg_text_prefix_tl LaTeX / #1 / #2` takes four arguments and builds the error text, which is fed to `__msg_expandable_error:n` with appropriate expansion: just as for usual messages the arguments are first turned to strings, then the message is fully expanded.

```

12338 \exp_args_generate:n { oooo }
12339 \cs_new:Npn \__kernel_msg_expandable_error:nnnnnn #1#2#3#4#5#6
12340 {
12341   \exp_args:Ne \__msg_expandable_error:n
12342   {
12343     \exp_args:Nc \exp_args:Noooo
12344     { \c__msg_text_prefix_tl LaTeX / #1 / #2 }
12345     { \tl_to_str:n {#3} }
12346     { \tl_to_str:n {#4} }
12347     { \tl_to_str:n {#5} }
12348     { \tl_to_str:n {#6} }
12349   }
12350 }
12351 \cs_new:Npn \__kernel_msg_expandable_error:nnnnnn #1#2#3#4#5
12352 {
12353   \__kernel_msg_expandable_error:nnnnnn
12354   {#1} {#2} {#3} {#4} {#5} { }
12355 }
12356 \cs_new:Npn \__kernel_msg_expandable_error:nnnn #1#2#3#4
12357 {
12358   \__kernel_msg_expandable_error:nnnnnn
12359   {#1} {#2} {#3} {#4} { } { }
12360 }
12361 \cs_new:Npn \__kernel_msg_expandable_error:nnn #1#2#3

```

```

12362 {
12363     \__kernel_msg_expandable_error:nnnnnn
12364     {#1} {#2} {#3} { } { } { }
12365 }
12366 \cs_new:Npn \__kernel_msg_expandable_error:nn #1#2
12367 {
12368     \__kernel_msg_expandable_error:nnnnnn
12369     {#1} {#2} { } { } { } { }
12370 }
12371 \cs_generate_variant:Nn \__kernel_msg_expandable_error:nnnnnn { nnffff }
12372 \cs_generate_variant:Nn \__kernel_msg_expandable_error:nnnnn { nnfff }
12373 \cs_generate_variant:Nn \__kernel_msg_expandable_error:nnnn { nnff }
12374 \cs_generate_variant:Nn \__kernel_msg_expandable_error:nnn { nnf }

```

(End definition for __kernel_msg_expandable_error:nnnnnn and others.)

```

\msg_expandable_error:nnnnnn Pass to an auxiliary the message to display and the module name
\msg_expandable_error:nnffff 12375 \cs_new:Npn \msg_expandable_error:nnnnnn #1#2#3#4#5#6
\msg_expandable_error:nnnnn 12376 {
\msg_expandable_error:nnfff 12377     \exp_args:Nc \__msg_expandable_error_module:nn
\msg_expandable_error:nnnn 12378     {
\msg_expandable_error:nnff 12379         \exp_args:Nc \exp_args:Noooo
\msg_expandable_error:nnn 12380         { \c_msg_text_prefix_tl #1 / #2 }
\msg_expandable_error:nnf 12381         { \tl_to_str:n {#3} }
\msg_expandable_error:nn 12382         { \tl_to_str:n {#4} }
\msg_expandable_error:nn 12383         { \tl_to_str:n {#5} }
\msg_expandable_error:nn 12384         { \tl_to_str:n {#6} }
12385     }
12386     {#1}
12387 }
12388 \cs_new:Npn \msg_expandable_error:nnnnn #1#2#3#4#5
12389 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }
12390 \cs_new:Npn \msg_expandable_error:nnnn #1#2#3#4
12391 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} { } { } }
12392 \cs_new:Npn \msg_expandable_error:nnn #1#2#3
12393 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} { } { } { } }
12394 \cs_new:Npn \msg_expandable_error:nn #1#2
12395 { \msg_expandable_error:nnnnnn {#1} {#2} { } { } { } { } }
12396 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnffff }
12397 \cs_generate_variant:Nn \msg_expandable_error:nnnnn { nnfff }
12398 \cs_generate_variant:Nn \msg_expandable_error:nnnn { nnff }
12399 \cs_generate_variant:Nn \msg_expandable_error:nnn { nnf }
12400 \cs_new:Npn \__msg_expandable_error_module:nn #1#2
12401 {
12402     \exp_after:wN \exp_after:wN
12403     \exp_after:wN \__msg_use_none_delimit_by_s_stop:w
12404     \use:n { \::error ! ~ #2 : ~ #1 } \s_msg_stop
12405 }

```

(End definition for \msg_expandable_error:nnnnnn and others. These functions are documented on page 157.)

```

12406 </package>

```

20 l3file implementation

The following test files are used for this code: *m3file001*.

```
12407 <{*package}
```

20.1 Input operations

```
12408 <@@=ior>
```

20.1.1 Variables and constants

```
\l__ior_internal_tl Used as a short-term scratch variable.
12409 \tl_new:N \l__ior_internal_tl
(End definition for \l__ior_internal_tl.)

\c__ior_term_ior Reading from the terminal (with a prompt) is done using a positive but non-existent
stream number. Unlike writing, there is no concept of reading from the log.
12410 \int_const:Nn \c__ior_term_ior { 16 }
(End definition for \c__ior_term_ior.)

\g__ior_streams_seq A list of the currently-available input streams to be used as a stack.
12411 \seq_new:N \g__ior_streams_seq
(End definition for \g__ior_streams_seq.)

\l__ior_stream_tl Used to recover the raw stream number from the stack.
12412 \tl_new:N \l__ior_stream_tl
(End definition for \l__ior_stream_tl.)

\g__ior_streams_prop The name of the file attached to each stream is tracked in a property list. To get the
correct number of reserved streams in package mode the underlying mechanism needs to
be queried. For LATEX 2ε and plain TEX this data is stored in \count16: with the etex
package loaded we need to subtract 1 as the register holds the number of the next stream
to use. In ConTEXt, we need to look at \count38 but there is no subtraction: like the
original plain TEX/LATEX 2ε mechanism it holds the value of the last stream allocated.
12413 \prop_new:N \g__ior_streams_prop
12414 \int_step_inline:nnn
12415 { 0 }
12416 {
12417   \cs_if_exist:NTF \normalend
12418   { \tex_count:D 38 ~ }
12419   {
12420     \tex_count:D 16 ~ %
12421     \cs_if_exist:NT \loccount { - 1 }
12422   }
12423 }
12424 {
12425   \prop_gput:Nnn \g__ior_streams_prop {#1} { Reserved-by~format }
12426 }
```

(End definition for \g__ior_streams_prop.)

20.1.2 Stream management

\ior_new:N Reserving a new stream is done by defining the name as equal to using the terminal.
\ior_new:c 12427 \cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c_ior_term_ior }
 12428 \cs_generate_variant:Nn \ior_new:N { c }

(End definition for \ior_new:N. This function is documented on page 159.)

\g_tmpa_ior The usual scratch space.
\g_tmpb_ior 12429 \ior_new:N \g_tmpa_ior
 12430 \ior_new:N \g_tmpb_ior

(End definition for \g_tmpa_ior and \g_tmpb_ior. These variables are documented on page 166.)

\ior_open:Nn Use the conditional version, with an error if the file is not found.
\ior_open:cn 12431 \cs_new_protected:Npn \ior_open:Nn #1#2
 12432 { \ior_open:NnF #1 {#2} { _kernel_file_missing:n {#2} } }
 12433 \cs_generate_variant:Nn \ior_open:Nn { c }

(End definition for \ior_open:Nn. This function is documented on page 159.)

\l_ior_file_name_tl Data storage.
 12434 \tl_new:N \l_ior_file_name_tl

(End definition for \l_ior_file_name_tl.)

\ior_open:NnTF An auxiliary searches for the file in the T_EX, L^AT_EX 2_ε and L^AT_EX 3 paths. Then pass the
\ior_open:cnTF file found to the lower-level function which deals with streams. The full_name is empty
 when the file is not found.

12435 \prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
 12436 {
 12437 \file_get_full_name:nNTF {#2} \l_ior_file_name_tl
 12438 {
 12439 _kernel_ior_open:No #1 \l_ior_file_name_tl
 12440 \prg_return_true:
 12441 }
 12442 { \prg_return_false: }
 12443 }
 12444 \prg_generate_conditional_variant:Nnn \ior_open:Nn { c } { T , F , TF }

(End definition for \ior_open:NnTF. This function is documented on page 160.)

__ior_new:N Streams are reserved using \newread before they can be managed by ior. To prevent
 ior from being affected by redefinitions of \newread (such as done by the third-party
 package morewrites), this macro is saved here under a private name. The complicated code
 ensures that __ior_new:N is not \outer despite plain T_EX's \newread being \outer.
 For ConT_EXt, we have to deal with the fact that \newread works like our own: it actually
 checks before altering definition.

12445 \exp_args:NNf \cs_new_protected:Npn __ior_new:N
 12446 { \exp_args:NNc \exp_after:wN \exp_stop_f: { newread } }
 12447 \cs_if_exist:NT \normalend
 12448 {
 12449 \cs_new_eq:NN __ior_new_aux:N __ior_new:N
 12450 \cs_set_protected:Npn __ior_new:N #1
 12451 {

```

12452         \cs_undefine:N #1
12453         \__ior_new_aux:N #1
12454     }
12455 }

```

(End definition for __ior_new:N.)

__kernel_ior_open:Nn
__kernel_ior_open:No
__ior_open_stream:Nn

The stream allocation itself uses the fact that there is a list of all of those available. Life gets more complex as it's important to keep things in sync. That is done using a two-part approach: any streams that have already been taken up by ior but are now free are tracked, so we first try those. If that fails, ask plain T_EX or L^AT_EX 2_ε for a new stream and use that number (after a bit of conversion).

```

12456 \cs_new_protected:Npn \__kernel_ior_open:Nn #1#2
12457 {
12458     \ior_close:N #1
12459     \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
12460     { \__ior_open_stream:Nn #1 {#2} }
12461     {
12462         \__ior_new:N #1
12463         \tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
12464         \__ior_open_stream:Nn #1 {#2}
12465     }
12466 }
12467 \cs_generate_variant:Nn \__kernel_ior_open:Nn { No }

```

Here, we act defensively in case LuaT_EX is in use with an extensionless file name.

```

12468 \cs_new_protected:Npx \__ior_open_stream:Nn #1#2
12469 {
12470     \tex_global:D \tex_chardef:D #1 = \exp_not:N \l__ior_stream_tl \scan_stop:
12471     \prop_gput:NVN \exp_not:N \g__ior_streams_prop #1 {#2}
12472     \tex_openin:D #1
12473     \sys_if_engine luatex:TF
12474     { {#2} }
12475     { \exp_not:N \__kernel_file_name_quote:n {#2} \scan_stop: }
12476 }

```

(End definition for __kernel_ior_open:Nn and __ior_open_stream:Nn.)

\ior_close:N
\ior_close:c

Closing a stream means getting rid of it at the T_EX level and removing from the various data structures. Unless the name passed is an invalid stream number (outside the range [0, 15]), it can be closed. On the other hand, it only gets added to the stack if it was not already there, to avoid duplicates building up.

```

12477 \cs_new_protected:Npn \ior_close:N #1
12478 {
12479     \int_compare:nT { -1 < #1 < \c__ior_term_ior }
12480     {
12481         \tex_closein:D #1
12482         \prop_gremove:NV \g__ior_streams_prop #1
12483         \seq_if_in:NVF \g__ior_streams_seq #1
12484         { \seq_gpush:NV \g__ior_streams_seq #1 }
12485         \cs_gset_eq:NN #1 \c__ior_term_ior
12486     }
12487 }
12488 \cs_generate_variant:Nn \ior_close:N { c }

```

(End definition for `\ior_close:N`. This function is documented on page 160.)

`\ior_show_list:` Show the property lists, but with some “pretty printing”. See the `l3msg` module. The first argument of the message is `ior` (as opposed to `iow`) and the second is empty if no read stream is open and non-empty (the list of streams formatted using `\msg_show_item_unbraced:nn`) otherwise. The code of the message `show-streams` takes care of translating `ior/iow` to English.

```

12489 \cs_new_protected:Npn \ior_show_list: { \__ior_list:N \msg_show:nnxxxx }
12490 \cs_new_protected:Npn \ior_log_list: { \__ior_list:N \msg_log:nnxxxx }
12491 \cs_new_protected:Npn \__ior_list:N #1
12492 {
12493   #1 { LaTeX / kernel } { show-streams }
12494   { ior }
12495   {
12496     \prop_map_function:NN \g__ior_streams_prop
12497     \msg_show_item_unbraced:nn
12498   }
12499   { } { }
12500 }
```

(End definition for `\ior_show_list:`, `\ior_log_list:`, and `__ior_list:N`. These functions are documented on page 160.)

20.1.3 Reading input

`\if_eof:w` The primitive conditional

```

12501 \cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

(End definition for `\if_eof:w`. This function is documented on page 166.)

`\ior_if_eof_p:N` To test if some particular input stream is exhausted the following conditional is provided.
`\ior_if_eof:N \underline{TF}` The primitive test can only deal with numbers in the range $[0, 15]$ so we catch outliers (they are exhausted).

```

12502 \prg_new_conditional:Npnn \ior_if_eof:N #1 { p , T , F , TF }
12503 {
12504   \cs_if_exist:NTF #1
12505   {
12506     \int_compare:nTF { -1 < #1 < \c__ior_term_ior }
12507     {
12508       \if_eof:w #1
12509       \prg_return_true:
12510       \else:
12511       \prg_return_false:
12512       \fi:
12513     }
12514     { \prg_return_true: }
12515   }
12516   { \prg_return_true: }
12517 }
```

(End definition for `\ior_if_eof:N \underline{TF}` . This function is documented on page 163.)

```

\ior_get:NN And here we read from files.
\__ior_get:NN 12518 \cs_new_protected:Npn \ior_get:NN #1#2
\ior_get:NNTF 12519 { \ior_get:NnF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
12520 \cs_new_protected:Npn \__ior_get:NN #1#2
12521 { \tex_read:D #1 to #2 }
12522 \prg_new_protected_conditional:Npnn \ior_get:NN #1#2 { T , F , TF }
12523 {
12524 \ior_if_eof:NTF #1
12525 { \prg_return_false: }
12526 {
12527 \__ior_get:NN #1 #2
12528 \prg_return_true:
12529 }
12530 }

```

(End definition for `\ior_get:NN`, `__ior_get:NN`, and `\ior_get:NNTF`. These functions are documented on page 161.)

```

\ior_str_get:NN Reading as strings is a more complicated wrapper, as we wish to remove the endline
\__ior_str_get:NN character and restore it afterwards.
\ior_str_get:NNTF 12531 \cs_new_protected:Npn \ior_str_get:NN #1#2
12532 { \ior_str_get:NnF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
12533 \cs_new_protected:Npn \__ior_str_get:NN #1#2
12534 {
12535 \exp_args:Nno \use:n
12536 {
12537 \int_set:Nn \tex_endlinechar:D { -1 }
12538 \tex_readline:D #1 to #2
12539 \int_set:Nn \tex_endlinechar:D
12540 } { \int_use:N \tex_endlinechar:D }
12541 }
12542 \prg_new_protected_conditional:Npnn \ior_str_get:NN #1#2 { T , F , TF }
12543 {
12544 \ior_if_eof:NTF #1
12545 { \prg_return_false: }
12546 {
12547 \__ior_str_get:NN #1 #2
12548 \prg_return_true:
12549 }
12550 }

```

(End definition for `\ior_str_get:NN`, `__ior_str_get:NN`, and `\ior_str_get:NNTF`. These functions are documented on page 161.)

```

\c__ior_term_noprompt_ior For reading without a prompt.
12551 \int_const:Nn \c__ior_term_noprompt_ior { -1 }

```

(End definition for `\c__ior_term_noprompt_ior`.)

```

\ior_get_term:nN Getting from the terminal is better with pretty-printing.
\ior_str_get_term:nN 12552 \cs_new_protected:Npn \ior_get_term:nN #1#2
\__ior_get_term:NnN 12553 { \__ior_get_term:NnN \__ior_get:NN {#1} #2 }
12554 \cs_new_protected:Npn \ior_str_get_term:nN #1#2
12555 { \__ior_get_term:NnN \__ior_str_get:NN {#1} #2 }
12556 \cs_new_protected:Npn \__ior_get_term:NnN #1#2#3

```

```

12557 {
12558   \group_begin:
12559   \tex_escapechar:D = -1 \scan_stop:
12560   \tl_if_blank:nTF {#2}
12561     { \exp_args:NNc #1 \c__ior_term_noprompt_ior }
12562     { \exp_args:NNc #1 \c__ior_term_ior }
12563     {#2}
12564   \exp_args:NNNv \group_end:
12565   \tl_set:Nn #3 {#2}
12566 }

```

(End definition for `\ior_get_term:nN`, `\ior_str_get_term:nN`, and `__ior_get_term:NnN`. These functions are documented on page 267.)

`\ior_map_break:` Usual map breaking functions.

```

\ior_map_break:n 12567 \cs_new:Npn \ior_map_break:
12568   { \prg_map_break:Nn \ior_map_break: { } }
12569 \cs_new:Npn \ior_map_break:n
12570   { \prg_map_break:Nn \ior_map_break: }

```

(End definition for `\ior_map_break:` and `\ior_map_break:n`. These functions are documented on page 162.)

`\ior_map_inline:Nn` Mapping to an input stream can be done on either a token or a string basis, hence the
`\ior_str_map_inline:Nn` set up. Within that, there is a check to avoid reading past the end of a file, hence the two
`__ior_map_inline:NNn` applications of `\ior_if_eof:N` and its lower-level analogue `\if_eof:w`. This mapping
`__ior_map_inline:NNNn` cannot be nested with twice the same stream, as the stream has only one “current line”.
`__ior_map_inline_loop:NNN`

```

12571 \cs_new_protected:Npn \ior_map_inline:Nn
12572   { \__ior_map_inline:NNn \__ior_get:NN }
12573 \cs_new_protected:Npn \ior_str_map_inline:Nn
12574   { \__ior_map_inline:NNn \__ior_str_get:NN }
12575 \cs_new_protected:Npn \__ior_map_inline:NNn
12576   {
12577     \int_gincr:N \g__kernel_prg_map_int
12578     \exp_args:Nc \__ior_map_inline:NNNn
12579       { __ior_map_ \int_use:N \g__kernel_prg_map_int :n }
12580   }
12581 \cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4
12582   {
12583     \cs_gset_protected:Npn #1 ##1 {#4}
12584     \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
12585     \prg_break_point:Nn \ior_map_break:
12586       { \int_gdecr:N \g__kernel_prg_map_int }
12587   }
12588 \cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
12589   {
12590     #2 #3 \l__ior_internal_tl
12591     \if_eof:w #3
12592       \exp_after:wN \ior_map_break:
12593     \fi:
12594     \exp_args:No #1 \l__ior_internal_tl
12595     \__ior_map_inline_loop:NNN #1#2#3
12596   }

```

(End definition for `\ior_map_inline:Nn` and others. These functions are documented on page 162.)

`\ior_map_variable:NNn` Since the `\TeX` primitive (`\read` or `\readline`) assigns the tokens read in the same way
`\ior_str_map_variable:NNn` as a token list assignment, we simply call the appropriate primitive. The end-of-loop is
`_ior_map_variable:NNNn` checked using the primitive conditional for speed.

```

12597 \cs_new_protected:Npn \ior_map_variable:NNn
12598   { \_ior_map_variable:NNNn \ior_get:NN }
12599 \cs_new_protected:Npn \ior_str_map_variable:NNn
12600   { \_ior_map_variable:NNNn \ior_str_get:NN }
12601 \cs_new_protected:Npn \_ior_map_variable:NNNn #1#2#3#4
12602   {
12603     \ior_if_eof:NF #2 { \_ior_map_variable_loop:NNNn #1#2#3 {#4} }
12604     \prg_break_point:Nn \ior_map_break: { }
12605   }
12606 \cs_new_protected:Npn \_ior_map_variable_loop:NNNn #1#2#3#4
12607   {
12608     #1 #2 #3
12609     \if_eof:w #2
12610       \exp_after:wN \ior_map_break:
12611     \fi:
12612     #4
12613     \_ior_map_variable_loop:NNNn #1#2#3 {#4}
12614   }

```

(End definition for `\ior_map_variable:NNn` and others. These functions are documented on page 162.)

20.2 Output operations

```
12615 <@@=iow>
```

There is a lot of similarity here to the input operations, at least for many of the basics. Thus quite a bit is copied from the earlier material with minor alterations.

20.2.1 Variables and constants

`\c_log_iow` Here we allocate two output streams for writing to the transcript file only (`\c_log_iow`)
`\c_term_iow` and to both the terminal and transcript file (`\c_term_iow`). Recent Lua \TeX provide 128
write streams; we also use `\c_term_iow` as the first non-allowed write stream so its value
depends on the engine.

```

12616 \int_const:Nn \c_log_iow { -1 }
12617 \int_const:Nn \c_term_iow
12618   {
12619     \bool_lazy_and:nnTF
12620       { \sys_if_engine luatex_p: }
12621       { \int_compare_p:nNn \tex_luatexversion:D > { 80 } }
12622     { 128 }
12623     { 16 }
12624   }

```

(End definition for `\c_log_iow` and `\c_term_iow`. These variables are documented on page 166.)

`\g__iow_streams_seq` A list of the currently-available output streams to be used as a stack.

```
12625 \seq_new:N \g__iow_streams_seq
```

(End definition for `\g__iow_streams_seq`.)

`\l__iow_stream_tl` Used to recover the raw stream number from the stack.

```
12626 \tl_new:N \l__iow_stream_tl
```

(End definition for \l__iow_stream_tl.)

\g__iow_streams_prop As for reads with the appropriate adjustment of the register numbers to check on.

```

12627 \prop_new:N \g__iow_streams_prop
12628 \int_step_inline:nnn
12629 { 0 }
12630 {
12631   \cs_if_exist:NTF \normalend
12632   { \tex_count:D 39 ~ }
12633   {
12634     \tex_count:D 17 ~
12635     \cs_if_exist:NT \loccount { - 1 }
12636   }
12637 }
12638 {
12639   \prop_gput:Nnn \g__iow_streams_prop {#1} { Reserved-by~format }
12640 }

```

(End definition for \g__iow_streams_prop.)

20.2.2 Internal auxiliaries

\s__iow_mark Internal scan marks.

```

\s__iow_stop 12641 \scan_new:N \s__iow_mark
12642 \scan_new:N \s__iow_stop

```

(End definition for \s__iow_mark and \s__iow_stop.)

__iow_use_i_delimit_by_s_stop:nw Functions to gobble up to a scan mark.

```

12643 \cs_new:Npn \__iow_use_i_delimit_by_s_stop:nw #1 #2 \s__iow_stop {#1}

```

(End definition for __iow_use_i_delimit_by_s_stop:nw.)

\q__iow_nil Internal quarks.

```

12644 \quark_new:N \q__iow_nil

```

(End definition for \q__iow_nil.)

20.3 Stream management

\iow_new:N Reserving a new stream is done by defining the name as equal to writing to the terminal:
 \iow_new:c odd but at least consistent.

```

12645 \cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
12646 \cs_generate_variant:Nn \iow_new:N { c }

```

(End definition for \iow_new:N. This function is documented on page 159.)

\g_tmpa_iow The usual scratch space.

```

\g_tmpb_iow 12647 \iow_new:N \g_tmpa_iow
12648 \iow_new:N \g_tmpb_iow

```

(End definition for \g_tmpa_iow and \g_tmpb_iow. These variables are documented on page 166.)

`__iow_new:N` As for read streams, copy `\newwrite`, making sure that it is not `\outer`.

```
12649 \exp_args:Nnf \cs_new_protected:Npn \__iow_new:N
12650 { \exp_args:Nnc \exp_after:wN \exp_stop_f: { newwrite } }
```

(End definition for `__iow_new:N`.)

`\l__iow_file_name_tl` Data storage.

```
12651 \tl_new:N \l__iow_file_name_tl
```

(End definition for `\l__iow_file_name_tl`.)

`\iow_open:Nn` The same idea as for reading, but without the path and without the need to allow for a conditional version.

`\iow_open:cn`

```
\__iow_open_stream:Nn 12652 \cs_new_protected:Npn \iow_open:Nn #1#2
\__iow_open_stream:NV 12653 {
12654   \tl_set:Nx \l__iow_file_name_tl
12655   { \__kernel_file_name_sanitiz:n {#2} }
12656   \iow_close:N #1
12657   \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
12658   { \__iow_open_stream:NV #1 \l__iow_file_name_tl }
12659   {
12660     \__iow_new:N #1
12661     \tl_set:Nx \l__iow_stream_tl { \int_eval:n {#1} }
12662     \__iow_open_stream:NV #1 \l__iow_file_name_tl
12663   }
12664 }
12665 \cs_generate_variant:Nn \iow_open:Nn { c }
12666 \cs_new_protected:Npn \__iow_open_stream:Nn #1#2
12667 {
12668   \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:
12669   \prop_gput:NVn \g__iow_streams_prop #1 {#2}
12670   \tex_immediate:D \tex_openout:D
12671   #1 \__kernel_file_name_quote:n {#2} \scan_stop:
12672 }
12673 \cs_generate_variant:Nn \__iow_open_stream:Nn { NV }
```

(End definition for `\iow_open:Nn` and `__iow_open_stream:Nn`. This function is documented on page 160.)

`\iow_close:N` Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.

`\iow_close:c`

```
12674 \cs_new_protected:Npn \iow_close:N #1
12675 {
12676   \int_compare:nT { - \c_log_iow < #1 < \c_term_iow }
12677   {
12678     \tex_immediate:D \tex_closeout:D #1
12679     \prop_gremove:NV \g__iow_streams_prop #1
12680     \seq_if_in:NVF \g__iow_streams_seq #1
12681     { \seq_gpush:NV \g__iow_streams_seq #1 }
12682     \cs_gset_eq:NN #1 \c_term_iow
12683   }
12684 }
12685 \cs_generate_variant:Nn \iow_close:N { c }
```


(End definition for `\iow_close:N`. This function is documented on page 160.)

`\iow_show_list:` Done as for input, but with a copy of the auxiliary so the name is correct.

`\iow_log_list:`

```

12686 \cs_new_protected:Npn \iow_show_list: { \__iow_list:N \msg_show:nnxxxx }
12687 \cs_new_protected:Npn \iow_log_list: { \__iow_list:N \msg_log:nnxxxx }
12688 \cs_new_protected:Npn \__iow_list:N #1
12689 {
12690   #1 { LaTeX / kernel } { show-streams }
12691   { iow }
12692   {
12693     \prop_map_function:NN \g__iow_streams_prop
12694     \msg_show_item_unbraced:nn
12695   }
12696   { } { }
12697 }
```

(End definition for `\iow_show_list:`, `\iow_log_list:`, and `__iow_list:N`. These functions are documented on page 160.)

20.3.1 Deferred writing

`\iow_shipout_x:Nn` First the easy part, this is the primitive, which expects its argument to be braced.

`\iow_shipout_x:Nx`

```

12698 \cs_new_protected:Npn \iow_shipout_x:Nn #1#2
12699 { \tex_write:D #1 {#2} }
```

`\iow_shipout_x:cn`

```

12700 \cs_generate_variant:Nn \iow_shipout_x:Nn { c, Nx, cx }
```

`\iow_shipout_x:cx`

(End definition for `\iow_shipout_x:Nn`. This function is documented on page 164.)

`\iow_shipout:Nn` With ε -TEX available deferred writing without expansion is easy.

`\iow_shipout:Nx`

```

12701 \cs_new_protected:Npn \iow_shipout:Nn #1#2
12702 { \tex_write:D #1 { \exp_not:n {#2} } }
```

`\iow_shipout:cn`

```

12703 \cs_generate_variant:Nn \iow_shipout:Nn { c, Nx, cx }
```

`\iow_shipout:cx`

(End definition for `\iow_shipout:Nn`. This function is documented on page 164.)

20.3.2 Immediate writing

`__kernel_iow_with:Nnn` If the integer #1 is equal to #2, just leave #3 in the input stream. Otherwise, pass the old value to an auxiliary, which sets the integer to the new value, runs the code, and restores the integer.

`__iow_with:nNnn`

```

12704 \cs_new_protected:Npn \__kernel_iow_with:Nnn #1#2
12705 {
12706   \int_compare:nNnTF {#1} = {#2}
12707   { \use:n }
12708   { \exp_args:No \__iow_with:nNnn { \int_use:N #1 } #1 {#2} }
12709 }
12710 \cs_new_protected:Npn \__iow_with:nNnn #1#2#3#4
12711 {
12712   \int_set:Nn #2 {#3}
12713   #4
12714   \int_set:Nn #2 {#1}
12715 }
```

(End definition for `__kernel_iow_with:Nnn` and `__iow_with:nNnn`.)

\iow_now:Nn This routine writes the second argument onto the output stream without expansion. If
\iow_now:Nx this stream isn't open, the output goes to the terminal instead. If the first argument is
\iow_now:cn no output stream at all, we get an internal error. We don't use the expansion done by
\iow_now:cx `\write` to get the `Nx` variant, because it differs in subtle ways from `x`-expansion, namely,
macro parameter characters would not need to be doubled. We set the `\newlinechar`
to 10 using `__kernel_iow_with:Nnn` to support formats such as plain `TEX`: otherwise,
`\iow_newline:` would not work. We do not do this for `\iow_shipout:Nn` or `\iow_`
`shipout_x:Nn`, as `TEX` looks at the value of the `\newlinechar` at shipout time in those
cases.

```

12716 \cs_new_protected:Npn \iow_now:Nn #1#2
12717 {
12718     \__kernel_iow_with:Nnn \tex_newlinechar:D { '\^^J }
12719     { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
12720 }
12721 \cs_generate_variant:Nn \iow_now:Nn { c, Nx, cx }

```

(End definition for `\iow_now:Nn`. This function is documented on page 163.)

\iow_log:n Writing to the log and the terminal directly are relatively easy.
\iow_log:x `\cs_set_protected:Npn \iow_log:x { \iow_now:Nx \c_log_iow }`
\iow_term:n `\cs_new_protected:Npn \iow_log:n { \iow_now:Nn \c_log_iow }`
\iow_term:x `\cs_set_protected:Npn \iow_term:x { \iow_now:Nx \c_term_iow }`
`\cs_new_protected:Npn \iow_term:n { \iow_now:Nn \c_term_iow }`

(End definition for `\iow_log:n` and `\iow_term:n`. These functions are documented on page 163.)

20.3.3 Special characters for writing

\iow_newline: Global variable holding the character that forces a new line when something is written
to an output stream.

```

12726 \cs_new:Npn \iow_newline: { ^^J }

```

(End definition for `\iow_newline:`. This function is documented on page 164.)

\iow_char:N Function to write any escaped char to an output stream.

```

12727 \cs_new_eq:NN \iow_char:N \cs_to_str:N

```

(End definition for `\iow_char:N`. This function is documented on page 164.)

20.3.4 Hard-wrapping lines to a character count

The code here implements a generic hard-wrapping function. This is used by the mes-
saging system, but is designed such that it is available for other uses.

\l_iow_line_count_int This is the “raw” number of characters in a line which can be written to the terminal.
The standard value is the line length typically used by `TEXLive` and `MiKTEX`.

```

12728 \int_new:N \l_iow_line_count_int
12729 \int_set:Nn \l_iow_line_count_int { 78 }

```

(End definition for `\l_iow_line_count_int`. This variable is documented on page 165.)

\l__iow_newline_tl The token list inserted to produce a new line, with the *⟨run-on text⟩*.

```

12730 \tl_new:N \l__iow_newline_tl

```

(End definition for \l__iow_newline_tl.)

\l__iow_line_target_int This stores the target line count: the full number of characters in a line, minus any part for a leader at the start of each line.

```
12731 \int_new:N \l__iow_line_target_int
```

(End definition for \l__iow_line_target_int.)

__iow_set_indent:n The `one_indent` variables hold one indentation marker and its length. The `__iow_unindent:w` auxiliary removes one indentation. The function `__iow_set_indent:n` (that could possibly be public) sets the indentation in a consistent way. We set it to four spaces by default.

```
12732 \tl_new:N \l__iow_one_indent_tl
12733 \int_new:N \l__iow_one_indent_int
12734 \cs_new:Npn \__iow_unindent:w { }
12735 \cs_new_protected:Npn \__iow_set_indent:n #1
12736 {
12737   \tl_set:Nx \l__iow_one_indent_tl
12738     { \exp_args:No \__kernel_str_to_other_fast:n { \tl_to_str:n {#1} } }
12739   \int_set:Nn \l__iow_one_indent_int
12740     { \str_count:N \l__iow_one_indent_tl }
12741   \exp_last_unbraced:NNo
12742     \cs_set:Npn \__iow_unindent:w \l__iow_one_indent_tl { }
12743 }
12744 \exp_args:Nx \__iow_set_indent:n { \prg_replicate:nn { 4 } { ~ } }
```

(End definition for __iow_set_indent:n and others.)

\l__iow_indent_tl The current indentation (some copies of \l__iow_one_indent_tl) and its number of characters.

\l__iow_indent_int

```
12745 \tl_new:N \l__iow_indent_tl
12746 \int_new:N \l__iow_indent_int
```

(End definition for \l__iow_indent_tl and \l__iow_indent_int.)

\l__iow_line_tl These hold the current line of text and a partial line to be added to it, respectively.

\l__iow_line_part_tl

```
12747 \tl_new:N \l__iow_line_tl
12748 \tl_new:N \l__iow_line_part_tl
```

(End definition for \l__iow_line_tl and \l__iow_line_part_tl.)

\l__iow_line_break_bool Indicates whether the line was broken precisely at a chunk boundary.

```
12749 \bool_new:N \l__iow_line_break_bool
```

(End definition for \l__iow_line_break_bool.)

\l__iow_wrap_tl Used for the expansion step before detokenizing, and for the output from wrapping text: fully expanded and with lines which are not overly long.

```
12750 \tl_new:N \l__iow_wrap_tl
```

(End definition for \l__iow_wrap_tl.)

`\c__iow_wrap_marker_tl` Every special action of the wrapping code is starts with the same recognizable string, `\c__iow_wrap_end_marker_tl` `\c__iow_wrap_marker_tl`. Upon seeing that “word”, the wrapping code reads one space-delimited argument to know what operation to perform. The setting of `\escapechar` here is not very important, but makes `\c__iow_wrap_marker_tl` look marginally nicer.

```

12751 \group_begin:
12752   \int_set:Nn \tex_escapechar:D { -1 }
12753   \tl_const:Nx \c__iow_wrap_marker_tl
12754     { \tl_to_str:n { \^^I \^^O \^^W \^^_ \^^W \^^R \^^A \^^P } }
12755 \group_end:
12756 \tl_map_inline:nn
12757   { { end } { newline } { allow_break } { indent } { unindent } }
12758   {
12759     \tl_const:cx { c__iow_wrap_ #1 _marker_tl }
12760     {
12761       \c__iow_wrap_marker_tl
12762       #1
12763       \c_catcode_other_space_tl
12764     }
12765   }

```

(End definition for `\c__iow_wrap_marker_tl` and others.)

`\iow_allow_break:` We set `\iow_allow_break:n` to produce an error when outside messages. Within wrapped message, it is set to `__iow_allow_break:` when valid and otherwise to `__iow_allow_break_error:`. The second produces an error expandably.

```

12766 \cs_new_protected:Npn \iow_allow_break:
12767   {
12768     \__kernel_msg_error:nnnn { kernel } { iow-indent }
12769     { \iow_wrap:nnnN } { \iow_allow_break: }
12770   }
12771 \cs_new:Npx \__iow_allow_break: { \c__iow_wrap_allow_break_marker_tl }
12772 \cs_new:Npn \__iow_allow_break_error:
12773   {
12774     \__kernel_msg_expandable_error:nnnn { kernel } { iow-indent }
12775     { \iow_wrap:nnnN } { \iow_allow_break: }
12776   }

```

(End definition for `\iow_allow_break:`, `__iow_allow_break:`, and `__iow_allow_break_error:`. This function is documented on page 266.)

`\iow_indent:n` We set `\iow_indent:n` to produce an error when outside messages. Within wrapped message, it is set to `__iow_indent:n` when valid and otherwise to `__iow_indent_error:n`. The first places the instruction for increasing the indentation before its argument, and the instruction for unindenting afterwards. The second produces an error expandably. Note that there are no forced line-break, so the indentation only changes when the next line is started.

```

12777 \cs_new_protected:Npn \iow_indent:n #1
12778   {
12779     \__kernel_msg_error:nnnnn { kernel } { iow-indent }
12780     { \iow_wrap:nnnN } { \iow_indent:n } { #1 }
12781     #1
12782   }
12783 \cs_new:Npx \__iow_indent:n #1
12784   {

```

```

12785 \c__iow_wrap_indent_marker_tl
12786 #1
12787 \c__iow_wrap_unindent_marker_tl
12788 }
12789 \cs_new:Npn \__iow_indent_error:n #1
12790 {
12791   \__kernel_msg_expandable_error:nnnnn { kernel } { iow-indent }
12792   { \iow_wrap:nnnN } { \iow_indent:n } {#1}
12793   #1
12794 }

```

(End definition for `\iow_indent:n`, `__iow_indent:n`, and `__iow_indent_error:n`. This function is documented on page 165.)

`\iow_wrap:nnnN` The main wrapping function works as follows. First give `\`, `_` and other formatting commands the correct definition for messages and perform the given setup #3. `\iow_wrap:nxnN` The definition of `_` uses an “other” space rather than a normal space, because the latter might be absorbed by \TeX to end a number or other f-type expansions. Use `\conditionally@traceoff` if defined; it is introduced by the `trace` package and suppresses uninteresting tracing of the wrapping code.

```

12795 \cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
12796 {
12797   \group_begin:
12798   \cs_if_exist_use:N \conditionally@traceoff
12799   \int_set:Nn \tex_escapechar:D { -1 }
12800   \cs_set:Npx \{ { \token_to_str:N \{ }
12801   \cs_set:Npx \# { \token_to_str:N \# }
12802   \cs_set:Npx \} { \token_to_str:N \} }
12803   \cs_set:Npx \% { \token_to_str:N \% }
12804   \cs_set:Npx \~ { \token_to_str:N \~ }
12805   \int_set:Nn \tex_escapechar:D { 92 }
12806   \cs_set_eq:NN \ \ \iow_newline:
12807   \cs_set_eq:NN \_ \c_catcode_other_space_tl
12808   \cs_set_eq:NN \iow_allow_break: \__iow_allow_break:
12809   \cs_set_eq:NN \iow_indent:n \__iow_indent:n
12810   #3

```

Then fully-expand the input: in package mode, the expansion uses $\LaTeX 2_{\epsilon}$ ’s `\protect` mechanism in the same way as `\typeout`. In generic mode this setting is useless but harmless. As soon as the expansion is done, reset `\iow_indent:n` to its error definition: it only works in the first argument of `\iow_wrap:nnnN`.

```

12811   \cs_set_eq:NN \protect \token_to_str:N
12812   \tl_set:Nx \l__iow_wrap_tl {#1}
12813   \cs_set_eq:NN \iow_allow_break: \__iow_allow_break_error:
12814   \cs_set_eq:NN \iow_indent:n \__iow_indent_error:n

```

Afterwards, set the newline marker (two assignments to fully expand, then convert to a string) and initialize the target count for lines (the first line has target count `\l__iow_line_count_int` instead).

```

12815   \tl_set:Nx \l__iow_newline_tl { \iow_newline: #2 }
12816   \tl_set:Nx \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
12817   \int_set:Nn \l__iow_line_target_int
12818   { \l__iow_line_count_int - \str_count:N \l__iow_newline_tl + 1 }

```

Sanity check.

```

12819 \int_compare:nNnT { \l__iow_line_target_int } < 0
12820 {
12821     \tl_set:Nn \l__iow_newline_tl { \iow_newline: }
12822     \int_set:Nn \l__iow_line_target_int
12823         { \l__iow_line_count_int + 1 }
12824 }

```

There is then a loop over the input, which stores the wrapped result in `\l__iow_wrap_tl`. After the loop, the resulting text is passed on to the function which has been given as a post-processor. The `\tl_to_str:N` step converts the “other” spaces back to normal spaces. The `f`-expansion removes a leading space from `\l__iow_wrap_tl`.

```

12825 \__iow_wrap_do:
12826 \exp_args:NNf \group_end:
12827 #4 { \tl_to_str:N \l__iow_wrap_tl }
12828 }
12829 \cs_generate_variant:Nn \iow_wrap:nnnN { nx }

```

(End definition for `\iow_wrap:nnnN`. This function is documented on page 165.)

<pre> __iow_wrap_do: __iow_wrap_fix_newline:w __iow_wrap_start:w </pre>	<p>Escape spaces and change newlines to <code>\c__iow_wrap_newline_marker_tl</code>. Set up a few variables, in particular the initial value of <code>\l__iow_wrap_tl</code>: the space stops the <code>f</code>-expansion of the main wrapping function and <code>\use_none:n</code> removes a newline marker inserted by later code. The main loop consists of repeatedly calling the <code>chunk</code> auxiliary to wrap chunks delimited by (newline or indentation) markers.</p>
--	--

```

12830 \cs_new_protected:Npn \__iow_wrap_do:
12831 {
12832     \tl_set:Nx \l__iow_wrap_tl
12833     {
12834         \exp_args:No \__kernel_str_to_other_fast:n \l__iow_wrap_tl
12835         \c__iow_wrap_end_marker_tl
12836     }
12837     \tl_set:Nx \l__iow_wrap_tl
12838     {
12839         \exp_after:wN \__iow_wrap_fix_newline:w \l__iow_wrap_tl
12840         ^^J \q__iow_nil ^^J \s__iow_stop
12841     }
12842     \exp_after:wN \__iow_wrap_start:w \l__iow_wrap_tl
12843 }
12844 \cs_new:Npn \__iow_wrap_fix_newline:w #1 ^^J #2 ^^J
12845 {
12846     #1
12847     \if_meaning:w \q__iow_nil #2
12848     \__iow_use_i_delimit_by_s_stop:nw
12849     \fi:
12850     \c__iow_wrap_newline_marker_tl
12851     \__iow_wrap_fix_newline:w #2 ^^J
12852 }
12853 \cs_new_protected:Npn \__iow_wrap_start:w
12854 {
12855     \bool_set_false:N \l__iow_line_break_bool
12856     \tl_clear:N \l__iow_line_tl
12857     \tl_clear:N \l__iow_line_part_tl
12858     \tl_set:Nn \l__iow_wrap_tl { ~ \use_none:n }

```

```

12859 \int_zero:N \l__iow_indent_int
12860 \tl_clear:N \l__iow_indent_tl
12861 \__iow_wrap_chunk:nw { \l_iow_line_count_int }
12862 }

```

(End definition for `__iow_wrap_do:`, `__iow_wrap_fix_newline:w`, and `__iow_wrap_start:w`.)

`__iow_wrap_chunk:nw` The `chunk` and `next` auxiliaries are defined indirectly to obtain the expansions of `\c_catcode_other_space_tl` and `\c__iow_wrap_marker_tl` in their definition. The `next` auxiliary calls a function corresponding to the type of marker (its `##2`), which can be `newline` or `indent` or `unindent` or `end`. The first argument of the `chunk` auxiliary is a target number of characters and the second is some string to wrap. If the chunk is empty simply call `next`. Otherwise, set up a call to `__iow_wrap_line:nw`, including the indentation if the current line is empty, and including a trailing space (`#1`) before the `__iow_wrap_end_chunk:w` auxiliary.

```

12863 \cs_set_protected:Npn \__iow_tmp:w #1#2
12864 {
12865   \cs_new_protected:Npn \__iow_wrap_chunk:nw ##1##2 #2
12866   {
12867     \tl_if_empty:NTF {##2}
12868     {
12869       \tl_clear:N \l__iow_line_part_tl
12870       \__iow_wrap_next:nw {##1}
12871     }
12872     {
12873       \tl_if_empty:NTF \l__iow_line_tl
12874       {
12875         \__iow_wrap_line:nw
12876         { \l__iow_indent_tl }
12877         ##1 - \l__iow_indent_int ;
12878       }
12879       { \__iow_wrap_line:nw { } ##1 ; }
12880       ##2 #1
12881       \__iow_wrap_end_chunk:w 7 6 5 4 3 2 1 0 \s__iow_stop
12882     }
12883   }
12884   \cs_new_protected:Npn \__iow_wrap_next:nw ##1##2 #1
12885   { \use:c { __iow_wrap_##2:n } {##1} }
12886 }
12887 \exp_args:NVV \__iow_tmp:w \c_catcode_other_space_tl \c__iow_wrap_marker_tl

```

(End definition for `__iow_wrap_chunk:nw` and `__iow_wrap_next:nw`.)

`__iow_wrap_line:nw` This is followed by `{\langle string \rangle \langle intexpr \rangle ;}`. It stores the `\langle string \rangle` and up to `\langle intexpr \rangle` characters from the current chunk into `\l__iow_line_part_tl`. Characters are grabbed 8 at a time and left in `\l__iow_line_part_tl` by the `line_loop` auxiliary. When $k < 8$ remain to be found, the `line_aux` auxiliary calls the `line_end` auxiliary followed by (the single digit) k , then $7 - k$ empty brace groups, then the chunk's remaining characters. The `line_end` auxiliary leaves k characters from the chunk in the line part, then ends the assignment. Ignore the `\use_none:nnnnn` line for now. If the next character is a space the line can be broken there: store what we found into the result and get the next line. Otherwise some work is needed to find a break-point. So far we have ignored what happens if the chunk is shorter than the requested number of characters: this is dealt

with by the `end_chunk` auxiliary, which gets treated like a character by the rest of the code. It ends up being called either as one of the arguments #2-#9 of the `line_loop` auxiliary or as one of the arguments #2-#8 of the `line_end` auxiliary. In both cases stop the assignment and work out how many characters are still needed. Notice that when we have exactly seven arguments to clean up, a `\exp_stop_f:` has to be inserted to stop the `\exp:w`. The weird `\use_none:nnnnn` ensures that the required data is in the right place.

```

12888 \cs_new_protected:Npn \__iow_wrap_line:nw #1
12889 {
12890   \tex_edef:D \l__iow_line_part_tl { \if_false: } \fi:
12891   #1
12892   \exp_after:wN \__iow_wrap_line_loop:w
12893   \int_value:w \int_eval:w
12894 }
12895 \cs_new:Npn \__iow_wrap_line_loop:w #1 ; #2#3#4#5#6#7#8#9
12896 {
12897   \if_int_compare:w #1 < 8 \exp_stop_f:
12898     \__iow_wrap_line_aux:Nw #1
12899   \fi:
12900   #2 #3 #4 #5 #6 #7 #8 #9
12901   \exp_after:wN \__iow_wrap_line_loop:w
12902   \int_value:w \int_eval:w #1 - 8 ;
12903 }
12904 \cs_new:Npn \__iow_wrap_line_aux:Nw #1#2#3 \exp_after:wN #4 ;
12905 {
12906   #2
12907   \exp_after:wN \__iow_wrap_line_end:NnnnnnnnN
12908   \exp_after:wN #1
12909   \exp:w \exp_end_continue_f:w
12910   \exp_after:wN \exp_after:wN
12911   \if_case:w #1 \exp_stop_f:
12912     \prg_do_nothing:
12913   \or: \use_none:n
12914   \or: \use_none:nn
12915   \or: \use_none:nnn
12916   \or: \use_none:nnnn
12917   \or: \use_none:nnnnn
12918   \or: \use_none:nnnnnn
12919   \or: \__iow_wrap_line_seven:nnnnnnn
12920   \fi:
12921   { } { } { } { } { } { } { } { } { } #3
12922 }
12923 \cs_new:Npn \__iow_wrap_line_seven:nnnnnnn #1#2#3#4#5#6#7 { \exp_stop_f: }
12924 \cs_new:Npn \__iow_wrap_line_end:NnnnnnnnN #1#2#3#4#5#6#7#8#9
12925 {
12926   #2 #3 #4 #5 #6 #7 #8
12927   \use_none:nnnnn \int_eval:w 8 - ; #9
12928   \token_if_eq_charcode:NNTF \c_space_token #9
12929     { \__iow_wrap_line_end:nw { } }
12930     { \if_false: { \fi: } \__iow_wrap_break:w #9 }
12931 }
12932 \cs_new:Npn \__iow_wrap_line_end:nw #1
12933 {

```



```

12934     \if_false: { \fi: }
12935     \__iow_wrap_store_do:n {#1}
12936     \__iow_wrap_next_line:w
12937   }
12938 \cs_new:Npn \__iow_wrap_end_chunk:w
12939   #1 \int_eval:w #2 - #3 ; #4#5 \s__iow_stop
12940 {
12941   \if_false: { \fi: }
12942   \exp_args:Nf \__iow_wrap_next:nw { \int_eval:n { #2 - #4 } }
12943 }

```

(End definition for __iow_wrap_line:nw and others.)

__iow_wrap_break:w Functions here are defined indirectly: __iow_tmp:w is eventually called with an “other”
 __iow_wrap_break_first:w space as its argument. The goal is to remove from \l__iow_line_part_tl the part
 __iow_wrap_break_none:w after the last space. In most cases this is done by repeatedly calling the **break_loop**
 __iow_wrap_break_loop:w auxiliary, which leaves “words” (delimited by spaces) until it hits the trailing space: then
 __iow_wrap_break_end:w its argument **##3** is ? __iow_wrap_break_end:w instead of a single token, and that
break_end auxiliary leaves in the assignment the line until the last space, then calls
 __iow_wrap_line_end:nw to finish up the line and move on to the next. If there is
 no space in \l__iow_line_part_tl then the **break_first** auxiliary calls the **break_**
none auxiliary. In that case, if the current line is empty, the complete word (including
##4, characters beyond what we had grabbed) is added to the line, making it over-long.
 Otherwise, the word is used for the following line (and the last space of the line so far is
 removed because it was inserted due to the presence of a marker).

```

12944 \cs_set_protected:Npn \__iow_tmp:w #1
12945 {
12946   \cs_new:Npn \__iow_wrap_break:w
12947   {
12948     \tex_edef:D \l__iow_line_part_tl
12949     { \if_false: } \fi:
12950     \exp_after:wN \__iow_wrap_break_first:w
12951     \l__iow_line_part_tl
12952     #1
12953     { ? \__iow_wrap_break_end:w }
12954     \s__iow_mark
12955   }
12956 \cs_new:Npn \__iow_wrap_break_first:w ##1 #1 ##2
12957 {
12958   \use_none:nn ##2 \__iow_wrap_break_none:w
12959   \__iow_wrap_break_loop:w ##1 #1 ##2
12960 }
12961 \cs_new:Npn \__iow_wrap_break_none:w ##1##2 #1 ##3 \s__iow_mark ##4 #1
12962 {
12963   \tl_if_empty:NTF \l__iow_line_tl
12964   { ##2 ##4 \__iow_wrap_line_end:nw { } }
12965   { \__iow_wrap_line_end:nw { \__iow_wrap_trim:N } ##2 ##4 #1 }
12966 }
12967 \cs_new:Npn \__iow_wrap_break_loop:w ##1 #1 ##2 #1 ##3
12968 {
12969   \use_none:n ##3
12970   ##1 #1
12971   \__iow_wrap_break_loop:w ##2 #1 ##3

```

```

12972     }
12973     \cs_new:Npn \__iow_wrap_break_end:w ##1 #1 ##2 ##3 #1 ##4 \s__iow_mark
12974     { ##1 \__iow_wrap_line_end:nw { } ##3 }
12975   }
12976   \exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl

```

(End definition for __iow_wrap_break:w and others.)

`__iow_wrap_next_line:w` The special case where the end of a line coincides with the end of a chunk is detected here, to avoid a spurious empty line. Otherwise, call `__iow_wrap_line:nw` to find characters for the next line (remembering to account for the indentation).

```

12977 \cs_new_protected:Npn \__iow_wrap_next_line:w #1#2 \s__iow_stop
12978 {
12979   \tl_clear:N \l__iow_line_tl
12980   \token_if_eq_meaning:NNTF #1 \__iow_wrap_end_chunk:w
12981   {
12982     \tl_clear:N \l__iow_line_part_tl
12983     \bool_set_true:N \l__iow_line_break_bool
12984     \__iow_wrap_next:nw { \l__iow_line_target_int }
12985   }
12986   {
12987     \__iow_wrap_line:nw
12988     { \l__iow_indent_tl }
12989     \l__iow_line_target_int - \l__iow_indent_int ;
12990     #1 #2 \s__iow_stop
12991   }
12992 }

```

(End definition for __iow_wrap_next_line:w.)

`__iow_wrap_allow_break:n` This is called after a chunk has been wrapped. The `\l__iow_line_part_tl` typically ends with a space (except at the beginning of a line?), which we remove since the `allow-break` marker should not insert a space. Then move on with the next chunk, making sure to adjust the target number of characters for the line in case we did remove a space.

```

12993 \cs_new_protected:Npn \__iow_wrap_allow_break:n #1
12994 {
12995   \tl_set:Nx \l__iow_line_tl
12996   { \l__iow_line_tl \__iow_wrap_trim:N \l__iow_line_part_tl }
12997   \bool_set_false:N \l__iow_line_break_bool
12998   \tl_if_empty:NNTF \l__iow_line_part_tl
12999   { \__iow_wrap_chunk:nw {#1} }
13000   { \exp_args:Nf \__iow_wrap_chunk:nw { \int_eval:n { #1 + 1 } } }
13001 }

```

(End definition for __iow_wrap_allow_break:n.)

`__iow_wrap_indent:n` These functions are called after a chunk has been wrapped, when encountering `indent/unindent` markers. Add the line part (last line part of the previous chunk) to the line so far and reset a boolean denoting the presence of a line-break. Most importantly, add or remove one indent from the current indent (both the integer and the token list). Finally, continue wrapping.

`__iow_wrap_unindent:n`

```

13002 \cs_new_protected:Npn \__iow_wrap_indent:n #1
13003 {
13004   \tl_put_right:Nx \l__iow_line_tl { \l__iow_line_part_tl }

```

```

13005     \bool_set_false:N \l__iow_line_break_bool
13006     \int_add:Nn \l__iow_indent_int { \l__iow_one_indent_int }
13007     \tl_put_right:No \l__iow_indent_tl { \l__iow_one_indent_tl }
13008     \__iow_wrap_chunk:nw {#1}
13009   }
13010 \cs_new_protected:Npn \__iow_wrap_unindent:n #1
13011 {
13012   \tl_put_right:Nx \l__iow_line_tl { \l__iow_line_part_tl }
13013   \bool_set_false:N \l__iow_line_break_bool
13014   \int_sub:Nn \l__iow_indent_int { \l__iow_one_indent_int }
13015   \tl_set:Nx \l__iow_indent_tl
13016     { \exp_after:wN \__iow_unindent:w \l__iow_indent_tl }
13017   \__iow_wrap_chunk:nw {#1}
13018 }

```

(End definition for __iow_wrap_indent:n and __iow_wrap_unindent:n.)

__iow_wrap_newline:n These functions are called after a chunk has been line-wrapped, when encountering a
 __iow_wrap_end:n **newline/end** marker. Unless we just took a line-break, store the line part and the line
 so far into the whole \l__iow_wrap_tl, trimming a trailing space. In the **newline** case
 look for a new line (of length \l__iow_line_target_int) in a new chunk.

```

13019 \cs_new_protected:Npn \__iow_wrap_newline:n #1
13020 {
13021   \bool_if:NF \l__iow_line_break_bool
13022     { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
13023   \bool_set_false:N \l__iow_line_break_bool
13024   \__iow_wrap_chunk:nw { \l__iow_line_target_int }
13025 }
13026 \cs_new_protected:Npn \__iow_wrap_end:n #1
13027 {
13028   \bool_if:NF \l__iow_line_break_bool
13029     { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
13030   \bool_set_false:N \l__iow_line_break_bool
13031 }

```

(End definition for __iow_wrap_newline:n and __iow_wrap_end:n.)

__iow_wrap_store_do:n First add the last line part to the line, then append it to \l__iow_wrap_tl with the
 appropriate new line (with “run-on” text), possibly with its last space removed (#1 is
 empty or __iow_wrap_trim:N).

```

13032 \cs_new_protected:Npn \__iow_wrap_store_do:n #1
13033 {
13034   \tl_set:Nx \l__iow_line_tl
13035     { \l__iow_line_tl \l__iow_line_part_tl }
13036   \tl_set:Nx \l__iow_wrap_tl
13037     {
13038       \l__iow_wrap_tl
13039       \l__iow_newline_tl
13040       #1 \l__iow_line_tl
13041     }
13042   \tl_clear:N \l__iow_line_tl
13043 }

```

(End definition for __iow_wrap_store_do:n.)

```

\__iow_wrap_trim:N Remove one trailing “other” space from the argument if present.
\__iow_wrap_trim:w
\__iow_wrap_trim_aux:w
13044 \cs_set_protected:Npn \__iow_tmp:w #1
13045 {
13046   \cs_new:Npn \__iow_wrap_trim:N ##1
13047     { \exp_after:wN \__iow_wrap_trim:w ##1 \s__iow_mark #1 \s__iow_mark \s__iow_stop }
13048   \cs_new:Npn \__iow_wrap_trim:w ##1 #1 \s__iow_mark
13049     { \__iow_wrap_trim_aux:w ##1 \s__iow_mark }
13050   \cs_new:Npn \__iow_wrap_trim_aux:w ##1 \s__iow_mark ##2 \s__iow_stop {##1}
13051 }
13052 \exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl

(End definition for \__iow_wrap_trim:N, \__iow_wrap_trim:w, and \__iow_wrap_trim_aux:w.)

13053 <@@=file>

```

20.4 File operations

`\l__file_internal_tl` Used as a short-term scratch variable.

```
13054 \tl_new:N \l__file_internal_tl
```

(End definition for `\l__file_internal_tl`.)

`\g_file_curr_dir_str` `\g_file_curr_ext_str` `\g_file_curr_name_str` The name of the current file should be available at all times: the name itself is set dynamically.

```

13055 \str_new:N \g_file_curr_dir_str
13056 \str_new:N \g_file_curr_ext_str
13057 \str_new:N \g_file_curr_name_str

```

(End definition for `\g_file_curr_dir_str`, `\g_file_curr_ext_str`, and `\g_file_curr_name_str`. These variables are documented on page 166.)

`\g__file_stack_seq` The input list of files is stored as a sequence stack. In package mode we can recover the information from the details held by $\text{\LaTeX} 2_{\epsilon}$ (we must be in the preamble and loaded using `\usepackage` or `\RequirePackage`). As $\text{\LaTeX} 2_{\epsilon}$ doesn’t store directory and name separately, we stick to the same convention here. In pre-loading, `\@currnamestack` is empty so is skipped.

```

13058 \seq_new:N \g__file_stack_seq
13059 \group_begin:
13060   \cs_set_protected:Npn \__file_tmp:w #1#2#3
13061   {
13062     \tl_if_blank:nTF {#1}
13063     {
13064       \cs_set:Npn \__file_tmp:w ##1 " ##2 " ##3 \s__file_stop
13065       { { } {##2} { } }
13066       \seq_gput_right:Nx \g__file_stack_seq
13067       {
13068         \exp_after:wN \__file_tmp:w \tex_jobname:D
13069         " \tex_jobname:D " \s__file_stop
13070       }
13071     }
13072   {
13073     \seq_gput_right:Nn \g__file_stack_seq { { } {#1} {#2} }
13074     \__file_tmp:w
13075   }

```

```

13076     }
13077     \cs_if_exist:NT \@currnamestack
13078     {
13079         \tl_if_empty:NF \@currnamestack
13080         { \exp_after:wN \__file_tmp:w \@currnamestack }
13081     }
13082 \group_end:

```

(End definition for \g__file_stack_seq.)

`\g__file_record_seq` The total list of files used is recorded separately from the current file stack, as nothing is ever popped from this list. The current file name should be included in the file list! We will eventually copy the contents of `\@filelist`.

```
13083 \seq_new:N \g__file_record_seq
```

(End definition for \g__file_record_seq.)

`\l__file_base_name_tl` For storing the basename and full path whilst passing data internally.

```

\l__file_full_name_tl 13084 \tl_new:N \l__file_base_name_tl
13085 \tl_new:N \l__file_full_name_tl

```

(End definition for \l__file_base_name_tl and \l__file_full_name_tl.)

`\l__file_dir_str` Used in parsing a path into parts: in contrast to the above, these are never used outside of the current module.

```

\l__file_ext_str 13086 \str_new:N \l__file_dir_str
13087 \str_new:N \l__file_ext_str
\l__file_name_str 13088 \str_new:N \l__file_name_str

```

(End definition for \l__file_dir_str, \l__file_ext_str, and \l__file_name_str.)

`\l_file_search_path_seq` The current search path.

```
13089 \seq_new:N \l_file_search_path_seq
```

(End definition for \l_file_search_path_seq. This variable is documented on page 167.)

`\l__file_tmp_seq` Scratch space for comma list conversion.

```
13090 \seq_new:N \l__file_tmp_seq
```

(End definition for \l__file_tmp_seq.)

20.4.1 Internal auxiliaries

`\s__file_stop` Internal scan marks.

```
13091 \scan_new:N \s__file_stop
```

(End definition for \s__file_stop.)

`\q__file_nil` Internal quarks.

```
13092 \quark_new:N \q__file_nil
```

(End definition for \q__file_nil.)

`__file_quark_if_nil_p:n` Branching quark conditional.

```

\__file_quark_if_nil:nTF 13093 \__kernel_quark_new_conditional:Nn \__file_quark_if_nil:n { TF }

```

(End definition for _file_quark_if_nil:nTF.)

\q__file_recursion_tail Internal recursion quarks.

\q__file_recursion_stop 13094 \quark_new:N \q__file_recursion_tail
13095 \quark_new:N \q__file_recursion_stop

(End definition for \q__file_recursion_tail and \q__file_recursion_stop.)

_file_if_recursion_tail_break:NN Functions to query recursion quarks.

_file_if_recursion_tail_stop_do:Nn 13096 __kernel_quark_new_test:N _file_if_recursion_tail_stop:N
13097 __kernel_quark_new_test:N _file_if_recursion_tail_stop_do:nn

(End definition for _file_if_recursion_tail_break:NN and _file_if_recursion_tail_stop_do:Nn.)

_kernel_file_name_sanitize:n Expanding the file name without expanding active characters is done using the same token-by-token approach as for example case changing. The final outcome only needs to be e-type expandable, so there is no need for the shuffling that is seen in other locations.

_kernel_file_name_expand_loop:w 13098 \cs_new:Npn _kernel_file_name_sanitize:n #1
_kernel_file_name_expand_N_type:Nw 13099 {
_kernel_file_name_expand_group:nw 13100 \exp_args:Ne _kernel_file_name_trim_spaces:n
_kernel_file_name_expand_space:w 13101 {
_kernel_file_name_strip_quotes:n 13102 \exp_args:Ne _kernel_file_name_strip_quotes:n
_kernel_file_name_strip_quotes:nnw 13103 {
_kernel_file_name_strip_quotes:nnn 13104 _kernel_file_name_expand_loop:w #1
_kernel_file_name_trim_spaces:n 13105 \q__file_recursion_tail \q__file_recursion_stop
_kernel_file_name_trim_spaces:nw 13106 }
_kernel_file_name_trim_spaces_aux:n 13107 }
_kernel_file_name_trim_spaces_aux:w 13108 }
13109 \cs_new:Npn _kernel_file_name_expand_loop:w #1 \q__file_recursion_stop
13110 {
13111 \tl_if_head_is_N_type:nTF {#1}
13112 { _kernel_file_name_expand_N_type:Nw }
13113 {
13114 \tl_if_head_is_group:nTF {#1}
13115 { _kernel_file_name_expand_group:nw }
13116 { _kernel_file_name_expand_space:w }
13117 }
13118 #1 \q__file_recursion_stop
13119 }
13120 \cs_new:Npn _kernel_file_name_expand_N_type:Nw #1
13121 {
13122 _file_if_recursion_tail_stop:N #1
13123 \bool_lazy_and:nnTF
13124 { \token_if_expandable_p:N #1 }
13125 {
13126 \bool_not_p:n
13127 {
13128 \bool_lazy_any_p:n
13129 {
13130 { \token_if_protected_macro_p:N #1 }
13131 { \token_if_protected_long_macro_p:N #1 }
13132 { \token_if_active_p:N #1 }
13133 }

```

13134     }
13135   }
13136   { \exp_after:wN \__kernel_file_name_expand_loop:w #1 }
13137   {
13138     \token_to_str:N #1
13139     \__kernel_file_name_expand_loop:w
13140   }
13141 }
13142 \cs_new:Npx \__kernel_file_name_expand_group:nw #1
13143 {
13144   \c_left_brace_str
13145   \exp_not:N \__kernel_file_name_expand_loop:w
13146   #1
13147   \c_right_brace_str
13148 }
13149 \exp_last_unbraced:NNo
13150 \cs_new:Npx \__kernel_file_name_expand_space:w \c_space_tl
13151 {
13152   \c_space_tl
13153   \exp_not:N \__kernel_file_name_expand_loop:w
13154 }

```

Quoting file name uses basically the same approach as for luaquotejobname: count the " tokens and remove them.

```

13155 \cs_new:Npn \__kernel_file_name_strip_quotes:n #1
13156 {
13157   \__kernel_file_name_strip_quotes:nnnw {#1} { 0 } { }
13158   #1 " \q__file_recursion_tail " \q__file_recursion_stop
13159 }
13160 \cs_new:Npn \__kernel_file_name_strip_quotes:nnnw #1#2#3#4 "
13161 {
13162   \__file_if_recursion_tail_stop_do:nn {#4}
13163   { \__kernel_file_name_strip_quotes:nnn {#1} {#2} {#3} }
13164   \__kernel_file_name_strip_quotes:nnnw {#1} { #2 + 1 } { #3#4 }
13165 }
13166 \cs_new:Npn \__kernel_file_name_strip_quotes:nnn #1#2#3
13167 {
13168   \int_if_even:nT {#2}
13169   {
13170     \__kernel_msg_expandable_error:nnn
13171     { kernel } { unbalanced-quote-in-filename } {#1}
13172   }
13173   #3
13174 }

```

Spaces need to be trimmed from the start of the name and from the end of any extension. However, the name we are passed might not have an extension: that means we have to look for one. If there is no extension, we still use the standard trimming function but deliberately prevent any spaces being removed at the end.

```

13175 \cs_new:Npn \__kernel_file_name_trim_spaces:n #1
13176 { \__kernel_file_name_trim_spaces:nw {#1} #1 . \q__file_nil . \s__file_stop }
13177 \cs_new:Npn \__kernel_file_name_trim_spaces:nw #1#2 . #3 . #4 \s__file_stop
13178 {
13179   \__file_quark_if_nil:nTF {#3}
13180   {

```

```

13181         \exp_args:Ne \__kernel_file_name_trim_spaces_aux:n
13182         { \tl_trim_spaces:n { #1 \s__file_stop } }
13183     }
13184     { \tl_trim_spaces:n {#1} }
13185 }
13186 \cs_new:Npn \__kernel_file_name_trim_spaces_aux:n #1
13187 { \__kernel_file_name_trim_spaces_aux:w #1 }
13188 \cs_new:Npn \__kernel_file_name_trim_spaces_aux:w #1 \s__file_stop {#1}

```

(End definition for __kernel_file_name_sanitize:n and others.)

```

\__kernel_file_name_quote:n
\__kernel_file_name_quote:nw
13189 \cs_new:Npn \__kernel_file_name_quote:n #1
13190 { \__kernel_file_name_quote:nw {#1} #1 ~ \q__file_nil \s__file_stop }
13191 \cs_new:Npn \__kernel_file_name_quote:nw #1 #2 ~ #3 \s__file_stop
13192 {
13193     \__file_quark_if_nil:nTF {#3}
13194     { #1 }
13195     { "#1" }
13196 }

```

(End definition for __kernel_file_name_quote:n and __kernel_file_name_quote:nw.)

\c__file_marker_tl The same idea as the marker for rescanning token lists: this pair of tokens cannot appear in a file that is being input.

```

13197 \tl_const:Nx \c__file_marker_tl { : \token_to_str:N : }

```

(End definition for \c__file_marker_tl.)

\file_get:nnNTF The approach here is similar to that for \tl_set_rescan:Nnn. The file contents are grabbed as an argument delimited by \c__file_marker_tl. A few subtleties: braces in **\file_get:nnN** **\if_false: ... \fi:** to deal with possible alignment tabs, \tracingnesting to avoid a warning about a group being closed inside the \scantokens, and \prg_return_true: is placed after the end-of-file marker.

```

13198 \cs_new_protected:Npn \file_get:nnN #1#2#3
13199 {
13200     \file_get:nnNF {#1} {#2} #3
13201     { \tl_set:Nn #3 { \q_no_value } }
13202 }
13203 \prg_new_protected_conditional:Npnn \file_get:nnN #1#2#3 { T , F , TF }
13204 {
13205     \file_get_full_name:nNTF {#1} \l__file_full_name_tl
13206     {
13207         \exp_args:NV \__file_get_aux:nnN
13208         \l__file_full_name_tl
13209         {#2} #3
13210         \prg_return_true:
13211     }
13212     { \prg_return_false: }
13213 }
13214 \cs_new_protected:Npx \__file_get_aux:nnN #1#2#3
13215 {
13216     \exp_not:N \if_false: { \exp_not:N \fi:
13217     \group_begin:

```



```

13218 \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
13219 \exp_not:N \exp_args:No \tex_everyeof:D
13220 { \exp_not:N \c__file_marker_tl }
13221 #2 \scan_stop:
13222 \exp_not:N \exp_after:wN \exp_not:N \__file_get_do:Nw
13223 \exp_not:N \exp_after:wN #3
13224 \exp_not:N \exp_after:wN \exp_not:N \prg_do_nothing:
13225 \exp_not:N \tex_input:D
13226 \sys_if_engine luatex:TF
13227 { {#1} }
13228 { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
13229 \exp_not:N \if_false: } \exp_not:N \fi:
13230 }
13231 \exp_args:Nno \use:nn
13232 { \cs_new_protected:Npn \__file_get_do:Nw #1#2 }
13233 { \c__file_marker_tl }
13234 {
13235 \group_end:
13236 \tl_set:No #1 {#2}
13237 }

```

(End definition for `\file_get:nnNTF` and others. These functions are documented on page 167.)

`__file_size:n` A copy of the primitive where it's available, or the LuaTeX equivalent if relevant.

```

13238 \cs_new_eq:NN \__file_size:n \tex_filesize:D
13239 \sys_if_engine luatex:T
13240 {
13241 \cs_gset:Npn \__file_size:n #1
13242 {
13243 \lua_now:e
13244 { l3kernel.filesize ( " \lua_escape:e {#1} " ) }
13245 }
13246 }

```

(End definition for `__file_size:n`.)

`\file_full_name:n` File searching can be carried out if the `\pdffilesize` primitive or an equivalent is available. That of course means we need to arrange for everything else to here to be done by expansion too. We start off by sanitizing the name and quoting if required: we may need to remove those quotes, so the raw name is passed too.

```

\__file_full_name:n
\__file_full_name_aux:Nnn
\__file_full_name_aux:nN
\__file_name_cleanup:w
\__file_name_end:
\__file_name_ext_check:n
\__file_name_ext_check:nw
\__file_name_ext_check:nnw
\__file_name_ext_check:nn
13247 \cs_new:Npn \file_full_name:n #1
13248 {
13249 \exp_args:Ne \__file_full_name:n
13250 { \__kernel_file_name_sanitiz:n {#1} }
13251 }

```

First, we check if the file is just here: no mapping so we do not need the break part of the broader auxiliary. We are using the fact that the primitive here returns nothing if the file is entirely absent. For package mode, `\input@path` is a token list not a sequence.

```

13252 \cs_new:Npn \__file_full_name:n #1
13253 {
13254 \tl_if_blank:nF {#1}
13255 {
13256 \tl_if_blank:eTF { \__file_size:n {#1} }

```

```

13257     {
13258         \seq_map_tokens:Nn \l_file_search_path_seq
13259         { \__file_full_name_aux:Nnn \seq_map_break:n {#1} }
13260         \cs_if_exist:NT \input@path
13261         {
13262             \tl_map_tokens:Nn \input@path
13263             { \__file_full_name_aux:Nnn \tl_map_break:n {#1} }
13264         }
13265         \__file_name_end:
13266     }
13267     { \__file_ext_check:n {#1} }
13268 }
13269 }

```

Two pars to the auxiliary here so we can avoid doing quoting twice in the event we find the right file.

```

13270 \cs_new:Npn \__file_full_name_aux:Nnn #1#2#3
13271 { \exp_args:Ne \__file_full_name_aux:nN { \tl_to_str:n {#3} / #2 } #1 }
13272 \cs_new:Npn \__file_full_name_aux:nN #1 #2
13273 {
13274     \tl_if_blank:eF { \__file_size:n {#1} }
13275     {
13276         #2
13277         {
13278             \__file_ext_check:n {#1}
13279             \__file_name_cleanup:w
13280         }
13281     }
13282 }
13283 \cs_new:Npn \__file_name_cleanup:w #1 \__file_name_end: { }
13284 \cs_new:Npn \__file_name_end: { }

```

As T_EX automatically adds .tex if there is no extension, there is a little clean up to do here. First, make sure we are not in the directory part, saving that. Then check for an extension.

```

13285 \cs_new:Npn \__file_ext_check:n #1
13286 { \__file_ext_check:nw { / } #1 / \q__file_nil / \s__file_stop }
13287 \cs_new:Npn \__file_ext_check:nw #1 #2 / #3 / #4 \s__file_stop
13288 {
13289     \__file_quark_if_nil:nTF {#3}
13290     {
13291         \exp_args:No \__file_ext_check:nnw
13292         { \use_none:n #1 } {#2} #2 . \q__file_nil . \s__file_stop
13293     }
13294     { \__file_ext_check:nw { #1 #2 / } #3 / #4 \s__file_stop }
13295 }
13296 \cs_new:Npx \__file_ext_check:nnw #1#2#3 . #4 . #5 \s__file_stop
13297 {
13298     \exp_not:N \__file_quark_if_nil:nTF {#4}
13299     {
13300         \exp_not:N \__file_ext_check:nn
13301         { #1 #2 } { #1 #2 \tl_to_str:n { .tex } }
13302     }
13303     { #1 #2 }
13304 }

```

```

13305 \cs_new:Npn \__file_ext_check:nn #1#2
13306 {
13307   \tl_if_blank:ETF { \__file_size:n {#2} }
13308   {#1}
13309   {
13310     \int_compare:nNnTF
13311     { \__file_size:n {#1} } = { \__file_size:n {#2} }
13312     {#2}
13313     {#1}
13314   }
13315 }

```

Deal with the fact that the primitive might not be available.

```

13316 \bool_lazy_or:nnF
13317 { \cs_if_exist_p:N \tex_filesize:D }
13318 { \sys_if_engine luatex_p: }
13319 {
13320   \cs_gset:Npn \file_full_name:n #1
13321   {
13322     \__kernel_msg_expandable_error:nnn
13323     { kernel } { primitive-not-available }
13324     { \pdffilesize }
13325   }
13326 }
13327 \__kernel_msg_new:nnnn { kernel } { primitive-not-available }
13328 { Primitive~\token_to_str:N #1 not-available }
13329 {
13330   The-version-of-your-TeX-engine-does-not~provide-functionality-equivalent-to-
13331   the-#1~primitive.
13332 }

```

(End definition for \file_full_name:n and others. This function is documented on page 167.)

```

\file_get_full_name:nN These functions pre-date using \tex_filesize:D for file searching, so are get functions
\file_get_full_name:VN with protection. To avoid having different search set ups, they are simply wrappers
\file_get_full_name:nNTF around the code above.
\file_get_full_name:VNTF
  \__file_get_full_name_search:nN
13333 \cs_new_protected:Npn \file_get_full_name:nN #1#2
13334 {
13335   \file_get_full_name:nNF {#1} #2
13336   { \tl_set:Nn #2 { \q_no_value } }
13337 }
13338 \cs_generate_variant:Nn \file_get_full_name:nN { V }
13339 \prg_new_protected_conditional:Npnn \file_get_full_name:nN #1#2 { T , F , TF }
13340 {
13341   \tl_set:Nx #2
13342   { \file_full_name:n {#1} }
13343   \tl_if_empty:NTF #2
13344   { \prg_return_false: }
13345   { \prg_return_true: }
13346 }
13347 \cs_generate_variant:Nn \file_get_full_name:nNT { V }
13348 \cs_generate_variant:Nn \file_get_full_name:nNF { V }
13349 \cs_generate_variant:Nn \file_get_full_name:nNTF { V }

```

If `\tex_filesize:D` is not available, the way to test if a file exists is to try to open it: if it does not exist then `TEX` reports end-of-file. A search is made looking at each potential path in turn (starting from the current directory). The first location is of course treated as the correct one: this is done by jumping to `\prg_break_point:`. If nothing is found, `#2` is returned empty. A special case when there is no extension is that once the first location is found we test the existence of the file with `.tex` extension in that directory, and if it exists we include the `.tex` extension in the result.

```

13350 \bool_lazy_or:nnF
13351 { \cs_if_exist_p:N \tex_filesize:D }
13352 { \sys_if_engine luatex_p: }
13353 {
13354   \prg_set_protected_conditional:Npnn \file_get_full_name:nN #1#2 { T , F , TF }
13355   {
13356     \tl_set:Nx \l__file_base_name_tl
13357       { \__kernel_file_name_sanitiz:n {#1} }
13358     \__file_get_full_name_search:nN { } \use:n
13359     \seq_map_inline:Nn \l_file_search_path_seq
13360       { \__file_get_full_name_search:nN { ##1 / } \seq_map_break:n }
13361     \cs_if_exist:NT \input@path
13362     {
13363       \tl_map_inline:Nn \input@path
13364         { \__file_get_full_name_search:nN { ##1 } \tl_map_break:n }
13365     }
13366     \tl_set:Nn \l__file_full_name_tl { \q_no_value }
13367     \prg_break_point:
13368     \quark_if_no_value:NTF \l__file_full_name_tl
13369     {
13370       \ior_close:N \g__file_internal_ior
13371       \prg_return_false:
13372     }
13373     {
13374       \file_parse_full_name:VNNN \l__file_full_name_tl
13375       \l__file_dir_str \l__file_name_str \l__file_ext_str
13376       \str_if_empty:NT \l__file_ext_str
13377       {
13378         \__kernel_ior_open:No \g__file_internal_ior
13379         { \l__file_full_name_tl .tex }
13380         \ior_if_eof:NF \g__file_internal_ior
13381         { \tl_put_right:Nn \l__file_full_name_tl { .tex } }
13382       }
13383       \ior_close:N \g__file_internal_ior
13384       \tl_set_eq:NN #2 \l__file_full_name_tl
13385       \prg_return_true:
13386     }
13387   }
13388 }
13389 \cs_new_protected:Npn \__file_get_full_name_search:nN #1#2
13390 {
13391   \tl_set:Nx \l__file_full_name_tl
13392     { \tl_to_str:n {#1} \l__file_base_name_tl }
13393   \__kernel_ior_open:No \g__file_internal_ior \l__file_full_name_tl
13394   \ior_if_eof:NF \g__file_internal_ior { #2 { \prg_break: } }
13395 }

```

(End definition for `\file_get_full_name:nN`, `\file_get_full_name:nNTF`, and `__file_get_full_name_search:nN`. These functions are documented on page 167.)

`\g__file_internal_ior` A reserved stream to test for file existence, if required.

```
13396 \bool_lazy_or:nnF
13397   { \cs_if_exist_p:N \tex_filesize:D }
13398   { \sys_if_engine luatex_p: }
13399   { \ior_new:N \g__file_internal_ior }
```

(End definition for `\g__file_internal_ior`.)

`\file_md5five_hash:n` Getting file details by expansion is relatively easy if a bit repetitive. As the MD5 function has a slightly different syntax from the other commands, there is a little cleaning up to do.

```
\file_size:n
\file_timestamp:n
\__file_details:nn
\__file_details_aux:nn
\__file_md5five_hash:n
13400 \cs_new:Npn \file_md5five_hash:n #1
13401   { \__file_details:nn {#1} { md5sum } }
13402 \cs_new:Npn \file_size:n #1
13403   { \__file_details:nn {#1} { size } }
13404 \cs_new:Npn \file_timestamp:n #1
13405   { \__file_details:nn {#1} { moddate } }
13406 \cs_new:Npn \__file_details:nn #1#2
13407   {
13408     \exp_args:Ne \__file_details_aux:nn
13409       { \file_full_name:n {#1} } {#2}
13410   }
13411 \cs_new:Npn \__file_details_aux:nn #1#2
13412   {
13413     \tl_if_blank:nF {#1}
13414       { \use:c { tex_file #2 :D } {#1} }
13415   }
13416 \sys_if_engine luatex:TF
13417   {
13418     \cs_gset:Npn \__file_details_aux:nn #1#2
13419       {
13420         \lua_now:e
13421           { l3kernel.file#2 ( " \lua_escape:e { #1 } " ) }
13422       }
13423   }
13424   {
13425     \cs_gset:Npn \file_md5five_hash:n #1
13426       { \exp_args:Ne \__file_md5five_hash:n { \file_full_name:n {#1} } }
13427     \cs_new:Npn \__file_md5five_hash:n #1
13428       { \tex_md5sum:D file {#1} }
13429   }
```

(End definition for `\file_md5five_hash:n` and others. These functions are documented on page 169.)

`\file_hex_dump:nnn` These are separate as they need multiple arguments *or* the file size. For LuaTeX, the emulation does not need the file size so we save a little on expansion.

```
\__file_hex_dump_auxi:nnn
\__file_hex_dump_auxii:nnnn
\__file_hex_dump_auxiii:nnnn
\__file_hex_dump_auxiiv:nnn
\file_hex_dump:n
\__file_hex_dump:n
13430 \cs_new:Npn \file_hex_dump:nnn #1#2#3
13431   {
13432     \exp_args:Neee \__file_hex_dump_auxi:nnn
13433       { \file_full_name:n {#1} }
13434       { \int_eval:n {#2} }
```

```

13435     { \int_eval:n {#3} }
13436   }
13437 \cs_new:Npn \__file_hex_dump_auxi:nnn #1#2#3
13438 {
13439   \bool_lazy_any:nF
13440   {
13441     { \tl_if_blank_p:n {#1} }
13442     { \int_compare_p:nNn {#2} = 0 }
13443     { \int_compare_p:nNn {#3} = 0 }
13444   }
13445   {
13446     \exp_args:Ne \__file_hex_dump_auxii:nnnn
13447       { \__file_details_aux:nn {#1} { size } }
13448       {#1} {#2} {#3}
13449   }
13450 }
13451 \cs_new:Npn \__file_hex_dump_auxii:nnnn #1#2#3#4
13452 {
13453   \int_compare:nNnTF {#3} > 0
13454   { \__file_hex_dump_auxiii:nnnn {#3} }
13455   {
13456     \exp_args:Ne \__file_hex_dump_auxiii:nnnn
13457       { \int_eval:n { #1 + #3 } }
13458   }
13459   {#1} {#2} {#4}
13460 }
13461 \cs_new:Npn \__file_hex_dump_auxiii:nnnn #1#2#3#4
13462 {
13463   \int_compare:nNnTF {#4} > 0
13464   { \__file_hex_dump_auxiv:nnn {#4} }
13465   {
13466     \exp_args:Ne \__file_hex_dump_auxiv:nnn
13467       { \int_eval:n { #2 + #4 } }
13468   }
13469   {#1} {#3}
13470 }
13471 \cs_new:Npn \__file_hex_dump_auxiv:nnn #1#2#3
13472 {
13473   \tex_filedump:D
13474   offset ~ \int_eval:n { #2 - 1 } ~
13475   length ~ \int_eval:n { #1 - #2 + 1 }
13476   {#3}
13477 }
13478 \sys_if_engine luatex:T
13479 {
13480   \cs_gset:Npn \__file_hex_dump_auxiv:nnn #1#2#3
13481   {
13482     \lua_now:e
13483     {
13484       l3kernel.filedump
13485       (
13486         " \lua_escape:e {#3} " ,
13487         \int_eval:n { #2 - 1 } ,
13488         \int_eval:n { #1 - #2 + 1 }

```

```

13489         )
13490     }
13491 }
13492 }
13493 \cs_new:Npn \file_hex_dump:n #1
13494 { \exp_args:Ne \__file_hex_dump:n { \file_full_name:n {#1} } }
13495 \cs_new:Npn \__file_hex_dump:n #1
13496 {
13497     \tl_if_blank:nF {#1}
13498     { \tex_dump:D length \tex_filesize:D {#1} {#1} }
13499 }
13500 \sys_if_engine luatex:T
13501 {
13502     \cs_gset:Npn \__file_hex_dump:n #1
13503     {
13504         \lua_now:e
13505         { l3kernel.filedump ( " \lua_escape:e { #1 } " ) }
13506     }
13507 }

```

(End definition for `\file_hex_dump:nnn` and others. These functions are documented on page 168.)

```

\file_get_hex_dump:nN Non-expandable wrappers around the above in the case where appropriate primitive
\file_get_hex_dump:nNTF support exists.
\file_get_md5fif_hash:nN\file_get_size:nN
\file_get_md5fif_hash:nNTF\file_get_size:nNTF
\file_get_timestamp:nN
\file_get_timestamp:nNTF
\__file_get_details:nnN
13508 \cs_new_protected:Npn \file_get_hex_dump:nN #1#2
13509 { \file_get_hex_dump:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
13510 \cs_new_protected:Npn \file_get_md5fif_hash:nN #1#2
13511 { \file_get_md5fif_hash:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
13512 \cs_new_protected:Npn \file_get_size:nN #1#2
13513 { \file_get_size:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
13514 \cs_new_protected:Npn \file_get_timestamp:nN #1#2
13515 { \file_get_timestamp:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
13516 \prg_new_protected_conditional:Npnn \file_get_hex_dump:nN #1#2 { T , F , TF }
13517 { \__file_get_details:nnN {#1} { hex_dump } #2 }
13518 \prg_new_protected_conditional:Npnn \file_get_md5fif_hash:nN #1#2 { T , F , TF }
13519 { \__file_get_details:nnN {#1} { md5fif_hash } #2 }
13520 \prg_new_protected_conditional:Npnn \file_get_size:nN #1#2 { T , F , TF }
13521 { \__file_get_details:nnN {#1} { size } #2 }
13522 \prg_new_protected_conditional:Npnn \file_get_timestamp:nN #1#2 { T , F , TF }
13523 { \__file_get_details:nnN {#1} { timestamp } #2 }
13524 \cs_new_protected:Npn \__file_get_details:nnN #1#2#3
13525 {
13526     \tl_set:Nx #3
13527     { \use:c { file_ #2 :n } {#1} }
13528     \tl_if_empty:NTF #3
13529     { \prg_return_false: }
13530     { \prg_return_true: }
13531 }

```

Where the primitive is not available, issue an error: this is a little more conservative than absolutely needed, but does work.

```

13532 \bool_lazy_or:nnF
13533 { \cs_if_exist_p:N \tex_filesize:D }
13534 { \sys_if_engine luatex_p: }
13535 {

```

```

13536 \cs_set_protected:Npn \__file_get_details:nnN #1#2#3
13537 {
13538   \tl_clear:N #3
13539   \__kernel_msg_error:nnx
13540   { kernel } { primitive-not-available }
13541   {
13542     \token_to_str:N \(\pdf)file
13543     \str_case:nn {#2}
13544     {
13545       { hex_dump } { dump }
13546       { mdfive_hash } { mdfivesum }
13547       { timestamp } { moddate }
13548       { size } { size }
13549     }
13550   }
13551   \prg_return_false:
13552 }
13553 }

```

(End definition for `\file_get_hex_dump:nnTF` and others. These functions are documented on page 168.)

`\file_get_hex_dump:nnnN`
`\file_get_hex_dump:nnnN`TF

Custom code due to the additional arguments.

```

13554 \cs_new_protected:Npn \file_get_hex_dump:nnnN #1#2#3#4
13555 {
13556   \file_get_hex_dump:nnnNF {#1} {#2} {#3} #4
13557   { \tl_set:Nn #4 { \q_no_value } }
13558 }
13559 \prg_new_protected_conditional:Npnn \file_get_hex_dump:nnnN #1#2#3#4
13560 { T , F , TF }
13561 {
13562   \tl_set:Nx #4
13563   { \file_hex_dump:nnn {#1} {#2} {#3} }
13564   \tl_if_empty:NTF #4
13565   { \prg_return_false: }
13566   { \prg_return_true: }
13567 }

```

(End definition for `\file_get_hex_dump:nnnN`. This function is documented on page 168.)

`__file_str_cmp:nn`
`__file_str_escape:n`

As we are doing a fixed-length “big” integer comparison, it is easiest to use the low-level behavior of string comparisons.

```

13568 \cs_new:Npn \__file_str_cmp:nn #1#2 { \tex_strcmp:D {#1} {#2} }
13569 \sys_if_engine luatex:T
13570 {
13571   \cs_set:Npn \__file_str_cmp:nn #1#2
13572   {
13573     \lua_now:e
13574     {
13575       l3kernel_strcmp
13576       (
13577         " \__file_str_escape:n {#1}",
13578         " \__file_str_escape:n {#2}"
13579       )
13580     }

```



```

13581     }
13582     \cs_new:Npn \__file_str_escape:n #1
13583     {
13584         \lua_escape:e
13585         { \__kernel_tl_to_str:w \use:e { {#1} } }
13586     }
13587 }

```

(End definition for __file_str_cmp:nn and __file_str_escape:n.)

`\file_compare_timestamp_p:nNn` Comparison of file date can be done by using the low-level nature of the string comparison functions.

`\file_compare_timestamp:nNnTF`
`__file_compare_timestamp:nnN`
`__file_timestamp:n`

```

13588 \prg_new_conditional:Npnn \file_compare_timestamp:nNn #1#2#3
13589 { p , T , F , TF }
13590 {
13591     \exp_args:Nee \__file_compare_timestamp:nnN
13592     { \file_full_name:n {#1} }
13593     { \file_full_name:n {#3} }
13594     #2
13595 }
13596 \cs_new:Npn \__file_compare_timestamp:nnN #1#2#3
13597 {
13598     \tl_if_blank:nTF {#1}
13599     {
13600         \if_charcode:w #3 <
13601         \prg_return_true:
13602     \else:
13603         \prg_return_false:
13604     \fi:
13605 }
13606 {
13607     \tl_if_blank:nTF {#2}
13608     {
13609         \if_charcode:w #3 >
13610         \prg_return_true:
13611     \else:
13612         \prg_return_false:
13613     \fi:
13614 }
13615 {
13616     \if_int_compare:w
13617     \__file_str_cmp:nn
13618     { \__file_timestamp:n {#1} }
13619     { \__file_timestamp:n {#2} }
13620     #3 0 \exp_stop_f:
13621     \prg_return_true:
13622     \else:
13623     \prg_return_false:
13624     \fi:
13625 }
13626 }
13627 }
13628 \sys_if_engine luatex:TF
13629 {

```

```

13630 \cs_new:Npn \__file_timestamp:n #1
13631 {
13632   \lua_now:e
13633   { l3kernel.filemoddate ( " \lua_escape:e {#1} " ) }
13634 }
13635 }
13636 { \cs_new_eq:NN \__file_timestamp:n \tex_filemoddate:D }
13637 \cs_if_exist:NF \tex_filemoddate:D
13638 {
13639   \prg_set_conditional:Npnn \file_compare_timestamp:nNn #1#2#3
13640   { p , T , F , TF }
13641   {
13642     \__kernel_msg_expandable_error:nnn
13643     { kernel } { primitive-not-available }
13644     { \pdffilemoddate }
13645     \prg_return_false:
13646   }
13647 }

```

(End definition for `\file_compare_timestamp:nNnTF`, `__file_compare_timestamp:nnN`, and `__file_timestamp:n`. This function is documented on page 170.)

`\file_if_exist:nTF` The test for the existence of a file is a wrapper around the function to add a path to a file. If the file was found, the path contains something, whereas if the file was not located then the return value is empty.

```

13648 \prg_new_protected_conditional:Npnn \file_if_exist:n #1 { T , F , TF }
13649 {
13650   \file_get_full_name:nNTF {#1} \l__file_full_name_tl
13651   { \prg_return_true: }
13652   { \prg_return_false: }
13653 }

```

(End definition for `\file_if_exist:nTF`. This function is documented on page 167.)

`\file_if_exist_input:n` Input of a file with a test for existence. We do not define the T or TF variants because the most useful place to place the *<true code>* would be inconsistent with other conditionals.

`\file_if_exist_input:nF`

```

13654 \cs_new_protected:Npn \file_if_exist_input:n #1
13655 {
13656   \file_get_full_name:nNT {#1} \l__file_full_name_tl
13657   { \__file_input:V \l__file_full_name_tl }
13658 }
13659 \cs_new_protected:Npn \file_if_exist_input:nF #1#2
13660 {
13661   \file_get_full_name:nNTF {#1} \l__file_full_name_tl
13662   { \__file_input:V \l__file_full_name_tl }
13663   {#2}
13664 }

```

(End definition for `\file_if_exist_input:n` and `\file_if_exist_input:nF`. These functions are documented on page 170.)

`\file_input_stop:` A simple rename.

```

13665 \cs_new_protected:Npn \file_input_stop: { \tex_endinput:D }

```

(End definition for `\file_input_stop:`. This function is documented on page 170.)

_kernel_file_missing:n An error message for a missing file, also used in \ior_open:Nn.

```

13666 \cs_new_protected:Npn \_kernel\_file\_missing:n #1
13667 {
13668   \_kernel\_msg\_error:nnx { kernel } { file-not-found }
13669   { \_kernel\_file\_name\_sanitize:n {#1} }
13670 }

```

(End definition for _kernel_file_missing:n.)

\file_input:n Loading a file is done in a safe way, checking first that the file exists and loading only if it does. Push the file name on the \g_file_stack_seq, and add it to the file list, either \g_file_record_seq, or \@filelist in package mode.

_file_input:n
_file_input:V

```

\_file\_input\_push:n 13671 \cs_new_protected:Npn \file\_input:n #1
\_kernel\_file\_input\_push:n 13672 {
  \_file\_input\_pop: 13673   \file\_get\_full\_name:nNTF {#1} \l\_file\_full\_name\_tl
\_kernel\_file\_input\_pop: 13674   { \_file\_input:V \l\_file\_full\_name\_tl }
\_file\_input\_pop:nnn 13675   { \_kernel\_file\_missing:n {#1} }
13676 }
13677 \cs_new_protected:Npx \_file\_input:n #1
13678 {
13679   \exp\_not:N \clist\_if\_exist:NTF \exp\_not:N \@filelist
13680   { \exp\_not:N \@addtofilelist {#1} }
13681   { \seq\_gput\_right:Nn \exp\_not:N \g\_file\_record\_seq {#1} }
13682   \exp\_not:N \_file\_input\_push:n {#1}
13683   \exp\_not:N \tex\_input:D
13684   \sys\_if\_engine\_luatex:TF
13685   { {#1} }
13686   { \exp\_not:N \_kernel\_file\_name\_quote:n {#1} \scan\_stop: }
13687   \exp\_not:N \_file\_input\_pop:
13688 }
13689 \cs_generate\_variant:Nn \_file\_input:n { V }

```

Keeping a track of the file data is easy enough: we store the separated parts so we do not need to parse them twice.

```

13690 \cs_new_protected:Npn \_file\_input\_push:n #1
13691 {
13692   \seq\_gpush:Nx \g\_file\_stack\_seq
13693   {
13694     { \g\_file\_curr\_dir\_str }
13695     { \g\_file\_curr\_name\_str }
13696     { \g\_file\_curr\_ext\_str }
13697   }
13698   \file\_parse\_full\_name:nNNN {#1}
13699   \l\_file\_dir\_str \l\_file\_name\_str \l\_file\_ext\_str
13700   \str\_gset\_eq:NN \g\_file\_curr\_dir\_str \l\_file\_dir\_str
13701   \str\_gset\_eq:NN \g\_file\_curr\_name\_str \l\_file\_name\_str
13702   \str\_gset\_eq:NN \g\_file\_curr\_ext\_str \l\_file\_ext\_str
13703 }
13704 \cs_new\_eq:NN \_kernel\_file\_input\_push:n \_file\_input\_push:n
13705 \cs_new_protected:Npn \_file\_input\_pop:
13706 {
13707   \seq\_gpop:NN \g\_file\_stack\_seq \l\_file\_internal\_tl
13708   \exp\_after:wN \_file\_input\_pop:nnn \l\_file\_internal\_tl
13709 }

```

```

13710 \cs_new_eq:NN \__kernel_file_input_pop: \__file_input_pop:
13711 \cs_new_protected:Npn \__file_input_pop:nnn #1#2#3
13712 {
13713   \str_gset:Nn \g_file_curr_dir_str {#1}
13714   \str_gset:Nn \g_file_curr_name_str {#2}
13715   \str_gset:Nn \g_file_curr_ext_str {#3}
13716 }

```

(End definition for \file_input:n and others. This function is documented on page 170.)

\file_parse_full_name:n The main parsing macro \file_parse_full_name_apply:nN passes the file name #1 through __kernel_file_name_sanitiz:n so that we have a single normalised way to treat files internally. \file_parse_full_name:n uses the former, with \prg_do_nothing: to leave each part of the name within a pair of braces.

```

13717 \cs_new:Npn \file_parse_full_name:n #1
13718 {
13719   \file_parse_full_name_apply:nN {#1}
13720   \prg_do_nothing:
13721 }
13722 \cs_new:Npn \file_parse_full_name_apply:nN #1
13723 {
13724   \exp_args:Ne \__file_parse_full_name_auxi:nN
13725   { \__kernel_file_name_sanitiz:n {#1} }
13726 }

```

_file_parse_full_name_auxi:nN
_file_parse_full_name_area:nw

_file_parse_full_name_area:nw splits the file name into chunks separated by /, until the last one is reached. The last chunk is the file name plus the extension, and everything before that is the path. When _file_parse_full_name_area:nw is done, it leaves the path within braces after the scan mark \s__file_stop and proceeds parsing the actual file name.

```

13727 \cs_new:Npn \__file_parse_full_name_auxi:nN #1
13728 {
13729   \__file_parse_full_name_area:nw { } #1
13730   / \s__file_stop
13731 }
13732 \cs_new:Npn \__file_parse_full_name_area:nw #1 #2 / #3 \s__file_stop
13733 {
13734   \tl_if_empty:nTF {#3}
13735   { \__file_parse_full_name_base:nw { } #2 . \s__file_stop {#1} }
13736   { \__file_parse_full_name_area:nw { #1 / #2 } #3 \s__file_stop }
13737 }

```

_file_parse_full_name_base:nw

_file_parse_full_name_base:nw does roughly the same as above, but it separates the chunks at each period. However here there's some extra complications: In case #1 is empty, it is assumed that the extension is actually empty, and the file name is #2. Besides, an extra . has to be added to #2 because it is later removed in __file_parse_full_name_tidy:nnnN. In any case, if there's an extension, it is returned with a leading ..

```

13738 \cs_new:Npn \__file_parse_full_name_base:nw #1 #2 . #3 \s__file_stop
13739 {
13740   \tl_if_empty:nTF {#3}
13741   {
13742     \tl_if_empty:nTF {#1}

```

```

13743     {
13744         \tl_if_empty:nTF {#2}
13745         { \__file_parse_full_name_tidy:nnnN { } { } }
13746         { \__file_parse_full_name_tidy:nnnN { .#2 } { } }
13747     }
13748     { \__file_parse_full_name_tidy:nnnN {#1} { .#2 } }
13749 }
13750 { \__file_parse_full_name_base:nw { #1 . #2 } #3 \s__file_stop }
13751 }

```

Now we just need to tidy some bits left loose before. The loop used in the two macros above start with a leading / and . in the file path an name, so here we need to remove them, except in the path, if it is a single /, in which case it's left as is. After all's done, pass to #4.

```

13752 \cs_new:Npn \__file_parse_full_name_tidy:nnnN #1 #2 #3 #4
13753 {
13754     \exp_args:Nee #4
13755     {
13756         \str_if_eq:nnF {#3} { / } { \use_none:n }
13757         #3 \prg_do_nothing:
13758     }
13759     { \use_none:n #1 \prg_do_nothing: }
13760     {#2}
13761 }

```

(End definition for \file_parse_full_name:n and others. These functions are documented on page 168.)

\file_parse_full_name:nNNN
\file_parse_full_name:VNNN

```

13762 \cs_new_protected:Npn \file_parse_full_name:nNNN #1 #2 #3 #4
13763 {
13764     \file_parse_full_name_apply:nN {#1}
13765     \__file_full_name_assign:nnnNNN #2 #3 #4
13766 }
13767 \cs_new_protected:Npn \__file_full_name_assign:nnnNNN #1 #2 #3 #4 #5 #6
13768 {
13769     \str_set:Nn #4 {#1}
13770     \str_set:Nn #5 {#2}
13771     \str_set:Nn #6 {#3}
13772 }
13773 \cs_generate_variant:Nn \file_parse_full_name:nNNN { V }

```

(End definition for \file_parse_full_name:nNNN. This function is documented on page 168.)

\file_show_list: A function to list all files used to the log, without duplicates. In package mode, if **\file_log_list:** **\@filelist** is still defined, we need to take this list of file names into account (we capture it **\AtBeginDocument** into **\g__file_record_seq**), turning it to a string (this does not affect the commas of this comma list).
__file_list:N
__file_list_aux:n

```

13774 \cs_new_protected:Npn \file_show_list: { \__file_list:N \msg_show:nnxxxx }
13775 \cs_new_protected:Npn \file_log_list: { \__file_list:N \msg_log:nnxxxx }
13776 \cs_new_protected:Npn \__file_list:N #1
13777 {
13778     \seq_clear:N \l__file_tmp_seq
13779     \clist_if_exist:NT \@filelist

```

```

13780     {
13781       \exp_args:NNx \seq_set_from_clist:Nn \l__file_tmp_seq
13782       { \tl_to_str:N \@filelist }
13783     }
13784     \seq_concat:NNN \l__file_tmp_seq \l__file_tmp_seq \g__file_record_seq
13785     \seq_remove_duplicates:N \l__file_tmp_seq
13786     #1 { LaTeX/kernel } { file-list }
13787     { \seq_map_function:NN \l__file_tmp_seq \__file_list_aux:n }
13788     { } { } { }
13789   }
13790   \cs_new:Npn \__file_list_aux:n #1 { \iow_newline: #1 }

```

(End definition for `\file_show_list:` and others. These functions are documented on page 170.)

When used as a package, there is a need to hold onto the standard file list as well as the new one here. File names recorded in `\@filelist` must be turned to strings before being added to `\g__file_record_seq`.

```

13791   \cs_if_exist:NT \@filelist
13792   {
13793     \AtBeginDocument
13794     {
13795       \exp_args:NNx \seq_set_from_clist:Nn \l__file_tmp_seq
13796       { \tl_to_str:N \@filelist }
13797       \seq_gconcat:NNN
13798       \g__file_record_seq
13799       \g__file_record_seq
13800       \l__file_tmp_seq
13801     }
13802   }

```

20.5 GetIdInfo

\GetIdInfo As documented in `expl3.dtx` this function extracts file name etc from an SVN Id line. This used to be how we got version number and so on in all modules, so it had to be defined in `l3bootstrap`. Now it's more convenient to define it after we have set up quite a lot of tools, and `l3file` seems the least unreasonable place for it.

The idea here is to extract out the information needed from a standard SVN Id line, but to avoid a line that would get changed when the file is checked in. Hence the fact that none of the lines here include both a dollar sign and the Id keyword!

```

13803   \cs_new_protected:Npn \GetIdInfo
13804   {
13805     \tl_clear_new:N \ExplFileDescription
13806     \tl_clear_new:N \ExplFileDate
13807     \tl_clear_new:N \ExplFileName
13808     \tl_clear_new:N \ExplFileExtension
13809     \tl_clear_new:N \ExplFileVersion
13810     \group_begin:
13811     \char_set_catcode_space:n { 32 }
13812     \exp_after:wN
13813     \group_end:
13814     \__file_id_info_auxi:w
13815   }

```

A first check for a completely empty SVN field. If that is not the case, there is a second case when a file created using `svn cp` but has not been checked in. That leaves a special marker `-1` version, which has no further data. Dealing correctly with that is the reason for the space in the line to use `_file_id_info_auxii:w`.

```

13816 \cs_new_protected:Npn \_file_id_info_auxii:w $ #1 $ #2
13817 {
13818   \tl_set:Nn \ExplFileDescription {#2}
13819   \str_if_eq:nnTF {#1} { Id }
13820   {
13821     \tl_set:Nn \ExplFileDate { 0000/00/00 }
13822     \tl_set:Nn \ExplFileName { [unknown] }
13823     \tl_set:Nn \ExplFileExtension { [unknown~extension] }
13824     \tl_set:Nn \ExplFileVersion {-1}
13825   }
13826   { \_file_id_info_auxii:w #1 ~ \s__file_stop }
13827 }

```

Here, `#1` is `Id`, `#2` is the file name, `#3` is the extension, `#4` is the version, `#5` is the check in date and `#6` is the check in time and user, plus some trailing spaces. If `#4` is the marker `-1` value then `#5` and `#6` are empty.

```

13828 \cs_new_protected:Npn \_file_id_info_auxiii:w
13829   #1 ~ #2.#3 ~ #4 ~ #5 ~ #6 \s__file_stop
13830 {
13831   \tl_set:Nn \ExplFileName {#2}
13832   \tl_set:Nn \ExplFileExtension {#3}
13833   \tl_set:Nn \ExplFileVersion {#4}
13834   \str_if_eq:nnTF {#4} {-1}
13835   { \tl_set:Nn \ExplFileDate { 0000/00/00 } }
13836   { \_file_id_info_auxiii:w #5 - 0 - 0 - \s__file_stop }
13837 }

```

Convert an SVN-style date into a L^AT_EX-style one.

```

13838 \cs_new_protected:Npn \_file_id_info_auxiiii:w #1 - #2 - #3 - #4 \s__file_stop
13839 { \tl_set:Nn \ExplFileDate { #1/#2/#3 } }

```

(End definition for `\GetIdInfo` and others. This function is documented on page 7.)

20.6 Messages

```

13840 \__kernel_msg_new:nnnn { kernel } { file-not-found }
13841 { File~'~#1'~not~found. }
13842 {
13843   The~requested~file~could~not~be~found~in~the~current~directory,~
13844   in~the~TeX~search~path~or~in~the~LaTeX~search~path.
13845 }
13846 \__kernel_msg_new:nnn { kernel } { file-list }
13847 {
13848   >~File~List~<
13849   #1 \\
13850   .....
13851 }
13852 \__kernel_msg_new:nnnn { kernel } { unbalanced-quote-in-filename }
13853 { Unbalanced~quotes~in~file~name~'~#1'. }
13854 {

```

```

13855     File~names~must~contain~balanced~numbers~of~quotes~(").
13856   }
13857   \_kernel_msg_new:nnnn { kernel } { iow-indent }
13858   { Only~#1 (arg~1)~allows~#2 }
13859   {
13860     The~command~#2 can~only~be~used~in~messages~
13861     which~will~be~wrapped~using~#1.
13862     \tl_if_empty:nF {#3} { ~ It~was~called~with~argument~'~#3'. }
13863   }

```

20.7 Functions delayed from earlier modules

<@@=sys>

`\c_sys_platform_str` Detecting the platform on LuaTeX is easy: for other engines, we use the fact that the two common cases have special null files. It is possible to probe further (see package `platform`), but that requires shell escape and seems unlikely to be useful. This is set up here as it requires file searching.

```

13864 \sys_if_engine luatex:TF
13865 {
13866   \str_const:Nx \c_sys_platform_str
13867   { \tex_directlua:D { tex.print(os.type) } }
13868 }
13869 {
13870   \file_if_exist:nTF { nul: }
13871   {
13872     \file_if_exist:nF { /dev/null }
13873     { \str_const:Nn \c_sys_platform_str { windows } }
13874   }
13875   {
13876     \file_if_exist:nT { /dev/null }
13877     { \str_const:Nn \c_sys_platform_str { unix } }
13878   }
13879 }
13880 \cs_if_exist:NF \c_sys_platform_str
13881 { \str_const:Nn \c_sys_platform_str { unknown } }

```

(End definition for `\c_sys_platform_str`. This variable is documented on page 117.)

`\sys_if_platform_unix_p:` We can now set up the tests.

```

\sys_if_platform_unix:TF
\sys_if_platform_windows_p:
\sys_if_platform_windows:TF
13882 \clist_map_inline:nn { unix , windows }
13883 {
13884   \__file_const:nn { sys_if_platform_ #1 }
13885   { \str_if_eq_p:Vn \c_sys_platform_str { #1 } }
13886 }

```

(End definition for `\sys_if_platform_unix:TF` and `\sys_if_platform_windows:TF`. These functions are documented on page 117.)

```

13887 \</package>

```


21 l3skip implementation

13888 $\langle *package \rangle$

13889 $\langle @@=dim \rangle$

21.1 Length primitives renamed

$\backslash if_dim:w$ Primitives renamed.
 $\backslash_dim_eval:w$ 13890 $\backslash cs_new_eq:NN \backslash if_dim:w \quad \backslash tex_ifdim:D$
 $\backslash_dim_eval_end:$ 13891 $\backslash cs_new_eq:NN \backslash_dim_eval:w \quad \backslash tex_dimexpr:D$
13892 $\backslash cs_new_eq:NN \backslash_dim_eval_end: \quad \backslash tex_relax:D$

(End definition for $\backslash if_dim:w$, $\backslash_dim_eval:w$, and $\backslash_dim_eval_end:$. This function is documented on page 185.)

21.2 Internal auxiliaries

$\backslash s_dim_mark$ Internal scan marks.
 $\backslash s_dim_stop$ 13893 $\backslash scan_new:N \backslash s_dim_mark$
13894 $\backslash scan_new:N \backslash s_dim_stop$

(End definition for $\backslash s_dim_mark$ and $\backslash s_dim_stop$.)

$\backslash_dim_use_none_delimit_by_s_stop:w$ Functions to gobble up to a scan mark.
13895 $\backslash cs_new:Npn \backslash_dim_use_none_delimit_by_s_stop:w \#1 \backslash s_dim_stop \{ \}$
(End definition for $\backslash_dim_use_none_delimit_by_s_stop:w$.)

21.3 Creating and initialising dim variables

$\backslash dim_new:N$ Allocating $\langle dim \rangle$ registers ...
 $\backslash dim_new:c$ 13896 $\backslash cs_new_protected:Npn \backslash dim_new:N \#1$
13897 $\{$
13898 $\quad \backslash_kernel_chk_if_free_cs:N \#1$
13899 $\quad \backslash cs:w newdimen \backslash cs_end: \#1$
13900 $\}$
13901 $\backslash cs_generate_variant:Nn \backslash dim_new:N \{ c \}$
(End definition for $\backslash dim_new:N$. This function is documented on page 171.)

 $\backslash dim_const:Nn$ Contrarily to integer constants, we cannot avoid using a register, even for constants. We
 $\backslash dim_const:cn$ cannot use $\backslash dim_gset:Nn$ because debugging code would complain that the constant is not a global variable. Since $\backslash dim_const:Nn$ does not need to be fast, use $\backslash dim_eval:n$ to avoid needing a debugging patch that wraps the expression in checking code.
13902 $\backslash cs_new_protected:Npn \backslash dim_const:Nn \#1\#2$
13903 $\{$
13904 $\quad \backslash dim_new:N \#1$
13905 $\quad \backslash tex_global:D \#1 \sim \backslash dim_eval:n \{ \#2 \} \backslash scan_stop:$
13906 $\}$
13907 $\backslash cs_generate_variant:Nn \backslash dim_const:Nn \{ c \}$

(End definition for $\backslash dim_const:Nn$. This function is documented on page 171.)

\dim_zero:N Reset the register to zero. Using `\c_zero_skip` deals with the case where the variable passed is incorrectly a skip (for example a $\text{\LaTeX} 2_{\epsilon}$ length).

```

\dim_zero:c
\dim_gzero:N 13908 \cs_new_protected:Npn \dim_zero:N #1 { #1 \c_zero_skip }
\dim_gzero:c 13909 \cs_new_protected:Npn \dim_gzero:N #1
               { \tex_global:D #1 \c_zero_skip }
               13910
               13911 \cs_generate_variant:Nn \dim_zero:N { c }
               13912 \cs_generate_variant:Nn \dim_gzero:N { c }

```

(End definition for `\dim_zero:N` and `\dim_gzero:N`. These functions are documented on page 171.)

\dim_zero_new:N Create a register if needed, otherwise clear it.

```

\dim_zero_new:c 13913 \cs_new_protected:Npn \dim_zero_new:N #1
\dim_gzero_new:N 13914 { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
\dim_gzero_new:c 13915 \cs_new_protected:Npn \dim_gzero_new:N #1
               13916 { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
               13917 \cs_generate_variant:Nn \dim_zero_new:N { c }
               13918 \cs_generate_variant:Nn \dim_gzero_new:N { c }

```

(End definition for `\dim_zero_new:N` and `\dim_gzero_new:N`. These functions are documented on page 171.)

\dim_if_exist_p:N Copies of the `cs` functions defined in `l3basics`.

```

\dim_if_exist_p:c 13919 \prg_new_eq_conditional:Nnn \dim_if_exist:N \cs_if_exist:N
\dim_if_exist:NTF 13920 { TF , T , F , p }
\dim_if_exist:cTF 13921 \prg_new_eq_conditional:Nnn \dim_if_exist:c \cs_if_exist:c
                  13922 { TF , T , F , p }

```

(End definition for `\dim_if_exist:NTF`. This function is documented on page 171.)

21.4 Setting dim variables

\dim_set:Nn Setting dimensions is easy enough but when debugging we want both to check that the variable is correctly local/global and to wrap the expression in some code. The `\scan_stop:` deals with the case where the variable passed is a skip (for example a $\text{\LaTeX} 2_{\epsilon}$ length).

```

13923 \cs_new_protected:Npn \dim_set:Nn #1#2
13924 { #1 ~ \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
13925 \cs_new_protected:Npn \dim_gset:Nn #1#2
13926 { \tex_global:D #1 ~ \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
13927 \cs_generate_variant:Nn \dim_set:Nn { c }
13928 \cs_generate_variant:Nn \dim_gset:Nn { c }

```

(End definition for `\dim_set:Nn` and `\dim_gset:Nn`. These functions are documented on page 172.)

\dim_set_eq:NN All straightforward, with a `\scan_stop:` to deal with the case where `#1` is (incorrectly) a skip.

```

\dim_set_eq:cN 13929 \cs_new_protected:Npn \dim_set_eq:NN #1#2
\dim_set_eq:Nc 13930 { #1 = #2 \scan_stop: }
\dim_set_eq:cc 13931 \cs_generate_variant:Nn \dim_set_eq:NN { c , Nc , cc }
\dim_gset_eq:NN 13932 \cs_new_protected:Npn \dim_gset_eq:NN #1#2
\dim_gset_eq:cN 13933 { \tex_global:D #1 = #2 \scan_stop: }
\dim_gset_eq:Nc 13934 \cs_generate_variant:Nn \dim_gset_eq:NN { c , Nc , cc }
\dim_gset_eq:cc

```

(End definition for `\dim_set_eq:NN` and `\dim_gset_eq:NN`. These functions are documented on page 172.)

`\dim_add:Nn` Using by here deals with the (incorrect) case `\dimen123`. Using `\scan_stop:` deals with skip variables. Since debugging checks that the variable is correctly local/global, the global versions cannot be defined as `\tex_global:D` followed by the local versions. The debugging code is inserted by `__dim_tmp:w`.

```

\dim_add:cn
\dim_gadd:Nn
\dim_gadd:cn
\dim_sub:Nn
\dim_sub:cn
\dim_gsub:Nn
\dim_gsub:cn
13935 \cs_new_protected:Npn \dim_add:Nn #1#2
13936 { \tex_advance:D #1 by \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
13937 \cs_new_protected:Npn \dim_gadd:Nn #1#2
13938 {
13939   \tex_global:D \tex_advance:D #1 by
13940   \__dim_eval:w #2 \__dim_eval_end: \scan_stop:
13941 }
13942 \cs_generate_variant:Nn \dim_add:Nn { c }
13943 \cs_generate_variant:Nn \dim_gadd:Nn { c }
13944 \cs_new_protected:Npn \dim_sub:Nn #1#2
13945 { \tex_advance:D #1 by - \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
13946 \cs_new_protected:Npn \dim_gsub:Nn #1#2
13947 {
13948   \tex_global:D \tex_advance:D #1 by
13949   - \__dim_eval:w #2 \__dim_eval_end: \scan_stop:
13950 }
13951 \cs_generate_variant:Nn \dim_sub:Nn { c }
13952 \cs_generate_variant:Nn \dim_gsub:Nn { c }

```

(End definition for `\dim_add:Nn` and others. These functions are documented on page 172.)

21.5 Utilities for dimension calculations

`\dim_abs:n` Functions for min, max, and absolute value with only one evaluation. The absolute value is evaluated by removing a leading `-` if present.

```

\__dim_abs:N
\dim_max:nn
\dim_min:nn
\__dim_maxmin:wwN
13953 \cs_new:Npn \dim_abs:n #1
13954 {
13955   \exp_after:wN \__dim_abs:N
13956   \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
13957 }
13958 \cs_new:Npn \__dim_abs:N #1
13959 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
13960 \cs_new:Npn \dim_max:nn #1#2
13961 {
13962   \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
13963   \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
13964   \dim_use:N \__dim_eval:w #2 ;
13965   >
13966   \__dim_eval_end:
13967 }
13968 \cs_new:Npn \dim_min:nn #1#2
13969 {
13970   \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
13971   \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
13972   \dim_use:N \__dim_eval:w #2 ;
13973   <
13974   \__dim_eval_end:
13975 }
13976 \cs_new:Npn \__dim_maxmin:wwN #1 ; #2 ; #3

```

```

13977 {
13978   \if_dim:w #1 #3 #2 ~
13979     #1
13980   \else:
13981     #2
13982   \fi:
13983 }

```

(End definition for `\dim_abs:n` and others. These functions are documented on page 172.)

`\dim_ratio:nn` With dimension expressions, something like `10 pt * (5 pt / 10 pt)` does not work. Instead, the ratio part needs to be converted to an integer expression. Using `\int_value:w` forces everything into `sp`, avoiding any decimal parts.

```

13984 \cs_new:Npn \dim_ratio:nn #1#2
13985 { \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
13986 \cs_new:Npn \__dim_ratio:n #1
13987 { \int_value:w \__dim_eval:w (#1) \__dim_eval_end: }

```

(End definition for `\dim_ratio:nn` and `__dim_ratio:n`. This function is documented on page 173.)

21.6 Dimension expression conditionals

`\dim_compare_p:nNn` Simple comparison.

```

\dim_compare:nNnTF
13988 \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
13989 {
13990   \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
13991     \prg_return_true: \else: \prg_return_false: \fi:
13992 }

```

(End definition for `\dim_compare:nNnTF`. This function is documented on page 173.)

`\dim_compare_p:n` This code is adapted from the `\int_compare:nTF` function. First make sure that there is at least one relation operator, by evaluating a dimension expression with a trailing `__dim_compare_error:.` Just like for integers, the looping auxiliary `__dim_compare:wNN` closes a primitive conditional and opens a new one. It is actually easier to grab a dimension operand than an integer one, because once evaluated, dimensions all end with `pt` (with category other). Thus we do not need specific auxiliaries for the three “simple” relations `<`, `=`, and `>`.

```

\dim_compare:nTF
\__dim_compare:w
\__dim_compare:wNN
\__dim_compare:=:w
\__dim_compare_!=:w
\__dim_compare_<:w
\__dim_compare_>:w
\__dim_compare_error:
13993 \prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
13994 {
13995   \exp_after:wN \__dim_compare:w
13996   \dim_use:N \__dim_eval:w #1 \__dim_compare_error:
13997 }
13998 \cs_new:Npn \__dim_compare:w #1 \__dim_compare_error:
13999 {
14000   \exp_after:wN \if_false: \exp:w \exp_end_continue_f:w
14001   \__dim_compare:wNN #1 ? { = \__dim_compare_end:w \else: } \s__dim_stop
14002 }
14003 \exp_args:Nno \use:nn
14004 { \cs_new:Npn \__dim_compare:wNN #1 } { \tl_to_str:n {pt} #2#3 }
14005 {
14006   \if_meaning:w = #3
14007     \use:c { __dim_compare_#2:w }
14008   \fi:

```

```

14009         #1 pt \exp_stop_f:
14010         \prg_return_false:
14011         \exp_after:wN \__dim_use_none_delimit_by_s_stop:w
14012         \fi:
14013         \reverse_if:N \if_dim:w #1 pt #2
14014         \exp_after:wN \__dim_compare:wNN
14015         \dim_use:N \__dim_eval:w #3
14016     }
14017 \cs_new:cpn { __dim_compare_ ! :w }
14018     #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
14019 \cs_new:cpn { __dim_compare_ = :w }
14020     #1 \__dim_eval:w = { #1 \__dim_eval:w }
14021 \cs_new:cpn { __dim_compare_ < :w }
14022     #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
14023 \cs_new:cpn { __dim_compare_ > :w }
14024     #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
14025 \cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \s__dim_stop
14026     { #1 \prg_return_false: \else: \prg_return_true: \fi: }
14027 \cs_new_protected:Npn \__dim_compare_error:
14028     {
14029         \if_int_compare:w \c_zero_int \c_zero_int \fi:
14030         =
14031         \__dim_compare_error:
14032     }

```

(End definition for \dim_compare:nTF and others. This function is documented on page 174.)

<pre> \dim_case:nn \dim_case:nnTF __dim_case:nnTF __dim_case:nw __dim_case_end:nw </pre>	<p>For dimension cases, the first task to fully expand the check condition. The over all idea is then much the same as for \str_case:nn(TF) as described in l3basics.</p> <pre> 14033 \cs_new:Npn \dim_case:nnTF #1 14034 { 14035 \exp:w 14036 \exp_args:Nf __dim_case:nnTF { \dim_eval:n {#1} } 14037 } 14038 \cs_new:Npn \dim_case:nnT #1#2#3 14039 { 14040 \exp:w 14041 \exp_args:Nf __dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { } 14042 } 14043 \cs_new:Npn \dim_case:nnF #1#2 14044 { 14045 \exp:w 14046 \exp_args:Nf __dim_case:nnTF { \dim_eval:n {#1} } {#2} { } 14047 } 14048 \cs_new:Npn \dim_case:nn #1#2 14049 { 14050 \exp:w 14051 \exp_args:Nf __dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { } 14052 } 14053 \cs_new:Npn __dim_case:nnTF #1#2#3#4 14054 { __dim_case:nw {#1} #2 {#1} { } \s__dim_mark {#3} \s__dim_mark {#4} \s__dim_stop } 14055 \cs_new:Npn __dim_case:nw #1#2#3 14056 { 14057 \dim_compare:nNnTF {#1} = {#2} </pre>
---	--

```

14058         { \_dim_case_end:nw {#3} }
14059         { \_dim_case:nw {#1} }
14060     }
14061 \cs_new:Npn \_dim_case_end:nw #1#2#3 \s__dim_mark #4#5 \s__dim_stop
14062 { \exp_end: #1 #4 }

```

(End definition for `\dim_case:nnTF` and others. This function is documented on page 175.)

21.7 Dimension expression loops

`\dim_while_do:nn` `while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```

\dim_until_do:nn
\dim_do_while:nn
\dim_do_until:nn
14063 \cs_new:Npn \dim_while_do:nn #1#2
14064 {
14065     \dim_compare:nT {#1}
14066     {
14067         #2
14068         \dim_while_do:nn {#1} {#2}
14069     }
14070 }
14071 \cs_new:Npn \dim_until_do:nn #1#2
14072 {
14073     \dim_compare:nF {#1}
14074     {
14075         #2
14076         \dim_until_do:nn {#1} {#2}
14077     }
14078 }
14079 \cs_new:Npn \dim_do_while:nn #1#2
14080 {
14081     #2
14082     \dim_compare:nT {#1}
14083     { \dim_do_while:nn {#1} {#2} }
14084 }
14085 \cs_new:Npn \dim_do_until:nn #1#2
14086 {
14087     #2
14088     \dim_compare:nF {#1}
14089     { \dim_do_until:nn {#1} {#2} }
14090 }

```

(End definition for `\dim_while_do:nn` and others. These functions are documented on page 176.)

`\dim_while_do:nNnn` `while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```

\dim_until_do:nNnn
\dim_do_while:nNnn
\dim_do_until:nNnn
14091 \cs_new:Npn \dim_while_do:nNnn #1#2#3#4
14092 {
14093     \dim_compare:nNnT {#1} #2 {#3}
14094     {
14095         #4
14096         \dim_while_do:nNnn {#1} #2 {#3} {#4}
14097     }
14098 }
14099 \cs_new:Npn \dim_until_do:nNnn #1#2#3#4

```

```

14100 {
14101   \dim_compare:nNnF {#1} #2 {#3}
14102   {
14103     #4
14104     \dim_until_do:nNnn {#1} #2 {#3} {#4}
14105   }
14106 }
14107 \cs_new:Npn \dim_do_while:nNnn #1#2#3#4
14108 {
14109   #4
14110   \dim_compare:nNnT {#1} #2 {#3}
14111   { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
14112 }
14113 \cs_new:Npn \dim_do_until:nNnn #1#2#3#4
14114 {
14115   #4
14116   \dim_compare:nNnF {#1} #2 {#3}
14117   { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
14118 }

```

(End definition for `\dim_while_do:nNnn` and others. These functions are documented on page 176.)

21.8 Dimension step functions

`\dim_step_function:nnnN`
`__dim_step:wwwN`
`__dim_step:NnnnN`

Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```

14119 \cs_new:Npn \dim_step_function:nnnN #1#2#3
14120 {
14121   \exp_after:wN \__dim_step:wwwN
14122   \tex_the:D \__dim_eval:w #1 \exp_after:wN ;
14123   \tex_the:D \__dim_eval:w #2 \exp_after:wN ;
14124   \tex_the:D \__dim_eval:w #3 ;
14125 }
14126 \cs_new:Npn \__dim_step:wwwN #1; #2; #3; #4
14127 {
14128   \dim_compare:nNnTF {#2} > \c_zero_dim
14129   { \__dim_step:NnnnN > }
14130   {
14131     \dim_compare:nNnTF {#2} = \c_zero_dim
14132     {
14133       \__kernel_msg_expandable_error:nnn { kernel } { zero-step } {#4}
14134       \use_none:nnnn
14135     }
14136     { \__dim_step:NnnnN < }
14137   }
14138   {#1} {#2} {#3} #4
14139 }
14140 \cs_new:Npn \__dim_step:NnnnN #1#2#3#4#5
14141 {
14142   \dim_compare:nNnF {#2} #1 {#4}
14143   {

```

```

14144         #5 {#2}
14145         \exp_args:Nnf \__dim_step:NnnnN
14146         #1 { \dim_eval:n { #2 + #3 } } {#3} {#4} #5
14147     }
14148 }

```

(End definition for `\dim_step_function:nnnN`, `__dim_step:wwwN`, and `__dim_step:NnnnN`. This function is documented on page 176.)

`\dim_step_inline:nnnn`
`\dim_step_variable:nnnNn`
`__dim_step:NNnnnn`

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\dim_step_function:nnnN`. We put a `\prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g__kernel_prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so that no breaking function recognizes this break point as its own.

```

14149 \cs_new_protected:Npn \dim_step_inline:nnnn
14150 {
14151     \int_gincr:N \g__kernel_prg_map_int
14152     \exp_args:NNc \__dim_step:NNnnnn
14153     \cs_gset_protected:Npn
14154     { __dim_map_ \int_use:N \g__kernel_prg_map_int :w }
14155 }
14156 \cs_new_protected:Npn \dim_step_variable:nnnNn #1#2#3#4#5
14157 {
14158     \int_gincr:N \g__kernel_prg_map_int
14159     \exp_args:NNc \__dim_step:NNnnnn
14160     \cs_gset_protected:Npx
14161     { __dim_map_ \int_use:N \g__kernel_prg_map_int :w }
14162     {#1}{#2}{#3}
14163     {
14164         \tl_set:Nn \exp_not:N #4 {##1}
14165         \exp_not:n {#5}
14166     }
14167 }
14168 \cs_new_protected:Npn \__dim_step:NNnnnn #1#2#3#4#5#6
14169 {
14170     #1 #2 ##1 {#6}
14171     \dim_step_function:nnnN {#3} {#4} {#5} #2
14172     \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
14173 }

```

(End definition for `\dim_step_inline:nnnn`, `\dim_step_variable:nnnNn`, and `__dim_step:NNnnnn`. These functions are documented on page 176.)

21.9 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```

14174 \cs_new:Npn \dim_eval:n #1
14175 { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_eval:n`. This function is documented on page 177.)

`\dim_sign:n` See `\dim_abs:n`. Contrarily to `\int_sign:n` the case of a zero dimension cannot be distinguished from a positive dimension by looking only at the first character, since 0.2pt

and `Opt` start the same way. We need explicit comparisons. We start by distinguishing the most common case of a positive dimension.

```

14176 \cs_new:Npn \dim_sign:n #1
14177 {
14178   \int_value:w \exp_after:wN \__dim_sign:Nw
14179   \dim_use:N \__dim_eval:w #1 \__dim_eval_end: ;
14180   \exp_stop_f:
14181 }
14182 \cs_new:Npn \__dim_sign:Nw #1#2 ;
14183 {
14184   \if_dim:w #1#2 > \c_zero_dim
14185     1
14186   \else:
14187     \if_meaning:w - #1
14188       -1
14189     \else:
14190       0
14191     \fi:
14192   \fi:
14193 }
```

(End definition for `\dim_sign:n` and `__dim_sign:Nw`. This function is documented on page 177.)

`\dim_use:N` Accessing a $\langle dim \rangle$.

`\dim_use:c` 14194 \cs_new_eq:NN \dim_use:N \tex_the:D

We hand-code this for some speed gain:

```

14195 %\cs_generate_variant:Nn \dim_use:N { c }
14196 \cs_new:Npn \dim_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(End definition for `\dim_use:N`. This function is documented on page 177.)

`\dim_to_decimal:n` A function which comes up often enough to deserve a place in the kernel. Evaluate the dimension expression `#1` then remove the trailing `pt`. When debugging is enabled, the argument is put in parentheses as this prevents the dimension expression from terminating early and leaving extra tokens lying around. This is used a lot by low-level manipulations.

```

\__dim_to_decimal:w
14197 \cs_new:Npn \dim_to_decimal:n #1
14198 {
14199   \exp_after:wN
14200   \__dim_to_decimal:w \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
14201 }
14202 \use:x
14203 {
14204   \cs_new:Npn \exp_not:N \__dim_to_decimal:w
14205     ##1 . ##2 \tl_to_str:n { pt }
14206 }
14207 {
14208   \int_compare:nNnTF {#2} > { 0 }
14209     { #1 . #2 }
14210     { #1 }
14211 }
```

(End definition for `\dim_to_decimal:n` and `__dim_to_decimal:w`. This function is documented on page 177.)

`\dim_to_decimal_in_bp:n` Conversion to big points is done using a scaling inside `__dim_eval:w` as ϵ -TeX does that using 64-bit precision. Here, 800/803 is the integer fraction for 72/72.27. This is a common case so is hand-coded for accuracy (and speed).

```
14212 \cs_new:Npn \dim_to_decimal_in_bp:n #1
14213 { \dim_to_decimal:n { ( #1 ) * 800 / 803 } }
```

(End definition for `\dim_to_decimal_in_bp:n`. This function is documented on page 178.)

`\dim_to_decimal_in_sp:n` Another hard-coded conversion: this one is necessary to avoid things going off-scale.

```
14214 \cs_new:Npn \dim_to_decimal_in_sp:n #1
14215 { \int_value:w \__dim_eval:w #1 \__dim_eval_end: }
```

(End definition for `\dim_to_decimal_in_sp:n`. This function is documented on page 178.)

`\dim_to_decimal_in_unit:nn` An analogue of `\dim_ratio:nn` that produces a decimal number as its result, rather than a rational fraction for use within dimension expressions.

```
14216 \cs_new:Npn \dim_to_decimal_in_unit:nn #1#2
14217 {
14218   \dim_to_decimal:n
14219   {
14220     1pt *
14221     \dim_ratio:nn {#1} {#2}
14222   }
14223 }
```

(End definition for `\dim_to_decimal_in_unit:nn`. This function is documented on page 178.)

`\dim_to_fp:n` Defined in `l3fp-convert`, documented here.

(End definition for `\dim_to_fp:n`. This function is documented on page 178.)

21.10 Viewing dim variables

`\dim_show:N` Diagnostics.

```
\dim_show:c 14224 \cs_new_eq:NN \dim_show:N \__kernel_register_show:N
14225 \cs_generate_variant:Nn \dim_show:N { c }
```

(End definition for `\dim_show:N`. This function is documented on page 178.)

`\dim_show:n` Diagnostics. We don't use the TeX primitive `\showthe` to show dimension expressions: this gives a more unified output.

```
14226 \cs_new_protected:Npn \dim_show:n
14227 { \msg_show_eval:Nn \dim_eval:n }
```

(End definition for `\dim_show:n`. This function is documented on page 179.)

`\dim_log:N` Diagnostics. Redirect output of `\dim_show:n` to the log.

```
\dim_log:c 14228 \cs_new_eq:NN \dim_log:N \__kernel_register_log:N
\dim_log:n 14229 \cs_new_eq:NN \dim_log:c \__kernel_register_log:c
14230 \cs_new_protected:Npn \dim_log:n
14231 { \msg_log_eval:Nn \dim_eval:n }
```

(End definition for `\dim_log:N` and `\dim_log:n`. These functions are documented on page 179.)

21.11 Constant dimensions

`\c_zero_dim` Constant dimensions.

`\c_max_dim` 14232 `\dim_const:Nn \c_zero_dim { 0 pt }`
14233 `\dim_const:Nn \c_max_dim { 16383.99999 pt }`

(End definition for `\c_zero_dim` and `\c_max_dim`. These variables are documented on page 179.)

21.12 Scratch dimensions

`\l_tmpa_dim` We provide two local and two global scratch registers, maybe we need more or less.

`\l_tmpb_dim` 14234 `\dim_new:N \l_tmpa_dim`
`\g_tmpa_dim` 14235 `\dim_new:N \l_tmpb_dim`
`\g_tmpb_dim` 14236 `\dim_new:N \g_tmpa_dim`
14237 `\dim_new:N \g_tmpb_dim`

(End definition for `\l_tmpa_dim` and others. These variables are documented on page 179.)

21.13 Creating and initialising skip variables

14238 `<@@=skip>`

`\s_skip_stop` Internal scan marks.

14239 `\scan_new:N \s_skip_stop`

(End definition for `\s_skip_stop`.)

`\skip_new:N` Allocation of a new internal registers.

`\skip_new:c` 14240 `\cs_new_protected:Npn \skip_new:N #1`
14241 `{`
14242 `__kernel_chk_if_free_cs:N #1`
14243 `\cs:w newskip \cs_end: #1`
14244 `}`
14245 `\cs_generate_variant:Nn \skip_new:N { c }`

(End definition for `\skip_new:N`. This function is documented on page 179.)

`\skip_const:Nn` Contrarily to integer constants, we cannot avoid using a register, even for constants. See
`\skip_const:cn` `\dim_const:Nn` for why we cannot use `\skip_gset:Nn`.

14246 `\cs_new_protected:Npn \skip_const:Nn #1#2`
14247 `{`
14248 `\skip_new:N #1`
14249 `\tex_global:D #1 ~ \skip_eval:n {#2} \scan_stop:`
14250 `}`
14251 `\cs_generate_variant:Nn \skip_const:Nn { c }`

(End definition for `\skip_const:Nn`. This function is documented on page 180.)

`\skip_zero:N` Reset the register to zero.

`\skip_zero:c` 14252 `\cs_new_protected:Npn \skip_zero:N #1 { #1 \c_zero_skip }`
`\skip_gzero:N` 14253 `\cs_new_protected:Npn \skip_gzero:N #1 { \tex_global:D #1 \c_zero_skip }`
`\skip_gzero:c` 14254 `\cs_generate_variant:Nn \skip_zero:N { c }`
14255 `\cs_generate_variant:Nn \skip_gzero:N { c }`

(End definition for `\skip_zero:N` and `\skip_gzero:N`. These functions are documented on page 180.)

\skip_zero_new:N Create a register if needed, otherwise clear it.

```

\skip_zero_new:c 14256 \cs_new_protected:Npn \skip_zero_new:N #1
\skip_gzero_new:N 14257 { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
\skip_gzero_new:c 14258 \cs_new_protected:Npn \skip_gzero_new:N #1
14259 { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
14260 \cs_generate_variant:Nn \skip_zero_new:N { c }
14261 \cs_generate_variant:Nn \skip_gzero_new:N { c }

(End definition for \skip_zero_new:N and \skip_gzero_new:N. These functions are documented on page 180.)

```

\skip_if_exist_p:N Copies of the cs functions defined in l3basics.

```

\skip_if_exist_p:c 14262 \prg_new_eq_conditional:NNn \skip_if_exist:N \cs_if_exist:N
\skip_if_exist:NTF 14263 { TF , T , F , p }
\skip_if_exist:cTF 14264 \prg_new_eq_conditional:NNn \skip_if_exist:c \cs_if_exist:c
14265 { TF , T , F , p }

(End definition for \skip_if_exist:NTF. This function is documented on page 180.)

```

21.14 Setting skip variables

\skip_set:Nn Much the same as for dimensions.

```

\skip_set:cn 14266 \cs_new_protected:Npn \skip_set:Nn #1#2
\skip_gset:Nn 14267 { #1 ~ \tex_glueexpr:D #2 \scan_stop: }
\skip_gset:cn 14268 \cs_new_protected:Npn \skip_gset:Nn #1#2
14269 { \tex_global:D #1 ~ \tex_glueexpr:D #2 \scan_stop: }
14270 \cs_generate_variant:Nn \skip_set:Nn { c }
14271 \cs_generate_variant:Nn \skip_gset:Nn { c }

(End definition for \skip_set:Nn and \skip_gset:Nn. These functions are documented on page 180.)

```

\skip_set_eq:NN All straightforward.

```

\skip_set_eq:cn 14272 \cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
\skip_set_eq:Nc 14273 \cs_generate_variant:Nn \skip_set_eq:NN { c , Nc , cc }
\skip_set_eq:cc 14274 \cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\skip_gset_eq:NN 14275 \cs_generate_variant:Nn \skip_gset_eq:NN { c , Nc , cc }
\skip_gset_eq:cn
\skip_gset_eq:Nc
\skip_gset_eq:cc

(End definition for \skip_set_eq:NN and \skip_gset_eq:NN. These functions are documented on page 180.)

```

\skip_add:Nn Using by here deals with the (incorrect) case \skip123.

```

\skip_add:cn 14276 \cs_new_protected:Npn \skip_add:Nn #1#2
\skip_gadd:Nn 14277 { \tex_advance:D #1 by \tex_glueexpr:D #2 \scan_stop: }
\skip_gadd:cn 14278 \cs_new_protected:Npn \skip_gadd:Nn #1#2
\skip_sub:Nn 14279 { \tex_global:D \tex_advance:D #1 by \tex_glueexpr:D #2 \scan_stop: }
\skip_sub:cn 14280 \cs_generate_variant:Nn \skip_add:Nn { c }
\skip_gsub:Nn 14281 \cs_generate_variant:Nn \skip_gadd:Nn { c }
\skip_gsub:cn 14282 \cs_new_protected:Npn \skip_sub:Nn #1#2
14283 { \tex_advance:D #1 by - \tex_glueexpr:D #2 \scan_stop: }
14284 \cs_new_protected:Npn \skip_gsub:Nn #1#2
14285 { \tex_global:D \tex_advance:D #1 by - \tex_glueexpr:D #2 \scan_stop: }
14286 \cs_generate_variant:Nn \skip_sub:Nn { c }
14287 \cs_generate_variant:Nn \skip_gsub:Nn { c }

(End definition for \skip_add:Nn and others. These functions are documented on page 180.)

```

21.15 Skip expression conditionals

`\skip_if_eq_p:nn` Comparing skips means doing two expansions to make strings, and then testing them.
`\skip_if_eq:nnTF` As a result, only equality is tested.

```

14288 \prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
14289 {
14290   \str_if_eq:eeTF { \skip_eval:n { #1 } } { \skip_eval:n { #2 } }
14291   { \prg_return_true: }
14292   { \prg_return_false: }
14293 }
```

(End definition for `\skip_if_eq:nnTF`. This function is documented on page 181.)

`\skip_if_finite_p:n` With ε -TeX, we have an easy access to the order of infinities of the stretch and shrink components of a skip. However, to access both, we either need to evaluate the expression twice, or evaluate it, then call an auxiliary to extract both pieces of information from the result. Since we are going to need an auxiliary anyways, it is quicker to make it search for the string `fil` which characterizes infinite glue.
`\skip_if_finite:nTF`
`__skip_if_finite:wwNw`

```

14294 \cs_set_protected:Npn \__skip_tmp:w #1
14295 {
14296   \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
14297   {
14298     \exp_after:wN \__skip_if_finite:wwNw
14299     \skip_use:N \tex_glueexpr:D ##1 ; \prg_return_false:
14300     #1 ; \prg_return_true: \s__skip_stop
14301   }
14302   \cs_new:Npn \__skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \s__skip_stop {##3}
14303 }
14304 \exp_args:No \__skip_tmp:w { \tl_to_str:n { fil } }
```

(End definition for `\skip_if_finite:nTF` and `__skip_if_finite:wwNw`. This function is documented on page 181.)

21.16 Using skip expressions and variables

`\skip_eval:n` Evaluating a skip expression expandably.

```

14305 \cs_new:Npn \skip_eval:n #1
14306 { \skip_use:N \tex_glueexpr:D #1 \scan_stop: }
```

(End definition for `\skip_eval:n`. This function is documented on page 181.)

`\skip_use:N` Accessing a $\langle skip \rangle$.

```

\skip_use:c 14307 \cs_new_eq:NN \skip_use:N \tex_the:D
14308 %\cs_generate_variant:Nn \skip_use:N { c }
14309 \cs_new:Npn \skip_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(End definition for `\skip_use:N`. This function is documented on page 181.)

21.17 Inserting skips into the output

`\skip_horizontal:N` Inserting skips.
`\skip_horizontal:c` 14310 `\cs_new_eq:NN \skip_horizontal:N \tex_hskip:D`
`\skip_horizontal:n` 14311 `\cs_new:Npn \skip_horizontal:n #1`
`\skip_vertical:N` 14312 `{ \skip_horizontal:N \tex_glueexpr:D #1 \scan_stop: }`
`\skip_vertical:c` 14313 `\cs_new_eq:NN \skip_vertical:N \tex_vskip:D`
`\skip_vertical:n` 14314 `\cs_new:Npn \skip_vertical:n #1`
14315 `{ \skip_vertical:N \tex_glueexpr:D #1 \scan_stop: }`
14316 `\cs_generate_variant:Nn \skip_horizontal:N { c }`
14317 `\cs_generate_variant:Nn \skip_vertical:N { c }`

(End definition for `\skip_horizontal:N` and others. These functions are documented on page 182.)

21.18 Viewing skip variables

`\skip_show:N` Diagnostics.
`\skip_show:c` 14318 `\cs_new_eq:NN \skip_show:N __kernel_register_show:N`
14319 `\cs_generate_variant:Nn \skip_show:N { c }`

(End definition for `\skip_show:N`. This function is documented on page 181.)

`\skip_show:n` Diagnostics. We don't use the T_EX primitive `\showthe` to show skip expressions: this gives a more unified output.

14320 `\cs_new_protected:Npn \skip_show:n`
14321 `{ \msg_show_eval:Nn \skip_eval:n }`

(End definition for `\skip_show:n`. This function is documented on page 181.)

`\skip_log:N` Diagnostics. Redirect output of `\skip_show:n` to the log.
`\skip_log:c` 14322 `\cs_new_eq:NN \skip_log:N __kernel_register_log:N`
`\skip_log:n` 14323 `\cs_new_eq:NN \skip_log:c __kernel_register_log:c`
14324 `\cs_new_protected:Npn \skip_log:n`
14325 `{ \msg_log_eval:Nn \skip_eval:n }`

(End definition for `\skip_log:N` and `\skip_log:n`. These functions are documented on page 182.)

21.19 Constant skips

`\c_zero_skip` Skips with no rubber component are just dimensions but need to terminate correctly.
`\c_max_skip` 14326 `\skip_const:Nn \c_zero_skip { \c_zero_dim }`
14327 `\skip_const:Nn \c_max_skip { \c_max_dim }`

(End definition for `\c_zero_skip` and `\c_max_skip`. These functions are documented on page 182.)

21.20 Scratch skips

`\l_tmpa_skip` We provide two local and two global scratch registers, maybe we need more or less.
`\l_tmpb_skip` 14328 `\skip_new:N \l_tmpa_skip`
`\g_tmpa_skip` 14329 `\skip_new:N \l_tmpb_skip`
`\g_tmpb_skip` 14330 `\skip_new:N \g_tmpa_skip`
14331 `\skip_new:N \g_tmpb_skip`

(End definition for `\l_tmpa_skip` and others. These variables are documented on page 182.)

21.21 Creating and initialising muskip variables

\muskip_new:N And then we add muskips.

```
\muskip_new:c 14332 \cs_new_protected:Npn \muskip_new:N #1
               14333 {
               14334     \__kernel_chk_if_free_cs:N #1
               14335     \cs:w newmuskip \cs_end: #1
               14336 }
               14337 \cs_generate_variant:Nn \muskip_new:N { c }
```

(End definition for \muskip_new:N. This function is documented on page 183.)

\muskip_const:Nn See \skip_const:Nn.

```
\muskip_const:cn 14338 \cs_new_protected:Npn \muskip_const:Nn #1#2
                 14339 {
                 14340     \muskip_new:N #1
                 14341     \tex_global:D #1 ~ \muskip_eval:n {#2} \scan_stop:
                 14342 }
                 14343 \cs_generate_variant:Nn \muskip_const:Nn { c }
```

(End definition for \muskip_const:Nn. This function is documented on page 183.)

\muskip_zero:N Reset the register to zero.

```
\muskip_zero:c 14344 \cs_new_protected:Npn \muskip_zero:N #1
\muskip_gzero:N 14345 { #1 \c_zero_muskip }
\muskip_gzero:c 14346 \cs_new_protected:Npn \muskip_gzero:N #1
                 14347 { \tex_global:D #1 \c_zero_muskip }
                 14348 \cs_generate_variant:Nn \muskip_zero:N { c }
                 14349 \cs_generate_variant:Nn \muskip_gzero:N { c }
```

(End definition for \muskip_zero:N and \muskip_gzero:N. These functions are documented on page 183.)

\muskip_zero_new:N Create a register if needed, otherwise clear it.

```
\muskip_zero_new:c 14350 \cs_new_protected:Npn \muskip_zero_new:N #1
\muskip_gzero_new:N 14351 { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
\muskip_gzero_new:c 14352 \cs_new_protected:Npn \muskip_gzero_new:N #1
                 14353 { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
                 14354 \cs_generate_variant:Nn \muskip_zero_new:N { c }
                 14355 \cs_generate_variant:Nn \muskip_gzero_new:N { c }
```

(End definition for \muskip_zero_new:N and \muskip_gzero_new:N. These functions are documented on page 183.)

\muskip_if_exist_p:N Copies of the cs functions defined in l3basics.

```
\muskip_if_exist_p:c 14356 \prg_new_eq_conditional:NNn \muskip_if_exist:N \cs_if_exist:N
\muskip_if_exist:N $\underline{TF}$  14357 { TF , T , F , p }
\muskip_if_exist:c $\underline{TF}$  14358 \prg_new_eq_conditional:NNn \muskip_if_exist:c \cs_if_exist:c
                 14359 { TF , T , F , p }
```

(End definition for \muskip_if_exist:N \underline{TF} . This function is documented on page 183.)

21.22 Setting muskip variables

`\muskip_set:Nn` This should be pretty familiar.

```

\muskip_set:cn 14360 \cs_new_protected:Npn \muskip_set:Nn #1#2
\muskip_gset:Nn 14361 { #1 ~ \tex_muexpr:D #2 \scan_stop: }
\muskip_gset:cn 14362 \cs_new_protected:Npn \muskip_gset:Nn #1#2
14363 { \tex_global:D #1 ~ \tex_muexpr:D #2 \scan_stop: }
14364 \cs_generate_variant:Nn \muskip_set:Nn { c }
14365 \cs_generate_variant:Nn \muskip_gset:Nn { c }
```

(End definition for `\muskip_set:Nn` and `\muskip_gset:Nn`. These functions are documented on page 184.)

`\muskip_set_eq:NN` All straightforward.

```

\muskip_set_eq:cn 14366 \cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
\muskip_set_eq:Nc 14367 \cs_generate_variant:Nn \muskip_set_eq:NN { c , Nc , cc }
\muskip_set_eq:cc 14368 \cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\muskip_gset_eq:NN 14369 \cs_generate_variant:Nn \muskip_gset_eq:NN { c , Nc , cc }
```

(End definition for `\muskip_set_eq:NN` and `\muskip_gset_eq:NN`. These functions are documented on page 184.)

`\muskip_add:Nn` Using by here deals with the (incorrect) case `\muskip123`.

```

\muskip_add:cn 14370 \cs_new_protected:Npn \muskip_add:Nn #1#2
\muskip_gadd:Nn 14371 { \tex_advance:D #1 by \tex_muexpr:D #2 \scan_stop: }
\muskip_gadd:cn 14372 \cs_new_protected:Npn \muskip_gadd:Nn #1#2
\muskip_sub:Nn 14373 { \tex_global:D \tex_advance:D #1 by \tex_muexpr:D #2 \scan_stop: }
\muskip_sub:cn 14374 \cs_generate_variant:Nn \muskip_add:Nn { c }
\muskip_gsub:Nn 14375 \cs_generate_variant:Nn \muskip_gadd:Nn { c }
\muskip_gsub:cn 14376 \cs_new_protected:Npn \muskip_sub:Nn #1#2
14377 { \tex_advance:D #1 by - \tex_muexpr:D #2 \scan_stop: }
14378 \cs_new_protected:Npn \muskip_gsub:Nn #1#2
14379 { \tex_global:D \tex_advance:D #1 by - \tex_muexpr:D #2 \scan_stop: }
14380 \cs_generate_variant:Nn \muskip_sub:Nn { c }
14381 \cs_generate_variant:Nn \muskip_gsub:Nn { c }
```

(End definition for `\muskip_add:Nn` and others. These functions are documented on page 183.)

21.23 Using muskip expressions and variables

`\muskip_eval:n` Evaluating a muskip expression expandably.

```

14382 \cs_new:Npn \muskip_eval:n #1
14383 { \muskip_use:N \tex_muexpr:D #1 \scan_stop: }
```

(End definition for `\muskip_eval:n`. This function is documented on page 184.)

`\muskip_use:N` Accessing a $\langle muskip \rangle$.

```

\muskip_use:c 14384 \cs_new_eq:NN \muskip_use:N \tex_the:D
14385 \cs_generate_variant:Nn \muskip_use:N { c }
```

(End definition for `\muskip_use:N`. This function is documented on page 184.)

21.24 Viewing muskip variables

\muskip_show:N Diagnostics.

\muskip_show:c 14386 \cs_new_eq:NN \muskip_show:N __kernel_register_show:N
14387 \cs_generate_variant:Nn \muskip_show:N { c }

(End definition for \muskip_show:N. This function is documented on page 184.)

\muskip_show:n Diagnostics. We don't use the TeX primitive \showthe to show muskip expressions: this gives a more unified output.

14388 \cs_new_protected:Npn \muskip_show:n
14389 { \msg_show_eval:Nn \muskip_eval:n }

(End definition for \muskip_show:n. This function is documented on page 185.)

\muskip_log:N Diagnostics. Redirect output of \muskip_show:n to the log.

\muskip_log:c 14390 \cs_new_eq:NN \muskip_log:N __kernel_register_log:N
\muskip_log:n 14391 \cs_new_eq:NN \muskip_log:c __kernel_register_log:c
14392 \cs_new_protected:Npn \muskip_log:n
14393 { \msg_log_eval:Nn \muskip_eval:n }

(End definition for \muskip_log:N and \muskip_log:n. These functions are documented on page 185.)

21.25 Constant muskips

\c_zero_muskip Constant muskips given by their value.

\c_max_muskip 14394 \muskip_const:Nn \c_zero_muskip { 0 mu }
14395 \muskip_const:Nn \c_max_muskip { 16383.99999 mu }

(End definition for \c_zero_muskip and \c_max_muskip. These functions are documented on page 185.)

21.26 Scratch muskips

\l_tmpa_muskip We provide two local and two global scratch registers, maybe we need more or less.

\l_tmpb_muskip 14396 \muskip_new:N \l_tmpa_muskip
\g_tmpa_muskip 14397 \muskip_new:N \l_tmpb_muskip
\g_tmpb_muskip 14398 \muskip_new:N \g_tmpa_muskip
14399 \muskip_new:N \g_tmpb_muskip

(End definition for \l_tmpa_muskip and others. These variables are documented on page 185.)

14400 </package>

22 l3keys Implementation

14401 <*package>

22.1 Low-level interface

The low-level key parser's implementation is based heavily on `expkv`. Compared to `keyval` it adds a number of additional "safety" requirements and allows to process the parsed list of key-value pairs in a variety of ways. The net result is that this code needs around one and a half the amount of time as `keyval` to parse the same list of keys. To optimise speed as far as reasonably practical, a number of lower-level approaches are taken rather than using the higher-level `expl3` interfaces.

14402 <@@=keyval>

```

\s__keyval_nil
\s__keyval_mark 14403 \scan_new:N \s__keyval_nil
\s__keyval_stop 14404 \scan_new:N \s__keyval_mark
\s__keyval_tail 14405 \scan_new:N \s__keyval_stop
14406 \scan_new:N \s__keyval_tail

```

(End definition for `\s__keyval_nil` and others.)

This temporary macro will be used since some of the definitions will need an active comma or equals sign. Inside of this macro #1 will be the active comma and #2 will be the active equals sign.

```

14407 \group_begin:
14408   \cs_set_protected:Npn \__keyval_tmp:NN #1#2
14409   {

```

\keyval_parse:NNn The main function starts the first of two loops. The outer loop splits the key–value list at active commas, the inner loop will do so at other commas. The use of `\s__keyval_mark` here prevents loss of braces from the key argument.

```

14410   \cs_new:Npn \keyval_parse:NNn ##1 ##2 ##3
14411   { \__keyval_loop_active:NNw ##1 ##2 \s__keyval_mark ##3 #1 \s__keyval_tail #1 }

```

(End definition for `\keyval_parse:NNn`. This function is documented on page 198.)

__keyval_loop_active:NNw First a fast test for the end of the loop is done, it'll gobble everything up to a `\s__keyval_tail`. The loop ending macro will gobble everything to the last `\s__keyval_mark` in this definition. If the end isn't reached yet, start the second loop splitting at other commas, and after that one iterate the current loop.

```

14412   \cs_new:Npn \__keyval_loop_active:NNw ##1 ##2 ##3 #1
14413   {
14414     \__keyval_if_recursion_tail:w ##3
14415     \__keyval_end_loop_active:w \s__keyval_tail
14416     \__keyval_loop_other:NNw ##1 ##2 ##3 , \s__keyval_tail ,
14417     \__keyval_loop_active:NNw ##1 ##2 \s__keyval_mark
14418   }

```

(End definition for `__keyval_loop_active:NNw`.)

__keyval_split_other:w **__keyval_split_active:w** These two macros allow to split at the first equals sign of category 12 or 13. At the same time they also execute branching by inserting the first token following `\s__keyval_mark` that followed the equals sign. Hence they also test for the presence of such an equals sign simultaneously.

```

14419   \cs_new:Npn \__keyval_split_other:w ##1 = ##2 \s__keyval_mark ##3 ##4 \s__keyval_stop
14420   { ##3 ##1 \s__keyval_stop \s__keyval_mark ##2 }
14421   \cs_new:Npn \__keyval_split_active:w ##1 #2 ##2 \s__keyval_mark ##3 ##4 \s__keyval_stop
14422   { ##3 ##1 \s__keyval_stop \s__keyval_mark ##2 }

```

(End definition for `__keyval_split_other:w` and `__keyval_split_active:w`.)

__keyval_loop_other:NNw The second loop uses the same test for its end as the first loop, next it splits at the first active equals sign using `__keyval_split_active:w`. The `\s__keyval_nil` prevents accidental brace stripping and acts as a delimiter in the next steps. First testing for an active equals sign will reduce the number of necessary expansion steps for the expected average use case of other equals signs and hence perform better on average.

```

14423   \cs_new:Npn \__keyval_loop_other:NNw ##1 ##2 ##3 ,

```

```

14424     {
14425         \__keyval_if_recursion_tail:w ##3
14426         \__keyval_end_loop_other:w \s__keyval_tail
14427         \__keyval_split_active:w ##3 \s__keyval_nil
14428         \s__keyval_mark \__keyval_split_active_auxi:w
14429         #2 \s__keyval_mark \__keyval_clean_up_active:w
14430         \s__keyval_stop
14431         ##1 ##2
14432         \__keyval_loop_other:NNw ##1 ##2 \s__keyval_mark
14433     }

```

(End definition for __keyval_loop_other:NNw.)

_keyval_split_active_auxi:w After __keyval_split_active:w the following will only be called if there was at least one active equals sign in the current key–value pair. Therefore this is the execution branch for a key–value pair with an active equals sign. ##1 will be everything up to the first active equals sign. First it tests for other equals signs in the key name, which will eventually throw an error via __keyval_misplaced_equal_after_active_error:w. If none was found we forward the key to __keyval_split_active_auxii:w.

```

14434     \cs_new:Npn \__keyval_split_active_auxi:w ##1 \s__keyval_stop
14435     {
14436         \__keyval_split_other:w ##1 \s__keyval_nil
14437         \s__keyval_mark \__keyval_misplaced_equal_after_active_error:w
14438         = \s__keyval_mark \__keyval_split_active_auxii:w
14439         \s__keyval_stop
14440     }

```

__keyval_split_active_auxii:w gets the correct key name with a leading \s__keyval_mark as ##1. It has to sanitise the remainder of the previous test and trims the key name which will be forwarded to __keyval_split_active_auxiii:w.

```

14441     \cs_new:Npn \__keyval_split_active_auxii:w
14442     ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_after_active_error:w
14443     \s__keyval_stop \s__keyval_mark
14444     { \__keyval_trim:nN { ##1 } \__keyval_split_active_auxiii:w }

```

Next we test for a misplaced active equals sign in the value, if none is found __keyval_split_active_auxiv:w will be called.

```

14445     \cs_new:Npn \__keyval_split_active_auxiii:w ##1 ##2 \s__keyval_nil
14446     {
14447         \__keyval_split_active:w ##2 \s__keyval_nil
14448         \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
14449         #2 \s__keyval_mark \__keyval_split_active_auxiv:w
14450         \s__keyval_stop
14451         { ##1 }
14452     }

```

This runs the last test after sanitising the remainder of the previous one. This time test for a misplaced equals sign of category 12 in the value. Finally the last auxiliary macro will be called.

```

14453     \cs_new:Npn \__keyval_split_active_auxiv:w
14454     ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
14455     \s__keyval_stop \s__keyval_mark
14456     {
14457         \__keyval_split_other:w ##1 \s__keyval_nil
14458         \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w

```

```

14459             = \s__keyval_mark \__keyval_split_active_auxv:w
14460             \s__keyval_stop
14461         }

```

This last macro in this execution branch sanitises the last test, trims the value and passes it to `__keyval_pair:nnNN`.

```

14462     \cs_new:Npn \__keyval_split_active_auxv:w
14463         ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
14464         \s__keyval_stop \s__keyval_mark
14465         { \__keyval_trim:nN { ##1 } \__keyval_pair:nnNN }

```

(End definition for __keyval_split_active_auxi:w and others.)

`__keyval_clean_up_active:w`

The following is the branch taken if the key-value pair doesn't contain an active equals sign. The remainder of that test will be cleaned up by `__keyval_clean_up_active:w` which will then split at an equals sign of category other.

```

14466     \cs_new:Npn \__keyval_clean_up_active:w
14467         ##1 \s__keyval_nil \s__keyval_mark \__keyval_split_active_auxi:w \s__keyval_stop
14468         {
14469             \__keyval_split_other:w ##1 \s__keyval_nil
14470             \s__keyval_mark \__keyval_split_other_auxi:w
14471             = \s__keyval_mark \__keyval_clean_up_other:w
14472             \s__keyval_stop
14473         }

```

(End definition for __keyval_clean_up_active:w.)

`__keyval_split_other_auxi:w`

`__keyval_split_other_auxii:w`

`__keyval_split_other_auxiii:w`

This is executed if the key-value pair doesn't contain an active equals sign but at least one other. `##1` of `__keyval_split_other_auxi:w` will contain the complete key name, which is trimmed and forwarded to the next auxiliary macro.

```

14474     \cs_new:Npn \__keyval_split_other_auxi:w ##1 \s__keyval_stop
14475         { \__keyval_trim:nN { ##1 } \__keyval_split_other_auxii:w }

```

We know that the value doesn't contain misplaced active equals signs but we have to test for others.

```

14476     \cs_new:Npn \__keyval_split_other_auxii:w ##1 ##2 \s__keyval_nil
14477         {
14478             \__keyval_split_other:w ##2 \s__keyval_nil
14479             \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
14480             = \s__keyval_mark \__keyval_split_other_auxiii:w
14481             \s__keyval_stop
14482             { ##1 }
14483         }

```

`__keyval_split_other_auxiii:w` sanitises the test for other equals signs, trims the value and forwards it to `__keyval_pair:nnNN`.

```

14484     \cs_new:Npn \__keyval_split_other_auxiii:w
14485         ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
14486         \s__keyval_stop \s__keyval_mark
14487         { \__keyval_trim:nN { ##1 } \__keyval_pair:nnNN }

```

(End definition for __keyval_split_other_auxi:w, __keyval_split_other_auxii:w, and __keyval_split_other_auxiii:w.)

`__keyval_clean_up_other:w` `__keyval_clean_up_other:w` is the last branch that might exist. It is called if no equals sign was found, hence the only possibilities left are a blank list element, which is to be skipped, or a lonely key. If it's no empty list element this will trim the key name and forward it to `__keyval_key:nnn`.

```

14488     \cs_new:Npn \__keyval_clean_up_other:w
14489         ##1 \s__keyval_nil \s__keyval_mark \__keyval_split_other_auxi:w \s__keyval_stop \
14490     {
14491         \__keyval_if_blank:w ##1 \s__keyval_nil \s__keyval_stop \__keyval_blank_true:w
14492         \s__keyval_mark \s__keyval_stop \use:n
14493         { \__keyval_trim:nn { ##1 } \__keyval_key:nnn }
14494     }

```

(End definition for `__keyval_clean_up_other:w`.)

`keyval_misplaced_equal_after_active_error:w` `__keyval_misplaced_equal_in_split_error:w` All these two macros do is gobble the remainder of the current other loop execution and throw an error.

```

14495     \cs_new:Npn \__keyval_misplaced_equal_after_active_error:w
14496         \s__keyval_mark ##1 \s__keyval_stop \s__keyval_mark ##2 \s__keyval_nil
14497         \s__keyval_mark ##3 \s__keyval_nil ##4 ##5
14498     {
14499         \__kernel_msg_expandable_error:nn
14500         { kernel } { misplaced-equals-sign }
14501     }
14502     \cs_new:Npn \__keyval_misplaced_equal_in_split_error:w
14503         \s__keyval_mark ##1 \s__keyval_stop \s__keyval_mark ##2 \s__keyval_nil
14504         ##3 ##4 ##5
14505     {
14506         \__kernel_msg_expandable_error:nn
14507         { kernel } { misplaced-equals-sign }
14508     }

```

(End definition for `__keyval_misplaced_equal_after_active_error:w` and `__keyval_misplaced_equal_in_split_error:w`.)

`__keyval_end_loop_other:w` `__keyval_end_loop_active:w` All that's left for the parsing loops are the macros which end the recursion. Both just gobble the remaining tokens of the respective loop including the next recursion call.

```

14509     \cs_new:Npn \__keyval_end_loop_other:w
14510         \s__keyval_tail
14511         \__keyval_split_active:w ##1 \s__keyval_nil
14512         \s__keyval_mark \__keyval_split_active_auxi:w
14513         #2 \s__keyval_mark \__keyval_clean_up_active:w
14514         \s__keyval_stop
14515         ##2 ##3
14516         \__keyval_loop_other:NNw ##4 \s__keyval_mark
14517     { }
14518     \cs_new:Npn \__keyval_end_loop_active:w
14519         \s__keyval_tail
14520         \__keyval_loop_other:NNw ##1 , \s__keyval_tail ,
14521         \__keyval_loop_active:NNw ##2 \s__keyval_mark
14522     { }

```

(End definition for `__keyval_end_loop_other:w` and `__keyval_end_loop_active:w`.)

The parsing loops are done, so here ends the definition of `__keyval_tmp:nn`, which will finally set up the macros.

```

14523     }
14524     \char_set_catcode_active:n { '\, }
14525     \char_set_catcode_active:n { '\= }
14526     \__keyval_tmp:NN , =
14527 \group_end:

```

`__keyval_pair:nnNN` These macros will be called on the parsed keys and values of the key–value list. All arguments are completely trimmed. They test for blank key names and call the functions passed to `\keyval_parse:NNn` inside of `\exp_not:n` with the correct arguments.

```

14528 \cs_new:Npn \__keyval_pair:nnNN #1 #2 #3 #4
14529 {
14530     \__keyval_if_blank:w \s__keyval_mark #2 \s__keyval_nil \s__keyval_stop \__keyval_blank_
14531     \s__keyval_mark \s__keyval_stop
14532     \exp_not:n { #4 { #2 } { #1 } }
14533 }
14534 \cs_new:Npn \__keyval_key:nnN #1 #2 #3
14535 {
14536     \__keyval_if_blank:w \s__keyval_mark #1 \s__keyval_nil \s__keyval_stop \__keyval_blank_
14537     \s__keyval_mark \s__keyval_stop
14538     \exp_not:n { #2 { #1 } }
14539 }

```

(End definition for `__keyval_pair:nnNN` and `__keyval_key:nnN`.)

`__keyval_if_empty:w` All these tests work by gobbling tokens until a certain combination is met, which makes them pretty fast. The test for a blank argument should be called with an arbitrary token following the argument. Each of these utilize the fact that the argument will contain a leading `\s__keyval_mark`.

```

14540 \cs_new:Npn \__keyval_if_empty:w #1 \s__keyval_mark \s__keyval_stop { }
14541 \cs_new:Npn \__keyval_if_blank:w \s__keyval_mark #1 { \__keyval_if_empty:w \s__keyval_mark
14542 \cs_new:Npn \__keyval_if_recursion_tail:w \s__keyval_mark #1 \s__keyval_tail { }

```

(End definition for `__keyval_if_empty:w`, `__keyval_if_blank:w`, and `__keyval_if_recursion_tail:w`.)

`__keyval_blank_true:w` These macros will be called if the tests above didn't gobble them, they execute the branching.

```

14543 \cs_new:Npn \__keyval_blank_true:w \s__keyval_mark \s__keyval_stop \use:n #1 #2 #3 { }
14544 \cs_new:Npn \__keyval_blank_key_error:w \s__keyval_mark \s__keyval_stop \exp_not:n #1
14545 {
14546     \__kernel_msg_expandable_error:nn
14547     { kernel } { blank-key-name }
14548 }

```

(End definition for `__keyval_blank_true:w` and `__keyval_blank_key_error:w`.)

Two messages for the low level parsing system.

```

14549 \__kernel_msg_new:nnn { kernel } { misplaced-equals-sign }
14550 { Misplaced~equals~sign~in~key~value~input~\msg_line_context: }
14551 \__kernel_msg_new:nnn { kernel } { blank-key-name }
14552 { Blank~key~name~in~key~value~input~\msg_line_context: }

```

```

    \__keyval_trim:nN
    \__keyval_trim_auxi:w
    \__keyval_trim_auxii:w
    \__keyval_trim_auxiii:w
    \__keyval_trim_auxiv:w

```

And an adapted version of `__tl_trim_spaces:nn` which is a bit faster for our use case, as it can strip the braces at the end. This is pretty much the same concept, so I won't comment on it here. The speed gain by using this instead of `\tl_trim_spaces_apply:nN` is about 10 % of the total time for `\keyval_parse:NNn` with one key and one key–value pair, so I think it's worth it.

```

14553 \group_begin:
14554   \cs_set_protected:Npn \__keyval_tmp:n #1
14555   {
14556     \cs_new:Npn \__keyval_trim:nN ##1
14557     {
14558       \__keyval_trim_auxi:w
14559       ##1
14560       \s__keyval_nil
14561       \s__keyval_mark #1 { }
14562       \s__keyval_mark \__keyval_trim_auxii:w
14563       \__keyval_trim_auxiii:w
14564       #1 \s__keyval_nil
14565       \__keyval_trim_auxiv:w
14566       \s__keyval_stop
14567     }
14568     \cs_new:Npn \__keyval_trim_auxi:w ##1 \s__keyval_mark #1 ##2 \s__keyval_mark ##3
14569     {
14570       ##3
14571       \__keyval_trim_auxi:w
14572       \s__keyval_mark
14573       ##2
14574       \s__keyval_mark #1 {##1}
14575     }
14576     \cs_new:Npn \__keyval_trim_auxii:w \__keyval_trim_auxi:w \s__keyval_mark \s__keyval_m
14577     {
14578       \__keyval_trim_auxiii:w
14579       ##1
14580     }
14581     \cs_new:Npn \__keyval_trim_auxiii:w ##1 #1 \s__keyval_nil ##2
14582     {
14583       ##2
14584       ##1 \s__keyval_nil
14585       \__keyval_trim_auxiii:w
14586     }

```

This is the one macro which differs from the original definition.

```

14587     \cs_new:Npn \__keyval_trim_auxiv:w \s__keyval_mark ##1 \s__keyval_nil ##2 \s__keyval_
14588     { ##3 { ##1 } }
14589   }
14590   \__keyval_tmp:n { ~ }
14591 \group_end:

```

(End definition for `__keyval_trim:nN` and others.)

22.2 Constants and variables

```

14592 <@@=keys>

```

Various storage areas for the different data which make up keys.

```

\c__keys_code_root_str
\c__keys_default_root_str
\c__keys_groups_root_str
\c__keys_inherit_root_str
\c__keys_type_root_str
\c__keys_validate_root_str

```

```

14593 \str_const:Nn \c__keys_code_root_str { key~code~>~ }
14594 \str_const:Nn \c__keys_default_root_str { key~default~>~ }
14595 \str_const:Nn \c__keys_groups_root_str { key~groups~>~ }
14596 \str_const:Nn \c__keys_inherit_root_str { key~inherit~>~ }
14597 \str_const:Nn \c__keys_type_root_str { key~type~>~ }
14598 \str_const:Nn \c__keys_validate_root_str { key~validate~>~ }

```

(End definition for \c__keys_code_root_str and others.)

\c__keys_props_root_str The prefix for storing properties.

```

14599 \str_const:Nn \c__keys_props_root_str { key~prop~>~ }

```

(End definition for \c__keys_props_root_str.)

\l_keys_choice_int Publicly accessible data on which choice is being used when several are generated as a set.
\l_keys_choice_tl

```

14600 \int_new:N \l_keys_choice_int
14601 \tl_new:N \l_keys_choice_tl

```

(End definition for \l_keys_choice_int and \l_keys_choice_tl. These variables are documented on page 192.)

\l__keys_groups_clist Used for storing and recovering the list of groups which apply to a key: set as a comma list but at one point we have to use this for a token list recovery.

```

14602 \clist_new:N \l__keys_groups_clist

```

(End definition for \l__keys_groups_clist.)

\l_keys_key_str The name of a key itself: needed when setting keys. The tl version is deprecated but
\l_keys_key_tl has to be handled manually.

```

14603 \str_new:N \l_keys_key_str
14604 \tl_new:N \l_keys_key_tl

```

(End definition for \l_keys_key_str and \l_keys_key_tl. These variables are documented on page 194.)

\l__keys_module_str The module for an entire set of keys.

```

14605 \str_new:N \l__keys_module_str

```

(End definition for \l__keys_module_str.)

\l__keys_no_value_bool A marker is needed internally to show if only a key or a key plus a value was seen: this is recorded here.

```

14606 \bool_new:N \l__keys_no_value_bool

```

(End definition for \l__keys_no_value_bool.)

\l__keys_only_known_bool Used to track if only “known” keys are being set.

```

14607 \bool_new:N \l__keys_only_known_bool

```

(End definition for \l__keys_only_known_bool.)

\l_keys_path_str The “path” of the current key is stored here: this is available to the programmer and so
\l_keys_path_tl is public. The older version is deprecated but has to be handled manually.

```

14608 \str_new:N \l_keys_path_str
14609 \tl_new:N \l_keys_path_tl

```


(End definition for `\l_keys_path_str` and `\l_keys_path_tl`. These variables are documented on page 194.)

`\l__keys_inherit_str`

14610 `\str_new:N \l__keys_inherit_str`

(End definition for `\l__keys_inherit_str`.)

`\l__keys_relative_tl` The relative path for passing keys back to the user. As this can be explicitly no-value, it must be a token list.

14611 `\tl_new:N \l__keys_relative_tl`

14612 `\tl_set:Nn \l__keys_relative_tl { \q_keys_no_value }`

(End definition for `\l__keys_relative_tl`.)

`\l__keys_property_str` The “property” begin set for a key at definition time is stored here.

14613 `\str_new:N \l__keys_property_str`

(End definition for `\l__keys_property_str`.)

`\l__keys_selective_bool` Two flags for using key groups: one to indicate that “selective” setting is active, a second
`\l__keys_filtered_bool` to specify which type (“opt-in” or “opt-out”).

14614 `\bool_new:N \l__keys_selective_bool`

14615 `\bool_new:N \l__keys_filtered_bool`

(End definition for `\l__keys_selective_bool` and `\l__keys_filtered_bool`.)

`\l__keys_selective_seq` The list of key groups being filtered in or out during selective setting.

14616 `\seq_new:N \l__keys_selective_seq`

(End definition for `\l__keys_selective_seq`.)

`\l__keys_unused_clist` Used when setting only some keys to store those left over.

14617 `\tl_new:N \l__keys_unused_clist`

(End definition for `\l__keys_unused_clist`.)

`\l_keys_value_tl` The value given for a key: may be empty if no value was given.

14618 `\tl_new:N \l_keys_value_tl`

(End definition for `\l_keys_value_tl`. This variable is documented on page 194.)

`\l__keys_tmp_bool` Scratch space.

`\l__keys_tmpa_tl` 14619 `\bool_new:N \l__keys_tmp_bool`

`\l__keys_tmpp_tl` 14620 `\tl_new:N \l__keys_tmpa_tl`

14621 `\tl_new:N \l__keys_tmpp_tl`

(End definition for `\l__keys_tmp_bool`, `\l__keys_tmpa_tl`, and `\l__keys_tmpp_tl`.)

22.2.1 Internal auxiliaries

`\s__keys_nil` Internal scan marks.

```
\s__keys_mark      14622 \scan_new:N \s__keys_nil
\s__keys_stop      14623 \scan_new:N \s__keys_mark
                   14624 \scan_new:N \s__keys_stop
```

(End definition for `\s__keys_nil`, `\s__keys_mark`, and `\s__keys_stop`.)

`\q__keys_no_value` Internal quarks.

```
14625 \quark_new:N \q__keys_no_value
```

(End definition for `\q__keys_no_value`.)

`__keys_quark_if_no_value_p:N` Branching quark conditional.

```
\__keys_quark_if_no_value:N $\underline{TF}$  14626 \__kernel_quark_new_conditional:Nn \__keys_quark_if_no_value:N { TF }
```

(End definition for `__keys_quark_if_no_value:N \underline{TF}` .)

22.3 The key defining mechanism

`\keys_define:nn` The public function for definitions is just a wrapper for the lower level mechanism, more or less. The outer function is designed to keep a track of the current module, to allow safe nesting. The module is set removing any leading / (which is not needed here).

```
14627 \cs_new_protected:Npn \keys_define:nn
14628 { \__keys_define:onn \l__keys_module_str }
14629 \cs_new_protected:Npn \__keys_define:nnn #1#2#3
14630 {
14631   \str_set:Nx \l__keys_module_str { \__keys_trim_spaces:n {#2} }
14632   \keyval_parse:NNn \__keys_define:n \__keys_define:nn {#3}
14633   \str_set:Nn \l__keys_module_str {#1}
14634 }
14635 \cs_generate_variant:Nn \__keys_define:nnn { o }
```

(End definition for `\keys_define:nn` and `__keys_define:nnn`. This function is documented on page 187.)

`__keys_define:n` The outer functions here record whether a value was given and then converge on a common internal mechanism. There is first a search for a property in the current key name, then a check to make sure it is known before the code hands off to the next step.

```
14636 \cs_new_protected:Npn \__keys_define:n #1
14637 {
14638   \bool_set_true:N \l__keys_no_value_bool
14639   \__keys_define_aux:nn {#1} { }
14640 }
14641 \cs_new_protected:Npn \__keys_define:nn #1#2
14642 {
14643   \bool_set_false:N \l__keys_no_value_bool
14644   \__keys_define_aux:nn {#1} {#2}
14645 }
14646 \cs_new_protected:Npn \__keys_define_aux:nn #1#2
14647 {
14648   \__keys_property_find:n {#1}
14649   \cs_if_exist:cTF { \c__keys_props_root_str \l__keys_property_str }
```

```

14650     { \__keys_define_code:n {#2} }
14651     {
14652         \str_if_empty:NF \l__keys_property_str
14653         {
14654             \__kernel_msg_error:nxxx { kernel } { key-property-unknown }
14655             \l__keys_property_str \l_keys_path_str
14656         }
14657     }
14658 }

```

(End definition for __keys_define:n, __keys_define:nn, and __keys_define_aux:nn.)

```

\__keys_property_find:n Searching for a property means finding the last . in the input, and storing the text before
\__keys_property_find_auxi:w and after it. Everything is turned into strings, so there is no problem using an x-type
    \__keys_property_find_auxii:w expansion. Since \__keys_trim_spaces:n will turn its argument into a string anyway,
\__keys_property_find_auxiii:w this function uses \cs_set_nopar:Npx instead of \tl_set:Nx to gain some speed.
    \__keys_property_find_auxiv:w
14659 \cs_new_protected:Npn \__keys_property_find:n #1
14660 {
14661     \cs_set_nopar:Npx \l__keys_property_str { \__keys_trim_spaces:n { #1 } }
14662     \exp_after:wN \__keys_property_find_auxi:w \l__keys_property_str
14663     \s__keys_nil \__keys_property_find_auxii:w
14664     . \s__keys_nil \__keys_property_find_err:w
14665 }
14666 \cs_new_protected:Npn \__keys_property_find_auxi:w #1 . #2 \s__keys_nil #3
14667 {
14668     #3 #1 \s__keys_mark #2 \s__keys_nil #3
14669 }
14670 \cs_new_protected:Npn \__keys_property_find_auxii:w
14671 #1 \s__keys_mark #2 \s__keys_nil \__keys_property_find_auxii:w . \s__keys_nil
14672 \__keys_property_find_err:w
14673 {
14674     \cs_set_nopar:Npx \l_keys_path_str
14675     { \str_if_empty:NF \l__keys_module_str { \l__keys_module_str / } #1 }
14676     \__keys_property_find_auxi:w #2 \s__keys_nil \__keys_property_find_auxiii:w . \s__keys_
14677     \__keys_property_find_auxiv:w
14678 }
14679 \cs_new_protected:Npn \__keys_property_find_auxiii:w #1 \s__keys_mark
14680 {
14681     \cs_set_nopar:Npx \l_keys_path_str { \l_keys_path_str . #1 }
14682     \__keys_property_find_auxi:w
14683 }
14684 \cs_new_protected:Npn \__keys_property_find_auxiv:w
14685 #1 \s__keys_nil \__keys_property_find_auxiii:w
14686 \s__keys_mark \s__keys_nil \__keys_property_find_auxiv:w
14687 {
14688     \cs_set_nopar:Npx \l__keys_property_str { . #1 }
14689     \cs_set_nopar:Npx \l_keys_path_str
14690     { \exp_after:wN \__keys_trim_spaces:n \exp_after:wN { \l_keys_path_str } }
14691     \tl_set_eq:NN \l_keys_path_tl \l_keys_path_str
14692 }
14693 \cs_new_protected:Npn \__keys_property_find_err:w
14694 #1 \s__keys_nil #2 \__keys_property_find_err:w
14695 {
14696     \str_clear:N \l__keys_property_str

```

```

14697     \__kernel_msg_error:nnn { kernel } { key-no-property } {#1}
14698 }

```

(End definition for __keys_property_find:n and others.)

__keys_define_code:n Two possible cases. If there is a value for the key, then just use the function. If not, then
__keys_define_code:w a check to make sure there is no need for a value with the property. If there should be
one then complain, otherwise execute it. There is no need to check for a : as if it was
missing the earlier tests would have failed.

```

14699 \cs_new_protected:Npn \__keys_define_code:n #1
14700 {
14701     \bool_if:NTF \l__keys_no_value_bool
14702     {
14703         \exp_after:wN \__keys_define_code:w
14704         \l__keys_property_str \s__keys_stop
14705         { \use:c { \c__keys_props_root_str \l__keys_property_str } }
14706         {
14707             \__kernel_msg_error:nnxx { kernel } { key-property-requires-value }
14708             \l__keys_property_str \l_keys_path_str
14709         }
14710     }
14711     { \use:c { \c__keys_props_root_str \l__keys_property_str } {#1} }
14712 }
14713 \exp_last_unbraced:NNNNo
14714 \cs_new:Npn \__keys_define_code:w #1 \c_colon_str #2 \s__keys_stop
14715 { \tl_if_empty:NTF {#2} }

```

(End definition for __keys_define_code:n and __keys_define_code:w.)

22.4 Turning properties into actions

__keys_bool_set:Nn Boolean keys are really just choices, but all done by hand. The second argument here is
__keys_bool_set:cn the scope: either empty or g for global.

```

14716 \cs_new_protected:Npn \__keys_bool_set:Nn #1#2
14717 {
14718     \bool_if_exist:NF #1 { \bool_new:N #1 }
14719     \__keys_choice_make:
14720     \__keys_cmd_set:nx { \l_keys_path_str / true }
14721     { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
14722     \__keys_cmd_set:nx { \l_keys_path_str / false }
14723     { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
14724     \__keys_cmd_set:nn { \l_keys_path_str / unknown }
14725     {
14726         \__kernel_msg_error:nnx { kernel } { boolean-values-only }
14727         \l_keys_key_str
14728     }
14729     \__keys_default_set:n { true }
14730 }
14731 \cs_generate_variant:Nn \__keys_bool_set:Nn { c }

```

(End definition for __keys_bool_set:Nn.)

_keys_bool_set_inverse:Nn

Inverse boolean setting is much the same.

_keys_bool_set_inverse:cn

```
14732 \\cs_new_protected:Npn \\_keys_bool_set_inverse:Nn #1#2
14733 {
14734   \\bool_if_exist:NF #1 { \\bool_new:N #1 }
14735   \\_keys_choice_make:
14736   \\_keys_cmd_set:nx { \\l_keys_path_str / true }
14737   { \\exp_not:c { bool_ #2 set_false:N } \\exp_not:N #1 }
14738   \\_keys_cmd_set:nx { \\l_keys_path_str / false }
14739   { \\exp_not:c { bool_ #2 set_true:N } \\exp_not:N #1 }
14740   \\_keys_cmd_set:nn { \\l_keys_path_str / unknown }
14741   {
14742     \\_kernel_msg_error:nxx { kernel } { boolean-values-only }
14743     \\l_keys_key_str
14744   }
14745   \\_keys_default_set:n { true }
14746 }
14747 \\cs_generate_variant:Nn \\_keys_bool_set_inverse:Nn { c }
```

(End definition for _keys_bool_set_inverse:Nn.)

_keys_choice_make:

To make a choice from a key, two steps: set the code, and set the unknown key. As multichoice and choices are essentially the same bar one function, the code is given together.

_keys_multichoice_make:

_keys_choice_make:N

_keys_choice_make_aux:N

```
14748 \\cs_new_protected:Npn \\_keys_choice_make:
14749 { \\_keys_choice_make:N \\_keys_choice_find:n }
14750 \\cs_new_protected:Npn \\_keys_multichoice_make:
14751 { \\_keys_choice_make:N \\_keys_multichoice_find:n }
14752 \\cs_new_protected:Npn \\_keys_choice_make:N #1
14753 {
14754   \\cs_if_exist:cTF
14755   { \\c__keys_type_root_str \\_keys_parent:o \\l_keys_path_str }
14756   {
14757     \\str_if_eq:vnTF
14758     { \\c__keys_type_root_str \\_keys_parent:o \\l_keys_path_str }
14759     { choice }
14760     {
14761       \\_kernel_msg_error:nxxx { kernel } { nested-choice-key }
14762       \\l_keys_path_tl { \\_keys_parent:o \\l_keys_path_str }
14763     }
14764     { \\_keys_choice_make_aux:N #1 }
14765   }
14766   { \\_keys_choice_make_aux:N #1 }
14767 }
14768 \\cs_new_protected:Npn \\_keys_choice_make_aux:N #1
14769 {
14770   \\cs_set_nopar:cpn { \\c__keys_type_root_str \\l_keys_path_str }
14771   { choice }
14772   \\_keys_cmd_set:nn \\l_keys_path_str { #1 {##1} }
14773   \\_keys_cmd_set:nn { \\l_keys_path_str / unknown }
14774   {
14775     \\_kernel_msg_error:nxxx { kernel } { key-choice-unknown }
14776     \\l_keys_path_str {##1}
14777   }
14778 }
```

(End definition for `_keys_choice_make:` and others.)

Auto-generating choices means setting up the root key as a choice, then defining each choice in turn.

```

\_keys_choices_make:nn
\_keys_multichoices_make:nn
\_keys_choices_make:Nnn
14779 \cs_new_protected:Npn \_keys_choices_make:nn
14780   { \_keys_choices_make:Nnn \_keys_choice_make: }
14781 \cs_new_protected:Npn \_keys_multichoices_make:nn
14782   { \_keys_choices_make:Nnn \_keys_multichoice_make: }
14783 \cs_new_protected:Npn \_keys_choices_make:Nnn #1#2#3
14784   {
14785     #1
14786     \int_zero:N \l_keys_choice_int
14787     \clist_map_inline:nn {#2}
14788     {
14789       \int_incr:N \l_keys_choice_int
14790       \_keys_cmd_set:nx
14791         { \l_keys_path_str / \_keys_trim_spaces:n {##1} }
14792         {
14793           \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
14794           \int_set:Nn \exp_not:N \l_keys_choice_int
14795             { \int_use:N \l_keys_choice_int }
14796           \exp_not:n {#3}
14797         }
14798     }
14799   }

```

(End definition for `_keys_choices_make:nn`, `_keys_multichoices_make:nn`, and `_keys_choices_make:Nnn`.)

Setting the code for a key first logs if appropriate that we are defining a new key, then saves the code.

```

\_keys_cmd_set:nn
\_keys_cmd_set:nx
\_keys_cmd_set:Vn
\_keys_cmd_set:Vo
14800 \cs_new_protected:Npn \_keys_cmd_set:nn #1#2
14801   { \cs_set_protected:cpn { \c__keys_code_root_str #1 } ##1 {#2} }
14802 \cs_generate_variant:Nn \_keys_cmd_set:nn { nx , Vn , Vo }

```

(End definition for `_keys_cmd_set:nn`.)

Creating control sequences is a bit more tricky than other cases as we need to pick up the `p` argument. To make the internals look clearer, the trailing `n` argument here is just for appearance.

```

\_keys_cs_set:NNpn
\_keys_cs_set:Ncpn
14803 \cs_new_protected:Npn \_keys_cs_set:NNpn #1#2#3#
14804   {
14805     \cs_set_protected:cpx { \c__keys_code_root_str \l_keys_path_str } ##1
14806     { #1 \exp_not:N #2 \exp_not:n {#3} {##1} }
14807     \use_none:n
14808   }
14809 \cs_generate_variant:Nn \_keys_cs_set:NNpn { Nc }

```

(End definition for `_keys_cs_set:NNpn`.)

Setting a default value is easy. These are stored using `\cs_set:cpx` as this avoids any worries about whether a token list exists.

```

\_keys_default_set:n
14810 \cs_new_protected:Npn \_keys_default_set:n #1
14811   {

```

```

14812 \tl_if_empty:nTF {#1}
14813 {
14814   \cs_set_eq:cN
14815   { \c__keys_default_root_str \l_keys_path_str }
14816   \tex_undefined:D
14817 }
14818 {
14819   \cs_set_nopar:cpx
14820   { \c__keys_default_root_str \l_keys_path_str }
14821   { \exp_not:n {#1} }
14822   \__keys_value_requirement:nn { required } { false }
14823 }
14824 }

```

(End definition for __keys_default_set:n.)

__keys_groups_set:n Assigning a key to one or more groups uses comma lists. As the list of groups only exists if there is anything to do, the setting is done using a scratch list. For the usual grouping reasons we use the low-level approach to undefining a list. We also use the low-level approach for the other case to avoid tripping up the `check-declarations` code.

```

14825 \cs_new_protected:Npn \__keys_groups_set:n #1
14826 {
14827   \clist_set:Nn \l__keys_groups_clist {#1}
14828   \clist_if_empty:NTF \l__keys_groups_clist
14829   {
14830     \cs_set_eq:cN { \c__keys_groups_root_str \l_keys_path_str }
14831     \tex_undefined:D
14832   }
14833   {
14834     \cs_set_eq:cN { \c__keys_groups_root_str \l_keys_path_str }
14835     \l__keys_groups_clist
14836   }
14837 }

```

(End definition for __keys_groups_set:n.)

__keys_inherit:n Inheritance means ignoring anything already said about the key: zap the lot and set up.

```

14838 \cs_new_protected:Npn \__keys_inherit:n #1
14839 {
14840   \__keys_undefine:
14841   \cs_set_nopar:cpn { \c__keys_inherit_root_str \l_keys_path_str } {#1}
14842 }

```

(End definition for __keys_inherit:n.)

__keys_initialise:n A set up for initialisation: just run the code if it exists.

```

14843 \cs_new_protected:Npn \__keys_initialise:n #1
14844 {
14845   \cs_if_exist:cTF
14846   { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
14847   { \__keys_execute_inherit: }
14848   {
14849     \str_clear:N \l__keys_inherit_str
14850     \cs_if_exist:cT { \c__keys_code_root_str \l_keys_path_str }

```

```

14851         { \__keys_execute:nn \l_keys_path_str {#1} }
14852     }
14853 }

```

(End definition for __keys_initialise:n.)

__keys_meta_make:n To create a meta-key, simply set up to pass data through.
__keys_meta_make:nn

```

14854 \cs_new_protected:Npn \__keys_meta_make:n #1
14855 {
14856     \__keys_cmd_set:Vo \l_keys_path_str
14857     {
14858         \exp_after:wN \keys_set:nn \exp_after:wN { \l__keys_module_str } {#1}
14859     }
14860 }
14861 \cs_new_protected:Npn \__keys_meta_make:nn #1#2
14862 { \__keys_cmd_set:Vn \l_keys_path_str { \keys_set:nn {#1} {#2} } }

```

(End definition for __keys_meta_make:n and __keys_meta_make:nn.)

__keys_prop_put:Nn Much the same as other variables, but needs a dedicated auxiliary.
__keys_prop_put:cn

```

14863 \cs_new_protected:Npn \__keys_prop_put:Nn #1#2
14864 {
14865     \prop_if_exist:NF #1 { \prop_new:N #1 }
14866     \exp_after:wN \__keys_find_key_module:wNN \l_keys_path_str \s__keys_stop
14867     \l__keys_tmpa_tl \l__keys_tmpb_tl
14868     \__keys_cmd_set:nx \l_keys_path_str
14869     {
14870         \exp_not:c { prop_ #2 put:Nnn }
14871         \exp_not:N #1
14872         { \l__keys_tmpb_tl }
14873         \exp_not:n { {##1} }
14874     }
14875 }
14876 \cs_generate_variant:Nn \__keys_prop_put:Nn { c }

```

(End definition for __keys_prop_put:Nn.)

__keys_undefine: Undefining a key has to be done without \cs_undefine:c as that function acts globally.

```

14877 \cs_new_protected:Npn \__keys_undefine:
14878 {
14879     \clist_map_inline:nn
14880     { code , default , groups , inherit , type , validate }
14881     {
14882         \cs_set_eq:cN
14883         { \tl_use:c { c__keys_ ##1 _root_str } \l_keys_path_str }
14884         \tex_undefined:D
14885     }
14886 }

```

(End definition for __keys_undefine:.)

__keys_value_requirement:nn Validating key input is done using a second function which runs before the main key
__keys_validate_forbidden: code. Setting that up means setting it equal to a generic stub which does the check. This
__keys_validate_required: approach makes the lookup very fast at the cost of one additional csname per key that
needs it. The cleanup here has to know the structure of the following code.


```

14887 \cs_new_protected:Npn \__keys_value_requirement:nn #1#2
14888 {
14889   \str_case:nnF {#2}
14890   {
14891     { true }
14892     {
14893       \cs_set_eq:cc
14894       { \c__keys_validate_root_str \l_keys_path_str }
14895       { __keys_validate_ #1 : }
14896     }
14897     { false }
14898     {
14899       \cs_if_eq:ccT
14900       { \c__keys_validate_root_str \l_keys_path_str }
14901       { __keys_validate_ #1 : }
14902       {
14903         \cs_set_eq:cN
14904         { \c__keys_validate_root_str \l_keys_path_str }
14905         \tex_undefined:D
14906       }
14907     }
14908   }
14909   {
14910     \__kernel_msg_error:nnx { kernel }
14911     { key-property-boolean-values-only }
14912     { .value_ #1 :n }
14913   }
14914 }
14915 \cs_new_protected:Npn \__keys_validate_forbidden:
14916 {
14917   \bool_if:NF \l__keys_no_value_bool
14918   {
14919     \__kernel_msg_error:nnxx { kernel } { value-forbidden }
14920     \l_keys_path_str \l_keys_value_tl
14921     \use_none:nnn
14922   }
14923 }
14924 \cs_new_protected:Npn \__keys_validate_required:
14925 {
14926   \bool_if:NT \l__keys_no_value_bool
14927   {
14928     \__kernel_msg_error:nnx { kernel } { value-required }
14929     \l_keys_path_str
14930     \use_none:nnn
14931   }
14932 }

```

(End definition for __keys_value_requirement:nn, __keys_validate_forbidden:, and __keys_validate_required:.)

```

\__keys_variable_set:NnnN
\__keys_variable_set:cnnN
  \_keys_variable_set_required:NnnN
  \_keys_variable_set_required:cnnN

```

Setting a variable takes the type and scope separately so that it is easy to make a new variable if needed.

```

14933 \cs_new_protected:Npn \__keys_variable_set:NnnN #1#2#3#4
14934 {

```

```

14935     \use:c { #2_if_exist:Nf } #1 { \use:c { #2_new:N } #1 }
14936     \__keys_cmd_set:nx \l_keys_path_str
14937     {
14938         \exp_not:c { #2 _ #3 set:N #4 }
14939         \exp_not:N #1
14940         \exp_not:n { {##1} }
14941     }
14942 }
14943 \cs_generate_variant:Nn \__keys_variable_set:NnnN { c }
14944 \cs_new_protected:Npn \__keys_variable_set_required:NnnN #1#2#3#4
14945 {
14946     \__keys_variable_set:NnnN #1 {#2} {#3} #4
14947     \__keys_value_requirement:nn { required } { true }
14948 }
14949 \cs_generate_variant:Nn \__keys_variable_set_required:NnnN { c }

```

(End definition for __keys_variable_set:NnnN and __keys_variable_set_required:NnnN.)

22.5 Creating key properties

The key property functions are all wrappers for internal functions, meaning that things stay readable and can also be altered later on.

Importantly, while key properties have “normal” argument specs, the underlying code always supplies one braced argument to these. As such, argument expansion is handled by hand rather than using the standard tools. This shows up particularly for the two-argument properties, where things would otherwise go badly wrong.

```

.bool_set:N One function for this.
.bool_set:c 14950 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set:N } #1
.bool_gset:N 14951 { \__keys_bool_set:Nn #1 { } }
.bool_gset:c 14952 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set:c } #1
14953 { \__keys_bool_set:cn {#1} { } }
14954 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset:N } #1
14955 { \__keys_bool_set:Nn #1 { g } }
14956 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset:c } #1
14957 { \__keys_bool_set:cn {#1} { g } }

```

(End definition for .bool_set:N and .bool_gset:N. These functions are documented on page 188.)

```

.bool_set_inverse:N One function for this.
.bool_set_inverse:c 14958 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set_inverse:N } #1
.bool_gset_inverse:N 14959 { \__keys_bool_set_inverse:Nn #1 { } }
.bool_gset_inverse:c 14960 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set_inverse:c } #1
14961 { \__keys_bool_set_inverse:cn {#1} { } }
14962 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset_inverse:N } #1
14963 { \__keys_bool_set_inverse:Nn #1 { g } }
14964 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset_inverse:c } #1
14965 { \__keys_bool_set_inverse:cn {#1} { g } }

```

(End definition for .bool_set_inverse:N and .bool_gset_inverse:N. These functions are documented on page 188.)

```

.choice: Making a choice is handled internally, as it is also needed by .generate_choices:n.
14966 \cs_new_protected:cpn { \c__keys_props_root_str .choice: }
14967 { \__keys_choice_make: }

```

(End definition for `.choice:`. This function is documented on page 188.)

.choices:nn For auto-generation of a series of mutually-exclusive choices. Here, #1 consists of two
.choices:Vn separate arguments, hence the slightly odd-looking implementation.

```
.choices:on
14968 \cs_new_protected:cpn { \c__keys_props_root_str .choices:nn } #1
.choices:xn 14969 { \__keys_choices_make:nn #1 }
14970 \cs_new_protected:cpn { \c__keys_props_root_str .choices:Vn } #1
14971 { \exp_args:NV \__keys_choices_make:nn #1 }
14972 \cs_new_protected:cpn { \c__keys_props_root_str .choices:on } #1
14973 { \exp_args:No \__keys_choices_make:nn #1 }
14974 \cs_new_protected:cpn { \c__keys_props_root_str .choices:xn } #1
14975 { \exp_args:Nx \__keys_choices_make:nn #1 }
```

(End definition for `.choices:nn`. This function is documented on page 188.)

.code:n Creating code is simply a case of passing through to the underlying set function.

```
14976 \cs_new_protected:cpn { \c__keys_props_root_str .code:n } #1
14977 { \__keys_cmd_set:nn \l_keys_path_str {#1} }
```

(End definition for `.code:n`. This function is documented on page 188.)

.clist_set:N

```
.clist_set:c 14978 \cs_new_protected:cpn { \c__keys_props_root_str .clist_set:N } #1
14979 { \__keys_variable_set:NnnN #1 { clist } { } n }
.clist_gset:N 14980 \cs_new_protected:cpn { \c__keys_props_root_str .clist_set:c } #1
14981 { \__keys_variable_set:cnN {#1} { clist } { } n }
.clist_gset:c 14982 \cs_new_protected:cpn { \c__keys_props_root_str .clist_gset:N } #1
14983 { \__keys_variable_set:NnnN #1 { clist } { g } n }
14984 \cs_new_protected:cpn { \c__keys_props_root_str .clist_gset:c } #1
14985 { \__keys_variable_set:cnN {#1} { clist } { g } n }
```

(End definition for `.clist_set:N` and `.clist_gset:N`. These functions are documented on page 188.)

.cs_set:Np

```
.cs_set:cp 14986 \cs_new_protected:cpn { \c__keys_props_root_str .cs_set:Np } #1
14987 { \__keys_cs_set:NNpn \cs_set:Npn #1 { } }
.cs_set_protected:Np 14988 \cs_new_protected:cpn { \c__keys_props_root_str .cs_set:cp } #1
14989 { \__keys_cs_set:Ncpn \cs_set:Npn #1 { } }
.cs_gset:Np 14990 \cs_new_protected:cpn { \c__keys_props_root_str .cs_set_protected:Np } #1
14991 { \__keys_cs_set:NNpn \cs_set_protected:Npn #1 { } }
.cs_gset:cp 14992 \cs_new_protected:cpn { \c__keys_props_root_str .cs_set_protected:cp } #1
14993 { \__keys_cs_set:Ncpn \cs_set_protected:Npn #1 { } }
.cs_gset_protected:Np 14994 \cs_new_protected:cpn { \c__keys_props_root_str .cs_gset:Np } #1
14995 { \__keys_cs_set:NNpn \cs_gset:Npn #1 { } }
14996 \cs_new_protected:cpn { \c__keys_props_root_str .cs_gset:cp } #1
14997 { \__keys_cs_set:Ncpn \cs_gset:Npn #1 { } }
14998 \cs_new_protected:cpn { \c__keys_props_root_str .cs_gset_protected:Np } #1
14999 { \__keys_cs_set:NNpn \cs_gset_protected:Npn #1 { } }
15000 \cs_new_protected:cpn { \c__keys_props_root_str .cs_gset_protected:cp } #1
15001 { \__keys_cs_set:Ncpn \cs_gset_protected:Npn #1 { } }
```

(End definition for `.cs_set:Np` and others. These functions are documented on page 188.)

.default:n Expansion is left to the internal functions.

```
.default:V 15002 \cs_new_protected:cpn { \c__keys_props_root_str .default:n } #1
.default:o 15003 { \__keys_default_set:n {#1} }
.default:x 15004 \cs_new_protected:cpn { \c__keys_props_root_str .default:V } #1
15005 { \exp_args:NV \__keys_default_set:n #1 }
15006 \cs_new_protected:cpn { \c__keys_props_root_str .default:o } #1
15007 { \exp_args:No \__keys_default_set:n {#1} }
15008 \cs_new_protected:cpn { \c__keys_props_root_str .default:x } #1
15009 { \exp_args:Nx \__keys_default_set:n {#1} }
```

(End definition for .default:n. This function is documented on page 189.)

.dim_set:N Setting a variable is very easy: just pass the data along.

```
.dim_set:c 15010 \cs_new_protected:cpn { \c__keys_props_root_str .dim_set:N } #1
.dim_gset:N 15011 { \__keys_variable_set_required:NnnN #1 { dim } { } n }
15012 \cs_new_protected:cpn { \c__keys_props_root_str .dim_set:c } #1
15013 { \__keys_variable_set_required:cnnN {#1} { dim } { } n }
15014 \cs_new_protected:cpn { \c__keys_props_root_str .dim_gset:N } #1
15015 { \__keys_variable_set_required:NnnN #1 { dim } { g } n }
15016 \cs_new_protected:cpn { \c__keys_props_root_str .dim_gset:c } #1
15017 { \__keys_variable_set_required:cnnN {#1} { dim } { g } n }
```

(End definition for .dim_set:N and .dim_gset:N. These functions are documented on page 189.)

.fp_set:N Setting a variable is very easy: just pass the data along.

```
.fp_set:c 15018 \cs_new_protected:cpn { \c__keys_props_root_str .fp_set:N } #1
.fp_gset:N 15019 { \__keys_variable_set_required:NnnN #1 { fp } { } n }
15020 \cs_new_protected:cpn { \c__keys_props_root_str .fp_set:c } #1
15021 { \__keys_variable_set_required:cnnN {#1} { fp } { } n }
15022 \cs_new_protected:cpn { \c__keys_props_root_str .fp_gset:N } #1
15023 { \__keys_variable_set_required:NnnN #1 { fp } { g } n }
15024 \cs_new_protected:cpn { \c__keys_props_root_str .fp_gset:c } #1
15025 { \__keys_variable_set_required:cnnN {#1} { fp } { g } n }
```

(End definition for .fp_set:N and .fp_gset:N. These functions are documented on page 189.)

.groups:n A single property to create groups of keys.

```
15026 \cs_new_protected:cpn { \c__keys_props_root_str .groups:n } #1
15027 { \__keys_groups_set:n {#1} }
```

(End definition for .groups:n. This function is documented on page 189.)

.inherit:n Nothing complex: only one variant at the moment!

```
15028 \cs_new_protected:cpn { \c__keys_props_root_str .inherit:n } #1
15029 { \__keys_inherit:n {#1} }
```

(End definition for .inherit:n. This function is documented on page 189.)

.initial:n The standard hand-off approach.

```
.initial:V 15030 \cs_new_protected:cpn { \c__keys_props_root_str .initial:n } #1
.initial:o 15031 { \__keys_initialise:n {#1} }
.initial:x 15032 \cs_new_protected:cpn { \c__keys_props_root_str .initial:V } #1
15033 { \exp_args:NV \__keys_initialise:n #1 }
15034 \cs_new_protected:cpn { \c__keys_props_root_str .initial:o } #1
15035 { \exp_args:No \__keys_initialise:n {#1} }
15036 \cs_new_protected:cpn { \c__keys_props_root_str .initial:x } #1
15037 { \exp_args:Nx \__keys_initialise:n {#1} }
```

(End definition for `.initial:n`. This function is documented on page 190.)

.int_set:N Setting a variable is very easy: just pass the data along.

```
.int_set:c 15038 \cs_new_protected:cpn { \c__keys_props_root_str .int_set:N } #1
           15039 { \__keys_variable_set_required:NnnN #1 { int } { } n }
.int_gset:N
.int_gset:c 15040 \cs_new_protected:cpn { \c__keys_props_root_str .int_set:c } #1
           15041 { \__keys_variable_set_required:cnnN {#1} { int } { } n }
           15042 \cs_new_protected:cpn { \c__keys_props_root_str .int_gset:N } #1
           15043 { \__keys_variable_set_required:NnnN #1 { int } { g } n }
           15044 \cs_new_protected:cpn { \c__keys_props_root_str .int_gset:c } #1
           15045 { \__keys_variable_set_required:cnnN {#1} { int } { g } n }
```

(End definition for `.int_set:N` and `.int_gset:N`. These functions are documented on page 190.)

.meta:n Making a meta is handled internally.

```
15046 \cs_new_protected:cpn { \c__keys_props_root_str .meta:n } #1
15047 { \__keys_meta_make:n {#1} }
```

(End definition for `.meta:n`. This function is documented on page 190.)

.meta:nn Meta with path: potentially lots of variants, but for the moment no so many defined.

```
15048 \cs_new_protected:cpn { \c__keys_props_root_str .meta:nn } #1
15049 { \__keys_meta_make:nn #1 }
```

(End definition for `.meta:nn`. This function is documented on page 190.)

.multichoice: The same idea as `.choice:` and `.choices:nn`, but where more than one choice is allowed.

```
.multichoices:nn 15050 \cs_new_protected:cpn { \c__keys_props_root_str .multichoice: }
.multichoices:Vn 15051 { \__keys_multichoice_make: }
.multichoices:on 15052 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:nn } #1
.multichoices:xn 15053 { \__keys_multichoices_make:nn #1 }
           15054 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:Vn } #1
           15055 { \exp_args:NV \__keys_multichoices_make:nn #1 }
           15056 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:on } #1
           15057 { \exp_args:No \__keys_multichoices_make:nn #1 }
           15058 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:xn } #1
           15059 { \exp_args:Nx \__keys_multichoices_make:nn #1 }
```

(End definition for `.multichoice:` and `.multichoices:nn`. These functions are documented on page 190.)

.muskip_set:N Setting a variable is very easy: just pass the data along.

```
.muskip_set:c 15060 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_set:N } #1
.muskip_gset:N 15061 { \__keys_variable_set_required:NnnN #1 { muskip } { } n }
.muskip_gset:c 15062 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_set:c } #1
           15063 { \__keys_variable_set_required:cnnN {#1} { muskip } { } n }
           15064 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_gset:N } #1
           15065 { \__keys_variable_set_required:NnnN #1 { muskip } { g } n }
           15066 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_gset:c } #1
           15067 { \__keys_variable_set_required:cnnN {#1} { muskip } { g } n }
```

(End definition for `.muskip_set:N` and `.muskip_gset:N`. These functions are documented on page 190.)

.prop_put:N Setting a variable is very easy: just pass the data along.

```

15068 \cs_new_protected:cpn { \c__keys_props_root_str .prop_put:N } #1
15069 { \__keys_prop_put:Nn #1 { } }
.prop_gput:N
15070 \cs_new_protected:cpn { \c__keys_props_root_str .prop_put:c } #1
15071 { \__keys_prop_put:cn {#1} { } }
.prop_gput:c
15072 \cs_new_protected:cpn { \c__keys_props_root_str .prop_gput:N } #1
15073 { \__keys_prop_put:Nn #1 { g } }
15074 \cs_new_protected:cpn { \c__keys_props_root_str .prop_gput:c } #1
15075 { \__keys_prop_put:cn {#1} { g } }

```

(End definition for .prop_put:N and .prop_gput:N. These functions are documented on page 190.)

.skip_set:N Setting a variable is very easy: just pass the data along.

```

15076 \cs_new_protected:cpn { \c__keys_props_root_str .skip_set:N } #1
15077 { \__keys_variable_set_required:NnnN #1 { skip } { } n }
.skip_gset:N
15078 \cs_new_protected:cpn { \c__keys_props_root_str .skip_set:c } #1
15079 { \__keys_variable_set_required:cnnN {#1} { skip } { } n }
.skip_gset:c
15080 \cs_new_protected:cpn { \c__keys_props_root_str .skip_gset:N } #1
15081 { \__keys_variable_set_required:NnnN #1 { skip } { g } n }
15082 \cs_new_protected:cpn { \c__keys_props_root_str .skip_gset:c } #1
15083 { \__keys_variable_set_required:cnnN {#1} { skip } { g } n }

```

(End definition for .skip_set:N and .skip_gset:N. These functions are documented on page 191.)

.tl_set:N Setting a variable is very easy: just pass the data along.

```

15084 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set:N } #1
15085 { \__keys_variable_set:NnnN #1 { tl } { } n }
.tl_gset:N
15086 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set:c } #1
15087 { \__keys_variable_set:cnnN {#1} { tl } { } n }
.tl_set_x:N
15088 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set_x:N } #1
15089 { \__keys_variable_set:NnnN #1 { tl } { } x }
.tl_set_x:c
15090 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set_x:c } #1
15091 { \__keys_variable_set:cnnN {#1} { tl } { } x }
.tl_gset_x:N
15092 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset:N } #1
15093 { \__keys_variable_set:NnnN #1 { tl } { g } n }
.tl_gset_x:c
15094 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset:c } #1
15095 { \__keys_variable_set:cnnN {#1} { tl } { g } n }
15096 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_x:N } #1
15097 { \__keys_variable_set:NnnN #1 { tl } { g } x }
15098 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_x:c } #1
15099 { \__keys_variable_set:cnnN {#1} { tl } { g } x }

```

(End definition for .tl_set:N and others. These functions are documented on page 191.)

.undefine: Another simple wrapper.

```

15100 \cs_new_protected:cpn { \c__keys_props_root_str .undefine: }
15101 { \__keys_undefine: }

```

(End definition for .undefine:. This function is documented on page 191.)

.value_forbidden:n These are very similar, so both call the same function.

```

15102 \cs_new_protected:cpn { \c__keys_props_root_str .value_forbidden:n } #1
15103 { \__keys_value_requirement:nn { forbidden } {#1} }
.value_required:n
15104 \cs_new_protected:cpn { \c__keys_props_root_str .value_required:n } #1
15105 { \__keys_value_requirement:nn { required } {#1} }

```

(End definition for .value_forbidden:n and .value_required:n. These functions are documented on page 191.)

22.6 Setting keys

```

\keys_set:nn A simple wrapper allowing for nesting.
\keys_set:nV 15106 \cs_new_protected:Npn \keys_set:nn #1#2
\keys_set:nv 15107 {
\keys_set:no 15108   \use:x
\__keys_set:nn 15109   {
\__keys_set:nnn 15110     \bool_set_false:N \exp_not:N \l__keys_only_known_bool
15111     \bool_set_false:N \exp_not:N \l__keys_filtered_bool
15112     \bool_set_false:N \exp_not:N \l__keys_selective_bool
15113     \tl_set:Nn \exp_not:N \l__keys_relative_tl
15114     { \exp_not:N \q__keys_no_value }
15115     \__keys_set:nn \exp_not:n { {#1} {#2} }
15116     \bool_if:NT \l__keys_only_known_bool
15117     { \bool_set_true:N \exp_not:N \l__keys_only_known_bool }
15118     \bool_if:NT \l__keys_filtered_bool
15119     { \bool_set_true:N \exp_not:N \l__keys_filtered_bool }
15120     \bool_if:NT \l__keys_selective_bool
15121     { \bool_set_true:N \exp_not:N \l__keys_selective_bool }
15122     \tl_set:Nn \exp_not:N \l__keys_relative_tl
15123     { \exp_not:o \l__keys_relative_tl }
15124   }
15125 }
15126 \cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
15127 \cs_new_protected:Npn \__keys_set:nn #1#2
15128 { \exp_args:No \__keys_set:nnn \l__keys_module_str {#1} {#2} }
15129 \cs_new_protected:Npn \__keys_set:nnn #1#2#3
15130 {
15131   \str_set:Nx \l__keys_module_str { \__keys_trim_spaces:n {#2} }
15132   \keyval_parse:NNn \__keys_set_keyval:n \__keys_set_keyval:nn {#3}
15133   \str_set:Nn \l__keys_module_str {#1}
15134 }

```

(End definition for `\keys_set:nn`, `__keys_set:nn`, and `__keys_set:nnn`. This function is documented on page 194.)

```

\keys_set_known:nnnN Setting known keys simply means setting the appropriate flag, then running the standard
\keys_set_known:nVN code. To allow for nested setting, any existing value of \l__keys_unused_clist is saved
\keys_set_known:nvN on the stack and reset afterwards. Note that for speed/simplicity reasons we use a tl
\keys_set_known:noN operation to set the clist here!
\keys_set_known:nnnN 15135 \cs_new_protected:Npn \keys_set_known:nnN #1#2#3
\keys_set_known:nVnN 15136 {
\keys_set_known:nvnN 15137   \exp_args:No \__keys_set_known:nnnnN
\keys_set_known:nonN 15138   \l__keys_unused_clist \q__keys_no_value {#1} {#2} #3
\__keys_set_known:nnnnN 15139 }
\keys_set_known:nn 15140 \cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
\keys_set_known:nV 15141 \cs_new_protected:Npn \keys_set_known:nnnN #1#2#3#4
\keys_set_known:nv 15142 {
\keys_set_known:no 15143   \exp_args:No \__keys_set_known:nnnnN
\__keys_set_known:nnn 15144   \l__keys_unused_clist {#3} {#1} {#2} #4
15145 }
15146 \cs_generate_variant:Nn \keys_set_known:nnnN { nV , nv , no }
15147 \cs_new_protected:Npn \__keys_set_known:nnnnN #1#2#3#4#5
15148 {

```

```

15149     \clist_clear:N \l__keys_unused_clist
15150     \__keys_set_known:nnn {#2} {#3} {#4}
15151     \tl_set:Nx #5 { \exp_not:o \l__keys_unused_clist }
15152     \tl_set:Nn \l__keys_unused_clist {#1}
15153 }
15154 \cs_new_protected:Npn \keys_set_known:nn #1#2
15155 { \__keys_set_known:nnn \q__keys_no_value {#1} {#2} }
15156 \cs_generate_variant:Nn \keys_set_known:nn { nV , nv , no }
15157 \cs_new_protected:Npn \__keys_set_known:nnn #1#2#3
15158 {
15159     \use:x
15160     {
15161         \bool_set_true:N \exp_not:N \l__keys_only_known_bool
15162         \bool_set_false:N \exp_not:N \l__keys_filtered_bool
15163         \bool_set_false:N \exp_not:N \l__keys_selective_bool
15164         \tl_set:Nn \exp_not:N \l__keys_relative_tl { \exp_not:n {#1} }
15165         \__keys_set:nn \exp_not:n { {#2} {#3} }
15166         \bool_if:NF \l__keys_only_known_bool
15167         { \bool_set_false:N \exp_not:N \l__keys_only_known_bool }
15168         \bool_if:NT \l__keys_filtered_bool
15169         { \bool_set_true:N \exp_not:N \l__keys_filtered_bool }
15170         \bool_if:NT \l__keys_selective_bool
15171         { \bool_set_true:N \exp_not:N \l__keys_selective_bool }
15172         \tl_set:Nn \exp_not:N \l__keys_relative_tl
15173         { \exp_not:o \l__keys_relative_tl }
15174     }
15175 }

```

(End definition for \keys_set_known:nnN and others. These functions are documented on page 195.)

\keys_set_filter:nnnN The idea of setting keys in a selective manner again uses flags wrapped around the basic code. The comments on \keys_set_known:nnN also apply here. We have a bit more shuffling to do to keep everything nestable.

```

15176 \cs_new_protected:Npn \keys_set_filter:nnnN #1#2#3#4
15177 {
15178     \exp_args:No \__keys_set_filter:nnnnN
15179     \l__keys_unused_clist
15180     \q__keys_no_value {#1} {#2} {#3} #4
15181 }
15182 \cs_generate_variant:Nn \keys_set_filter:nnnN { nnV , nnv , nno }
15183 \cs_new_protected:Npn \keys_set_filter:nnnnN #1#2#3#4#5
15184 {
15185     \exp_args:No \__keys_set_filter:nnnnN
15186     \l__keys_unused_clist {#4} {#1} {#2} {#3} #5
15187 }
15188 \cs_generate_variant:Nn \keys_set_filter:nnnnN { nnV , nnv , nno }
15189 \cs_new_protected:Npn \__keys_set_filter:nnnnN #1#2#3#4#5#6
15190 {
15191     \clist_clear:N \l__keys_unused_clist
15192     \__keys_set_filter:nnnn {#2} {#3} {#4} {#5}
15193     \tl_set:Nx #6 { \exp_not:o \l__keys_unused_clist }
15194     \tl_set:Nn \l__keys_unused_clist {#1}
15195 }
15196 \cs_new_protected:Npn \keys_set_filter:nnn #1#2#3

```

\keys_set_filter:nnV

\keys_set_filter:nnv

\keys_set_filter:nno

\keys_set_filter:nnnnN

\keys_set_filter:nnn

\keys_set_filter:nnV

\keys_set_filter:nnv

\keys_set_filter:nno

__keys_set_filter:nnnnN

\keys_set_groups:nnn

\keys_set_groups:nnV

\keys_set_groups:nnv

\keys_set_groups:nno

__keys_set_selective:nnn

__keys_set_selective:nnnn


```

15197     { \__keys_set_filter:nnnn \q__keys_no_value {#1} {#2} {#3} }
15198 \cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }
15199 \cs_new_protected:Npn \__keys_set_filter:nnnn #1#2#3#4
15200 {
15201     \use:x
15202     {
15203         \bool_set_false:N \exp_not:N \l__keys_only_known_bool
15204         \bool_set_true:N \exp_not:N \l__keys_filtered_bool
15205         \bool_set_true:N \exp_not:N \l__keys_selective_bool
15206         \tl_set:Nn \exp_not:N \l__keys_relative_tl { \exp_not:n {#1} }
15207         \__keys_set_selective:nnn \exp_not:n { {#2} {#3} {#4} }
15208         \bool_if:NT \l__keys_only_known_bool
15209             { \bool_set_true:N \exp_not:N \l__keys_only_known_bool }
15210         \bool_if:NF \l__keys_filtered_bool
15211             { \bool_set_false:N \exp_not:N \l__keys_filtered_bool }
15212         \bool_if:NF \l__keys_selective_bool
15213             { \bool_set_false:N \exp_not:N \l__keys_selective_bool }
15214         \tl_set:Nn \exp_not:N \l__keys_relative_tl
15215             { \exp_not:o \l__keys_relative_tl }
15216     }
15217 }
15218 \cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
15219 {
15220     \use:x
15221     {
15222         \bool_set_false:N \exp_not:N \l__keys_only_known_bool
15223         \bool_set_false:N \exp_not:N \l__keys_filtered_bool
15224         \bool_set_true:N \exp_not:N \l__keys_selective_bool
15225         \tl_set:Nn \exp_not:N \l__keys_relative_tl
15226             { \exp_not:N \q__keys_no_value }
15227         \__keys_set_selective:nnn \exp_not:n { {#1} {#2} {#3} }
15228         \bool_if:NT \l__keys_only_known_bool
15229             { \bool_set_true:N \exp_not:N \l__keys_only_known_bool }
15230         \bool_if:NF \l__keys_filtered_bool
15231             { \bool_set_true:N \exp_not:N \l__keys_filtered_bool }
15232         \bool_if:NF \l__keys_selective_bool
15233             { \bool_set_false:N \exp_not:N \l__keys_selective_bool }
15234         \tl_set:Nn \exp_not:N \l__keys_relative_tl
15235             { \exp_not:o \l__keys_relative_tl }
15236     }
15237 }
15238 \cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }
15239 \cs_new_protected:Npn \__keys_set_selective:nnn
15240 { \exp_args:No \__keys_set_selective:nnnn \l__keys_selective_seq }
15241 \cs_new_protected:Npn \__keys_set_selective:nnnn #1#2#3#4
15242 {
15243     \seq_set_from_clist:Nn \l__keys_selective_seq {#3}
15244     \__keys_set:nn {#2} {#4}
15245     \tl_set:Nn \l__keys_selective_seq {#1}
15246 }

```

(End definition for `\keys_set_filter:nnnN` and others. These functions are documented on page 196.)

A shared system once again. First, set the current path and add a default if needed. There are then checks to see if the a value is required or forbidden. If everything passes,

```

\__keys_set_keyval:n
\__keys_set_keyval:nn
\__keys_set_keyval:nnn
\__keys_set_keyval:onn
\__keys_find_key_module:wNN
\__keys_find_key_module_auxi:Nw
\__keys_find_key_module_auxii:Nw
\__keys_find_key_module_auxiii:Nn
\__keys_find_key_module_auxiv:Nw
\__keys_set_selective:

```

move on to execute the code.

```

15247 \cs_new_protected:Npn \__keys_set_keyval:n #1
15248 {
15249     \bool_set_true:N \l__keys_no_value_bool
15250     \__keys_set_keyval:onn \l__keys_module_str {#1} { }
15251 }
15252 \cs_new_protected:Npn \__keys_set_keyval:nn #1#2
15253 {
15254     \bool_set_false:N \l__keys_no_value_bool
15255     \__keys_set_keyval:onn \l__keys_module_str {#1} {#2}
15256 }

```

The key path here can be fully defined, after which there is a search for the key and module names: the user may have passed them with part of what is actually the module (for our purposes) in the key name. As that happens on a per-key basis, we use the stack approach to restore the module name without a group.

```

15257 \cs_new_protected:Npn \__keys_set_keyval:nnn #1#2#3
15258 {
15259     \tl_set:Nx \l_keys_path_str
15260     {
15261         \tl_if_blank:nF {#1}
15262         { #1 / }
15263         \__keys_trim_spaces:n {#2}
15264     }
15265     \str_clear:N \l__keys_module_str
15266     \str_clear:N \l__keys_inherit_str
15267     \exp_after:wN \__keys_find_key_module:wNN \l_keys_path_str \s__keys_stop
15268     \l__keys_module_str \l_keys_key_str
15269     \tl_set_eq:NN \l_keys_key_tl \l_keys_key_str
15270     \__keys_value_or_default:n {#3}
15271     \bool_if:NTF \l__keys_selective_bool
15272     \__keys_set_selective:
15273     \__keys_execute:
15274     \str_set:Nn \l__keys_module_str {#1}
15275 }
15276 \cs_generate_variant:Nn \__keys_set_keyval:nnn { o }

```

This function uses `\cs_set_nopar:Npx` internally for performance reasons, the argument `#1` is already a string in every usage, so turning it into a string again seems unnecessary.

```

15277 \cs_new_protected:Npn \__keys_find_key_module:wNN #1 \s__keys_stop #2 #3
15278 {
15279     \__keys_find_key_module_auxi:Nw #2 #1 \s__keys_nil \__keys_find_key_module_auxii:Nw
15280     / \s__keys_nil \__keys_find_key_module_auxiv:Nw #3
15281 }
15282 \cs_new_protected:Npn \__keys_find_key_module_auxi:Nw #1 #2 / #3 \s__keys_nil #4
15283 {
15284     #4 #1 #2 \s__keys_mark #3 \s__keys_nil #4
15285 }
15286 \cs_new_protected:Npn \__keys_find_key_module_auxii:Nw
15287     #1 #2 \s__keys_mark #3 \s__keys_nil \__keys_find_key_module_auxiii:Nw
15288 {
15289     \cs_set_nopar:Npx #1 { \tl_if_empty:NF #1 { #1 / } #2 }
15290     \__keys_find_key_module_auxi:Nw #1 #3 \s__keys_nil \__keys_find_key_module_auxiii:Nw
15291 }

```

```

15292 \cs_new_protected:Npn \__keys_find_key_module_auxiii:Nw #1 #2 \s__keys_mark
15293 {
15294   \cs_set_nopar:Npx #1 { \tl_if_empty:NF #1 { #1 / } #2 }
15295   \__keys_find_key_module_auxi:Nw #1
15296 }
15297 \cs_new_protected:Npn \__keys_find_key_module_auxiv:Nw
15298   #1 #2 \s__keys_nil #3 \s__keys_mark
15299   \s__keys_nil \__keys_find_key_module_auxiv:Nw #4
15300 {
15301   \cs_set_nopar:Npn #4 { #2 }
15302 }

```

If selective setting is active, there are a number of possible sub-cases to consider. The key name may not be known at all or if it is, it may not have any groups assigned. There is then the question of whether the selection is opt-in or opt-out.

```

15303 \cs_new_protected:Npn \__keys_set_selective:
15304 {
15305   \cs_if_exist:cTF { \c__keys_groups_root_str \l_keys_path_str }
15306   {
15307     \clist_set_eq:Nc \l__keys_groups_clist
15308       { \c__keys_groups_root_str \l_keys_path_str }
15309     \__keys_check_groups:
15310   }
15311   {
15312     \bool_if:NTF \l__keys_filtered_bool
15313       \__keys_execute:
15314       \__keys_store_unused:
15315   }
15316 }

```

In the case where selective setting requires a comparison of the list of groups which apply to a key with the list of those which have been set active. That requires two mappings, and again a different outcome depending on whether opt-in or opt-out is set.

```

15317 \cs_new_protected:Npn \__keys_check_groups:
15318 {
15319   \bool_set_false:N \l__keys_tmp_bool
15320   \seq_map_inline:Nn \l__keys_selective_seq
15321   {
15322     \clist_map_inline:Nn \l__keys_groups_clist
15323     {
15324       \str_if_eq:nnT {##1} {####1}
15325       {
15326         \bool_set_true:N \l__keys_tmp_bool
15327         \clist_map_break:n \seq_map_break:
15328       }
15329     }
15330   }
15331   \bool_if:NTF \l__keys_tmp_bool
15332   {
15333     \bool_if:NTF \l__keys_filtered_bool
15334       \__keys_store_unused:
15335       \__keys_execute:
15336   }
15337   {
15338     \bool_if:NTF \l__keys_filtered_bool

```

```

15339         \__keys_execute:
15340         \__keys_store_unused:
15341     }
15342 }

```

(End definition for __keys_set_keyval:n and others.)

__keys_value_or_default:n
 __keys_default_inherit:

If a value is given, return it as #1, otherwise send a default if available.

```

15343 \cs_new_protected:Npn \__keys_value_or_default:n #1
15344 {
15345   \bool_if:NTF \l__keys_no_value_bool
15346   {
15347     \cs_if_exist:cTF { \c__keys_default_root_str \l_keys_path_str }
15348     {
15349       \tl_set_eq:Nc
15350       \l_keys_value_tl
15351       { \c__keys_default_root_str \l_keys_path_str }
15352     }
15353     {
15354       \tl_clear:N \l_keys_value_tl
15355       \cs_if_exist:cT
15356       { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
15357       { \__keys_default_inherit: }
15358     }
15359   }
15360   { \tl_set:Nn \l_keys_value_tl {#1} }
15361 }
15362 \cs_new_protected:Npn \__keys_default_inherit:
15363 {
15364   \clist_map_inline:cn
15365   { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
15366   {
15367     \cs_if_exist:cT
15368     { \c__keys_default_root_str ##1 / \l_keys_key_str }
15369     {
15370       \tl_set_eq:Nc
15371       \l_keys_value_tl
15372       { \c__keys_default_root_str ##1 / \l_keys_key_str }
15373       \clist_map_break:
15374     }
15375   }
15376 }

```

(End definition for __keys_value_or_default:n and __keys_default_inherit:.)

__keys_execute:
 __keys_execute_inherit:
 __keys_execute_unknown:
 __keys_execute:nn
 __keys_execute:no
 __keys_store_unused:
 __keys_store_unused_aux:

Actually executing a key is done in two parts. First, look for the key itself, then look for the **unknown** key with the same path. If both of these fail, complain. What exactly happens if a key is unknown depends on whether unknown keys are being skipped or if an error should be raised.

```

15377 \cs_new_protected:Npn \__keys_execute:
15378 {
15379   \cs_if_exist:cTF { \c__keys_code_root_str \l_keys_path_str }
15380   {
15381     \cs_if_exist_use:c { \c__keys_validate_root_str \l_keys_path_str }

```

```

15382     \_keys_execute:no \l_keys_path_str \l_keys_value_tl
15383   }
15384   {
15385     \cs_if_exist:cTF
15386     { \c__keys_inherit_root_str \_keys_parent:o \l_keys_path_str }
15387     { \_keys_execute_inherit: }
15388     { \_keys_execute_unknown: }
15389   }
15390 }

```

To deal with the case where there is no hit, we leave `_keys_execute_unknown:` in the input stream and clean it up using the break function: that avoids needing a boolean.

```

15391 \cs_new_protected:Npn \_keys_execute_inherit:
15392 {
15393   \clist_map_inline:cn
15394   { \c__keys_inherit_root_str \_keys_parent:o \l_keys_path_str }
15395   {
15396     \cs_if_exist:cT
15397     { \c__keys_code_root_str ##1 / \l_keys_key_str }
15398     {
15399       \str_set:Nn \l__keys_inherit_str {##1}
15400       \cs_if_exist_use:c { \c__keys_validate_root_str ##1 / \l_keys_key_str }
15401       \_keys_execute:no { ##1 / \l_keys_key_str } \l_keys_value_tl
15402       \clist_map_break:n \use_none:n
15403     }
15404   }
15405   \_keys_execute_unknown:
15406 }
15407 \cs_new_protected:Npn \_keys_execute_unknown:
15408 {
15409   \bool_if:NTF \l__keys_only_known_bool
15410   { \_keys_store_unused: }
15411   {
15412     \cs_if_exist:cTF
15413     { \c__keys_code_root_str \l__keys_module_str / unknown }
15414     { \_keys_execute:no { \l__keys_module_str / unknown } \l_keys_value_tl }
15415     {
15416       \__kernel_msg_error:nxxx { kernel } { key-unknown }
15417       \l_keys_path_str \l__keys_module_str
15418     }
15419   }
15420 }

```

A key's code is in the control sequence with csname `\c__keys_code_root_str #1`. We expand it once to get the replacement text (with argument #2) and call `\use:n` with this replacement as its argument. This ensures that any undefined control sequence error in the key's code will lead to an error message of the form `<argument>...<control sequence>` in which one can read the (undefined) `<control sequence>` in full, rather than an error message that starts with the potentially very long key name, which would make the (undefined) `<control sequence>` be truncated or sometimes completely hidden. See <https://github.com/latex3/latex2e/issues/351>.

```

15421 \cs_new:Npn \_keys_execute:nn #1#2
15422 { \_keys_execute:no {#1} { \prg_do_nothing: #2 } }
15423 \cs_new:Npn \_keys_execute:no #1#2

```

```

15424 {
15425     \exp_args:NNo \exp_args:No \use:n
15426     {
15427         \cs:w \c__keys_code_root_str #1 \exp_after:wN \cs_end:
15428         \exp_after:wN {#2}
15429     }
15430 }

```

When there is no relative path, things here are easy: just save the key name and value. When we are working with a relative path, first we need to turn it into a string: that can't happen earlier as we need to store `\q__keys_no_value`. Then, use a standard delimited approach to fish out the partial path.

```

15431 \cs_new_protected:Npn \__keys_store_unused:
15432 {
15433     \__keys_quark_if_no_value:NTF \l__keys_relative_tl
15434     {
15435         \clist_put_right:Nx \l__keys_unused_clist
15436         {
15437             \l_keys_key_str
15438             \bool_if:NF \l__keys_no_value_bool
15439             { = { \exp_not:o \l_keys_value_tl } }
15440         }
15441     }
15442     {
15443         \tl_if_empty:NTF \l__keys_relative_tl
15444         {
15445             \clist_put_right:Nx \l__keys_unused_clist
15446             {
15447                 \l_keys_path_str
15448                 \bool_if:NF \l__keys_no_value_bool
15449                 { = { \exp_not:o \l_keys_value_tl } }
15450             }
15451         }
15452         { \__keys_store_unused_aux: }
15453     }
15454 }
15455 \cs_new_protected:Npn \__keys_store_unused_aux:
15456 {
15457     \tl_set:Nx \l__keys_relative_tl
15458     { \exp_args:No \__keys_trim_spaces:n \l__keys_relative_tl }
15459     \use:x
15460     {
15461         \cs_set_protected:Npn \__keys_store_unused:w
15462         ###1 \l__keys_relative_tl /
15463         ###2 \l__keys_relative_tl /
15464         ###3 \s__keys_stop
15465     }
15466     {
15467         \tl_if_blank:nF {##1}
15468         {
15469             \__kernel_msg_error:nxxx { kernel } { bad-relative-key-path }
15470             \l_keys_path_str
15471             \l__keys_relative_tl
15472         }

```

```

15473         \clist_put_right:Nx \l__keys_unused_clist
15474         {
15475             \exp_not:n {##2}
15476             \bool_if:NF \l__keys_no_value_bool
15477             { = { \exp_not:o \l_keys_value_tl } }
15478         }
15479     }
15480     \use:x
15481     {
15482         \__keys_store_unused:w \l_keys_path_str
15483         \l__keys_relative_tl / \l__keys_relative_tl /
15484         \s__keys_stop
15485     }
15486 }
15487 \cs_new_protected:Npn \__keys_store_unused:w { }

```

(End definition for `__keys_execute:` and others.)

`__keys_choice_find:n` Executing a choice has two parts. First, try the choice given, then if that fails call the
`__keys_choice_find:nn` unknown key. That always exists, as it is created when a choice is first made. So there
`__keys_multichoice_find:n` is no need for any escape code. For multiple choices, the same code ends up used in a
mapping.

```

15488 \cs_new:Npn \__keys_choice_find:n #1
15489 {
15490     \str_if_empty:NTF \l__keys_inherit_str
15491     { \__keys_choice_find:nn \l_keys_path_str {#1} }
15492     {
15493         \__keys_choice_find:nn
15494         { \l__keys_inherit_str / \l_keys_key_str } {#1}
15495     }
15496 }
15497 \cs_new:Npn \__keys_choice_find:nn #1#2
15498 {
15499     \cs_if_exist:cTF { \c__keys_code_root_str #1 / \__keys_trim_spaces:n {#2} }
15500     { \__keys_execute:nn { #1 / \__keys_trim_spaces:n {#2} } {#2} }
15501     { \__keys_execute:nn { #1 / unknown } {#2} }
15502 }
15503 \cs_new:Npn \__keys_multichoice_find:n #1
15504 { \clist_map_function:nN {#1} \__keys_choice_find:n }

```

(End definition for `__keys_choice_find:n`, `__keys_choice_find:nn`, and `__keys_multichoice_find:n`.)

22.7 Utilities

`__keys_parent:o` Used to strip off the ending part of the key path after the last `/`.
`__keys_parent_auxi:w`
`__keys_parent_auxii:w`
`__keys_parent_auxiii:n`
`__keys_parent_auxiv:w`

```

15505 \cs_new:Npn \__keys_parent:o #1
15506 {
15507     \exp_after:wN \__keys_parent_auxi:w #1 \q_nil \__keys_parent_auxii:w
15508     / \q_nil \__keys_parent_auxiv:w
15509 }
15510 \cs_new:Npn \__keys_parent_auxi:w #1 / #2 \q_nil #3
15511 {
15512     #3 { #1 } #2 \q_nil #3

```

```

15513 }
15514 \cs_new:Npn \__keys_parent_auxii:w #1 #2 \q_nil \__keys_parent_auxii:w
15515 {
15516     #1 \__keys_parent_auxi:w #2 \q_nil \__keys_parent_auxiii:n
15517 }
15518 \cs_new:Npn \__keys_parent_auxiii:n #1
15519 {
15520     / #1 \__keys_parent_auxi:w
15521 }
15522 \cs_new:Npn \__keys_parent_auxiv:w #1 \q_nil \__keys_parent_auxiv:w
15523 {
15524 }

```

(End definition for __keys_parent:o and others.)

__keys_trim_spaces:n Space stripping has to allow for the fact that the key here might have several parts, and spaces need to be stripped from each part. Since the key name is turned into a string groups can't be stripped accidentally and the precautions of \tl_trim_spaces:n aren't necessary, in this case it is much faster to just directly strip spaces around /.

```

15525 \group_begin:
15526     \cs_set:Npn \__keys_tmp:n #1
15527     {
15528         \cs_new:Npn \__keys_trim_spaces:n ##1
15529         {
15530             \exp_after:wN \__keys_trim_spaces_auxi:w \tl_to_str:n { / ##1 } /
15531             \s_keys_nil \__keys_trim_spaces_auxi:w
15532             \s_keys_mark \__keys_trim_spaces_auxii:w
15533             #1 / #1
15534             \s_keys_nil \__keys_trim_spaces_auxii:w
15535             \s_keys_mark \__keys_trim_spaces_auxiii:w
15536         }
15537     }
15538     \__keys_tmp:n { ~ }
15539 \group_end:
15540 \cs_new:Npn \__keys_trim_spaces_auxi:w #1 ~ / #2 \s_keys_nil #3
15541 {
15542     #3 #1 / #2 \s_keys_nil #3
15543 }
15544 \cs_new:Npn \__keys_trim_spaces_auxii:w #1 / ~ #2 \s_keys_mark #3
15545 {
15546     #3 #1 / #2 \s_keys_mark #3
15547 }
15548 \cs_new:Npn \__keys_trim_spaces_auxiii:w
15549 / #1 /
15550 \s_keys_nil \__keys_trim_spaces_auxi:w
15551 \s_keys_mark \__keys_trim_spaces_auxii:w
15552 /
15553 \s_keys_nil \__keys_trim_spaces_auxii:w
15554 \s_keys_mark \__keys_trim_spaces_auxiii:w
15555 {
15556     #1
15557 }

```

(End definition for __keys_trim_spaces:n and others.)

`\keys_if_exist_p:nn` A utility for others to see if a key exists.
`\keys_if_exist:nnTF`

```

15558 \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
15559 {
15560   \cs_if_exist:cTF
15561     { \c__keys_code_root_str \__keys_trim_spaces:n { #1 / #2 } }
15562     { \prg_return_true: }
15563     { \prg_return_false: }
15564 }

```

(End definition for `\keys_if_exist:nnTF`. This function is documented on page 196.)

`\keys_if_choice_exist_p:nnn` Just an alternative view on `\keys_if_exist:nnTF`.
`\keys_if_choice_exist:nnnTF`

```

15565 \prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3
15566 { p , T , F , TF }
15567 {
15568   \cs_if_exist:cTF
15569     { \c__keys_code_root_str \__keys_trim_spaces:n { #1 / #2 / #3 } }
15570     { \prg_return_true: }
15571     { \prg_return_false: }
15572 }

```

(End definition for `\keys_if_choice_exist:nnnTF`. This function is documented on page 197.)

`\keys_show:nn` To show a key, show its code using a message.
`\keys_log:nn`
`__keys_show:Nnn`

```

15573 \cs_new_protected:Npn \keys_show:nn
15574 { \__keys_show:Nnn \msg_show:nnxxxx }
15575 \cs_new_protected:Npn \keys_log:nn
15576 { \__keys_show:Nnn \msg_log:nnxxxx }
15577 \cs_new_protected:Npn \__keys_show:Nnn #1#2#3
15578 {
15579   #1 { LaTeX / kernel } { show-key }
15580   { \__keys_trim_spaces:n { #2 / #3 } }
15581   {
15582     \keys_if_exist:nnT {#2} {#3}
15583     {
15584       \exp_args:Nnf \msg_show_item_unbraced:nn { code }
15585       {
15586         \exp_args:Nc \cs_replacement_spec:N
15587         {
15588           \c__keys_code_root_str
15589           \__keys_trim_spaces:n { #2 / #3 }
15590         }
15591       }
15592     }
15593   }
15594   { } { } { }
15595 }

```

(End definition for `\keys_show:nn`, `\keys_log:nn`, and `__keys_show:Nnn`. These functions are documented on page 197.)

22.8 Messages

For when there is a need to complain.

```
15596 \__kernel_msg_new:nnnn { kernel } { bad-relative-key-path }
15597 { The-key~'#1'~is-not~inside~the~'#2'~path. }
15598 { The-key~'#1'~cannot~be~expressed~relative~to~path~'#2'. }
15599 \__kernel_msg_new:nnnn { kernel } { boolean-values-only }
15600 { Key~'#1'~accepts~boolean~values~only. }
15601 { The-key~'#1'~only~accepts~the~values~'true'~and~'false'. }
15602 \__kernel_msg_new:nnnn { kernel } { key-choice-unknown }
15603 { Key~'#1'~accepts~only~a~fixed~set~of~choices. }
15604 {
15605     The-key~'#1'~only~accepts~predefined~values,~
15606     and~'#2'~is~not~one~of~these.
15607 }
15608 \__kernel_msg_new:nnnn { kernel } { key-unknown }
15609 { The-key~'#1'~is~unknown~and~is~being~ignored. }
15610 {
15611     The-module~'#2'~does~not~have~a~key~called~'#1'.\\
15612     Check~that~you~have~spelled~the~key~name~correctly.
15613 }
15614 \__kernel_msg_new:nnnn { kernel } { nested-choice-key }
15615 { Attempt~to~define~'#1'~as~a~nested~choice~key. }
15616 {
15617     The-key~'#1'~cannot~be~defined~as~a~choice~as~the~parent~key~'#2'~is~
15618     itself~a~choice.
15619 }
15620 \__kernel_msg_new:nnnn { kernel } { value-forbidden }
15621 { The-key~'#1'~does~not~take~a~value. }
15622 {
15623     The-key~'#1'~should~be~given~without~a~value.\\
15624     The-value~'#2'~was~present:~the~key~will~be~ignored.
15625 }
15626 \__kernel_msg_new:nnnn { kernel } { value-required }
15627 { The-key~'#1'~requires~a~value. }
15628 {
15629     The-key~'#1'~must~have~a~value.\\
15630     No~value~was~present:~the~key~will~be~ignored.
15631 }
15632 \__kernel_msg_new:nnn { kernel } { show-key }
15633 {
15634     The-key~'#1'~
15635     \tl_if_empty:nTF {#2}
15636     { is~undefined. }
15637     { has~the~properties: #2 . }
15638 }
15639 </package>
```

23 l3intarray implementation

```
15640 <*package>
15641 <@@=intarray>
```

23.1 Allocating arrays

`__intarray_entry:w` We use these primitives quite a lot in this module.

`__intarray_count:w` 15642 `\cs_new_eq:NN __intarray_entry:w \tex_fontdimen:D`
15643 `\cs_new_eq:NN __intarray_count:w \tex_hyphenchar:D`
(End definition for `__intarray_entry:w` and `__intarray_count:w`.)

`\l__intarray_loop_int` A loop index.
15644 `\int_new:N \l__intarray_loop_int`
(End definition for `\l__intarray_loop_int`.)

`\c__intarray_sp_dim` Used to convert integers to dimensions fast.
15645 `\dim_const:Nn \c__intarray_sp_dim { 1 sp }`
(End definition for `\c__intarray_sp_dim`.)

`\g__intarray_font_int` Used to assign one font per array.
15646 `\int_new:N \g__intarray_font_int`
(End definition for `\g__intarray_font_int`.)
15647 `__kernel_msg_new:nnn { kernel } { negative-array-size }`
15648 `{ Size-of-array-may-not-be-negative:~#1 }`

`\intarray_new:Nn` Declare `#1` to be a font (arbitrarily `cmr10` at a never-used size). Store the array's size as the `\hyphenchar` of that font and make sure enough `\fontdimen` are allocated, by setting the last one. Then clear any `\fontdimen` that `cmr10` starts with. It seems LuaTeX's `cmr10` has an extra `\fontdimen` parameter number 8 compared to other engines (for a math font we would replace 8 by 22 or some such). Every `intarray` must be global; it's enough to run this check in `\intarray_new:Nn`.

`\intarray_new:cn`

`__intarray_new:N`
15649 `\cs_new_protected:Npn __intarray_new:N #1`
15650 `{`
15651 `__kernel_chk_if_free_cs:N #1`
15652 `\int_gincr:N \g__intarray_font_int`
15653 `\tex_global:D \tex_font:D #1`
15654 `= cmr10~at~ \g__intarray_font_int \c__intarray_sp_dim \scan_stop:`
15655 `\int_step_inline:nn { 8 }`
15656 `{ __kernel_intarray_gset:Nnn #1 {##1} \c_zero_int }`
15657 `}`
15658 `\cs_new_protected:Npn \intarray_new:Nn #1#2`
15659 `{`
15660 `__intarray_new:N #1`
15661 `__intarray_count:w #1 = \int_eval:n {#2} \scan_stop:`
15662 `\int_compare:nNnT { \intarray_count:N #1 } < 0`
15663 `{`
15664 `__kernel_msg_error:nnx { kernel } { negative-array-size }`
15665 `{ \intarray_count:N #1 }`
15666 `}`
15667 `\int_compare:nNnT { \intarray_count:N #1 } > 0`
15668 `{ __kernel_intarray_gset:Nnn #1 { \intarray_count:N #1 } { 0 } }`
15669 `}`
15670 `\cs_generate_variant:Nn \intarray_new:Nn { c }`

(End definition for \intarray_new:Nn and __intarray_new:N. This function is documented on page 199.)

\intarray_count:N Size of an array.

\intarray_count:c 15671 \cs_new:Npn \intarray_count:N #1 { \int_value:w __intarray_count:w #1 }
15672 \cs_generate_variant:Nn \intarray_count:N { c }

(End definition for \intarray_count:N. This function is documented on page 199.)

23.2 Array items

__intarray_signed_max_dim:n Used when an item to be stored is larger than \c_max_dim in absolute value; it is replaced by $\pm\c_max_dim$.

15673 \cs_new:Npn __intarray_signed_max_dim:n #1
15674 { \int_value:w \int_compare:nNnT {#1} < 0 { - } \c_max_dim }

(End definition for __intarray_signed_max_dim:n.)

__intarray_bounds:NNnTF The functions \intarray_gset:Nnn and \intarray_item:Nn share bounds checking.
__intarray_bounds_error:NNnw The T branch is used if #3 is within bounds of the array #2.

15675 \cs_new:Npn __intarray_bounds:NNnTF #1#2#3
15676 {
15677 \if_int_compare:w 1 > #3 \exp_stop_f:
15678 __intarray_bounds_error:NNnw #1 #2 {#3}
15679 \else:
15680 \if_int_compare:w #3 > \intarray_count:N #2 \exp_stop_f:
15681 __intarray_bounds_error:NNnw #1 #2 {#3}
15682 \fi:
15683 \fi:
15684 \use_i:nn
15685 }
15686 \cs_new:Npn __intarray_bounds_error:NNnw #1#2#3#4 \use_i:nn #5#6
15687 {
15688 #4
15689 #1 { kernel } { out-of-bounds }
15690 { \token_to_str:N #2 } {#3} { \intarray_count:N #2 }
15691 #6
15692 }

(End definition for __intarray_bounds:NNnTF and __intarray_bounds_error:NNnw.)

\intarray_gset:Nnn Set the appropriate \fontdimen. The __kernel_intarray_gset:Nnn function does not

\intarray_gset:cnn use \int_eval:n, namely its arguments must be suitable for \int_value:w. The user

__kernel_intarray_gset:Nnn version checks the position and value are within bounds.

__intarray_gset:Nnn 15693 \cs_new_protected:Npn __kernel_intarray_gset:Nnn #1#2#3
__intarray_gset_overflow:Nnn 15694 { __intarray_entry:w #2 #1 #3 \c__intarray_sp_dim }
15695 \cs_new_protected:Npn \intarray_gset:Nnn #1#2#3
15696 {
15697 \exp_after:wN __intarray_gset:Nnw
15698 \exp_after:wN #1
15699 \int_value:w \int_eval:n {#2} \exp_after:wN ;
15700 \int_value:w \int_eval:n {#3} ;
15701 }
15702 \cs_generate_variant:Nn \intarray_gset:Nnn { c }

```

15703 \cs_new_protected:Npn \__intarray_gset:Nww #1#2 ; #3 ;
15704 {
15705   \__intarray_bounds:NNnTF \__kernel_msg_error:nnxxx #1 {#2}
15706   {
15707     \__intarray_gset_overflow_test:nw {#3}
15708     \__kernel_intarray_gset:Nnn #1 {#2} {#3}
15709   }
15710   { }
15711 }
15712 \cs_if_exist:NTF \tex_ifabsnum:D
15713 {
15714   \cs_new_protected:Npn \__intarray_gset_overflow_test:nw #1
15715   {
15716     \tex_ifabsnum:D #1 > \c_max_dim
15717     \exp_after:wN \__intarray_gset_overflow:NNnn
15718     \fi:
15719   }
15720 }
15721 {
15722   \cs_new_protected:Npn \__intarray_gset_overflow_test:nw #1
15723   {
15724     \if_int_compare:w \int_abs:n {#1} > \c_max_dim
15725     \exp_after:wN \__intarray_gset_overflow:NNnn
15726     \fi:
15727   }
15728 }
15729 \cs_new_protected:Npn \__intarray_gset_overflow:NNnn #1#2#3#4
15730 {
15731   \__kernel_msg_error:nnxxxx { kernel } { overflow }
15732   { \token_to_str:N #2 } {#3} {#4} { \__intarray_signed_max_dim:n {#4} }
15733   #1 #2 {#3} { \__intarray_signed_max_dim:n {#4} }
15734 }

```

(End definition for `\intarray_gset:Nnn` and others. This function is documented on page 199.)

`\intarray_gzero:N` Set the appropriate `\fontdimen` to zero. No bound checking needed. The `\prg_replicate:nn` possibly uses quite a lot of memory, but this is somewhat comparable to the size of the array, and it is much faster than an `\int_step_inline:nn` loop.

`\intarray_gzero:c`

```

15735 \cs_new_protected:Npn \intarray_gzero:N #1
15736 {
15737   \int_zero:N \l__intarray_loop_int
15738   \prg_replicate:nn { \intarray_count:N #1 }
15739   {
15740     \int_incr:N \l__intarray_loop_int
15741     \__intarray_entry:w \l__intarray_loop_int #1 \c_zero_dim
15742   }
15743 }
15744 \cs_generate_variant:Nn \intarray_gzero:N { c }

```

(End definition for `\intarray_gzero:N`. This function is documented on page 199.)

`\intarray_item:Nn` Get the appropriate `\fontdimen` and perform bound checks. The `__kernel_intarray_item:Nn` function omits bound checks and omits `\int_eval:n`, namely its argument must be a TeX integer suitable for `\int_value:w`.

`\intarray_item:cn`

`__kernel_intarray_item:Nn`

`__intarray_item:Nn`

```

15745 \cs_new:Npn \__kernel_intarray_item:Nn #1#2
15746 { \int_value:w \__intarray_entry:w #2 #1 }
15747 \cs_new:Npn \intarray_item:Nn #1#2
15748 {
15749   \exp_after:wN \__intarray_item:Nw
15750   \exp_after:wN #1
15751   \int_value:w \int_eval:n {#2} ;
15752 }
15753 \cs_generate_variant:Nn \intarray_item:Nn { c }
15754 \cs_new:Npn \__intarray_item:Nw #1#2 ;
15755 {
15756   \__intarray_bounds:NNnTF \__kernel_msg_expandable_error:nnfff #1 {#2}
15757   { \__kernel_intarray_item:Nn #1 {#2} }
15758   { 0 }
15759 }

```

(End definition for `\intarray_item:Nn`, `__kernel_intarray_item:Nn`, and `__intarray_item:Nn`. This function is documented on page 200.)

`\intarray_rand_item:N` Importantly, `\intarray_item:Nn` only evaluates its argument once.

```

\intarray_rand_item:c
15760 \cs_new:Npn \intarray_rand_item:N #1
15761 { \intarray_item:Nn #1 { \int_rand:n { \intarray_count:N #1 } } }
15762 \cs_generate_variant:Nn \intarray_rand_item:N { c }

```

(End definition for `\intarray_rand_item:N`. This function is documented on page 200.)

23.3 Working with contents of integer arrays

`\intarray_const_from_clist:Nn` Similar to `\intarray_new:Nn` (which we don't use because when debugging is enabled that function checks the variable name starts with `g_`). We make use of the fact that `TEX` allows allocation of successive `\fontdimen` as long as no other font has been declared: no need to count the comma list items first. We need the code in `\intarray_gset:Nnn` that checks the item value is not too big, namely `__intarray_gset_overflow_test:nw`, but not the code that checks bounds. At the end, set the size of the intarray.

```

15763 \cs_new_protected:Npn \intarray_const_from_clist:Nn #1#2
15764 {
15765   \__intarray_new:N #1
15766   \int_zero:N \l__intarray_loop_int
15767   \clist_map_inline:nn {#2}
15768   { \exp_args:Nf \__intarray_const_from_clist:nN { \int_eval:n {##1} } #1 }
15769   \__intarray_count:w #1 \l__intarray_loop_int
15770 }
15771 \cs_generate_variant:Nn \intarray_const_from_clist:Nn { c }
15772 \cs_new_protected:Npn \__intarray_const_from_clist:nN #1#2
15773 {
15774   \int_incr:N \l__intarray_loop_int
15775   \__intarray_gset_overflow_test:nw {#1}
15776   \__kernel_intarray_gset:Nnn #2 \l__intarray_loop_int {#1}
15777 }

```

(End definition for `\intarray_const_from_clist:Nn` and `__intarray_const_from_clist:nN`. This function is documented on page 199.)

\intarray_to_clist:N Loop through the array, putting a comma before each item. Remove the leading comma with f-expansion. We also use the auxiliary in **\intarray_show:N** with argument comma, space.

```

\intarray_to_clist:c
\__intarray_to_clist:Nn
\__intarray_to_clist:w
15778 \cs_new:Npn \intarray_to_clist:N #1 { \__intarray_to_clist:Nn #1 { , } }
15779 \cs_generate_variant:Nn \intarray_to_clist:N { c }
15780 \cs_new:Npn \__intarray_to_clist:Nn #1#2
15781 {
15782   \int_compare:nNf { \intarray_count:N #1 } = \c_zero_int
15783   {
15784     \exp_last_unbraced:Nf \use_none:n
15785     { \__intarray_to_clist:w 1 ; #1 {#2} \prg_break_point: }
15786   }
15787 }
15788 \cs_new:Npn \__intarray_to_clist:w #1 ; #2#3
15789 {
15790   \if_int_compare:w #1 > \__intarray_count:w #2
15791   \prg_break:n
15792   \fi:
15793   #3 \__kernel_intarray_item:Nn #2 {#1}
15794   \exp_after:wN \__intarray_to_clist:w
15795   \int_value:w \int_eval:w #1 + \c_one_int ; #2 {#3}
15796 }

```

(End definition for **\intarray_to_clist:N**, **__intarray_to_clist:Nn**, and **__intarray_to_clist:w**. This function is documented on page 267.)

\intarray_show:N Convert the list to a comma list (with spaces after each comma)

\intarray_show:c

\intarray_log:N

\intarray_log:c

```

15797 \cs_new_protected:Npn \intarray_show:N { \__intarray_show:NN \msg_show:nnxxxx }
15798 \cs_generate_variant:Nn \intarray_show:N { c }
15799 \cs_new_protected:Npn \intarray_log:N { \__intarray_show:NN \msg_log:nnxxxx }
15800 \cs_generate_variant:Nn \intarray_log:N { c }
15801 \cs_new_protected:Npn \__intarray_show:NN #1#2
15802 {
15803   \__kernel_chk_defined:NT #2
15804   {
15805     #1 { LaTeX/kernel } { show-intarray }
15806     { \token_to_str:N #2 }
15807     { \intarray_count:N #2 }
15808     { >~ \__intarray_to_clist:Nn #2 { , ~ } }
15809   }
15810 }
15811 }

```

(End definition for **\intarray_show:N** and **\intarray_log:N**. These functions are documented on page 200.)

23.4 Random arrays

We only perform the bounds checks once. This is done by two **__intarray_gset_**-**overflow_test:nw**, with an appropriate empty argument to avoid a spurious “at position #1” part in the error message. Then calculate the number of choices: this is at most $(2^{30} - 1) - (-(2^{30} - 1)) + 1 = 2^{31} - 1$, which just barely does not overflow. For small ranges use **__kernel_randint:n** (making sure to subtract 1 *before* adding the random number

\intarray_gset_rand:Nn

\intarray_gset_rand:cn

\intarray_gset_rand:Nnn

\intarray_gset_rand:cnn

__intarray_gset_rand:Nnn

__intarray_gset_rand:Nff

__intarray_gset_rand_auxi:Nnn

__intarray_gset_rand_auxii:Nnn

__intarray_gset_rand_auxiii:Nnn

__intarray_gset_all_same:Nn

to the $\langle min \rangle$, to avoid overflow when $\langle min \rangle$ or $\langle max \rangle$ are $\pm \backslash c_max_int$), otherwise $\backslash_kernel_randint:nn$. Finally, if there are no random numbers do not define any of the auxiliaries.

```

15812 \cs_new_protected:Npn \intarray_gset_rand:Nn #1
15813 { \intarray_gset_rand:Nnn #1 { 1 } }
15814 \cs_generate_variant:Nn \intarray_gset_rand:Nn { c }
15815 \sys_if_rand_exist:TF
15816 {
15817   \cs_new_protected:Npn \intarray_gset_rand:Nnn #1#2#3
15818   {
15819     \__intarray_gset_rand:Nff #1
15820     { \int_eval:n {#2} } { \int_eval:n {#3} }
15821   }
15822   \cs_new_protected:Npn \__intarray_gset_rand:Nnn #1#2#3
15823   {
15824     \int_compare:nNnTF {#2} > {#3}
15825     {
15826       \__kernel_msg_expandable_error:nnnn
15827       { kernel } { randint-backward-range } {#2} {#3}
15828       \__intarray_gset_rand:Nnn #1 {#3} {#2}
15829     }
15830     {
15831       \__intarray_gset_overflow_test:nw {#2}
15832       \__intarray_gset_rand_auxi:Nnnn #1 { } {#2} {#3}
15833     }
15834   }
15835   \cs_generate_variant:Nn \__intarray_gset_rand:Nnn { Nff }
15836   \cs_new_protected:Npn \__intarray_gset_rand_auxi:Nnnn #1#2#3#4
15837   {
15838     \__intarray_gset_overflow_test:nw {#4}
15839     \__intarray_gset_rand_auxii:Nnnn #1 { } {#4} {#3}
15840   }
15841   \cs_new_protected:Npn \__intarray_gset_rand_auxii:Nnnn #1#2#3#4
15842   {
15843     \exp_args:NNf \__intarray_gset_rand_auxiii:Nnnn #1
15844     { \int_eval:n { #3 - #4 + 1 } } {#4} {#3}
15845   }
15846   \cs_new_protected:Npn \__intarray_gset_rand_auxiii:Nnnn #1#2#3#4
15847   {
15848     \exp_args:NNf \__intarray_gset_all_same:Nn #1
15849     {
15850       \int_compare:nNnTF {#2} > \c__kernel_randint_max_int
15851       {
15852         \exp_stop_f:
15853         \int_eval:n { \__kernel_randint:nn {#3} {#4} }
15854       }
15855       {
15856         \exp_stop_f:
15857         \int_eval:n { \__kernel_randint:n {#2} - 1 + #3 }
15858       }
15859     }
15860   }
15861   \cs_new_protected:Npn \__intarray_gset_all_same:Nn #1#2
15862   {

```



```

15863     \int_zero:N \l__intarray_loop_int
15864     \prg_replicate:nn { \intarray_count:N #1 }
15865     {
15866         \int_incr:N \l__intarray_loop_int
15867         \__kernel_intarray_gset:Nnn #1 \l__intarray_loop_int {#2}
15868     }
15869 }
15870 }
15871 {
15872     \cs_new_protected:Npn \intarray_gset_rand:Nnn #1#2#3
15873     {
15874         \__kernel_msg_error:nnn { kernel } { fp-no-random }
15875         { \intarray_gset_rand:Nnn #1 {#2} {#3} }
15876     }
15877 }
15878 \cs_generate_variant:Nn \intarray_gset_rand:Nnn { c }

```

(End definition for `\intarray_gset_rand:Nn` and others. These functions are documented on page 267.)

```

15879 \endpackage

```

24 l3fp implementation

Nothing to see here: everything is in the subfiles!

25 l3fp-aux implementation

```

15880 \begin{package}
15881 \begin{@@=fp}

```

25.1 Access to primitives

`__fp_int_eval:w` Largely for performance reasons, we need to directly access primitives rather than use `\int_eval:n`. This happens *a lot*, so we use private names. The same is true for `__fp_int_eval_end:` `\romannumeral`, although it is used much less widely.

```

15882 \cs_new_eq:NN \__fp_int_eval:w \tex_numexpr:D
15883 \cs_new_eq:NN \__fp_int_eval_end: \scan_stop:
15884 \cs_new_eq:NN \__fp_int_to_roman:w \tex_romannumeral:D

```

(End definition for `__fp_int_eval:w`, `__fp_int_eval_end:`, and `__fp_int_to_roman:w`.)

25.2 Internal representation

Internally, a floating point number $\langle X \rangle$ is a token list containing

$$\backslash s_fp \backslash _fp_chk:w \langle case \rangle \langle sign \rangle \langle body \rangle ;$$

Let us explain each piece separately.

Internal floating point numbers are used in expressions, and in this context are subject to `f`-expansion. They must leave a recognizable mark after `f`-expansion, to prevent the floating point number from being re-parsed. Thus, `\s_fp` is simply another name for `\relax`.

Table 3: Internal representation of floating point numbers.

Representation	Meaning
0 0 \s_fp_... ;	Positive zero.
0 2 \s_fp_... ;	Negative zero.
1 0 {⟨exponent⟩} {⟨X ₁ ⟩} {⟨X ₂ ⟩} {⟨X ₃ ⟩} {⟨X ₄ ⟩} ;	Positive floating point.
1 2 {⟨exponent⟩} {⟨X ₁ ⟩} {⟨X ₂ ⟩} {⟨X ₃ ⟩} {⟨X ₄ ⟩} ;	Negative floating point.
2 0 \s_fp_... ;	Positive infinity.
2 2 \s_fp_... ;	Negative infinity.
3 1 \s_fp_... ;	Quiet nan.
3 1 \s_fp_... ;	Signalling nan.

When used directly without an accessor function, floating points should produce an error: this is the role of `_fp_chk:w`. We could make floating point variables be protected to prevent them from expanding under `x`-expansion, but it seems more convenient to treat them as a subcase of token list variables.

The (decimal part of the) IEEE-754-2008 standard requires the format to be able to represent special floating point numbers besides the usual positive and negative cases. We distinguish the various possibilities by their $\langle case \rangle$, which is a single digit:

- 0 zeros: `+0` and `-0`,
- 1 “normal” numbers (positive and negative),
- 2 infinities: `+inf` and `-inf`,
- 3 quiet and signalling nan.

The $\langle sign \rangle$ is 0 (positive) or 2 (negative), except in the case of nan, which have $\langle sign \rangle = 1$. This ensures that changing the $\langle sign \rangle$ digit to $2 - \langle sign \rangle$ is exactly equivalent to changing the sign of the number.

Special floating point numbers have the form

`\s_fp _fp_chk:w ⟨case⟩ ⟨sign⟩ \s_fp_... ;`

where `\s_fp_...` is a scan mark carrying information about how the number was formed (useful for debugging).

Normal floating point numbers ($\langle case \rangle = 1$) have the form

`\s_fp _fp_chk:w 1 ⟨sign⟩ {⟨exponent⟩} {⟨X1⟩} {⟨X2⟩} {⟨X3⟩} {⟨X4⟩} ;`

Here, the $\langle exponent \rangle$ is an integer, between -10000 and 10000 . The body consists in four blocks of exactly 4 digits, $0000 \leq \langle X_i \rangle \leq 9999$, and the floating point is

$$(-1)^{\langle sign \rangle / 2} \langle X_1 \rangle \langle X_2 \rangle \langle X_3 \rangle \langle X_4 \rangle \cdot 10^{\langle exponent \rangle - 16}$$

where we have concatenated the 16 digits. Currently, floating point numbers are normalized such that the $\langle exponent \rangle$ is minimal, in other words, $1000 \leq \langle X_1 \rangle \leq 9999$.

Calculations are done in base 10000, *i.e.* one myriad.

25.3 Using arguments and semicolons

`_fp_use_none_stop_f:n` This function removes an argument (typically a digit) and replaces it by `\exp_stop_f:`, a marker which stops f-type expansion.

```
15885 \cs_new:Npn \_fp_use_none_stop_f:n #1 { \exp_stop_f: }
```

(End definition for _fp_use_none_stop_f:n.)

`_fp_use_s:n` Those functions place a semicolon after one or two arguments (typically digits).

```
\_fp_use_s:nn
15886 \cs_new:Npn \_fp_use_s:n #1 { #1; }
15887 \cs_new:Npn \_fp_use_s:nn #1#2 { #1#2; }
```

(End definition for _fp_use_s:n and _fp_use_s:nn.)

`_fp_use_none_until_s:w` Those functions select specific arguments among a set of arguments delimited by a semicolon.
`_fp_use_i_until_s:nw`
`_fp_use_ii_until_s:nnw`

```
15888 \cs_new:Npn \_fp_use_none_until_s:w #1; { }
15889 \cs_new:Npn \_fp_use_i_until_s:nw #1#2; {#1}
15890 \cs_new:Npn \_fp_use_ii_until_s:nnw #1#2#3; {#2}
```

(End definition for _fp_use_none_until_s:w, _fp_use_i_until_s:nw, and _fp_use_ii_until_s:nnw.)

`_fp_reverse_args:Nww` Many internal functions take arguments delimited by semicolons, and it is occasionally useful to swap two such arguments.

```
15891 \cs_new:Npn \_fp_reverse_args:Nww #1 #2; #3; { #1 #3; #2; }
```

(End definition for _fp_reverse_args:Nww.)

`_fp_rrot:www` Rotate three arguments delimited by semicolons. This is the inverse (or the square) of the Forth primitive ROT, hence the name.

```
15892 \cs_new:Npn \_fp_rrot:www #1; #2; #3; { #2; #3; #1; }
```

(End definition for _fp_rrot:www.)

`_fp_use_i:ww` Many internal functions take arguments delimited by semicolons, and it is occasionally useful to remove one or two such arguments.
`_fp_use_i:www`

```
15893 \cs_new:Npn \_fp_use_i:ww #1; #2; { #1; }
15894 \cs_new:Npn \_fp_use_i:www #1; #2; #3; { #1; }
```

(End definition for _fp_use_i:ww and _fp_use_i:www.)

25.4 Constants, and structure of floating points

`_fp_misused:n` This receives a floating point object (floating point number or tuple) and generates an error stating that it was misused. This is called when for instance an `fp` variable is left in the input stream and its contents reach TeX's stomach.

```
15895 \cs_new_protected:Npn \_fp_misused:n #1
15896 { \_kernel_msg_error:nnx { kernel } { misused-fp } { \fp_to_tl:n {#1} } }
```

(End definition for _fp_misused:n.)

`\s__fp` Floating points numbers all start with `\s__fp __fp_chk:w`, where `\s__fp` is equal to the `\relax` primitive, and `__fp_chk:w` is protected. The rest of the floating point number is made of characters (or `\relax`). This ensures that nothing expands under `f`-expansion, nor under `x`-expansion. However, when typeset, `\s__fp` does nothing, and `__fp_chk:w` is expanded. We define `__fp_chk:w` to produce an error.

```

15897 \scan_new:N \s__fp
15898 \cs_new_protected:Npn \__fp_chk:w #1 ;
15899 { \__fp_misused:n { \s__fp \__fp_chk:w #1 ; } }

```

(End definition for `\s__fp` and `__fp_chk:w`.)

`\s__fp_expr_mark` Aliases of `\tex_relax:D`, used to terminate expressions.

`\s__fp_expr_stop`

```

15900 \scan_new:N \s__fp_expr_mark
15901 \scan_new:N \s__fp_expr_stop

```

(End definition for `\s__fp_expr_mark` and `\s__fp_expr_stop`.)

`\s__fp_mark` Generic scan marks used throughout the module.

`\s__fp_stop`

```

15902 \scan_new:N \s__fp_mark
15903 \scan_new:N \s__fp_stop

```

(End definition for `\s__fp_mark` and `\s__fp_stop`.)

`_fp_use_i_delimit_by_s_stop:nw` Functions to gobble up to a scan mark.

```

15904 \cs_new:Npn \_fp_use_i_delimit_by_s_stop:nw #1 #2 \s__fp_stop {#1}

```

(End definition for `_fp_use_i_delimit_by_s_stop:nw`.)

`\s__fp_invalid` A couple of scan marks used to indicate where special floating point numbers come from.

`\s__fp_underflow`

`\s__fp_overflow`

`\s__fp_division`

`\s__fp_exact`

```

15905 \scan_new:N \s__fp_invalid
15906 \scan_new:N \s__fp_underflow
15907 \scan_new:N \s__fp_overflow
15908 \scan_new:N \s__fp_division
15909 \scan_new:N \s__fp_exact

```

(End definition for `\s__fp_invalid` and others.)

`\c_zero_fp` The special floating points. We define the floating points here as “exact”.

`\c_minus_zero_fp`

`\c_inf_fp`

`\c_minus_inf_fp`

`\c_nan_fp`

```

15910 \tl_const:Nn \c_zero_fp { \s__fp \__fp_chk:w 0 0 \s__fp_exact ; }
15911 \tl_const:Nn \c_minus_zero_fp { \s__fp \__fp_chk:w 0 2 \s__fp_exact ; }
15912 \tl_const:Nn \c_inf_fp { \s__fp \__fp_chk:w 2 0 \s__fp_exact ; }
15913 \tl_const:Nn \c_minus_inf_fp { \s__fp \__fp_chk:w 2 2 \s__fp_exact ; }
15914 \tl_const:Nn \c_nan_fp { \s__fp \__fp_chk:w 3 1 \s__fp_exact ; }

```

(End definition for `\c_zero_fp` and others. These variables are documented on page 208.)

`\c__fp_prec_int` The number of digits of floating points.

`\c__fp_half_prec_int`

`\c__fp_block_int`

```

15915 \int_const:Nn \c__fp_prec_int { 16 }
15916 \int_const:Nn \c__fp_half_prec_int { 8 }
15917 \int_const:Nn \c__fp_block_int { 4 }

```

(End definition for `\c__fp_prec_int`, `\c__fp_half_prec_int`, and `\c__fp_block_int`.)

`\c__fp_myriad_int` Blocks have 4 digits so this integer is useful.

```

15918 \int_const:Nn \c__fp_myriad_int { 10000 }

```

(End definition for `\c__fp_myriad_int`.)

`\c__fp_minus_min_exponent_int` `\c__fp_max_exponent_int` Normal floating point numbers have an exponent between `-minus_min_exponent` and `max_exponent` inclusive. Larger numbers are rounded to $\pm\infty$. Smaller numbers are rounded to ± 0 . It would be more natural to define a `min_exponent` with the opposite sign but that would waste one T_EX count.

```
15919 \int_const:Nn \c__fp_minus_min_exponent_int { 10000 }
15920 \int_const:Nn \c__fp_max_exponent_int { 10000 }
```

(End definition for `\c__fp_minus_min_exponent_int` and `\c__fp_max_exponent_int`.)

`\c__fp_max_exp_exponent_int` If a number's exponent is larger than that, its exponential overflows/underflows.

```
15921 \int_const:Nn \c__fp_max_exp_exponent_int { 5 }
```

(End definition for `\c__fp_max_exp_exponent_int`.)

`\c__fp_overflowing_fp` A floating point number that is bigger than all normal floating point numbers. This replaces infinities when converting to formats that do not support infinities.

```
15922 \tl_const:Nx \c__fp_overflowing_fp
15923 {
15924   \s__fp \__fp_chk:w 1 0
15925   { \int_eval:n { \c__fp_max_exponent_int + 1 } }
15926   {1000} {0000} {0000} {0000} ;
15927 }
```

(End definition for `\c__fp_overflowing_fp`.)

`__fp_zero_fp:N` `__fp_inf_fp:N` In case of overflow or underflow, we have to output a zero or infinity with a given sign.

```
15928 \cs_new:Npn \__fp_zero_fp:N #1
15929 { \s__fp \__fp_chk:w 0 #1 \s__fp_underflow ; }
15930 \cs_new:Npn \__fp_inf_fp:N #1
15931 { \s__fp \__fp_chk:w 2 #1 \s__fp_overflow ; }
```

(End definition for `__fp_zero_fp:N` and `__fp_inf_fp:N`.)

`__fp_exponent:w` For normal numbers, the function expands to the exponent, otherwise to 0. This is used in `l3str-format`.

```
15932 \cs_new:Npn \__fp_exponent:w \s__fp \__fp_chk:w #1
15933 {
15934   \if_meaning:w 1 #1
15935     \exp_after:wN \__fp_use_ii_until_s:nnw
15936   \else:
15937     \exp_after:wN \__fp_use_i_until_s:nw
15938     \exp_after:wN 0
15939   \fi:
15940 }
```

(End definition for `__fp_exponent:w`.)

`__fp_neg_sign:N` When appearing in an integer expression or after `\int_value:w`, this expands to the sign opposite to `#1`, namely 0 (positive) is turned to 2 (negative), 1 (nan) to 1, and 2 to 0.

```
15941 \cs_new:Npn \__fp_neg_sign:N #1
15942 { \__fp_int_eval:w 2 - #1 \__fp_int_eval_end: }
```

(End definition for `__fp_neg_sign:N`.)

`__fp_kind:w` Expands to 0 for zeros, 1 for normal floating point numbers, 2 for infinities, 3 for NaN, 4 for tuples.

```

15943 \cs_new:Npn \__fp_kind:w #1
15944 {
15945   \__fp_if_type_fp:NTwFw
15946   #1 \__fp_use_ii_until_s:nnw
15947   \s__fp { \__fp_use_i_until_s:nw 4 }
15948   \s__fp_stop
15949 }

```

(End definition for `__fp_kind:w`.)

25.5 Overflow, underflow, and exact zero

`__fp_sanitize:Nw` Expects the sign and the exponent in some order, then the significand (which we don't touch). Outputs the corresponding floating point number, possibly underflowed to ± 0
`__fp_sanitize:wN` or overflowed to $\pm\infty$. The functions `__fp_underflow:w` and `__fp_overflow:w` are defined in `l3fp-traps`.
`__fp_sanitize_zero:w`

```

15950 \cs_new:Npn \__fp_sanitize:Nw #1 #2;
15951 {
15952   \if_case:w
15953     \if_int_compare:w #2 > \c__fp_max_exponent_int 1 ~ \else:
15954     \if_int_compare:w #2 < - \c__fp_minus_min_exponent_int 2 ~ \else:
15955     \if_meaning:w 1 #1 3 ~ \fi: \fi: \fi: 0 ~
15956     \or: \exp_after:wN \__fp_overflow:w
15957     \or: \exp_after:wN \__fp_underflow:w
15958     \or: \exp_after:wN \__fp_sanitize_zero:w
15959     \fi:
15960     \s__fp \__fp_chk:w 1 #1 {#2}
15961   }
15962 \cs_new:Npn \__fp_sanitize:wN #1; #2 { \__fp_sanitize:Nw #2 #1; }
15963 \cs_new:Npn \__fp_sanitize_zero:w \s__fp \__fp_chk:w #1 #2 #3;
15964 { \c_zero_fp }

```

(End definition for `__fp_sanitize:Nw`, `__fp_sanitize:wN`, and `__fp_sanitize_zero:w`.)

25.6 Expanding after a floating point number

`__fp_exp_after_o:w` `__fp_exp_after_o:w` *<floating point>*
`__fp_exp_after_f:nw` `__fp_exp_after_f:nw` *{<tokens>}* *<floating point>*
Places *<tokens>* (empty in the case of `__fp_exp_after_o:w`) between the *<floating point>* and the following tokens, then hits those tokens with `o` or `f`-expansion, and leaves the floating point number unchanged.

We first distinguish normal floating points, which have a significand, from the much simpler special floating points.

```

15965 \cs_new:Npn \__fp_exp_after_o:w \s__fp \__fp_chk:w #1
15966 {
15967   \if_meaning:w 1 #1
15968     \exp_after:wN \__fp_exp_after_normal:nNNw
15969   \else:
15970     \exp_after:wN \__fp_exp_after_special:nNNw
15971   \fi:
15972   { }

```

```

15973     #1
15974   }
15975 \cs_new:Npn \__fp_exp_after_f:nw #1 \s__fp \__fp_chk:w #2
15976   {
15977     \if_meaning:w 1 #2
15978       \exp_after:wN \__fp_exp_after_normal:nNNw
15979     \else:
15980       \exp_after:wN \__fp_exp_after_special:nNNw
15981     \fi:
15982     { \exp:w \exp_end_continue_f:w #1 }
15983     #2
15984   }

```

(End definition for __fp_exp_after_o:w and __fp_exp_after_f:nw.)

__fp_exp_after_special:nNNw __fp_exp_after_special:nNNw {<after>} <case> <sign> <scan mark> ;
Special floating point numbers are easy to jump over since they contain few tokens.

```

15985 \cs_new:Npn \__fp_exp_after_special:nNNw #1#2#3#4;
15986   {
15987     \exp_after:wN \s__fp
15988     \exp_after:wN \__fp_chk:w
15989     \exp_after:wN #2
15990     \exp_after:wN #3
15991     \exp_after:wN #4
15992     \exp_after:wN ;
15993     #1
15994   }

```

(End definition for __fp_exp_after_special:nNNw.)

__fp_exp_after_normal:nNNw For normal floating point numbers, life is slightly harder, since we have many tokens to jump over. Here it would be slightly better if the digits were not braced but instead were delimited arguments (for instance delimited by ,). That may be changed some day.

```

15995 \cs_new:Npn \__fp_exp_after_normal:nNNw #1 1 #2 #3 #4#5#6#7;
15996   {
15997     \exp_after:wN \__fp_exp_after_normal:Nwwwww
15998     \exp_after:wN #2
15999     \int_value:w #3 \exp_after:wN ;
16000     \int_value:w 1 #4 \exp_after:wN ;
16001     \int_value:w 1 #5 \exp_after:wN ;
16002     \int_value:w 1 #6 \exp_after:wN ;
16003     \int_value:w 1 #7 \exp_after:wN ; #1
16004   }
16005 \cs_new:Npn \__fp_exp_after_normal:Nwwwww
16006   #1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;
16007   { \s__fp \__fp_chk:w 1 #1 {#2} {#3} {#4} {#5} {#6} ; }

```

(End definition for __fp_exp_after_normal:nNNw.)

25.7 Other floating point types

\s__fp_tuple Floating point tuples take the form \s__fp_tuple __fp_tuple_chk:w { <fp 1> <fp 2>
__fp_tuple_chk:w ... } ; where each <fp> is a floating point number or tuple, hence ends with ; itself. When
\c__fp_empty_tuple_fp

a tuple is typeset, `__fp_tuple_chk:w` produces an error, just like usual floating point numbers. Tuples may have zero or one element.

```

16008 \scan_new:N \s__fp_tuple
16009 \cs_new_protected:Npn \__fp_tuple_chk:w #1 ;
16010 { \__fp_misused:n { \s__fp_tuple \__fp_tuple_chk:w #1 ; } }
16011 \tl_const:Nn \c__fp_empty_tuple_fp
16012 { \s__fp_tuple \__fp_tuple_chk:w { } ; }

```

(End definition for `\s__fp_tuple`, `__fp_tuple_chk:w`, and `\c__fp_empty_tuple_fp`.)

`__fp_tuple_count:w` Count the number of items in a tuple of floating points by counting semicolons. The technique is very similar to `\tl_count:n`, but with the loop built-in. Checking for the end of the loop is done with the `\use_none:n #1` construction.

```

16013 \cs_new:Npn \__fp_array_count:n #1
16014 { \__fp_tuple_count:w \s__fp_tuple \__fp_tuple_chk:w {#1} ; }
16015 \cs_new:Npn \__fp_tuple_count:w \s__fp_tuple \__fp_tuple_chk:w #1 ;
16016 {
16017   \int_value:w \__fp_int_eval:w 0
16018   \__fp_tuple_count_loop:Nw #1 { ? \prg_break: } ;
16019   \prg_break_point:
16020   \__fp_int_eval_end:
16021 }
16022 \cs_new:Npn \__fp_tuple_count_loop:Nw #1#2;
16023 { \use_none:n #1 + 1 \__fp_tuple_count_loop:Nw }

```

(End definition for `__fp_tuple_count:w`, `__fp_array_count:n`, and `__fp_tuple_count_loop:Nw`.)

`__fp_if_type_fp:NTwFw` Used as `__fp_if_type_fp:NTwFw <marker> {<true code>} \s__fp {<false code>} \s__fp_stop`, this test whether the `<marker>` is `\s__fp` or not and runs the appropriate `<code>`. The very unusual syntax is for optimization purposes as that function is used for all floating point operations.

```

16024 \cs_new:Npn \__fp_if_type_fp:NTwFw #1 \s__fp #2 #3 \s__fp_stop {#2}

```

(End definition for `__fp_if_type_fp:NTwFw`.)

`__fp_array_if_all_fp:nTF` True if all items are floating point numbers. Used for min.
`__fp_array_if_all_fp_loop:w`

```

16025 \cs_new:Npn \__fp_array_if_all_fp:nTF #1
16026 {
16027   \__fp_array_if_all_fp_loop:w #1 { \s__fp \prg_break: } ;
16028   \prg_break_point: \use_i:nn
16029 }
16030 \cs_new:Npn \__fp_array_if_all_fp_loop:w #1#2 ;
16031 {
16032   \__fp_if_type_fp:NTwFw
16033   #1 \__fp_array_if_all_fp_loop:w
16034   \s__fp { \prg_break:n \use_iii:nnn }
16035   \s__fp_stop
16036 }

```

(End definition for `__fp_array_if_all_fp:nTF` and `__fp_array_if_all_fp_loop:w`.)

`__fp_type_from_scan:N` Used as `__fp_type_from_scan:N` *<token>*. Grabs the pieces of the stringified *<token>* which lies after the first `s__fp`. If the *<token>* does not contain that string, the result is `_?`.
`__fp_type_from_scan_other:N`
`__fp_type_from_scan:w`

```

16037 \cs_new:Npn \__fp_type_from_scan:N #1
16038 {
16039   \__fp_if_type_fp:NTwFw
16040   #1 { }
16041   \s__fp { \__fp_type_from_scan_other:N #1 }
16042   \s__fp_stop
16043 }
16044 \cs_new:Npx \__fp_type_from_scan_other:N #1
16045 {
16046   \exp_not:N \exp_after:wN \exp_not:N \__fp_type_from_scan:w
16047   \exp_not:N \token_to_str:N #1 \s__fp_mark
16048   \tl_to_str:n { s__fp _? } \s__fp_mark \s__fp_stop
16049 }
16050 \exp_last_unbraced:NNNNo
16051   \cs_new:Npn \__fp_type_from_scan:w #1
16052   { \tl_to_str:n { s__fp } } #2 \s__fp_mark #3 \s__fp_stop {#2}

```

(End definition for `__fp_type_from_scan:N`, `__fp_type_from_scan_other:N`, and `__fp_type_from_scan:w`.)

`__fp_change_func_type:NNN` Arguments are *<type marker>* *<function>* *<recovery>*. This gives the function obtained by placing the type after `@@`. If the function is not defined then *<recovery>* *<function>* is used instead; however that test is not run when the *<type marker>* is `\s__fp`.
`__fp_change_func_type_aux:w`
`_fp_change_func_type_chk:NNN`

```

16053 \cs_new:Npn \__fp_change_func_type:NNN #1#2#3
16054 {
16055   \__fp_if_type_fp:NTwFw
16056   #1 #2
16057   \s__fp
16058   {
16059     \exp_after:wN \__fp_change_func_type_chk:NNN
16060     \cs:w
16061     __fp \__fp_type_from_scan_other:N #1
16062     \exp_after:wN \__fp_change_func_type_aux:w \token_to_str:N #2
16063     \cs_end:
16064     #2 #3
16065   }
16066   \s__fp_stop
16067 }
16068 \exp_last_unbraced:NNNNo
16069   \cs_new:Npn \__fp_change_func_type_aux:w #1 { \tl_to_str:n { __fp } } { }
16070 \cs_new:Npn \__fp_change_func_type_chk:NNN #1#2#3
16071 {
16072   \if_meaning:w \scan_stop: #1
16073   \exp_after:wN #3 \exp_after:wN #2
16074   \else:
16075   \exp_after:wN #1
16076   \fi:
16077 }

```

(End definition for `__fp_change_func_type:NNN`, `__fp_change_func_type_aux:w`, and `__fp_change_func_type_chk:NNN`.)

```

\__fp_exp_after_any_f:Nnw
\__fp_exp_after_any_f:nw
  \__fp_exp_after_expr_stop_f:nw

```

The `Nnw` function simply dispatches to the appropriate `__fp_exp_after..._f:nw` with “...” (either empty or $\langle type \rangle$) extracted from `#1`, which should start with `\s__fp`. If it doesn’t start with `\s__fp` the function `__fp_exp_after?..._f:nw` defined in `l3fp-parse` gives an error; another special $\langle type \rangle$ is `stop`, useful for loops, see below. The `nw` function has an important optimization for floating points numbers; it also fetches its type marker `#2` from the floating point.

```

16078 \cs_new:Npn \__fp_exp_after_any_f:Nnw #1
16079   { \cs:w __fp_exp_after \__fp_type_from_scan_other:N #1 _f:nw \cs_end: }
16080 \cs_new:Npn \__fp_exp_after_any_f:nw #1#2
16081   {
16082     \__fp_if_type_fp:NTwFw
16083     #2 \__fp_exp_after_f:nw
16084     \s__fp { \__fp_exp_after_any_f:Nnw #2 }
16085     \s__fp_stop
16086     {#1} #2
16087   }
16088 \cs_new_eq:NN \__fp_exp_after_expr_stop_f:nw \use_none:nn

```

(End definition for `__fp_exp_after_any_f:Nnw`, `__fp_exp_after_any_f:nw`, and `__fp_exp_after_expr_stop_f:nw`.)

```

\__fp_exp_after_tuple_o:w
\__fp_exp_after_tuple_f:nw
\__fp_exp_after_array_f:w

```

The loop works by using the `n` argument of `__fp_exp_after_any_f:nw` to place the loop macro after the next item in the tuple and expand it.

```

\__fp_exp_after_array_f:w
 $\langle fp_1 \rangle$  ;
...
 $\langle fp_n \rangle$  ;
\s__fp_expr_stop

```

```

16089 \cs_new:Npn \__fp_exp_after_tuple_o:w
16090   { \__fp_exp_after_tuple_f:nw { \exp_after:wN \exp_stop_f: } }
16091 \cs_new:Npn \__fp_exp_after_tuple_f:nw
16092   #1 \s__fp_tuple \__fp_tuple_chk:w #2 ;
16093   {
16094     \exp_after:wN \s__fp_tuple
16095     \exp_after:wN \__fp_tuple_chk:w
16096     \exp_after:wN {
16097       \exp:w \exp_end_continue_f:w
16098       \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
16099       \exp_after:wN }
16100       \exp_after:wN ;
16101       \exp:w \exp_end_continue_f:w #1
16102     }
16103 \cs_new:Npn \__fp_exp_after_array_f:w
16104   { \__fp_exp_after_any_f:nw { \__fp_exp_after_array_f:w } }

```

(End definition for `__fp_exp_after_tuple_o:w`, `__fp_exp_after_tuple_f:nw`, and `__fp_exp_after_array_f:w`.)

25.8 Packing digits

When a positive integer `#1` is known to be less than 10^8 , the following trick splits it into two blocks of 4 digits, padding with zeros on the left.

```

\cs_new:Npn \pack:NNNNNw #1 #2#3#4#5 #6; { {#2#3#4#5} {#6} }
\exp_after:wN \pack:NNNNNw
  \__fp_int_value:w \__fp_int_eval:w 1 0000 0000 + #1 ;

```

The idea is that adding 10^8 to the number ensures that it has exactly 9 digits, and can then easily find which digits correspond to what position in the number. Of course, this can be modified for any number of digits less or equal to 9 (we are limited by $\text{T}_\text{E}\text{X}$'s integers). This method is very heavily relied upon in `l3fp-basics`.

More specifically, the auxiliary inserts `+ #1#2#3#4#5 ; {#6}`, which allows us to compute several blocks of 4 digits in a nested manner, performing carries on the fly. Say we want to compute 12345×66778899 . With simplified names, we would do

```

\exp_after:wN \post_processing:w
\__fp_int_value:w \__fp_int_eval:w - 5 0000
  \exp_after:wN \pack:NNNNNw
    \__fp_int_value:w \__fp_int_eval:w 4 9995 0000
      + 12345 * 6677
    \exp_after:wN \pack:NNNNNw
      \__fp_int_value:w \__fp_int_eval:w 5 0000 0000
        + 12345 * 8899 ;

```

The `\exp_after:wN` triggers `\int_value:w __fp_int_eval:w`, which starts a first computation, whose initial value is $-5\,0000$ (the “leading shift”). In that computation appears an `\exp_after:wN`, which triggers the nested computation `\int_value:w __fp_int_eval:w` with starting value $4\,9995\,0000$ (the “middle shift”). That, in turn, expands `\exp_after:wN` which triggers the third computation. The third computation's value is $5\,0000\,0000 + 12345 \times 8899$, which has 9 digits. Adding $5 \cdot 10^8$ to the product allowed us to know how many digits to expect as long as the numbers to multiply are not too big; it also works to some extent with negative results. The `pack` function puts the last 4 of those 9 digits into a brace group, moves the semi-colon delimiter, and inserts a `+`, which combines the carry with the previous computation. The shifts nicely combine into $5\,0000\,0000/10^4 + 4\,9995\,0000 = 5\,0000\,0000$. As long as the operands are in some range, the result of this second computation has 9 digits. The corresponding `pack` function, expanded after the result is computed, braces the last 4 digits, and leaves `+ {5 digits}` for the initial computation. The “leading shift” cancels the combination of the other shifts, and the `\post_processing:w` takes care of packing the last few digits.

Admittedly, this is quite intricate. It is probably the key in making `l3fp` as fast as other pure $\text{T}_\text{E}\text{X}$ floating point units despite its increased precision. In fact, this is used so much that we provide different sets of packing functions and shifts, depending on ranges of input.

```

\__fp_pack:NNNNNw This set of shifts allows for computations involving results in the range  $[-4 \cdot 10^8, 5 \cdot 10^8 - 1]$ .
\c__fp_trailing_shift_int Shifted values all have exactly 9 digits.
\c__fp_middle_shift_int
\c__fp_leading_shift_int
16105 \int_const:Nn \c__fp_leading_shift_int { - 5 0000 }
16106 \int_const:Nn \c__fp_middle_shift_int { 5 0000 * 9999 }
16107 \int_const:Nn \c__fp_trailing_shift_int { 5 0000 * 10000 }
16108 \cs_new:Npn \__fp_pack:NNNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }

```

(End definition for `__fp_pack:NNNNNw` and others.)

```

\__fp_pack_big:NNNNNNw This set of shifts allows for computations involving results in the range  $[-5 \cdot 10^8, 6 \cdot 10^8 - 1]$ 
\c__fp_big_trailing_shift_int (actually a bit more). Shifted values all have exactly 10 digits. Note that the upper
\c__fp_big_middle_shift_int
\c__fp_big_leading_shift_int

```

bound is due to $\text{T}_{\text{E}}\text{X}$'s limit of $2^{31} - 1$ on integers. The shifts are chosen to be roughly the mid-point of 10^9 and 2^{31} , the two bounds on 10-digit integers in $\text{T}_{\text{E}}\text{X}$.

```

16109 \int_const:Nn \c__fp_big_leading_shift_int { - 15 2374 }
16110 \int_const:Nn \c__fp_big_middle_shift_int { 15 2374 * 9999 }
16111 \int_const:Nn \c__fp_big_trailing_shift_int { 15 2374 * 10000 }
16112 \cs_new:Npn \__fp_pack_big:NNNNNNw #1#2 #3#4#5#6 #7;
16113 { + #1#2#3#4#5#6 ; {#7} }

```

(End definition for `__fp_pack_big:NNNNNNw` and others.)

```

\__fp_pack_Bigg:NNNNNNw
\c__fp_Bigg_trailing_shift_int
\c__fp_Bigg_middle_shift_int
\c__fp_Bigg_leading_shift_int

```

This set of shifts allows for computations with results in the range $[-1 \cdot 10^9, 147483647]$; the end-point is $2^{31} - 1 - 2 \cdot 10^9 \simeq 1.47 \cdot 10^8$. Shifted values all have exactly 10 digits.

```

16114 \int_const:Nn \c__fp_Bigg_leading_shift_int { - 20 0000 }
16115 \int_const:Nn \c__fp_Bigg_middle_shift_int { 20 0000 * 9999 }
16116 \int_const:Nn \c__fp_Bigg_trailing_shift_int { 20 0000 * 10000 }
16117 \cs_new:Npn \__fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7;
16118 { + #1#2#3#4#5#6 ; {#7} }

```

(End definition for `__fp_pack_Bigg:NNNNNNw` and others.)

```
\__fp_pack_twice_four:wNNNNNNNN
```

```
\__fp_pack_twice_four:wNNNNNNNN <tokens> ; <≥ 8 digits>
```

Grabs two sets of 4 digits and places them before the semi-colon delimiter. Putting several copies of this function before a semicolon packs more digits since each takes the digits packed by the others in its first argument.

```

16119 \cs_new:Npn \__fp_pack_twice_four:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
16120 { #1 {#2#3#4#5} {#6#7#8#9} ; }

```

(End definition for `__fp_pack_twice_four:wNNNNNNNN`.)

```
\__fp_pack_eight:wNNNNNNNN
```

```
\__fp_pack_eight:wNNNNNNNN <tokens> ; <≥ 8 digits>
```

Grabs one set of 8 digits and places them before the semi-colon delimiter as a single group. Putting several copies of this function before a semicolon packs more digits since each takes the digits packed by the others in its first argument.

```

16121 \cs_new:Npn \__fp_pack_eight:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
16122 { #1 {#2#3#4#5#6#7#8#9} ; }

```

(End definition for `__fp_pack_eight:wNNNNNNNN`.)

```

\__fp_basics_pack_low:NNNNNw
\__fp_basics_pack_high:NNNNNw
\__fp_basics_pack_high_carry:w

```

Addition and multiplication of significands are done in two steps: first compute a (more or less) exact result, then round and pack digits in the final (braced) form. These functions take care of the packing, with special attention given to the case where rounding has caused a carry. Since rounding can only shift the final digit by 1, a carry always produces an exact power of 10. Thus, `__fp_basics_pack_high_carry:w` is always followed by four times `{0000}`.

This is used in `l3fp-basics` and `l3fp-extended`.

```

16123 \cs_new:Npn \__fp_basics_pack_low:NNNNNw #1 #2#3#4#5 #6;
16124 { + #1 - 1 ; {#2#3#4#5} {#6} ; }
16125 \cs_new:Npn \__fp_basics_pack_high:NNNNNw #1 #2#3#4#5 #6;
16126 {
16127   \if_meaning:w 2 #1
16128     \__fp_basics_pack_high_carry:w
16129   \fi:
16130   ; {#2#3#4#5} {#6}

```

```

16131 }
16132 \cs_new:Npn \__fp_basics_pack_high_carry:w \fi: ; #1
16133 { \fi: + 1 ; {1000} }

```

(End definition for `__fp_basics_pack_low:NNNNw`, `__fp_basics_pack_high:NNNNw`, and `__fp_basics_pack_high_carry:w`.)

`__fp_basics_pack_weird_low:NNNNw`
`__fp_basics_pack_weird_high:NNNNNNNw`

This is used in `l3fp-basics` for additions and divisions. Their syntax is confusing, hence the name.

```

16134 \cs_new:Npn \__fp_basics_pack_weird_low:NNNNw #1 #2#3#4 #5;
16135 {
16136   \if_meaning:w 2 #1
16137     + 1
16138   \fi:
16139   \__fp_int_eval_end:
16140   #2#3#4; {#5} ;
16141 }
16142 \cs_new:Npn \__fp_basics_pack_weird_high:NNNNNNNw
16143 1 #1#2#3#4 #5#6#7#8 #9; { ; {#1#2#3#4} {#5#6#7#8} {#9} }

```

(End definition for `__fp_basics_pack_weird_low:NNNNw` and `__fp_basics_pack_weird_high:NNNNNNNw`.)

25.9 Decimate (dividing by a power of 10)

`__fp_decimate:nNnnnn`

`__fp_decimate:nNnnnn {⟨shift⟩} ⟨f1⟩`
`{⟨X1⟩} {⟨X2⟩} {⟨X3⟩} {⟨X4⟩}`

Each $\langle X_i \rangle$ consists in 4 digits exactly, and $1000 \leq \langle X_1 \rangle < 9999$. The first argument determines by how much we shift the digits. $\langle f_1 \rangle$ is called as follows:

$\langle f_1 \rangle \langle \text{rounding} \rangle \{ \langle X'_1 \rangle \} \{ \langle X'_2 \rangle \} \langle \text{extra-digits} \rangle ;$

where $0 \leq \langle X'_i \rangle < 10^8 - 1$ are 8 digit integers, forming the truncation of our number. In other words,

$$\left(\sum_{i=1}^4 \langle X_i \rangle \cdot 10^{-4i} \cdot 10^{-\langle \text{shift} \rangle} \right) - (\langle X'_1 \rangle \cdot 10^{-8} + \langle X'_2 \rangle \cdot 10^{-16}) = 0. \langle \text{extra-digits} \rangle \cdot 10^{-16} \in [0, 10^{-16}).$$

To round properly later, we need to remember some information about the difference. The $\langle \text{rounding} \rangle$ digit is 0 if and only if the difference is exactly 0, and 5 if and only if the difference is exactly $0.5 \cdot 10^{-16}$. Otherwise, it is the (non-0, non-5) digit closest to 10^{17} times the difference. In particular, if the shift is 17 or more, all the digits are dropped, $\langle \text{rounding} \rangle$ is 1 (not 0), and $\langle X'_1 \rangle$ and $\langle X'_2 \rangle$ are both zero.

If the shift is 1, the $\langle \text{rounding} \rangle$ digit is simply the only digit that was pushed out of the brace groups (this is important for subtraction). It would be more natural for the $\langle \text{rounding} \rangle$ digit to be placed after the $\langle X'_i \rangle$, but the choice we make involves less reshuffling.

Note that this function treats negative $\langle \text{shift} \rangle$ as 0.

```

16144 \cs_new:Npn \__fp_decimate:nNnnnn #1
16145 {
16146   \cs:w
16147   __fp_decimate_
16148   \if_int_compare:w \__fp_int_eval:w #1 > \c__fp_prec_int
16149     tiny

```

```

16150     \else:
16151       \__fp_int_to_roman:w \__fp_int_eval:w #1
16152     \fi:
16153     :Nnnnn
16154   \cs_end:
16155 }

```

Each of the auxiliaries see the function $\langle f_1 \rangle$, followed by 4 blocks of 4 digits.

(End definition for `__fp_decimate:nNnnnn`.)

```

\__fp_decimate_:Nnnnn
\__fp_decimate_tiny:Nnnnn

```

If the $\langle shift \rangle$ is zero, or too big, life is very easy.

```

16156 \cs_new:Npn \__fp_decimate_:Nnnnn #1 #2#3#4#5
16157 { #1 0 {#2#3} {#4#5} ; }
16158 \cs_new:Npn \__fp_decimate_tiny:Nnnnn #1 #2#3#4#5
16159 { #1 1 { 0000 0000 } { 0000 0000 } 0 #2#3#4#5 ; }

```

(End definition for `__fp_decimate_:Nnnnn` and `__fp_decimate_tiny:Nnnnn`.)

```

\__fp_decimate_auxi:Nnnnn
\__fp_decimate_auxii:Nnnnn
\__fp_decimate_auxiii:Nnnnn
\__fp_decimate_auxiv:Nnnnn
\__fp_decimate_auxv:Nnnnn
\__fp_decimate_auxvi:Nnnnn
\__fp_decimate_auxvii:Nnnnn
\__fp_decimate_auxviii:Nnnnn
\__fp_decimate_auxix:Nnnnn
\__fp_decimate_auxx:Nnnnn
\__fp_decimate_auxxi:Nnnnn
\__fp_decimate_auxxii:Nnnnn
\__fp_decimate_auxxiii:Nnnnn
\__fp_decimate_auxxiv:Nnnnn
\__fp_decimate_auxxv:Nnnnn
\__fp_decimate_auxxvi:Nnnnn

```

```

\__fp_decimate_auxi:Nnnnn \langle f_1 \rangle \{ \langle X_1 \rangle \} \{ \langle X_2 \rangle \} \{ \langle X_3 \rangle \} \{ \langle X_4 \rangle \}

```

Shifting happens in two steps: compute the $\langle rounding \rangle$ digit, and repack digits into two blocks of 8. The sixteen functions are very similar, and defined through `__fp_tmp:w`. The arguments are as follows: #1 indicates which function is being defined; after one step of expansion, #2 yields the “extra digits” which are then converted by `__fp_round_digit:Nw` to the $\langle rounding \rangle$ digit (note the + separating blocks of digits to avoid overflowing TeX’s integers). This triggers the f-expansion of `__fp_decimate_pack:nnnnnnnnnw`,⁸ responsible for building two blocks of 8 digits, and removing the rest. For this to work, #3 alternates between braced and unbraced blocks of 4 digits, in such a way that the 5 first and 5 next token groups yield the correct blocks of 8 digits.

```

16160 \cs_new:Npn \__fp_tmp:w #1 #2 #3
16161 {
16162   \cs_new:cpn { __fp_decimate_ #1 :Nnnnn } ##1 ##2##3##4##5
16163   {
16164     \exp_after:wN ##1
16165     \int_value:w
16166     \exp_after:wN \__fp_round_digit:Nw #2 ;
16167     \__fp_decimate_pack:nnnnnnnnnw #3 ;
16168   }
16169 }
16170 \__fp_tmp:w {i} {\use_none:nnn #50}{ 0{#2}#3{#4}#5 }
16171 \__fp_tmp:w {ii} {\use_none:nn #5 }{ 00{#2}#3{#4}#5 }
16172 \__fp_tmp:w {iii} {\use_none:n #5 }{ 000{#2}#3{#4}#5 }
16173 \__fp_tmp:w {iv} { #5 }{ {0000}#2{#3}#4 #5 }
16174 \__fp_tmp:w {v} {\use_none:nnn #4#5 }{ 0{0000}#2{#3}#4 #5 }
16175 \__fp_tmp:w {vi} {\use_none:nn #4#5 }{ 00{0000}#2{#3}#4 #5 }
16176 \__fp_tmp:w {vii} {\use_none:n #4#5 }{ 000{0000}#2{#3}#4 #5 }
16177 \__fp_tmp:w {viii}{ #4#5 }{ {0000}0000{#2}#3 #4 #5 }
16178 \__fp_tmp:w {ix} {\use_none:nnn #3#4+#5}{ 0{0000}0000{#2}#3 #4 #5 }
16179 \__fp_tmp:w {x} {\use_none:nn #3#4+#5}{ 00{0000}0000{#2}#3 #4 #5 }
16180 \__fp_tmp:w {xi} {\use_none:n #3#4+#5}{ 000{0000}0000{#2}#3 #4 #5 }
16181 \__fp_tmp:w {xii} { #3#4+#5}{ {0000}0000{0000}#2 #3 #4 #5 }
16182 \__fp_tmp:w {xiii}{\use_none:nnn#2#3+#4#5}{ 0{0000}0000{0000}#2 #3 #4 #5 }

```

⁸No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

```

16183 \__fp_tmp:w {xiv} {\use_none:nn #2#3+#4#5}{ 00{0000}0000{0000}#2 #3 #4 #5 }
16184 \__fp_tmp:w {xv} {\use_none:n #2#3+#4#5}{ 000{0000}0000{0000}#2 #3 #4 #5 }
16185 \__fp_tmp:w {xvi} { #2#3+#4#5}{0000}0000{0000}0000 #2 #3 #4 #5}

```

(End definition for __fp_decimate_auxi:Nnnnn and others.)

__fp_decimate_pack:nnnnnnnnnw

The computation of the *rounding* digit leaves an unfinished \int_value:w, which expands the following functions. This allows us to repack nicely the digits we keep. Those digits come as an alternation of unbraced and braced blocks of 4 digits, such that the first 5 groups of token consist in 4 single digits, and one brace group (in some order), and the next 5 have the same structure. This is followed by some digits and a semicolon.

```

16186 \cs_new:Npn \__fp_decimate_pack:nnnnnnnnnw #1#2#3#4#5
16187 { \__fp_decimate_pack:nnnnnnw { #1#2#3#4#5 } }
16188 \cs_new:Npn \__fp_decimate_pack:nnnnnnw #1 #2#3#4#5#6
16189 { {#1} {#2#3#4#5#6} }

```

(End definition for __fp_decimate_pack:nnnnnnnnnw.)

25.10 Functions for use within primitive conditional branches

The functions described in this section are not pretty and can easily be misused. When correctly used, each of them removes one \fi: as part of its parameter text, and puts one back as part of its replacement text.

Many computation functions in l3fp must perform tests on the type of floating points that they receive. This is often done in an \if_case:w statement or another conditional statement, and only a few cases lead to actual computations: most of the special cases are treated using a few standard functions which we define now. A typical use context for those functions would be

```

\if_case:w <integer> \exp_stop_f:
  \__fp_case_return_o:Nw <fp var>
\or: \__fp_case_use:nw {<some computation>}
\or: \__fp_case_return_same_o:w
\or: \__fp_case_return:nw {<something>}
\fi:
<junk>
<floating point>

```

In this example, the case 0 returns the floating point *<fp var>*, expanding once after that floating point. Case 1 does *<some computation>* using the *<floating point>* (presumably compute the operation requested by the user in that non-trivial case). Case 2 returns the *<floating point>* without modifying it, removing the *<junk>* and expanding once after. Case 3 closes the conditional, removes the *<junk>* and the *<floating point>*, and expands *<something>* next. In other cases, the “*<junk>*” is expanded, performing some other operation on the *<floating point>*. We provide similar functions with two trailing *<floating points>*.

__fp_case_use:nw

This function ends a TeX conditional, removes junk until the next floating point, and places its first argument before that floating point, to perform some operation on the floating point.

```

16190 \cs_new:Npn \__fp_case_use:nw #1#2 \fi: #3 \s__fp { \fi: #1 \s__fp }

```

(End definition for __fp_case_use:nw.)

`__fp_case_return:nw` This function ends a \TeX conditional, removes junk and a floating point, and places its first argument in the input stream. A quirk is that we don't define this function requiring a floating point to follow, simply anything ending in a semicolon. This, in turn, means that the *<junk>* may not contain semicolons.

```
16191 \cs_new:Npn \__fp_case_return:nw #1#2 \fi: #3 ; { \fi: #1 }
```

(End definition for `__fp_case_return:nw`.)

`__fp_case_return_o:Nw` This function ends a \TeX conditional, removes junk and a floating point, and returns its first argument (an *<fp var>*) then expands once after it.

```
16192 \cs_new:Npn \__fp_case_return_o:Nw #1#2 \fi: #3 \s__fp #4 ;
16193 { \fi: \exp_after:wN #1 }
```

(End definition for `__fp_case_return_o:Nw`.)

`__fp_case_return_same_o:w` This function ends a \TeX conditional, removes junk, and returns the following floating point, expanding once after it.

```
16194 \cs_new:Npn \__fp_case_return_same_o:w #1 \fi: #2 \s__fp
16195 { \fi: \__fp_exp_after_o:w \s__fp }
```

(End definition for `__fp_case_return_same_o:w`.)

`__fp_case_return_o:Nww` Same as `__fp_case_return_o:Nw` but with two trailing floating points.

```
16196 \cs_new:Npn \__fp_case_return_o:Nww #1#2 \fi: #3 \s__fp #4 ; #5 ;
16197 { \fi: \exp_after:wN #1 }
```

(End definition for `__fp_case_return_o:Nww`.)

`__fp_case_return_i_o:ww` Similar to `__fp_case_return_same_o:w`, but this returns the first or second of two trailing floating point numbers, expanding once after the result.

`__fp_case_return_ii_o:ww`

```
16198 \cs_new:Npn \__fp_case_return_i_o:ww #1 \fi: #2 \s__fp #3 ; \s__fp #4 ;
16199 { \fi: \__fp_exp_after_o:w \s__fp #3 ; }
16200 \cs_new:Npn \__fp_case_return_ii_o:ww #1 \fi: #2 \s__fp #3 ;
16201 { \fi: \__fp_exp_after_o:w }
```

(End definition for `__fp_case_return_i_o:ww` and `__fp_case_return_ii_o:ww`.)

25.11 Integer floating points

`__fp_int_p:w` Tests if the floating point argument is an integer. For normal floating point numbers, `__fp_int:wTF` this holds if the rounding digit resulting from `__fp_decimate:nNnnnn` is 0.

```
16202 \prg_new_conditional:Npnn \__fp_int:w \s__fp \__fp_chk:w #1 #2 #3 #4;
16203 { TF , T , F , p }
16204 {
16205   \if_case:w #1 \exp_stop_f:
16206     \prg_return_true:
16207   \or:
16208     \if_charcode:w 0
16209       \__fp_decimate:nNnnnn { \c__fp_prec_int - #3 }
16210       \__fp_use_i_until_s:nw #4
16211       \prg_return_true:
16212     \else:
16213       \prg_return_false:
16214     \fi:
```



```

16215     \else: \prg_return_false:
16216     \fi:
16217 }

```

(End definition for _fp_int:wTF.)

25.12 Small integer floating points

_fp_small_int:wTF Tests if the floating point argument is an integer or $\pm\infty$. If so, it is clipped to an integer in the range $[-10^8, 10^8]$ and fed as a braced argument to the *⟨true code⟩*. Otherwise, the *⟨false code⟩* is performed.

First filter special cases: zeros and infinities are integers, `nan` is not. For normal numbers, decimate. If the rounding digit is not 0 run the *⟨false code⟩*. If it is, then the integer is `#2 #3`; use `#3` if `#2` vanishes and otherwise `108`.

```

16218 \cs_new:Npn \_fp_small_int:wTF \s__fp \_fp_chk:w #1#2
16219 {
16220   \if_case:w #1 \exp_stop_f:
16221     \_fp_case_return:nw { \_fp_small_int_true:wTF 0 ; }
16222   \or:   \exp_after:wN \_fp_small_int_normal:NnwTF
16223   \or:
16224     \_fp_case_return:nw
16225     {
16226       \exp_after:wN \_fp_small_int_true:wTF \int_value:w
16227       \if_meaning:w 2 #2 - \fi: 1 0000 0000 ;
16228     }
16229   \else: \_fp_case_return:nw \use_ii:nn
16230   \fi:
16231   #2
16232 }
16233 \cs_new:Npn \_fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
16234 \cs_new:Npn \_fp_small_int_normal:NnwTF #1#2#3;
16235 {
16236   \_fp_decimate:nNnnnn { \c__fp_prec_int - #2 }
16237   \_fp_small_int_test:NnnwNw
16238   #3 #1
16239 }
16240 \cs_new:Npn \_fp_small_int_test:NnnwNw #1#2#3#4; #5
16241 {
16242   \if_meaning:w 0 #1
16243     \exp_after:wN \_fp_small_int_true:wTF
16244     \int_value:w \if_meaning:w 2 #5 - \fi:
16245     \if_int_compare:w #2 > 0 \exp_stop_f:
16246       1 0000 0000
16247     \else:
16248       #3
16249     \fi:
16250     \exp_after:wN ;
16251   \else:
16252     \exp_after:wN \use_ii:nn
16253   \fi:
16254 }

```

(End definition for _fp_small_int:wTF and others.)

25.13 Fast string comparison

`__fp_str_if_eq:nn` A private version of the low-level string comparison function. As the nature of the arguments is restricted and as speed is of the essence, this version does not seek to deal with `#` tokens. No `l3sys` or `l3luatex` just yet so we have to define in terms of primitives.

```

16255 \sys_if_engine luatex:TF
16256 {
16257   \cs_new:Npn \__fp_str_if_eq:nn #1#2
16258   {
16259     \tex_directlua:D
16260     {
16261       l3kernel.strcmp
16262       (
16263         " \tex_luaescapestring:D {#1}",
16264         " \tex_luaescapestring:D {#2}"
16265       )
16266     }
16267   }
16268 }
16269 { \cs_new_eq:NN \__fp_str_if_eq:nn \tex_strcmp:D }

```

(End definition for `__fp_str_if_eq:nn`.)

25.14 Name of a function from its `l3fp`-parse name

`__fp_func_to_name:N` The goal is to convert for instance `__fp_sin_o:w` to `sin`. This is used in error messages
`__fp_func_to_name_aux:w` hence does not need to be fast.

```

16270 \cs_new:Npn \__fp_func_to_name:N #1
16271 {
16272   \exp_last_unbraced:Nf
16273   \__fp_func_to_name_aux:w { \cs_to_str:N #1 } X
16274 }
16275 \cs_set_protected:Npn \__fp_tmp:w #1 #2
16276 { \cs_new:Npn \__fp_func_to_name_aux:w ##1 #1 ##2 #2 ##3 X {##2} }
16277 \exp_args:Nff \__fp_tmp:w { \tl_to_str:n { __fp_ } }
16278 { \tl_to_str:n { _o: } }

```

(End definition for `__fp_func_to_name:N` and `__fp_func_to_name_aux:w`.)

25.15 Messages

Using a floating point directly is an error.

```

16279 \__kernel_msg_new:nnnn { kernel } { misused-fp }
16280 { A~floating~point~with~value~'~#1'~was~misused. }
16281 {
16282   To~obtain~the~value~of~a~floating~point~variable,~use~
16283   '\token_to_str:N \fp_to_decimal:N',~
16284   '\token_to_str:N \fp_to_tl:N',~or~other~
16285   conversion~functions.
16286 }
16287 </package>

```

26 13fp-traps Implementation

16288 `<*package>`

16289 `<@@=fp>`

Exceptions should be accessed by an `n`-type argument, among

- `invalid_operation`
- `division_by_zero`
- `overflow`
- `underflow`
- `inexact` (actually never used).

26.1 Flags

`flag_fp_invalid_operation`
`flag_fp_division_by_zero`
`flag_fp_overflow`
`flag_fp_underflow`

Flags to denote exceptions.

16290 `\flag_new:n { fp_invalid_operation }`

16291 `\flag_new:n { fp_division_by_zero }`

16292 `\flag_new:n { fp_overflow }`

16293 `\flag_new:n { fp_underflow }`

(End definition for flag `fp_invalid_operation` and others. These variables are documented on page 210.)

26.2 Traps

Exceptions can be trapped to obtain custom behaviour. When an invalid operation or a division by zero is trapped, the trap receives as arguments the result as an `N`-type floating point number, the function name (multiple letters for prefix operations, or a single symbol for infix operations), and the operand(s). When an overflow or underflow is trapped, the trap receives the resulting overly large or small floating point number if it is not too big, otherwise it receives $+\infty$. Currently, the `inexact` exception is entirely ignored.

The behaviour when an exception occurs is controlled by the definitions of the functions

- `__fp_invalid_operation:nnw,`
- `__fp_invalid_operation_o:Nww,`
- `__fp_invalid_operation_tl_o:ff,`
- `__fp_division_by_zero_o:Nnw,`
- `__fp_division_by_zero_o:NNww,`
- `__fp_overflow:w,`
- `__fp_underflow:w.`

Rather than changing them directly, we provide a user interface as `\fp_trap:nn {<exception>} {<way of trapping>}`, where the `<way of trapping>` is one of `error`, `flag`, or `none`.

We also provide `__fp_invalid_operation_o:nw`, defined in terms of `__fp_invalid_operation:nnw`.

\fp_trap:nn

```
16294 \cs_new_protected:Npn \fp_trap:nn #1#2
16295 {
16296   \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
16297   {
16298     \clist_if_in:nnTF
16299     { invalid_operation , division_by_zero , overflow , underflow }
16300     {#1}
16301     {
16302       \__kernel_msg_error:nnxx { kernel }
16303       { unknown-fpu-trap-type } {#1} {#2}
16304     }
16305     {
16306       \__kernel_msg_error:nnx
16307       { kernel } { unknown-fpu-exception } {#1}
16308     }
16309   }
16310 }
```

(End definition for \fp_trap:nn. This function is documented on page 210.)

_fp_trap_invalid_operation_set_error: We provide three types of trapping for invalid operations: either produce an error and
_fp_trap_invalid_operation_set_flag: raise the relevant flag; or only raise the flag; or don't even raise the flag. In most cases,
_fp_trap_invalid_operation_set_none: the function produces as a result its first argument, possibly with post-expansion.

```
\_fp_trap_invalid_operation_set:N
16311 \cs_new_protected:Npn \__fp_trap_invalid_operation_set_error:
16312 { \__fp_trap_invalid_operation_set:N \prg_do_nothing: }
16313 \cs_new_protected:Npn \__fp_trap_invalid_operation_set_flag:
16314 { \__fp_trap_invalid_operation_set:N \use_none:nnnnn }
16315 \cs_new_protected:Npn \__fp_trap_invalid_operation_set_none:
16316 { \__fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
16317 \cs_new_protected:Npn \__fp_trap_invalid_operation_set:N #1
16318 {
16319   \exp_args:Nno \use:n
16320   { \cs_set:Npn \__fp_invalid_operation:nnw ##1##2##3; }
16321   {
16322     #1
16323     \__fp_error:nnfn { fp-invalid } {##2} { \fp_to_tl:n { ##3; } } { }
16324     \flag_raise_if_clear:n { fp_invalid_operation }
16325     ##1
16326   }
16327   \exp_args:Nno \use:n
16328   { \cs_set:Npn \__fp_invalid_operation_o:Nww ##1##2; ##3; }
16329   {
16330     #1
16331     \__fp_error:nffn { fp-invalid-ii }
16332     { \fp_to_tl:n { ##2; } } { \fp_to_tl:n { ##3; } } {##1}
16333     \flag_raise_if_clear:n { fp_invalid_operation }
16334     \exp_after:wN \c_nan_fp
16335   }
16336   \exp_args:Nno \use:n
16337   { \cs_set:Npn \__fp_invalid_operation_tl_o:ff ##1##2 }
16338   {
16339     #1
16340     \__fp_error:nffn { fp-invalid } {##1} {##2} { }
```

```

16341         \flag_raise_if_clear:n { fp_invalid_operation }
16342         \exp_after:wN \c_nan_fp
16343     }
16344 }

```

(End definition for `__fp_trap_invalid_operation_set_error:` and others.)

`__fp_trap_division_by_zero_set_error:` We provide three types of trapping for invalid operations and division by zero: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In all cases, the function must produce a result, namely its first argument, $\pm\infty$ or NaN.

```

16345 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_error:
16346 { \__fp_trap_division_by_zero_set:N \prg_do_nothing: }
16347 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_flag:
16348 { \__fp_trap_division_by_zero_set:N \use_none:nnnnn }
16349 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_none:
16350 { \__fp_trap_division_by_zero_set:N \use_none:nnnnnnn }
16351 \cs_new_protected:Npn \__fp_trap_division_by_zero_set:N #1
16352 {
16353     \exp_args:Nno \use:n
16354     { \cs_set:Npn \__fp_division_by_zero_o:NNw ##1##2##3; }
16355     {
16356         #1
16357         \__fp_error:nfn { fp-zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
16358         \flag_raise_if_clear:n { fp_division_by_zero }
16359         \exp_after:wN ##1
16360     }
16361     \exp_args:Nno \use:n
16362     { \cs_set:Npn \__fp_division_by_zero_o:NNww ##1##2##3; ##4; }
16363     {
16364         #1
16365         \__fp_error:nfn { fp-zero-div-ii }
16366         { \fp_to_tl:n { ##3; } } { \fp_to_tl:n { ##4; } } {##2}
16367         \flag_raise_if_clear:n { fp_division_by_zero }
16368         \exp_after:wN ##1
16369     }
16370 }

```

(End definition for `__fp_trap_division_by_zero_set_error:` and others.)

`__fp_trap_overflow_set_error:` Just as for invalid operations and division by zero, the three different behaviours are obtained by feeding `\prg_do_nothing:`, `\use_none:nnnnn` or `\use_none:nnnnnnn` to an auxiliary, with a further auxiliary common to overflow and underflow functions. In most cases, the argument of the `__fp_overflow:w` and `__fp_underflow:w` functions will be an (almost) normal number (with an exponent outside the allowed range), and the error message thus displays that number together with the result to which it overflowed or underflowed. For extreme cases such as `10 ** 1e9999`, the exponent would be too large for T_EX, and `__fp_overflow:w` receives $\pm\infty$ (`__fp_underflow:w` would receive ± 0); then we cannot do better than simply say an overflow or underflow occurred.

```

16371 \cs_new_protected:Npn \__fp_trap_overflow_set_error:
16372 { \__fp_trap_overflow_set:N \prg_do_nothing: }
16373 \cs_new_protected:Npn \__fp_trap_overflow_set_flag:
16374 { \__fp_trap_overflow_set:N \use_none:nnnnn }
16375 \cs_new_protected:Npn \__fp_trap_overflow_set_none:

```

```

16376 { \__fp_trap_overflow_set:N \use_none:nnnnnnn }
16377 \cs_new_protected:Npn \__fp_trap_overflow_set:N #1
16378 { \__fp_trap_overflow_set:NnNn #1 { overflow } \__fp_inf_fp:N { inf } }
16379 \cs_new_protected:Npn \__fp_trap_underflow_set_error:
16380 { \__fp_trap_underflow_set:N \prg_do_nothing: }
16381 \cs_new_protected:Npn \__fp_trap_underflow_set_flag:
16382 { \__fp_trap_underflow_set:N \use_none:nnnnn }
16383 \cs_new_protected:Npn \__fp_trap_underflow_set_none:
16384 { \__fp_trap_underflow_set:N \use_none:nnnnnnn }
16385 \cs_new_protected:Npn \__fp_trap_underflow_set:N #1
16386 { \__fp_trap_overflow_set:NnNn #1 { underflow } \__fp_zero_fp:N { 0 } }
16387 \cs_new_protected:Npn \__fp_trap_overflow_set:NnNn #1#2#3#4
16388 {
16389   \exp_args:Nno \use:n
16390   { \cs_set:cpn { \__fp_ #2 :w } \s__fp \__fp_chk:w ##1##2##3; }
16391   {
16392     #1
16393     \__fp_error:nffn
16394     { fp-flow \if_meaning:w 1 ##1 -to \fi: }
16395     { \fp_to_tl:n { \s__fp \__fp_chk:w ##1##2##3; } }
16396     { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }
16397     {#2}
16398     \flag_raise_if_clear:n { fp_#2 }
16399     #3 ##2
16400   }
16401 }

```

(End definition for __fp_trap_overflow_set_error: and others.)

<pre> __fp_invalid_operation:nnw __fp_invalid_operation_o:Nnw __fp_invalid_operation_tl_o:ff __fp_division_by_zero_o:Nnw __fp_division_by_zero_o:NNww __fp_overflow:w __fp_underflow:w </pre>	<p>Initialize the control sequences (to log properly their existence). Then set invalid operations to trigger an error, and division by zero, overflow, and underflow to act silently on their flag.</p> <pre> 16402 \cs_new:Npn __fp_invalid_operation:nnw #1#2#3; { } 16403 \cs_new:Npn __fp_invalid_operation_o:Nnw #1#2; #3; { } 16404 \cs_new:Npn __fp_invalid_operation_tl_o:ff #1 #2 { } 16405 \cs_new:Npn __fp_division_by_zero_o:Nnw #1#2#3; { } 16406 \cs_new:Npn __fp_division_by_zero_o:NNww #1#2#3; #4; { } 16407 \cs_new:Npn __fp_overflow:w { } 16408 \cs_new:Npn __fp_underflow:w { } 16409 \fp_trap:nn { invalid_operation } { error } 16410 \fp_trap:nn { division_by_zero } { flag } 16411 \fp_trap:nn { overflow } { flag } 16412 \fp_trap:nn { underflow } { flag } </pre>
--	--

(End definition for __fp_invalid_operation:nnw and others.)

<pre> __fp_invalid_operation_o:nw __fp_invalid_operation_o:fw </pre>	<p>Convenient short-hands for returning \c_nan_fp for a unary or binary operation, and expanding after.</p> <pre> 16413 \cs_new:Npn __fp_invalid_operation_o:nw 16414 { __fp_invalid_operation:nnw { \exp_after:wN \c_nan_fp } } 16415 \cs_generate_variant:Nn __fp_invalid_operation_o:nw { f } </pre>
--	--

(End definition for __fp_invalid_operation_o:nw.)

26.3 Errors

```

\__fp_error:nnnn
\__fp_error:nnfn 16416 \cs_new:Npn \__fp_error:nnnn
\__fp_error:nffn 16417 { \__kernel_msg_expandable_error:nnnnn { kernel } }
\__fp_error:nfff 16418 \cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff , nfff }

(End definition for \__fp_error:nnnn.)

```

26.4 Messages

Some messages.

```

16419 \__kernel_msg_new:nnnn { kernel } { unknown-fpu-exception }
16420 {
16421   The~FPU~exception~'#1'~is~not~known:~
16422   that~trap~will~never~be~triggered.
16423 }
16424 {
16425   The~only~exceptions~to~which~traps~can~be~attached~are \
16426   \iow_indent:n
16427   {
16428     * ~ invalid_operation \
16429     * ~ division_by_zero \
16430     * ~ overflow \
16431     * ~ underflow
16432   }
16433 }
16434 \__kernel_msg_new:nnnn { kernel } { unknown-fpu-trap-type }
16435 { The~FPU~trap~type~'#2'~is~not~known. }
16436 {
16437   The~trap~type~must~be~one~of \
16438   \iow_indent:n
16439   {
16440     * ~ error \
16441     * ~ flag \
16442     * ~ none
16443   }
16444 }
16445 \__kernel_msg_new:nnn { kernel } { fp-flow }
16446 { An ~ #3 ~ occurred. }
16447 \__kernel_msg_new:nnn { kernel } { fp-flow-to }
16448 { #1 ~ #3 ed ~ to ~ #2 . }
16449 \__kernel_msg_new:nnn { kernel } { fp-zero-div }
16450 { Division~by~zero~in~ #1 (#2) }
16451 \__kernel_msg_new:nnn { kernel } { fp-zero-div-ii }
16452 { Division~by~zero~in~ (#1) #3 (#2) }
16453 \__kernel_msg_new:nnn { kernel } { fp-invalid }
16454 { Invalid~operation~ #1 (#2) }
16455 \__kernel_msg_new:nnn { kernel } { fp-invalid-ii }
16456 { Invalid~operation~ (#1) #3 (#2) }
16457 \__kernel_msg_new:nnn { kernel } { fp-unknown-type }
16458 { Unknown~type~for~'#1' }
16459 \end{package}

```

27 13fp-round implementation

```

16460 (*package)
16461 (@@=fp)

__fp_parse_word_trunc:N
__fp_parse_word_floor:N
__fp_parse_word_ceil:N
16462 \cs_new:Npn __fp_parse_word_trunc:N
16463   { __fp_parse_function:NNN __fp_round_o:Nw __fp_round_to_zero:NNN }
16464 \cs_new:Npn __fp_parse_word_floor:N
16465   { __fp_parse_function:NNN __fp_round_o:Nw __fp_round_to_ninf:NNN }
16466 \cs_new:Npn __fp_parse_word_ceil:N
16467   { __fp_parse_function:NNN __fp_round_o:Nw __fp_round_to_pinf:NNN }

(End definition for __fp_parse_word_trunc:N, __fp_parse_word_floor:N, and __fp_parse_word_ceil:N.)

__fp_parse_word_round:N
__fp_parse_round:Nw
16468 \cs_new:Npn __fp_parse_word_round:N #1#2
16469   {
16470     __fp_parse_function:NNN
16471     __fp_round_o:Nw __fp_round_to_nearest:NNN #1
16472     #2
16473   }
16474 \cs_new:Npn __fp_parse_round:Nw #1 #2 __fp_round_to_nearest:NNN #3#4
16475   { #2 #1 #3 }
16476

(End definition for __fp_parse_word_round:N and __fp_parse_round:Nw.)

```

27.1 Rounding tools

`\c_fp_five_int` This is used as the half-point for which numbers are rounded up/down.

```

16477 \int_const:Nn \c_fp_five_int { 5 }

```

(End definition for `\c_fp_five_int`.)

Floating point operations often yield a result that cannot be exactly represented in a significand with 16 digits. In that case, we need to round the exact result to a representable number. The IEEE standard defines four rounding modes:

- Round to nearest: round to the representable floating point number whose absolute difference with the exact result is the smallest. If the exact result lies exactly at the mid-point between two consecutive representable floating point numbers, round to the floating point number whose last digit is even.
- Round towards negative infinity: round to the greatest floating point number not larger than the exact result.
- Round towards zero: round to a floating point number with the same sign as the exact result, with the largest absolute value not larger than the absolute value of the exact result.
- Round towards positive infinity: round to the least floating point number not smaller than the exact result.

This is not fully implemented in l3fp yet, and transcendental functions fall back on the “round to nearest” mode. All rounding for basic algebra is done through the functions defined in this module, which can be redefined to change their rounding behaviour (but there is not interface for that yet).

The rounding tools available in this module are many variations on a base function `__fp_round:NNN`, which expands to `0\exp_stop_f:` or `1\exp_stop_f:` depending on whether the final result should be rounded up or down.

- `__fp_round:NNN <sign> <digit1> <digit2>` can expand to `0\exp_stop_f:` or `1\exp_stop_f:.`
- `__fp_round_s:NNNw <sign> <digit1> <digit2> <more digits>;` can expand to `0\exp_stop_f:;` or `1\exp_stop_f:;`
- `__fp_round_neg:NNN <sign> <digit1> <digit2>` can expand to `0\exp_stop_f:` or `1\exp_stop_f:.`

See implementation comments for details on the syntax.

```
\__fp_round:NNN
\__fp_round_to_nearest:NNN
  \__fp_round_to_nearest_ninf:NNN
  \__fp_round_to_nearest_zero:NNN
  \__fp_round_to_nearest_pinf:NNN
\__fp_round_to_ninf:NNN
\__fp_round_to_zero:NNN
\__fp_round_to_pinf:NNN
```

`__fp_round:NNN <final sign> <digit1> <digit2>`
If rounding the number $\langle final\ sign\rangle\langle digit_1\rangle.\langle digit_2\rangle$ to an integer rounds it towards zero (truncates it), this function expands to `0\exp_stop_f:`, and otherwise to `1\exp_stop_f:.` Typically used within the scope of an `__fp_int_eval:w`, to add 1 if needed, and thereby round correctly. The result depends on the rounding mode.

It is very important that $\langle final\ sign\rangle$ be the final sign of the result. Otherwise, the result would be incorrect in the case of rounding towards $-\infty$ or towards $+\infty$. Also recall that $\langle final\ sign\rangle$ is 0 for positive, and 2 for negative.

By default, the functions below return `0\exp_stop_f:`, but this is superseded by `__fp_round_return_one:`, which instead returns `1\exp_stop_f:`, expanding everything and removing `0\exp_stop_f:` in the process. In the case of rounding towards $\pm\infty$ or towards 0, this is not really useful, but it prepares us for the “round to nearest, ties to even” mode.

The “round to nearest” mode is the default. If the $\langle digit_2\rangle$ is larger than 5, then round up. If it is less than 5, round down. If it is exactly 5, then round such that $\langle digit_1\rangle$ plus the result is even. In other words, round up if $\langle digit_1\rangle$ is odd.

The “round to nearest” mode has three variants, which differ in how ties are rounded: down towards $-\infty$, truncated towards 0, or up towards $+\infty$.

```
16478 \cs_new:Npn \__fp_round_return_one:
16479   { \exp_after:wN 1 \exp_after:wN \exp_stop_f: \exp:w }
16480 \cs_new:Npn \__fp_round_to_ninf:NNN #1 #2 #3
16481   {
16482     \if_meaning:w 2 #1
16483     \if_int_compare:w #3 > 0 \exp_stop_f:
16484       \__fp_round_return_one:
16485     \fi:
16486     \fi:
16487     0 \exp_stop_f:
16488   }
16489 \cs_new:Npn \__fp_round_to_zero:NNN #1 #2 #3 { 0 \exp_stop_f: }
16490 \cs_new:Npn \__fp_round_to_pinf:NNN #1 #2 #3
16491   {
16492     \if_meaning:w 0 #1
16493     \if_int_compare:w #3 > 0 \exp_stop_f:
```

```

16494         \_fp_round_return_one:
16495         \fi:
16496     \fi:
16497     0 \exp_stop_f:
16498 }
16499 \cs_new:Npn \_fp_round_to_nearest:NNN #1 #2 #3
16500 {
16501     \if_int_compare:w #3 > \c__fp_five_int
16502         \_fp_round_return_one:
16503     \else:
16504         \if_meaning:w 5 #3
16505             \if_int_odd:w #2 \exp_stop_f:
16506             \_fp_round_return_one:
16507         \fi:
16508     \fi:
16509     \fi:
16510     0 \exp_stop_f:
16511 }
16512 \cs_new:Npn \_fp_round_to_nearest_ninf:NNN #1 #2 #3
16513 {
16514     \if_int_compare:w #3 > \c__fp_five_int
16515         \_fp_round_return_one:
16516     \else:
16517         \if_meaning:w 5 #3
16518             \if_meaning:w 2 #1
16519                 \_fp_round_return_one:
16520             \fi:
16521         \fi:
16522     \fi:
16523     0 \exp_stop_f:
16524 }
16525 \cs_new:Npn \_fp_round_to_nearest_zero:NNN #1 #2 #3
16526 {
16527     \if_int_compare:w #3 > \c__fp_five_int
16528         \_fp_round_return_one:
16529     \fi:
16530     0 \exp_stop_f:
16531 }
16532 \cs_new:Npn \_fp_round_to_nearest_pinf:NNN #1 #2 #3
16533 {
16534     \if_int_compare:w #3 > \c__fp_five_int
16535         \_fp_round_return_one:
16536     \else:
16537         \if_meaning:w 5 #3
16538             \if_meaning:w 0 #1
16539                 \_fp_round_return_one:
16540             \fi:
16541         \fi:
16542     \fi:
16543     0 \exp_stop_f:
16544 }
16545 \cs_new_eq:NN \_fp_round:NNN \_fp_round_to_nearest:NNN

```

(End definition for _fp_round:NNN and others.)

_fp_round_s:NNNw

_fp_round_s:NNNw <final sign> <digit> <more digits> ;

Similar to _fp_round:NNN, but with an extra semicolon, this function expands to 0\exp_stop_f:; if rounding <final sign><digit>.<more digits> to an integer truncates, and to 1\exp_stop_f:; otherwise. The <more digits> part must be a digit, followed by something that does not overflow a \int_use:N _fp_int_eval:w construction. The only relevant information about this piece is whether it is zero or not.

```

16546 \cs_new:Npn \_fp_round_s:NNNw #1 #2 #3 #4;
16547 {
16548   \exp_after:wN \_fp_round:NNN
16549   \exp_after:wN #1
16550   \exp_after:wN #2
16551   \int_value:w \_fp_int_eval:w
16552   \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
16553   \if_meaning:w 5 #3 1 \fi:
16554   \exp_stop_f:
16555   \if_int_compare:w \_fp_int_eval:w #4 > 0 \exp_stop_f:
16556   1 +
16557   \fi:
16558   \fi:
16559   #3
16560   ;
16561 }

```

(End definition for _fp_round_s:NNNw.)

_fp_round_digit:Nw

\int_value:w _fp_round_digit:Nw <digit> <intexpr> ;

This function should always be called within an \int_value:w or _fp_int_eval:w expansion; it may add an extra _fp_int_eval:w, which means that the integer or integer expression should not be ended with a synonym of \relax, but with a semi-colon for instance.

```

16562 \cs_new:Npn \_fp_round_digit:Nw #1 #2;
16563 {
16564   \if_int_odd:w \if_meaning:w 0 #1 1 \else:
16565   \if_meaning:w 5 #1 1 \else:
16566   0 \fi: \fi: \exp_stop_f:
16567   \if_int_compare:w \_fp_int_eval:w #2 > 0 \exp_stop_f:
16568   \_fp_int_eval:w 1 +
16569   \fi:
16570   \fi:
16571   #1
16572 }

```

(End definition for _fp_round_digit:Nw.)

_fp_round_neg:NNN

_fp_round_neg:NNN <final sign> <digit₁> <digit₂>

_fp_round_to_nearest_neg:NNN

This expands to 0\exp_stop_f: or 1\exp_stop_f: after doing the following test.

_fp_round_to_nearest_ninf_neg:NNN

Starting from a number of the form <final sign>0.<15 digits><digit₁> with exactly 15 (non-all-zero) digits before <digit₁>, subtract from it <final sign>0.0...0<digit₂>, where there are 16 zeros. If in the current rounding mode the result should be rounded down, then this function returns 1\exp_stop_f:. Otherwise, i.e., if the result is rounded back to the first operand, then this function returns 0\exp_stop_f:.

_fp_round_to_nearest_zero_neg:NNN

_fp_round_to_nearest_pinf_neg:NNN

_fp_round_to_ninf_neg:NNN

_fp_round_to_zero_neg:NNN

_fp_round_to_pinf_neg:NNN

It turns out that this negative “round to nearest” is identical to the positive one. And this is the default mode.

```

16573 \cs_new_eq:NN \__fp_round_to_ninf_neg:NNN \__fp_round_to_pinf:NNN
16574 \cs_new:Npn \__fp_round_to_zero_neg:NNN #1 #2 #3
16575 {
16576     \if_int_compare:w #3 > 0 \exp_stop_f:
16577     \__fp_round_return_one:
16578     \fi:
16579     0 \exp_stop_f:
16580 }
16581 \cs_new_eq:NN \__fp_round_to_pinf_neg:NNN \__fp_round_to_ninf:NNN
16582 \cs_new_eq:NN \__fp_round_to_nearest_neg:NNN \__fp_round_to_nearest:NNN
16583 \cs_new_eq:NN \__fp_round_to_nearest_ninf_neg:NNN
16584 \__fp_round_to_nearest_pinf:NNN
16585 \cs_new:Npn \__fp_round_to_nearest_zero_neg:NNN #1 #2 #3
16586 {
16587     \if_int_compare:w #3 < \c__fp_five_int \else:
16588     \__fp_round_return_one:
16589     \fi:
16590     0 \exp_stop_f:
16591 }
16592 \cs_new_eq:NN \__fp_round_to_nearest_pinf_neg:NNN
16593 \__fp_round_to_nearest_ninf:NNN
16594 \cs_new_eq:NN \__fp_round_neg:NNN \__fp_round_to_nearest_neg:NNN

```

(End definition for __fp_round_neg:NNN and others.)

27.2 The round function

__fp_round_o:Nw
__fp_round_aux_o:Nw

First check that all arguments are floating point numbers. The `trunc`, `ceil` and `floor` functions expect one or two arguments (the second is 0 by default), and the `round` function also accepts a third argument (`nan` by default), which changes `#1` from `__fp_round_to_nearest:NNN` to one of its analogues.

```

16595 \cs_new:Npn \__fp_round_o:Nw #1
16596 {
16597     \__fp_parse_function_all_fp_o:fnw
16598     { \__fp_round_name_from_cs:N #1 }
16599     { \__fp_round_aux_o:Nw #1 }
16600 }
16601 \cs_new:Npn \__fp_round_aux_o:Nw #1#2 @
16602 {
16603     \if_case:w
16604     \__fp_int_eval:w \__fp_array_count:n {#2} \__fp_int_eval_end:
16605     \__fp_round_no_arg_o:Nw #1 \exp:w
16606     \or: \__fp_round:Nwn #1 #2 {0} \exp:w
16607     \or: \__fp_round:Nww #1 #2 \exp:w
16608     \else: \__fp_round:Nwww #1 #2 @ \exp:w
16609     \fi:
16610     \exp_after:wN \exp_end:
16611 }

```

(End definition for __fp_round_o:Nw and __fp_round_aux_o:Nw.)

__fp_round_no_arg_o:Nw

```

16612 \cs_new:Npn \__fp_round_no_arg_o:Nw #1
16613 {

```

```

16614 \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
16615 { \__fp_error:nnnn { fp-num-args } { round () } { 1 } { 3 } }
16616 {
16617   \__fp_error:nffn { fp-num-args }
16618   { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
16619 }
16620 \exp_after:wN \c_nan_fp
16621 }

```

(End definition for __fp_round_no_arg_o:Nw.)

__fp_round:Nwww Having three arguments is only allowed for round, not trunc, ceil, floor, so check for that case. If all is well, construct one of __fp_round_to_nearest:NNN, __fp_round_to_nearest_zero:NNN, __fp_round_to_nearest_ninf:NNN, __fp_round_to_nearest_pinf:NNN and act accordingly.

```

16622 \cs_new:Npn \__fp_round:Nwww #1#2 ; #3 ; \s__fp \__fp_chk:w #4#5#6 ; #7 @
16623 {
16624   \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
16625   {
16626     \tl_if_empty:nTF {#7}
16627     {
16628       \exp_args:Nc \__fp_round:Nww
16629       {
16630         __fp_round_to_nearest
16631         \if_meaning:w 0 #4 _zero \else:
16632         \if_case:w #5 \exp_stop_f: _pinf \or: \else: _ninf \fi: \fi:
16633         :NNN
16634       }
16635       #2 ; #3 ;
16636     }
16637     {
16638       \__fp_error:nnnn { fp-num-args } { round () } { 1 } { 3 }
16639       \exp_after:wN \c_nan_fp
16640     }
16641   }
16642   {
16643     \__fp_error:nffn { fp-num-args }
16644     { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
16645     \exp_after:wN \c_nan_fp
16646   }
16647 }

```

(End definition for __fp_round:Nwww.)

__fp_round_name_from_cs:N

```

16648 \cs_new:Npn \__fp_round_name_from_cs:N #1
16649 {
16650   \cs_if_eq:NNTF #1 \__fp_round_to_zero:NNN { trunc }
16651   {
16652     \cs_if_eq:NNTF #1 \__fp_round_to_ninf:NNN { floor }
16653     {
16654       \cs_if_eq:NNTF #1 \__fp_round_to_pinf:NNN { ceil }
16655       { round }
16656     }
16657   }

```

```

16657     }
16658 }

```

(End definition for _fp_round_name_from_cs:N.)

_fp_round:Nww

_fp_round:Nwn

If the number of digits to round to is an integer or infinity all is good; if it is nan then just produce a nan; otherwise invalid as we have something like round(1,3.14) where the number of digits is not an integer.

_fp_round_normal:NwNNnw

_fp_round_normal:NnnwNNnn

_fp_round_pack:Nw

_fp_round_normal:NNwNnn

_fp_round_normal_end:wwNnn

_fp_round_special:NwwNnn

_fp_round_special_aux:Nw

```

16659 \cs_new:Npn \_fp_round:Nww #1#2 ; #3 ;
16660 {
16661   \_fp_small_int:wTF #3; { \_fp_round:Nwn #1#2; }
16662   {
16663     \if:w 3 \_fp_kind:w #3 ;
16664     \exp_after:wN \use_i:nn
16665     \else:
16666       \exp_after:wN \use_ii:nn
16667     \fi:
16668     { \exp_after:wN \c_nan_fp }
16669     {
16670       \_fp_invalid_operation_tl_o:ff
16671       { \_fp_round_name_from_cs:N #1 }
16672       { \_fp_array_to_clist:n { #2; #3; } }
16673     }
16674   }
16675 }
16676 \cs_new:Npn \_fp_round:Nwn #1 \s_fp \_fp_chk:w #2#3#4; #5
16677 {
16678   \if_meaning:w 1 #2
16679     \exp_after:wN \_fp_round_normal:NwNNnw
16680     \exp_after:wN #1
16681     \int_value:w #5
16682   \else:
16683     \exp_after:wN \_fp_exp_after_o:w
16684   \fi:
16685   \s_fp \_fp_chk:w #2#3#4;
16686 }
16687 \cs_new:Npn \_fp_round_normal:NwNNnw #1#2 \s_fp \_fp_chk:w 1#3#4#5;
16688 {
16689   \_fp_decimate:nNnnnn { \c_fp_prec_int - #4 - #2 }
16690   \_fp_round_normal:NnnwNNnn #5 #1 #3 {#4} {#2}
16691 }
16692 \cs_new:Npn \_fp_round_normal:NnnwNNnn #1#2#3#4; #5#6
16693 {
16694   \exp_after:wN \_fp_round_normal:NNwNnn
16695   \int_value:w \_fp_int_eval:w
16696   \if_int_compare:w #2 > 0 \exp_stop_f:
16697     1 \int_value:w #2
16698     \exp_after:wN \_fp_round_pack:Nw
16699     \int_value:w \_fp_int_eval:w 1#3 +
16700   \else:
16701     \if_int_compare:w #3 > 0 \exp_stop_f:
16702       1 \int_value:w #3 +
16703     \fi:
16704   \fi:

```

```

16705     \exp_after:wN #5
16706     \exp_after:wN #6
16707     \use_none:nnnnnnn #3
16708     #1
16709     \__fp_int_eval_end:
16710     0000 0000 0000 0000 ; #6
16711 }
16712 \cs_new:Npn \__fp_round_pack:Nw #1
16713 { \if_meaning:w 2 #1 + 1 \fi: \__fp_int_eval_end: }
16714 \cs_new:Npn \__fp_round_normal:NNwNnn #1 #2
16715 {
16716     \if_meaning:w 0 #2
16717     \exp_after:wN \__fp_round_special:NwwNnn
16718     \exp_after:wN #1
16719     \fi:
16720     \__fp_pack_twice_four:wNNNNNNNNN
16721     \__fp_pack_twice_four:wNNNNNNNNN
16722     \__fp_round_normal_end:wwNnn
16723     ; #2
16724 }
16725 \cs_new:Npn \__fp_round_normal_end:wwNnn #1;#2;#3#4#5
16726 {
16727     \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
16728     \__fp_sanitizew:Nw #3 #4 ; #1 ;
16729 }
16730 \cs_new:Npn \__fp_round_special:NwwNnn #1#2;#3;#4#5#6
16731 {
16732     \if_meaning:w 0 #1
16733     \__fp_case_return:nw
16734     { \exp_after:wN \__fp_zero_fp:N \exp_after:wN #4 }
16735     \else:
16736     \exp_after:wN \__fp_round_special_aux:Nw
16737     \exp_after:wN #4
16738     \int_value:w \__fp_int_eval:w 1
16739     \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
16740     \fi:
16741     ;
16742 }
16743 \cs_new:Npn \__fp_round_special_aux:Nw #1#2;
16744 {
16745     \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
16746     \__fp_sanitizew:Nw #1#2; {1000}{0000}{0000}{0000};
16747 }

```

(End definition for __fp_round:Nww and others.)

```

16748 \end{package}

```

28 l3fp-parse implementation

```

16749 \begin{package}
16750 \set{fp}

```

28.1 Work plan

The task at hand is non-trivial, and some previous failed attempts show that the code leads to unreadable logs, so we had better get it (almost) right the first time. Let us first describe our goal, then discuss the design precisely before writing any code.

In this file at least, a *floating point object* is a floating point number or tuple. This can be extended to anything that starts with `\s__fp` or `\s__fp_⟨type⟩` and ends with `;` with some internal structure that depends on the *type*.

`__fp_parse:n`

`__fp_parse:n {⟨fexpr⟩}`

Evaluates the *floating point expression* and leaves the result in the input stream as a floating point object. This function forms the basis of almost all public `l3fp` functions. During evaluation, each token is fully *f*-expanded.

`__fp_parse_o:n` does the same but expands once after its result.

T_EXhackers note: Registers (integers, toks, etc.) are automatically unpacked, without requiring a function such as `\int_use:N`. Invalid tokens remaining after *f*-expansion lead to unrecoverable low-level T_EX errors.

(End definition for `__fp_parse:n`.)

`\c__fp_prec_func_int`
`\c__fp_prec_hatii_int`
`\c__fp_prec_hat_int`
`\c__fp_prec_not_int`
`\c__fp_prec_juxt_int`
`\c__fp_prec_times_int`
`\c__fp_prec_plus_int`
`\c__fp_prec_comp_int`
`\c__fp_prec_and_int`
`\c__fp_prec_or_int`
`\c__fp_prec_quest_int`
`\c__fp_prec_colon_int`
`\c__fp_prec_comma_int`
`\c__fp_prec_tuple_int`
`\c__fp_prec_end_int`

Floating point expressions are composed of numbers, given in various forms, infix operators, such as `+`, `**`, or `,` (which joins two numbers into a list), and prefix operators, such as the unary `-`, functions, or opening parentheses. Here is a list of precedences which control the order of evaluation (some distinctions are irrelevant for the order of evaluation, but serve as signals), from the tightest binding to the loosest binding.

16 Function calls.

13/14 Binary `**` and `^` (right to left).

12 Unary `+`, `-`, `!` (right to left).

11 Juxtaposition (implicit `*`) with no parenthesis.

10 Binary `*` and `/`.

9 Binary `+` and `-`.

7 Comparisons.

6 Logical **and**, denoted by `&&`.

5 Logical **or**, denoted by `||`.

4 Ternary operator `?:`, piece `?`.

3 Ternary operator `?:`, piece `:`.

2 Commas.

1 Place where a comma is allowed and generates a tuple.

0 Start and end of the expression.


```

16751 \int_const:Nn \c__fp_prec_func_int { 16 }
16752 \int_const:Nn \c__fp_prec_hatii_int { 14 }
16753 \int_const:Nn \c__fp_prec_hat_int { 13 }
16754 \int_const:Nn \c__fp_prec_not_int { 12 }
16755 \int_const:Nn \c__fp_prec_juxt_int { 11 }
16756 \int_const:Nn \c__fp_prec_times_int { 10 }
16757 \int_const:Nn \c__fp_prec_plus_int { 9 }
16758 \int_const:Nn \c__fp_prec_comp_int { 7 }
16759 \int_const:Nn \c__fp_prec_and_int { 6 }
16760 \int_const:Nn \c__fp_prec_or_int { 5 }
16761 \int_const:Nn \c__fp_prec_quest_int { 4 }
16762 \int_const:Nn \c__fp_prec_colon_int { 3 }
16763 \int_const:Nn \c__fp_prec_comma_int { 2 }
16764 \int_const:Nn \c__fp_prec_tuple_int { 1 }
16765 \int_const:Nn \c__fp_prec_end_int { 0 }

```

(End definition for `\c__fp_prec_func_int` and others.)

28.1.1 Storing results

The main question in parsing expressions expandably is to decide where to put the intermediate results computed for various subexpressions.

One option is to store the values at the start of the expression, and carry them together as the first argument of each macro. However, we want to `f-expand` tokens one by one in the expression (as `\int_eval:n` does), and with this approach, expanding the next unread token forces us to jump with `\exp_after:wN` over every value computed earlier in the expression. With this approach, the run-time grows at least quadratically in the length of the expression, if not as its cube (inserting the `\exp_after:wN` is tricky and slow).

A second option is to place those values at the end of the expression. Then expanding the next unread token is straightforward, but this still hits a performance issue: for long expressions we would be reaching all the way to the end of the expression at every step of the calculation. The run-time is again quadratic.

A variation of the above attempts to place the intermediate results which appear when computing a parenthesized expression near the closing parenthesis. This still lets us expand tokens as we go, and avoids performance problems as long as there are enough parentheses. However, it would be better to avoid requiring the closing parenthesis to be present as soon as the corresponding opening parenthesis is read: the closing parenthesis may still be hidden in a macro yet to be expanded.

Hence, we need to go for some fine expansion control: the result is stored *before* the start!

Let us illustrate this idea in a simple model: adding positive integers which may be resulting from the expansion of macros, or may be values of registers. Assume that one number, say, 12345, has already been found, and that we want to parse the next number. The current status of the code may look as follows.

```

\exp_after:wN \add:ww \int_value:w 12345 \exp_after:wN ;
\exp:w \operand:w <stuff>

```

One step of expansion expands `\exp_after:wN`, which triggers the primitive `\int_value:w`, which reads the five digits we have already found, 12345. This integer is unfinished, causing the second `\exp_after:wN` to expand, and to trigger the construction

`\exp:w`, which expands `\operand:w`, defined to read what follows and make a number out of it, then leave `\exp_end:`, the number, and a semicolon in the input stream. Once `\operand:w` is done expanding, we obtain essentially

```
\exp_after:wN \add:ww \int_value:w 12345 ;
\exp:w \exp_end: 333444 ;
```

where in fact `\exp_after:wN` has already been expanded, `\int_value:w` has already seen 12345, and `\exp:w` is still looking for a number. It finds `\exp_end:`, hence expands to nothing. Now, `\int_value:w` sees the `;`, which cannot be part of a number. The expansion stops, and we are left with

```
\add:ww 12345 ; 333444 ;
```

which can safely perform the addition by grabbing two arguments delimited by `;`.

If we were to continue parsing the expression, then the following number should also be cleaned up before the next use of a binary operation such as `\add:ww`. Just like `\int_value:w 12345 \exp_after:wN ;` expanded what follows once, we need `\add:ww` to do the calculation, and in the process to expand the following once. This is also true in our real application: all the functions of the form `__fp..._o:ww` expand what follows once. This comes at the cost of leaving tokens in the input stack, and we need to be careful not to waste this memory. All of our discussion above is nice but simplistic, as operations should not simply be performed in the order they appear.

28.1.2 Precedence and infix operators

The various operators we will encounter have different precedences, which influence the order of calculations: $1 + 2 \times 3 = 1 + (2 \times 3)$ because \times has a higher precedence than $+$. The true analog of our macro `\operand:w` must thus take care of that. When looking for an operand, it needs to perform calculations until reaching an operator which has lower precedence than the one which called `\operand:w`. This means that `\operand:w` must know what the previous binary operator is, or rather, its precedence: we thus rename it `\operand:Nw`. Let us describe as an example how we plan to do the calculation $41 - 2^3 * 4 + 5$. More precisely we describe how to perform the first operation in this expression. Here, we abuse notations: the first argument of `\operand:Nw` should be an integer constant (`\c__fp_prec_plus_int, ...`) equal to the precedence of the given operator, not directly the operator itself.

- Clean up 41 and find $-$. We call `\operand:Nw -` to find the second operand.
- Clean up 2 and find \wedge .
- Compare the precedences of $-$ and \wedge . Since the latter is higher, we need to compute the exponentiation. For this, find the second operand with a nested call to `\operand:Nw \wedge`.
- Clean up 3 and find $*$.
- Compare the precedences of \wedge and $*$. Since the former is higher, `\operand:Nw \wedge` has found the second operand of the exponentiation, which is computed: $2^3 = 8$.
- We now have $41 - 8 * 4 + 5$, and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 8?

- Compare the precedences of $-$ and $*$. Since the latter is higher, we are not done with 8. Call `\operand:Nw *` to find the second operand of the multiplication.
- Clean up 4, and find $+$.
- Compare the precedences of $*$ and $+$. Since the former is higher, `\operand:Nw *` has found the second operand of the multiplication, which is computed: $8 * 4 = 32$.
- We now have $41 - 32 + 5$, and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 32?
- Compare the precedences of $-$ and $+$. Since they are equal, `\operand:Nw -` has found the second operand for the subtraction, which is computed: $41 - 32 = 9$.
- We now have $9 + 5$.

The procedure above stops short of performing all computations, but adding a surrounding call to `\operand:Nw` with a very low precedence ensures that all computations are performed before `\operand:Nw` is done. Adding a trailing marker with the same very low precedence prevents the surrounding `\operand:Nw` from going beyond the marker.

The pattern above to find an operand for a given operator, is to find one number and the next operator, then compare precedences to know if the next computation should be done. If it should, then perform it after finding its second operand, and look at the next operator, then compare precedences to know if the next computation should be done. This continues until we find that the next computation should not be done. Then, we stop.

We are now ready to get a bit more technical and describe which of the `l3fp-parse` functions correspond to each step above.

First, `__fp_parse_operand:Nw` is the `\operand:Nw` function above, with small modifications due to expansion issues discussed later. We denote by $\langle precedence \rangle$ the argument of `__fp_parse_operand:Nw`, that is, the precedence of the binary operator whose operand we are trying to find. The basic action is to read numbers from the input stream. This is done by `__fp_parse_one:Nw`. A first approximation of this function is that it reads one $\langle number \rangle$, performing no computation, and finds the following binary $\langle operator \rangle$. Then it expands to

$$\langle number \rangle \\ __fp_parse_infix_ \langle operator \rangle : N \langle precedence \rangle$$

expanding the `infix` auxiliary before leaving the above in the input stream.

We now explain the `infix` auxiliaries. We need some flexibility in how we treat the case of equal precedences: most often, the first operation encountered should be performed, such as $1 - 2 - 3$ being computed as $(1 - 2) - 3$, but 2^3^4 should be evaluated as $2^{(3^4)}$ instead. For this reason, and to support the equivalence between $**$ and \wedge more easily, each binary operator is converted to a control sequence `__fp_parse_infix_ \langle operator \rangle : N` when it is encountered for the first time. Instead of passing both precedences to a test function to do the comparison steps above, we pass the $\langle precedence \rangle$ (of the earlier operator) to the `infix` auxiliary for the following $\langle operator \rangle$, to know whether to perform the computation of the $\langle operator \rangle$. If it should not be performed, the `infix` auxiliary expands to

$$@ __use_none : n __fp_parse_infix_ \langle operator \rangle : N$$

and otherwise it calls `__fp_parse_operand:Nw` with the precedence of the $\langle operator \rangle$ to find its second operand $\langle number_2 \rangle$ and the next $\langle operator_2 \rangle$, and expands to

```
@ \__fp_parse_apply_binary:NwNwN
  \langle operator \rangle \langle number_2 \rangle
@ \__fp_parse_infix_\langle operator_2 \rangle:N
```

The `infix` function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand $\langle number \rangle$ is before the `infix` function in the input stream. This is why we stop the expansion here and give control to another function to close the loop.

A definition of `__fp_parse_operand:Nw` $\langle precedence \rangle$ with some of the expansion control removed is

```
\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN \langle precedence \rangle
\exp:w \exp_end_continue_f:w
  \__fp_parse_one:Nw \langle precedence \rangle
```

This expands `__fp_parse_one:Nw` $\langle precedence \rangle$ completely, which finds a number, wraps the next $\langle operator \rangle$ into an `infix` function, feeds this function the $\langle precedence \rangle$, and expands it, yielding either

```
\__fp_parse_continue:NwN \langle precedence \rangle
\langle number \rangle @
\use_none:n \__fp_parse_infix_\langle operator \rangle:N
```

or

```
\__fp_parse_continue:NwN \langle precedence \rangle
\langle number \rangle @
\__fp_parse_apply_binary:NwNwN
  \langle operator \rangle \langle number_2 \rangle
@ \__fp_parse_infix_\langle operator_2 \rangle:N
```

The definition of `__fp_parse_continue:NwN` is then very simple:

```
\cs_new:Npn \__fp_parse_continue:NwN #1#2@#3 { #3 #1 #2 @ }
```

In the first case, `#3` is `\use_none:n`, yielding

```
\use_none:n \langle precedence \rangle \langle number \rangle @
\__fp_parse_infix_\langle operator \rangle:N
```

then $\langle number \rangle @ __fp_parse_infix_\langle operator \rangle:N$. In the second case, `#3` is `__fp_parse_apply_binary:NwNwN`, whose role is to compute $\langle number \rangle \langle operator \rangle \langle number_2 \rangle$ and to prepare for the next comparison of precedences: first we get

```
\__fp_parse_apply_binary:NwNwN
  \langle precedence \rangle \langle number \rangle @
  \langle operator \rangle \langle number_2 \rangle
@ \__fp_parse_infix_\langle operator_2 \rangle:N
```

then

```

\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN <precedence>
\exp:w \exp_end_continue_f:w
\__fp_<operator>_o:ww <number> <number2>
\exp:w \exp_end_continue_f:w
\__fp_parse_infix_<operator2>:N <precedence>

```

where `__fp_<operator>_o:ww` computes `<number> <operator> <number2>` and expands after the result, thus triggers the comparison of the precedence of the `<operator2>` and the `<precedence>`, continuing the loop.

We have introduced the most important functions here, and the next few paragraphs we describe various subtleties.

28.1.3 Prefix operators, parentheses, and functions

Prefix operators (unary `-`, `+`, `!`) and parentheses are taken care of by the same mechanism, and functions (`sin`, `exp`, etc.) as well. Finding the argument of the unary `-`, for instance, is very similar to grabbing the second operand of a binary infix operator, with a subtle precedence explained below. Once that operand is found, the operator can be applied to it (for the unary `-`, this simply flips the sign). A left parenthesis is just a prefix operator with a very low precedence equal to that of the closing parenthesis (which is treated as an infix operator, since it normally appears just after numbers), so that all computations are performed until the closing parenthesis. The prefix operator associated to the left parenthesis does not alter its argument, but it removes the closing parenthesis (with some checks).

Prefix operators are the reason why we only summarily described the function `__fp_parse_one:Nw` earlier. This function is responsible for reading in the input stream the first possible `<number>` and the next infix `<operator>`. If what follows `__fp_parse_one:Nw <precedence>` is a prefix operator, then we must find the operand of this prefix operator through a nested call to `__fp_parse_operand:Nw` with the appropriate precedence, then apply the operator to the operand found to yield the result of `__fp_parse_one:Nw`. So far, all is simple.

The unary operators `+`, `-`, `!` complicate things a little bit: `-3**2` should be $-(3^2) = -9$, and not $(-3)^2 = 9$. This would easily be done by giving `-` a lower precedence, equal to that of the infix `+` and `-`. Unfortunately, this fails in cases such as `3**-2*4`, yielding $3^{-2 \times 4}$ instead of the correct $3^{-2} \times 4$. A second attempt would be to call `__fp_parse_operand:Nw` with the `<precedence>` of the previous operator, but `0>-2+3` is then parsed as `0>-(2+3)`: the addition is performed because it binds more tightly than the comparison which precedes `-`. The correct approach is for a unary `-` to perform operations whose precedence is greater than both that of the previous operation, and that of the unary `-` itself. The unary `-` is given a precedence higher than multiplication and division. This does not lead to any surprising result, since $-(x/y) = (-x)/y$ and similarly for multiplication, and it reduces the number of nested calls to `__fp_parse_operand:Nw`.

Functions are implemented as prefix operators with very high precedence, so that their argument is the first number that can possibly be built.

Note that contrarily to the `infix` functions discussed earlier, the `prefix` functions do perform tests on the previous `<precedence>` to decide whether to find an argument or not, since we know that we need a number, and must never stop there.

28.1.4 Numbers and reading tokens one by one

So far, we have glossed over one important point: what is a “number”? A number is typically given in the form $\langle\textit{significand}\rangle\mathbf{e}\langle\textit{exponent}\rangle$, where the $\langle\textit{significand}\rangle$ is any non-empty string composed of decimal digits and at most one decimal separator (a period), the exponent “ $\mathbf{e}\langle\textit{exponent}\rangle$ ” is optional and is composed of an exponent mark **e** followed by a possibly empty string of signs + or - and a non-empty string of decimal digits. The $\langle\textit{significand}\rangle$ can also be an integer, dimension, skip, or muskip variable, in which case dimensions are converted from points (or mu units) to floating points, and the $\langle\textit{exponent}\rangle$ can also be an integer variable. Numbers can also be given as floating point variables, or as named constants such as **nan**, **inf** or **pi**. We may add more types in the future.

When `__fp_parse_one:Nw` is looking for a “number”, here is what happens.

- If the next token is a control sequence with the meaning of `\scan_stop:`, it can be: `\s__fp`, in which case our job is done, as what follows is an internal floating point number, or `\s__fp_expr_mark`, in which case the expression has come to an early end, as we are still looking for a number here, or something else, in which case we consider the control sequence to be a bad variable resulting from `c`-expansion.
- If the next token is a control sequence with a different meaning, we assume that it is a register, unpack it with `\tex_the:D`, and use its value (in `pt` for dimensions and skips, `mu` for muskips) as the $\langle\textit{significand}\rangle$ of a number: we look for an exponent.
- If the next token is a digit, we remove any leading zeros, then read a significand larger than 1 if the next character is a digit, read a significand smaller than 1 if the next character is a period, or we have found a significand equal to 0 otherwise, and look for an exponent.
- If the next token is a letter, we collect more letters until the first non-letter: the resulting word may denote a function such as **asin**, a constant such as **pi** or be unknown. In the first case, we call `__fp_parse_operand:Nw` to find the argument of the function, then apply the function, before declaring that we are done. Otherwise, we are done, either with the value of the constant, or with the value **nan** for unknown words.
- If the next token is anything else, we check whether it is a known prefix operator, in which case `__fp_parse_operand:Nw` finds its operand. If it is not known, then either a number is missing (if the token is a known infix operator) or the token is simply invalid in floating point expressions.

Once a number is found, `__fp_parse_one:Nw` also finds an infix operator. This goes as follows.

- If the next token is a control sequence, it could be the special marker `\s__fp_expr_mark`, and otherwise it is a case of juxtaposing numbers, such as `2\c_zero_int`, with an implied multiplication.
- If the next token is a letter, it is also a case of juxtaposition, as letters cannot be proper infix operators.
- Otherwise (including in the case of digits), if the token is a known infix operator, the appropriate `__fp_infix_<operator>:N` function is built, and if it does not exist, we complain. In particular, the juxtaposition `\c_zero_int 2` is disallowed.

In the above, we need to test whether a character token #1 is a digit:

```
\if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
  is a digit
\else:
  not a digit
\fi:
```

To exclude 0, replace 9 by 10. The use of `\token_to_str:N` ensures that a digit with any catcode is detected. To test if a character token is a letter, we need to work with its character code, testing if ‘#1 lies in [65,90] (uppercase letters) or [97,112] (lowercase letters)

```
\if_int_compare:w \__fp_int_eval:w
  ( ‘#1 \if_int_compare:w ‘#1 > ‘Z - 32 \fi: ) / 26 = 3 \exp_stop_f:
  is a letter
\else:
  not a letter
\fi:
```

At all steps, we try to accept all category codes: when #1 is kept to be used later, it is almost always converted to category code other through `\token_to_str:N`. More precisely, catcodes {3,6,7,8,11,12} should work without trouble, but not {1,2,4,10,13}, and of course {0,5,9} cannot become tokens.

Floating point expressions should behave as much as possible like ε -TeX-based integer expressions and dimension expressions. In particular, f-expansion should be performed as the expression is read, token by token, forcing the expansion of protected macros, and ignoring spaces. One advantage of expanding at every step is that restricted expandable functions can then be used in floating point expressions just as they can be in other kinds of expressions. Problematically, spaces stop f-expansion: for instance, the macro `\X` below would not be expanded if we simply performed f-expansion.

```
\DeclareDocumentCommand {\test} {m} { \fp_eval:n {#1} }
\ExplSyntaxOff
\test { 1 + \X }
```

Of course, spaces typically do not appear in a code setting, but may very easily come in document-level input, from which some expressions may come. To avoid this problem, at every step, we do essentially what `\use:f` would do: take an argument, put it back in the input stream, then f-expand it. This is not a complete solution, since a macro’s expansion could contain leading spaces which would stop the f-expansion before further macro calls are performed. However, in practice it should be enough: in particular, floating point numbers are correctly expanded to the underlying `\s__fp ...` structure. The f-expansion is performed by `__fp_parse_expand:w`.

28.2 Main auxiliary functions

```
\__fp_parse_operand:Nw \exp:w \__fp_parse_operand:Nw <precedence> \__fp_parse_expand:w
Reads the "...", performing every computation with a precedence higher than
<precedence>, then expands to

<result> @ \__fp_parse_infix_<operation>:N ...
```

where the $\langle operation \rangle$ is the first operation with a lower precedence, possibly `end`, and the “...” start just after the $\langle operation \rangle$.

(End definition for `_fp_parse_operand:Nw`.)

```
\_fp_parse_infix_+:N      \_fp_parse_infix_+:N  $\langle precedence \rangle$  ...
                          If + has a precedence higher than the  $\langle precedence \rangle$ , cleans up a second  $\langle operand \rangle$  and
                          finds the  $\langle operation_2 \rangle$  which follows, and expands to

                          @ \_fp_parse_apply_binary:NwNwN +  $\langle operand \rangle$  @ \_fp_parse_infix_ $\langle operation_2 \rangle$ :N
                          ...
```

Otherwise expands to

```
@ \use_none:n \_fp_parse_infix_+:N ...
```

A similar function exists for each infix operator.

(End definition for `_fp_parse_infix_+:N`.)

```
\_fp_parse_one:Nw      \_fp_parse_one:Nw  $\langle precedence \rangle$  ...
                        Cleans up one or two operands depending on how the precedence of the next operation
                        compares to the  $\langle precedence \rangle$ . If the following  $\langle operation \rangle$  has a precedence higher
                        than  $\langle precedence \rangle$ , expands to
```

```
 $\langle operand_1 \rangle$  @ \_fp_parse_apply_binary:NwNwN  $\langle operation \rangle$   $\langle operand_2 \rangle$  @
\_fp_parse_infix_ $\langle operation_2 \rangle$ :N ...
```

and otherwise expands to

```
 $\langle operand \rangle$  @ \use_none:n \_fp_parse_infix_ $\langle operation \rangle$ :N ...
```

(End definition for `_fp_parse_one:Nw`.)

28.3 Helpers

```
\_fp_parse_expand:w      \exp:w \_fp_parse_expand:w  $\langle tokens \rangle$ 
```

This function must always come within a `\exp:w` expansion. The $\langle tokens \rangle$ should be the part of the expression that we have not yet read. This requires in particular closing all conditionals properly before expanding.

```
16766 \cs_new:Npn \_fp_parse_expand:w #1 { \exp_end_continue_f:w #1 }
```

(End definition for `_fp_parse_expand:w`.)

```
\_fp_parse_return_semicolon:w This very odd function swaps its position with the following \fi: and removes \_fp_
parse_expand:w normally responsible for expansion. That turns out to be useful.
```

```
16767 \cs_new:Npn \_fp_parse_return_semicolon:w
16768 #1 \fi: \_fp_parse_expand:w { \fi: ; #1 }
```

(End definition for `_fp_parse_return_semicolon:w`.)

```
\_fp_parse_digits_vii:N These functions must be called within an \int_value:w or \_fp_int_eval:w construc-
\_fp_parse_digits_vi:N tion. The first token which follows must be f-expanded prior to calling those functions.
\_fp_parse_digits_v:N The functions read tokens one by one, and output digits into the input stream, until
\_fp_parse_digits_iv:N meeting a non-digit, or up to a number of digits equal to their index. The full expansion
\_fp_parse_digits_iii:N is
\_fp_parse_digits_ii:N
\_fp_parse_digits_i:N
\_fp_parse_digits_:N
```


$\langle \text{digits} \rangle$; $\langle \text{filling } 0 \rangle$; $\langle \text{length} \rangle$

where $\langle \text{filling } 0 \rangle$ is a string of zeros such that $\langle \text{digits} \rangle \langle \text{filling } 0 \rangle$ has the length given by the index of the function, and $\langle \text{length} \rangle$ is the number of zeros in the $\langle \text{filling } 0 \rangle$ string. Each function puts a digit into the input stream and calls the next function, until we find a non-digit. We are careful to pass the tested tokens through $\backslash \text{token_to_str:N}$ to normalize their category code.

```

16769 \cs_set_protected:Npn \__fp_tmp:w #1 #2 #3
16770 {
16771   \cs_new:cpn { __fp_parse_digits_ #1 :N } ##1
16772   {
16773     \if_int_compare:w 9 < 1 \token_to_str:N ##1 \exp_stop_f:
16774     \token_to_str:N ##1 \exp_after:wN #2 \exp:w
16775     \else:
16776     \__fp_parse_return_semicolon:w #3 ##1
16777     \fi:
16778     \__fp_parse_expand:w
16779   }
16780 }
16781 \__fp_tmp:w {vii} \__fp_parse_digits_vi:N { 0000000 ; 7 }
16782 \__fp_tmp:w {vi} \__fp_parse_digits_v:N { 000000 ; 6 }
16783 \__fp_tmp:w {v} \__fp_parse_digits_iv:N { 00000 ; 5 }
16784 \__fp_tmp:w {iv} \__fp_parse_digits_iii:N { 0000 ; 4 }
16785 \__fp_tmp:w {iii} \__fp_parse_digits_ii:N { 000 ; 3 }
16786 \__fp_tmp:w {ii} \__fp_parse_digits_i:N { 00 ; 2 }
16787 \__fp_tmp:w {i} \__fp_parse_digits_:N { 0 ; 1 }
16788 \cs_new:Npn \__fp_parse_digits_:N { ; ; 0 }

```

(End definition for $\backslash \text{__fp_parse_digits_vii:N}$ and others.)

28.4 Parsing one number

$\backslash \text{__fp_parse_one:Nw}$ This function finds one number, and packs the symbol which follows in an $\backslash \text{__fp_parse_infix_...}$ csname. #1 is the previous $\langle \text{precedence} \rangle$, and #2 the first token of the operand. We distinguish four cases: #2 is equal to $\backslash \text{scan_stop:}$ in meaning, #2 is a different control sequence, #2 is a digit, and #2 is something else (this last case is split further later). Despite the earlier f-expansion, #2 may still be expandable if it was protected by $\backslash \text{exp_not:N}$, as may happen with the $\text{\LaTeX 2}_{\epsilon}$ command $\backslash \text{protect}$. Using a well placed $\backslash \text{reverse_if:N}$, this case is sent to $\backslash \text{__fp_parse_one_fp:NN}$ which deals with it robustly.

```

16789 \cs_new:Npn \__fp_parse_one:Nw #1 #2
16790 {
16791   \if_catcode:w \scan_stop: \exp_not:N #2
16792   \exp_after:wN \if_meaning:w \exp_not:N #2 #2 \else:
16793   \exp_after:wN \reverse_if:N
16794   \fi:
16795   \if_meaning:w \scan_stop: #2
16796   \exp_after:wN \exp_after:wN
16797   \exp_after:wN \__fp_parse_one_fp:NN
16798   \else:
16799   \exp_after:wN \exp_after:wN
16800   \exp_after:wN \__fp_parse_one_register:NN
16801   \fi:

```

```

16802 \else:
16803 \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
16804 \exp_after:wN \exp_after:wN
16805 \exp_after:wN \__fp_parse_one_digit:NN
16806 \else:
16807 \exp_after:wN \exp_after:wN
16808 \exp_after:wN \__fp_parse_one_other:NN
16809 \fi:
16810 \fi:
16811 #1 #2
16812 }

```

(End definition for `__fp_parse_one:Nw`.)

```

\__fp_parse_one_fp:NN
\__fp_exp_after_expr_mark_f:nw
\__fp_exp_after_?_f:nw

```

This function receives a $\langle precedence \rangle$ and a control sequence equal to `\scan_stop:` in meaning. There are three cases.

- `\s__fp` starts a floating point number, and we call `__fp_exp_after_f:nw`, which f-expands after the floating point.
- `\s__fp_expr_mark` is a premature end, we call `__fp_exp_after_expr_mark_f:nw`, which triggers an `fp-early-end` error.
- For a control sequence not containing `\s__fp`, we call `__fp_exp_after_?_f:nw`, causing a `bad-variable` error.

This scheme is extensible: additional types can be added by starting the variables with a scan mark of the form `\s__fp_⟨type⟩` and defining `__fp_exp_after_⟨type⟩_f:nw`. In all cases, we make sure that the second argument of `__fp_parse_infix:NN` is correctly expanded. A special case only enabled in L^AT_EX 2_ε is that if `\protect` is encountered then the error message mentions the control sequence which follows it rather than `\protect` itself. The test for L^AT_EX 2_ε uses `\@unexpandable@protect` rather than `\protect` because `\protect` is often `\scan_stop:` hence “does not exist”.

```

16813 \cs_new:Npn \__fp_parse_one_fp:NN #1
16814 {
16815 \__fp_exp_after_any_f:nw
16816 {
16817 \exp_after:wN \__fp_parse_infix:NN
16818 \exp_after:wN #1 \exp:w \__fp_parse_expand:w
16819 }
16820 }
16821 \cs_new:Npn \__fp_exp_after_expr_mark_f:nw #1
16822 {
16823 \int_case:nnF { \exp_after:wN \use_i:nnn \use_none:nnn #1 }
16824 {
16825 \c__fp_prec_comma_int { }
16826 \c__fp_prec_tuple_int { }
16827 \c__fp_prec_end_int
16828 {
16829 \exp_after:wN \c__fp_empty_tuple_fp
16830 \exp:w \exp_end_continue_f:w
16831 }
16832 }
16833 {

```

```

16834         \_kernel_msg_expandable_error:nn { kernel } { fp-early-end }
16835         \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
16836     }
16837     #1
16838 }
16839 \cs_new:cpn { __fp_exp_after_?_f:nw } #1#2
16840 {
16841     \_kernel_msg_expandable_error:nnn { kernel } { bad-variable }
16842     {#2}
16843     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
16844 }
16845 \cs_set_protected:Npn \__fp_tmp:w #1
16846 {
16847     \cs_if_exist:NT #1
16848     {
16849         \cs_gset:cpn { __fp_exp_after_?_f:nw } ##1##2
16850         {
16851             \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w ##1
16852             \str_if_eq:nnTF {##2} { \protect }
16853             {
16854                 \cs_if_eq:NNTF ##2 #1 { \use_i:nn } { \use:n }
16855                 {
16856                     \_kernel_msg_expandable_error:nnn { kernel }
16857                     { fp-robust-cmd }
16858                 }
16859             }
16860             {
16861                 \_kernel_msg_expandable_error:nnn { kernel }
16862                 { bad-variable } {##2}
16863             }
16864         }
16865     }
16866 }
16867 \exp_args:Nc \__fp_tmp:w { @unexpandable@protect }

```

(End definition for __fp_parse_one_fp:NN, __fp_exp_after_expr_mark_f:nw, and __fp_exp_after_?_f:nw.)

__fp_parse_one_register:NN This is called whenever #2 is a control sequence other than \scan_stop: in meaning. We special-case \wd, \ht, \dp (see later) and otherwise assume that it is a register, but carefully unpack it with \tex_the:D within braces. First, we find the exponent following #2. Then we unpack #2 with \tex_the:D, and the auxii auxiliary distinguishes integer registers from dimensions/skips from muskips, according to the presence of a period and/or of pt. For integers, simply convert $\langle value \rangle e^{\langle exponent \rangle}$ to a floating point number with __fp_parse:n (this is somewhat wasteful). For other registers, the decimal rounding provided by T_EX does not accurately represent the binary value that it manipulates, so we extract this binary value as a number of scaled points with \int_value:w \dim_to_decimal_in_sp:n { $\langle decimal value \rangle$ pt }, and use an auxiliary of \dim_to_fp:n, which performs the multiplication by 2^{-16} , correctly rounded.

```

16868 \cs_new:Npn \__fp_parse_one_register:NN #1#2
16869 {
16870     \exp_after:wN \__fp_parse_infix_after_operand:NwN
16871     \exp_after:wN #1
16872     \exp:w \exp_end_continue_f:w

```

```

16873 \__fp_parse_one_register_special:N #2
16874 \exp_after:wN \__fp_parse_one_register_aux:Nw
16875 \exp_after:wN #2
16876 \int_value:w
16877 \exp_after:wN \__fp_parse_exponent:N
16878 \exp:w \__fp_parse_expand:w
16879 }
16880 \cs_new:Npx \__fp_parse_one_register_aux:Nw #1
16881 {
16882 \exp_not:n
16883 {
16884 \exp_after:wN \use:nn
16885 \exp_after:wN \__fp_parse_one_register_auxii:wwwNw
16886 }
16887 \exp_not:N \exp_after:wN { \exp_not:N \tex_the:D #1 }
16888 ; \exp_not:N \__fp_parse_one_register_dim:ww
16889 \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_mu:www
16890 . \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_int:www
16891 \s__fp_stop
16892 }
16893 \exp_args:Nno \use:nn
16894 { \cs_new:Npn \__fp_parse_one_register_auxii:wwwNw #1 . #2 }
16895 { \tl_to_str:n { pt } #3 ; #4#5 \s__fp_stop }
16896 { #4 #1.#2; }
16897 \exp_args:Nno \use:nn
16898 { \cs_new:Npn \__fp_parse_one_register_mu:www #1 }
16899 { \tl_to_str:n { mu } ; #2 ; }
16900 { \__fp_parse_one_register_dim:ww #1 ; }
16901 \cs_new:Npn \__fp_parse_one_register_int:www #1; #2.; #3;
16902 { \__fp_parse:n { #1 e #3 } }
16903 \cs_new:Npn \__fp_parse_one_register_dim:ww #1; #2;
16904 {
16905 \exp_after:wN \__fp_from_dim_test:ww
16906 \int_value:w #2 \exp_after:wN ,
16907 \int_value:w \dim_to_decimal_in_sp:n { #1 pt } ;
16908 }

```

(End definition for `__fp_parse_one_register:NN` and others.)

`__fp_parse_one_register_special:N` The `\wd`, `\dp`, `\ht` primitives expect an integer argument. We abuse the exponent parser to find the integer argument: simply include the exponent marker `e`. Once that “exponent” is found, use `\tex_the:D` to find the box dimension and then copy what we did for dimensions.

```

16909 \cs_new:Npn \__fp_parse_one_register_special:N #1
16910 {
16911 \if_meaning:w \box_wd:N #1 \__fp_parse_one_register_wd:w \fi:
16912 \if_meaning:w \box_ht:N #1 \__fp_parse_one_register_wd:w \fi:
16913 \if_meaning:w \box_dp:N #1 \__fp_parse_one_register_wd:w \fi:
16914 \if_meaning:w \infty #1
16915 \__fp_parse_one_register_math:NNw \infty #1
16916 \fi:
16917 \if_meaning:w \pi #1
16918 \__fp_parse_one_register_math:NNw \pi #1
16919 \fi:

```

```

16920 }
16921 \cs_new:Npn \__fp_parse_one_register_math:NNw
16922   #1#2#3#4 \__fp_parse_expand:w
16923   {
16924     #3
16925     \str_if_eq:nnTF {#1} {#2}
16926     {
16927       \__kernel_msg_expandable_error:nnn
16928       { kernel } { fp-infty-pi } {#1}
16929       \c_nan_fp
16930     }
16931     { #4 \__fp_parse_expand:w }
16932   }
16933 \cs_new:Npn \__fp_parse_one_register_wd:w
16934   #1#2 \exp_after:wN #3#4 \__fp_parse_expand:w
16935   {
16936     #1
16937     \exp_after:wN \__fp_parse_one_register_wd:Nw
16938     #4 \__fp_parse_expand:w e
16939   }
16940 \cs_new:Npn \__fp_parse_one_register_wd:Nw #1#2 ;
16941   {
16942     \exp_after:wN \__fp_from_dim_test:ww
16943     \exp_after:wN 0 \exp_after:wN ,
16944     \int_value:w \dim_to_decimal_in_sp:n { #1 #2 } ;
16945   }

```

(End definition for __fp_parse_one_register_special:N and others.)

__fp_parse_one_digit:NN A digit marks the beginning of an explicit floating point number. Once the number is found, we catch the case of overflow and underflow with __fp_sanitize:wN, then __fp_parse_infix_after_operand:NwN expands __fp_parse_infix:NN after the number we find, to wrap the following infix operator as required. Finding the number itself begins by removing leading zeros: further steps are described later.

```

16946 \cs_new:Npn \__fp_parse_one_digit:NN #1
16947   {
16948     \exp_after:wN \__fp_parse_infix_after_operand:NwN
16949     \exp_after:wN #1
16950     \exp:w \exp_end_continue_f:w
16951     \exp_after:wN \__fp_sanitize:wN
16952     \int_value:w \__fp_int_eval:w 0 \__fp_parse_trim_zeros:N
16953   }

```

(End definition for __fp_parse_one_digit:NN.)

__fp_parse_one_other:NN For this function, #2 is a character token which is not a digit. If it is an ASCII letter, __fp_parse_letters:N beyond this one and give the result to __fp_parse_word:Nw. Otherwise, the character is assumed to be a prefix operator, and we build __fp_parse_prefix_{operator}:Nw.

```

16954 \cs_new:Npn \__fp_parse_one_other:NN #1 #2
16955   {
16956     \if_int_compare:w
16957       \__fp_int_eval:w
16958       ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26

```

```

16959         = 3 \exp_stop_f:
16960         \exp_after:wN \__fp_parse_word:Nw
16961         \exp_after:wN #1
16962         \exp_after:wN #2
16963         \exp:w \exp_after:wN \__fp_parse_letters:N
16964         \exp:w
16965     \else:
16966         \exp_after:wN \__fp_parse_prefix:NNN
16967         \exp_after:wN #1
16968         \exp_after:wN #2
16969         \cs:w
16970         __fp_parse_prefix_ \token_to_str:N #2 :Nw
16971         \exp_after:wN
16972         \cs_end:
16973         \exp:w
16974     \fi:
16975     \__fp_parse_expand:w
16976 }

```

(End definition for __fp_parse_one_other:NN.)

__fp_parse_word:Nw
__fp_parse_letters:N

Finding letters is a simple recursion. Once __fp_parse_letters:N has done its job, we try to build a control sequence from the word #2. If it is a known word, then the corresponding action is taken, and otherwise, we complain about an unknown word, yield \c_nan_fp, and look for the following infix operator. Note that the unknown word could be a mistyped function as well as a mistyped constant, so there is no way to tell whether to look for arguments; we do not. The standard requires “inf” and “infinity” and “nan” to be recognized regardless of case, but we probably don’t want to allow every l3fp word to have an arbitrary mixture of lower and upper case, so we test and use a differently-named control sequence.

```

16977 \cs_new:Npn \__fp_parse_word:Nw #1#2;
16978 {
16979     \cs_if_exist_use:cF { __fp_parse_word_#2:N }
16980     {
16981         \cs_if_exist_use:cF
16982         { __fp_parse_caseless_ \str_foldcase:n {#2} :N }
16983         {
16984             \__kernel_msg_expandable_error:nnn
16985             { kernel } { unknown-fp-word } {#2}
16986             \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
16987             \__fp_parse_infix:NN
16988         }
16989     }
16990     #1
16991 }
16992 \cs_new:Npn \__fp_parse_letters:N #1
16993 {
16994     \exp_end_continue_f:w
16995     \if_int_compare:w
16996     \if_catcode:w \scan_stop: \exp_not:N #1
16997     0
16998     \else:
16999         \__fp_int_eval:w
17000         ( '#1 \if_int_compare:w '#1 > 'Z - 32 \fi: ) / 26

```

```

17001     \fi:
17002     = 3 \exp_stop_f:
17003     \exp_after:wN #1
17004     \exp:w \exp_after:wN \_fp_parse_letters:N
17005     \exp:w
17006   \else:
17007     \_fp_parse_return_semicolon:w #1
17008   \fi:
17009   \_fp_parse_expand:w
17010 }

```

(End definition for _fp_parse_word:Nw and _fp_parse_letters:N.)

```

\_fp_parse_prefix:NNN
\_fp_parse_prefix_unknown:NNN

```

For this function, #1 is the previous *<precedence>*, #2 is the operator just seen, and #3 is a control sequence which implements the operator if it is a known operator. If this control sequence is `\scan_stop:`, then the operator is in fact unknown. Either the expression is missing a number there (if the operator is valid as an infix operator), and we put `nan`, wrapping the infix operator in a csname as appropriate, or the character is simply invalid in floating point expressions, and we continue looking for a number, starting again from `_fp_parse_one:Nw`.

```

17011 \cs_new:Npn \_fp_parse_prefix:NNN #1#2#3
17012 {
17013   \if_meaning:w \scan_stop: #3
17014   \exp_after:wN \_fp_parse_prefix_unknown:NNN
17015   \exp_after:wN #2
17016   \fi:
17017   #3 #1
17018 }
17019 \cs_new:Npn \_fp_parse_prefix_unknown:NNN #1#2#3
17020 {
17021   \cs_if_exist:cTF { \_fp_parse_infix_ \token_to_str:N #1 :N }
17022   {
17023     \_kernel_msg_expandable_error:nnn
17024     { kernel } { fp-missing-number } {#1}
17025     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
17026     \_fp_parse_infix:NN #3 #1
17027   }
17028   {
17029     \_kernel_msg_expandable_error:nnn
17030     { kernel } { fp-unknown-symbol } {#1}
17031     \_fp_parse_one:Nw #3
17032   }
17033 }

```

(End definition for _fp_parse_prefix:NNN and _fp_parse_prefix_unknown:NNN.)

28.4.1 Numbers: trimming leading zeros

Numbers are parsed as follows: first we trim leading zeros, then if the next character is a digit, start reading a significand ≥ 1 with the set of functions `_fp_parse_large...`; if it is a period, the significand is < 1 ; and otherwise it is zero. In the second case, trim additional zeros after the period, counting them for an exponent shift $\langle exp_1 \rangle < 0$, then read the significand with the set of functions `_fp_parse_small...`. Once the significand is read, read the exponent if `e` is present.

`__fp_parse_trim_zeros:N` This function expects an already expanded token. It removes any leading zero, then distinguishes three cases: if the first non-zero token is a digit, then call `__fp_parse_large:N` (the significand is ≥ 1); if it is `.`, then continue trimming zeros with `__fp_parse_strim_zeros:N`; otherwise, our number is exactly zero, and we call `__fp_parse_zero:` to take care of that case.

```

17034 \cs_new:Npn \__fp_parse_trim_zeros:N #1
17035 {
17036   \if:w 0 \exp_not:N #1
17037     \exp_after:wN \__fp_parse_trim_zeros:N
17038     \exp:w
17039   \else:
17040     \if:w . \exp_not:N #1
17041       \exp_after:wN \__fp_parse_strim_zeros:N
17042       \exp:w
17043     \else:
17044       \__fp_parse_trim_end:w #1
17045     \fi:
17046   \fi:
17047   \__fp_parse_expand:w
17048 }
17049 \cs_new:Npn \__fp_parse_trim_end:w #1 \fi: \fi: \__fp_parse_expand:w
17050 {
17051   \fi:
17052   \fi:
17053   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
17054     \exp_after:wN \__fp_parse_large:N
17055   \else:
17056     \exp_after:wN \__fp_parse_zero:
17057   \fi:
17058   #1
17059 }

```

(End definition for `__fp_parse_trim_zeros:N` and `__fp_parse_trim_end:w`.)

`__fp_parse_strim_zeros:N` If we have removed all digits until a period (or if the body started with a period), then enter the “small_trim” loop which outputs `-1` for each removed 0. Those `-1` are added to an integer expression waiting for the exponent. If the first non-zero token is a digit, call `__fp_parse_small:N` (our significand is smaller than 1), and otherwise, the number is an exact zero. The name `strim` stands for “small trim”.

```

17060 \cs_new:Npn \__fp_parse_strim_zeros:N #1
17061 {
17062   \if:w 0 \exp_not:N #1
17063     - 1
17064     \exp_after:wN \__fp_parse_strim_zeros:N \exp:w
17065   \else:
17066     \__fp_parse_strim_end:w #1
17067   \fi:
17068   \__fp_parse_expand:w
17069 }
17070 \cs_new:Npn \__fp_parse_strim_end:w #1 \fi: \__fp_parse_expand:w
17071 {
17072   \fi:
17073   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:

```



```

17074     \exp_after:wN \_fp_parse_small:N
17075   \else:
17076     \exp_after:wN \_fp_parse_zero:
17077   \fi:
17078   #1
17079 }

```

(End definition for `_fp_parse_strim_zeros:N` and `_fp_parse_strim_end:w`.)

`_fp_parse_zero:` After reading a significand of 0, find any exponent, then put a sign of 1 for `_fp-sanitize:wN`, which removes everything and leaves an exact zero.

```

17080 \cs_new:Npn \_fp_parse_zero:
17081 {
17082   \exp_after:wN ; \exp_after:wN 1
17083   \int_value:w \_fp_parse_exponent:N
17084 }

```

(End definition for `_fp_parse_zero:.`)

28.4.2 Number: small significand

`_fp_parse_small:N` This function is called after we have passed the decimal separator and removed all leading zeros from the significand. It is followed by a non-zero digit (with any catcode). The goal is to read up to 16 digits. But we can't do that all at once, because `\int_value:w` (which allows us to collect digits and continue expanding) can only go up to 9 digits. Hence we grab digits in two steps of 8 digits. Since `#1` is a digit, read seven more digits using `_fp_parse_digits_vii:N`. The `small_leading` auxiliary leaves those digits in the `\int_value:w`, and grabs some more, or stops if there are no more digits. Then the `pack_leading` auxiliary puts the various parts in the appropriate order for the processing further up.

```

17085 \cs_new:Npn \_fp_parse_small:N #1
17086 {
17087   \exp_after:wN \_fp_parse_pack_leading:NNNNNww
17088   \int_value:w \_fp_int_eval:w 1 \token_to_str:N #1
17089   \exp_after:wN \_fp_parse_small_leading:wwNN
17090   \int_value:w 1
17091   \exp_after:wN \_fp_parse_digits_vii:N
17092   \exp:w \_fp_parse_expand:w
17093 }

```

(End definition for `_fp_parse_small:N`.)

`_fp_parse_small_leading:wwNN` `_fp_parse_small_leading:wwNN 1 <digits> ; <zeros> ; <number of zeros>`

We leave `<digits>` `<zeros>` in the input stream: the functions used to grab digits are such that this constitutes digits 1 through 8 of the significand. Then prepare to pack 8 more digits, with an exponent shift of zero (this shift is used in the case of a large significand). If `#4` is a digit, leave it behind for the packing function, and read 6 more digits to reach a total of 15 digits: further digits are involved in the rounding. Otherwise put 8 zeros in to complete the significand, then look for an exponent.

```

17094 \cs_new:Npn \_fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
17095 {
17096   #1 #2
17097   \exp_after:wN \_fp_parse_pack_trailing:NNNNNNww

```

```

17098 \exp_after:wN 0
17099 \int_value:w \_fp_int_eval:w 1
17100 \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
17101 \token_to_str:N #4
17102 \exp_after:wN \_fp_parse_small_trailing:wwNN
17103 \int_value:w 1
17104 \exp_after:wN \_fp_parse_digits_vi:N
17105 \exp:w
17106 \else:
17107 0000 0000 \_fp_parse_exponent:Nw #4
17108 \fi:
17109 \_fp_parse_expand:w
17110 }

```

(End definition for _fp_parse_small_leading:wwNN.)

```

\_fp_parse_small_trailing:wwNN \_fp_parse_small_trailing:wwNN 1 <digits> ; <zeros> ; <number of zeros>
<next token>

```

Leave digits 10 to 15 (arguments #1 and #2) in the input stream. If the *<next token>* is a digit, it is the 16th digit, we keep it, then the `small_round` auxiliary considers this digit and all further digits to perform the rounding: the function expands to nothing, to +0 or to +1. Otherwise, there is no 16-th digit, so we put a 0, and look for an exponent.

```

17111 \cs_new:Npn \_fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
17112 {
17113   #1 #2
17114   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
17115   \token_to_str:N #4
17116   \exp_after:wN \_fp_parse_small_round:NN
17117   \exp_after:wN #4
17118   \exp:w
17119   \else:
17120   0 \_fp_parse_exponent:Nw #4
17121   \fi:
17122   \_fp_parse_expand:w
17123 }

```

(End definition for _fp_parse_small_trailing:wwNN.)

```

\_fp_parse_pack_trailing:NNNNNNww
\_fp_parse_pack_leading:NNNNNNww
\_fp_parse_pack_carry:w

```

Those functions are expanded after all the digits are found, we took care of the rounding, as well as the exponent. The last argument is the exponent. The previous five arguments are 8 digits which we pack in groups of 4, and the argument before that is 1, except in the rare case where rounding lead to a carry, in which case the argument is 2. The `trailing` function has an exponent shift as its first argument, which we add to the exponent found in the `e...` syntax. If the trailing digits cause a carry, the integer expression for the leading digits is incremented (+1 in the code below). If the leading digits propagate this carry all the way up, the function `_fp_parse_pack_carry:w` increments the exponent, and changes the significand from 0000... to 1000...: this is simple because such a carry can only occur to give rise to a power of 10.

```

17124 \cs_new:Npn \_fp_parse_pack_trailing:NNNNNNww #1 #2 #3#4#5#6 #7; #8 ;
17125 {
17126   \if_meaning:w 2 #2 + 1 \fi:
17127   ; #8 + #1 ; {#3#4#5#6} {#7};
17128 }

```

```

17129 \cs_new:Npn \__fp_parse_pack_leading:NNNNNww #1 #2#3#4#5 #6; #7;
17130 {
17131   + #7
17132   \if_meaning:w 2 #1 \__fp_parse_pack_carry:w \fi:
17133   ; 0 {#2#3#4#5} {#6}
17134 }
17135 \cs_new:Npn \__fp_parse_pack_carry:w \fi: ; 0 #1
17136 { \fi: + 1 ; 0 {1000} }

```

(End definition for `__fp_parse_pack_trailing:NNNNNww`, `__fp_parse_pack_leading:NNNNNww`, and `__fp_parse_pack_carry:w`.)

28.4.3 Number: large significand

Parsing a significand larger than 1 is a little bit more difficult than parsing small significands. We need to count the number of digits before the decimal separator, and add that to the final exponent. We also need to test for the presence of a dot each time we run out of digits, and branch to the appropriate `parse_small` function in those cases.

`__fp_parse_large:N` This function is followed by the first non-zero digit of a “large” significand (≥ 1). It is called within an integer expression for the exponent. Grab up to 7 more digits, for a total of 8 digits.

```

17137 \cs_new:Npn \__fp_parse_large:N #1
17138 {
17139   \exp_after:wN \__fp_parse_large_leading:wwNN
17140   \int_value:w 1 \token_to_str:N #1
17141   \exp_after:wN \__fp_parse_digits_vii:N
17142   \exp:w \__fp_parse_expand:w
17143 }

```

(End definition for `__fp_parse_large:N`.)

`__fp_parse_large_leading:wwNN` `__fp_parse_large_leading:wwNN 1 <digits> ; <zeros> ; <number of zeros>`
`<next token>`

We shift the exponent by the number of digits in `#1`, namely the target number, 8, minus the `<number of zeros>` (number of digits missing). Then prepare to pack the 8 first digits. If the `<next token>` is a digit, read up to 6 more digits (digits 10 to 15). If it is a period, try to grab the end of our 8 first digits, branching to the `small` functions since the number of digit does not affect the exponent anymore. Finally, if this is the end of the significand, insert the `<zeros>` to complete the 8 first digits, insert 8 more, and look for an exponent.

```

17144 \cs_new:Npn \__fp_parse_large_leading:wwNN 1 #1 ; #2; #3 #4
17145 {
17146   + \c__fp_half_prec_int - #3
17147   \exp_after:wN \__fp_parse_pack_leading:NNNNNww
17148   \int_value:w \__fp_int_eval:w 1 #1
17149   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
17150     \exp_after:wN \__fp_parse_large_trailing:wwNN
17151     \int_value:w 1 \token_to_str:N #4
17152     \exp_after:wN \__fp_parse_digits_vi:N
17153     \exp:w
17154   \else:
17155     \if:w . \exp_not:N #4

```

```

17156         \exp_after:wN \__fp_parse_small_leading:wwNN
17157         \int_value:w 1
17158         \cs:w
17159         __fp_parse_digits_
17160         \__fp_int_to_roman:w #3
17161         :N \exp_after:wN
17162         \cs_end:
17163         \exp:w
17164     \else:
17165     #2
17166     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
17167     \exp_after:wN 0
17168     \int_value:w 1 0000 0000
17169     \__fp_parse_exponent:Nw #4
17170     \fi:
17171     \fi:
17172     \__fp_parse_expand:w
17173 }

```

(End definition for __fp_parse_large_leading:wwNN.)

```

\__fp_parse_large_trailing:wwNN
    \__fp_parse_large_trailing:wwNN 1 <digits> ; <zeros> ; <number of zeros>
    <next token>

```

We have just read 15 digits. If the *<next token>* is a digit, then the exponent shift caused by this block of 8 digits is 8, first argument to the `pack_trailing` function. We keep the *<digits>* and this 16-th digit, and find how this should be rounded using `__fp_parse_large_round:NN`. Otherwise, the exponent shift is the number of *<digits>*, 7 minus the *<number of zeros>*, and we test for a decimal point. This case happens in 123451234512345.67 with exactly 15 digits before the decimal separator. Then branch to the appropriate `small` auxiliary, grabbing a few more digits to complement the digits we already grabbed. Finally, if this is truly the end of the significand, look for an exponent after using the *<zeros>* and providing a 16-th digit of 0.

```

17174 \cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
17175 {
17176     \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
17177     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
17178     \exp_after:wN \c__fp_half_prec_int
17179     \int_value:w \__fp_int_eval:w 1 #1 \token_to_str:N #4
17180     \exp_after:wN \__fp_parse_large_round:NN
17181     \exp_after:wN #4
17182     \exp:w
17183     \else:
17184     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
17185     \int_value:w \__fp_int_eval:w 7 - #3 \exp_stop_f:
17186     \int_value:w \__fp_int_eval:w 1 #1
17187     \if:w . \exp_not:N #4
17188     \exp_after:wN \__fp_parse_small_trailing:wwNN
17189     \int_value:w 1
17190     \cs:w
17191     __fp_parse_digits_
17192     \__fp_int_to_roman:w #3
17193     :N \exp_after:wN
17194     \cs_end:

```

```

17195         \exp:w
17196     \else:
17197         #2 0 \__fp_parse_exponent:Nw #4
17198     \fi:
17199 \fi:
17200 \__fp_parse_expand:w
17201 }

```

(End definition for __fp_parse_large_trailing:wwNN.)

28.4.4 Number: beyond 16 digits, rounding

__fp_parse_round_loop:N This loop is called when rounding a number (whether the mantissa is small or large).
 __fp_parse_round_up:N It should appear in an integer expression. This function reads digits one by one, until reaching a non-digit, and adds 1 to the integer expression for each digit. If all digits found are 0, the function ends the expression by ;0, otherwise by ;1. This is done by switching the loop to round_up at the first non-zero digit, thus we avoid to test whether digits are 0 or not once we see a first non-zero digit.

```

17202 \cs_new:Npn \__fp_parse_round_loop:N #1
17203 {
17204     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
17205         + 1
17206     \if:w 0 \token_to_str:N #1
17207         \exp_after:wN \__fp_parse_round_loop:N
17208         \exp:w
17209     \else:
17210         \exp_after:wN \__fp_parse_round_up:N
17211         \exp:w
17212     \fi:
17213 \else:
17214     \__fp_parse_return_semicolon:w 0 #1
17215 \fi:
17216 \__fp_parse_expand:w
17217 }
17218 \cs_new:Npn \__fp_parse_round_up:N #1
17219 {
17220     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
17221         + 1
17222     \exp_after:wN \__fp_parse_round_up:N
17223     \exp:w
17224 \else:
17225     \__fp_parse_return_semicolon:w 1 #1
17226 \fi:
17227 \__fp_parse_expand:w
17228 }

```

(End definition for __fp_parse_round_loop:N and __fp_parse_round_up:N.)

__fp_parse_round_after:wN After the loop __fp_parse_round_loop:N, this function fetches an exponent with __fp_parse_exponent:N, and combines it with the number of digits counted by __fp_parse_round_loop:N. At the same time, the result 0 or 1 is added to the surrounding integer expression.

```

17229 \cs_new:Npn \__fp_parse_round_after:wN #1; #2
17230 {

```

```

17231     + #2 \exp_after:wN ;
17232     \int_value:w \__fp_int_eval:w #1 + \__fp_parse_exponent:N
17233   }

```

(End definition for __fp_parse_round_after:wN.)

```

\__fp_parse_small_round:NN
\__fp_parse_round_after:wN

```

Here, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If #2 is not a digit, then fetch an exponent and expand to ;*<exponent>* only. Otherwise, we expand to +0 or +1, then ;*<exponent>*. To decide which, call __fp_round_s:NNNw to know whether to round up, giving it as arguments a sign 0 (all explicit numbers are positive), the digit #1 to round, the first following digit #2, and either +0 or +1 depending on whether the following digits are all zero or not. This last argument is obtained by __fp_parse_round_loop:N, whose number of digits we discard by multiplying it by 0. The exponent which follows the number is also fetched by __fp_parse_round_after:wN.

```

17234 \cs_new:Npn \__fp_parse_small_round:NN #1#2
17235 {
17236   \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
17237     +
17238     \exp_after:wN \__fp_round_s:NNNw
17239     \exp_after:wN 0
17240     \exp_after:wN #1
17241     \exp_after:wN #2
17242     \int_value:w \__fp_int_eval:w
17243     \exp_after:wN \__fp_parse_round_after:wN
17244     \int_value:w \__fp_int_eval:w 0 * \__fp_int_eval:w 0
17245     \exp_after:wN \__fp_parse_round_loop:N
17246     \exp:w
17247   \else:
17248     \__fp_parse_exponent:Nw #2
17249   \fi:
17250   \__fp_parse_expand:w
17251 }

```

(End definition for __fp_parse_small_round:NN and __fp_parse_round_after:wN.)

```

\__fp_parse_large_round:NN
\__fp_parse_large_round_test:NN
\__fp_parse_large_round_aux:wNN

```

Large numbers are harder to round, as there may be a period in the way. Again, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If there are no more digits (#2 is not a digit), then we must test for a period: if there is one, then switch to the rounding function for small significands, otherwise fetch an exponent. If there are more digits (#2 is a digit), then round, checking with __fp_parse_round_loop:N if all further digits vanish, or some are non-zero. This loop is not enough, as it is stopped by a period. After the loop, the aux function tests for a period: if it is present, then we must continue looking for digits, this time discarding the number of digits we find.

```

17252 \cs_new:Npn \__fp_parse_large_round:NN #1#2
17253 {
17254   \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
17255     +
17256     \exp_after:wN \__fp_round_s:NNNw
17257     \exp_after:wN 0
17258     \exp_after:wN #1
17259     \exp_after:wN #2

```

```

17260     \int_value:w \__fp_int_eval:w
17261     \exp_after:wN \__fp_parse_large_round_aux:wNN
17262     \int_value:w \__fp_int_eval:w 1
17263     \exp_after:wN \__fp_parse_round_loop:N
17264 \else: %^^A could be dot, or e, or other
17265     \exp_after:wN \__fp_parse_large_round_test:NN
17266     \exp_after:wN #1
17267     \exp_after:wN #2
17268 \fi:
17269 }
17270 \cs_new:Npn \__fp_parse_large_round_test:NN #1#2
17271 {
17272     \if:w . \exp_not:N #2
17273     \exp_after:wN \__fp_parse_small_round:NN
17274     \exp_after:wN #1
17275     \exp:w
17276 \else:
17277     \__fp_parse_exponent:Nw #2
17278 \fi:
17279 \__fp_parse_expand:w
17280 }
17281 \cs_new:Npn \__fp_parse_large_round_aux:wNN #1 ; #2 #3
17282 {
17283     + #2
17284     \exp_after:wN \__fp_parse_round_after:wN
17285     \int_value:w \__fp_int_eval:w #1
17286     \if:w . \exp_not:N #3
17287         + 0 * \__fp_int_eval:w 0
17288         \exp_after:wN \__fp_parse_round_loop:N
17289         \exp:w \exp_after:wN \__fp_parse_expand:w
17290     \else:
17291         \exp_after:wN ;
17292         \exp_after:wN 0
17293         \exp_after:wN #3
17294     \fi:
17295 }

```

(End definition for `__fp_parse_large_round:NN`, `__fp_parse_large_round_test:NN`, and `__fp_parse_large_round_aux:wNN`.)

28.4.5 Number: finding the exponent

Expansion is a little bit tricky here, in part because we accept input where multiplication is implicit.

```

\__fp_parse:n { 3.2 erf(0.1) }
\__fp_parse:n { 3.2 e\l_my_int }
\__fp_parse:n { 3.2 \c_pi_fp }

```

The first case indicates that just looking one character ahead for an “e” is not enough, since we would mistake the function `erf` for an exponent of “rf”. An alternative would be to look two tokens ahead and check if what follows is a sign or a digit, considering in that case that we must be finding an exponent. But taking care of the second case requires that we unpack registers after `e`. However, blindly expanding the two tokens ahead completely would break the third example (unpacking is even worse). Indeed, in

the course of reading 3.2, `\c_pi_fp` is expanded to `\s__fp __fp_chk:w 1 0 {-1} {3141}` `...` ; and `\s__fp` stops the expansion. Expanding two tokens ahead would then force the expansion of `__fp_chk:w` (despite it being protected), and that function tries to produce an error.

What can we do? Really, the reason why this last case breaks is that just as `TeX` does, we should read ahead as little as possible. Here, the only case where there may be an exponent is if the first token ahead is `e`. Then we expand (and possibly unpack) the second token.

`__fp_parse_exponent:Nw` This auxiliary is convenient to smuggle some material through `\fi:` ending conditional processing. We place those `\fi:` (argument #2) at a very odd place because this allows us to insert `__fp_int_eval:w ...` there if needed.

```

17296 \cs_new:Npn \__fp_parse_exponent:Nw #1 #2 \__fp_parse_expand:w
17297 {
17298   \exp_after:wN ;
17299   \int_value:w #2 \__fp_parse_exponent:N #1
17300 }
```

(End definition for `__fp_parse_exponent:Nw`.)

`__fp_parse_exponent:N`
`__fp_parse_exponent_aux:NN` This function should be called within an `\int_value:w` expansion (or within an integer expression). It leaves digits of the exponent behind it in the input stream, and terminates the expansion with a semicolon. If there is no `e` (or `E`), leave an exponent of 0. If there is an `e` or `E`, expand the next token to run some tests on it. The first rough test is that if the character code of #1 is greater than that of 9 (largest code valid for an exponent, less than any code valid for an identifier), there was in fact no exponent; otherwise, we search for the sign of the exponent.

```

17301 \cs_new:Npn \__fp_parse_exponent:N #1
17302 {
17303   \if:w e \if:w E \exp_not:N #1 e \else: \exp_not:N #1 \fi:
17304   \exp_after:wN \__fp_parse_exponent_aux:NN
17305   \exp_after:wN #1
17306   \exp:w
17307   \else:
17308     0 \__fp_parse_return_semicolon:w #1
17309   \fi:
17310   \__fp_parse_expand:w
17311 }
17312 \cs_new:Npn \__fp_parse_exponent_aux:NN #1#2
17313 {
17314   \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #2
17315     0 \else: '#2 \fi: > '9 \exp_stop_f:
17316     0 \exp_after:wN ; \exp_after:wN #1
17317   \else:
17318     \exp_after:wN \__fp_parse_exponent_sign:N
17319   \fi:
17320   #2
17321 }
```

(End definition for `__fp_parse_exponent:N` and `__fp_parse_exponent_aux:NN`.)

`__fp_parse_exponent_sign:N` Read signs one by one (if there is any).

```

17322 \cs_new:Npn \__fp_parse_exponent_sign:N #1
```



```

17323 {
17324   \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
17325   \exp_after:wN \__fp_parse_exponent_sign:N
17326   \exp:w \exp_after:wN \__fp_parse_expand:w
17327   \else:
17328     \exp_after:wN \__fp_parse_exponent_body:N
17329     \exp_after:wN #1
17330   \fi:
17331 }

```

(End definition for `__fp_parse_exponent_sign:N`.)

`__fp_parse_exponent_body:N` An exponent can be an explicit integer (most common case), or various other things (most of which are invalid).

```

17332 \cs_new:Npn \__fp_parse_exponent_body:N #1
17333 {
17334   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
17335   \token_to_str:N #1
17336   \exp_after:wN \__fp_parse_exponent_digits:N
17337   \exp:w
17338   \else:
17339     \__fp_parse_exponent_keep:NTF #1
17340     { \__fp_parse_return_semicolon:w #1 }
17341     {
17342       \exp_after:wN ;
17343       \exp:w
17344     }
17345   \fi:
17346   \__fp_parse_expand:w
17347 }

```

(End definition for `__fp_parse_exponent_body:N`.)

`__fp_parse_exponent_digits:N` Read digits one by one, and leave them behind in the input stream. When finding a non-digit, stop, and insert a semicolon. Note that we do not check for overflow of the exponent, hence there can be a \TeX error. It is mostly harmless, except when parsing `0e9876543210`, which should be a valid representation of 0, but is not.

```

17348 \cs_new:Npn \__fp_parse_exponent_digits:N #1
17349 {
17350   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
17351   \token_to_str:N #1
17352   \exp_after:wN \__fp_parse_exponent_digits:N
17353   \exp:w
17354   \else:
17355     \__fp_parse_return_semicolon:w #1
17356   \fi:
17357   \__fp_parse_expand:w
17358 }

```

(End definition for `__fp_parse_exponent_digits:N`.)

`__fp_parse_exponent_keep:NTF` This is the last building block for parsing exponents. The argument `#1` is already fully expanded, and neither `+` nor `-` nor a digit. It can be:

- `\s__fp`, marking the start of an internal floating point, invalid here;

- another control sequence equal to `\relax`, probably a bad variable;
- a register: in this case we make sure that it is an integer register, not a dimension;
- a character other than +, - or digits, again, an error.

```

17359 \prg_new_conditional:Npnn \_fp_parse_exponent_keep:N #1 { TF }
17360 {
17361   \if_catcode:w \scan_stop: \exp_not:N #1
17362   \if_meaning:w \scan_stop: #1
17363   \if_int_compare:w
17364     \_fp_str_if_eq:nn { \s_fp } { \exp_not:N #1 }
17365     = 0 \exp_stop_f:
17366     0
17367     \_kernel_msg_expandable_error:nnn
17368     { kernel } { fp-after-e } { floating-point~ }
17369     \prg_return_true:
17370   \else:
17371     0
17372     \_kernel_msg_expandable_error:nnn
17373     { kernel } { bad-variable } { #1 }
17374     \prg_return_false:
17375   \fi:
17376 \else:
17377   \if_int_compare:w
17378     \_fp_str_if_eq:nn { \int_value:w #1 } { \tex_the:D #1 }
17379     = 0 \exp_stop_f:
17380     \int_value:w #1
17381   \else:
17382     0
17383     \_kernel_msg_expandable_error:nnn
17384     { kernel } { fp-after-e } { dimension~#1 }
17385   \fi:
17386   \prg_return_false:
17387 \fi:
17388 \else:
17389   0
17390   \_kernel_msg_expandable_error:nnn
17391   { kernel } { fp-missing } { exponent }
17392   \prg_return_true:
17393 \fi:
17394 }

```

(End definition for `_fp_parse_exponent_keep:N`.)

28.5 Constants, functions and prefix operators

28.5.1 Prefix operators

`_fp_parse_prefix_+:Nw` A unary + does nothing: we should continue looking for a number.

```

17395 \cs_new_eq:cN { \_fp_parse_prefix_+:Nw } \_fp_parse_one:Nw

```

(End definition for `_fp_parse_prefix_+:Nw`.)

_fp_parse_apply_function:NNWwN Here, #1 is a precedence, #2 is some extra data used by some functions, #3 is *e.g.*, _fp_sin_o:w, and expands once after the calculation, #4 is the operand, and #5 is a _fp_parse_infix...:N function. We feed the data #2, and the argument #4, to the function #3, which expands \exp:w thus the infix function #5.

```

17396 \cs_new:Npn \_fp_parse_apply_function:NNWwN #1#2#3#4#5
17397 {
17398     #3 #2 #4 @
17399     \exp:w \exp_end_continue_f:w #5 #1
17400 }

```

(End definition for _fp_parse_apply_function:NNWwN.)

_fp_parse_apply_unary:NNWwN In contrast to _fp_parse_apply_function:NNWwN, this checks that the operand #4 is a single argument (namely there is a single ;). We use the fact that any floating point starts with a “safe” token like \s__fp. If there is no argument produce the fp-no-arg error; if there are at least two produce fp-multi-arg. For the error message extract the mathematical function name (such as sin) from the expl3 function that computes it, such as _fp_sin_o:w.

_fp_parse_apply_unary_chk:NwNw
_fp_parse_apply_unary_chk:nNNWw
_fp_parse_apply_unary_type:NNN
_fp_parse_apply_unary_error:NNw

In addition, since there is a single argument we can dispatch on type and check that the resulting function exists. This catches things like sin((1,2)) where it does not make sense to take the sine of a tuple.

```

17401 \cs_new:Npn \_fp_parse_apply_unary:NNWwN #1#2#3#4#5
17402 {
17403     \_fp_parse_apply_unary_chk:NwNw #4 @ ; . \s__fp_stop
17404     \_fp_parse_apply_unary_type:NNN
17405     #3 #2 #4 @
17406     \exp:w \exp_end_continue_f:w #5 #1
17407 }
17408 \cs_new:Npn \_fp_parse_apply_unary_chk:NwNw #1#2 ; #3#4 \s__fp_stop
17409 {
17410     \if_meaning:w @ #3 \else:
17411         \token_if_eq_meaning:NNTF . #3
17412         { \_fp_parse_apply_unary_chk:nNNNNw { no } }
17413         { \_fp_parse_apply_unary_chk:nNNNNw { multi } }
17414     \fi:
17415 }
17416 \cs_new:Npn \_fp_parse_apply_unary_chk:nNNNNw #1#2#3#4#5#6 @
17417 {
17418     #2
17419     \_fp_error:nffn { fp-#1-arg } { \_fp_func_to_name:N #4 } { } { }
17420     \exp_after:wN #4 \exp_after:wN #5 \c_nan_fp @
17421 }
17422 \cs_new:Npn \_fp_parse_apply_unary_type:NNN #1#2#3
17423 {
17424     \_fp_change_func_type:NNN #3 #1 \_fp_parse_apply_unary_error:NNw
17425     #2 #3
17426 }
17427 \cs_new:Npn \_fp_parse_apply_unary_error:NNw #1#2#3 @
17428 { \_fp_invalid_operation_o:fw { \_fp_func_to_name:N #1 } #3 }

```

(End definition for _fp_parse_apply_unary:NNWwN and others.)

`__fp_parse_prefix_-:Nw` The unary `-` and boolean not are harder: we parse the operand using a precedence equal
`__fp_parse_prefix_!:Nw` to the maximum of the previous precedence `##1` and the precedence `\c__fp_prec_not_-`
`int` of the unary operator, then call the appropriate `__fp_⟨operation⟩_o:w` function,
where the `⟨operation⟩` is `set_sign` or `not`.

```

17429 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
17430 {
17431   \cs_new:cpn { __fp_parse_prefix_ #1 :Nw } ##1
17432   {
17433     \exp_after:wN \__fp_parse_apply_unary:NNwN
17434     \exp_after:wN ##1
17435     \exp_after:wN #4
17436     \exp_after:wN #3
17437     \exp:w
17438     \if_int_compare:w #2 < ##1
17439       \__fp_parse_operand:Nw ##1
17440     \else:
17441       \__fp_parse_operand:Nw #2
17442     \fi:
17443     \__fp_parse_expand:w
17444   }
17445 }
17446 \__fp_tmp:w - \c__fp_prec_not_int \__fp_set_sign_o:w 2
17447 \__fp_tmp:w ! \c__fp_prec_not_int \__fp_not_o:w ?

```

(End definition for `__fp_parse_prefix_-:Nw` and `__fp_parse_prefix_!:Nw`.)

`__fp_parse_prefix_:Nw` Numbers which start with a decimal separator (a period) end up here. Of course, we do
not look for an operand, but for the rest of the number. This function is very similar to
`__fp_parse_one_digit:NN` but calls `__fp_parse_strim_zeros:N` to trim zeros after
the decimal point, rather than the `trim_zeros` function for zeros before the decimal
point.

```

17448 \cs_new:cpn { __fp_parse_prefix_:Nw } #1
17449 {
17450   \exp_after:wN \__fp_parse_infix_after_operand:NwN
17451   \exp_after:wN #1
17452   \exp:w \exp_end_continue_f:w
17453   \exp_after:wN \__fp_sanitise:wN
17454   \int_value:w \__fp_int_eval:w 0 \__fp_parse_strim_zeros:N
17455 }

```

(End definition for `__fp_parse_prefix_:Nw`.)

`__fp_parse_prefix(:Nw` The left parenthesis is treated as a unary prefix operator because it appears in exactly
`__fp_parse_lparen_after:NwN` the same settings. If the previous precedence is `\c__fp_prec_func_int` we are parsing
arguments of a function and commas should not build tuples; otherwise commas should
build tuples. We distinguish these cases by precedence: `\c__fp_prec_comma_int` for the
case of arguments, `\c__fp_prec_tuple_int` for the case of tuples. Once the operand
is found, the `lparen_after` auxiliary makes sure that there was a closing parenthesis
(otherwise it complains), and leaves in the input stream an operand, fetching the following
infix operator.

```

17456 \cs_new:cpn { __fp_parse_prefix(:Nw } #1
17457 {
17458   \exp_after:wN \__fp_parse_lparen_after:NwN

```

```

17459     \exp_after:wN #1
17460     \exp:w
17461     \if_int_compare:w #1 = \c__fp_prec_func_int
17462       \__fp_parse_operand:Nw \c__fp_prec_comma_int
17463     \else:
17464       \__fp_parse_operand:Nw \c__fp_prec_tuple_int
17465     \fi:
17466     \__fp_parse_expand:w
17467   }
17468 \cs_new:Npx \__fp_parse_lparen_after:NwN #1#2 @ #3
17469 {
17470   \exp_not:N \token_if_eq_meaning:NNTF #3
17471   \exp_not:c { __fp_parse_infix_ }:N }
17472   {
17473     \exp_not:N \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
17474     \exp_not:N \exp_after:wN
17475     \exp_not:N \__fp_parse_infix_after_paren:NN
17476     \exp_not:N \exp_after:wN #1
17477     \exp_not:N \exp:w
17478     \exp_not:N \__fp_parse_expand:w
17479   }
17480   {
17481     \exp_not:N \__kernel_msg_expandable_error:nnn
17482     { kernel } { fp-missing } { } }
17483   \exp_not:N \tl_if_empty:nT {#2} \exp_not:N \c__fp_empty_tuple_fp
17484   #2 @
17485   \exp_not:N \use_none:n #3
17486 }
17487 }

```

(End definition for __fp_parse_prefix_(:Nw and __fp_parse_lparen_after:NwN.)

__fp_parse_prefix_):Nw The right parenthesis can appear as a prefix in two similar cases: in an empty tuple or tuple ending with a comma, or in an empty argument list or argument list ending with a comma, such as in `max(1,2,)` or in `rand()`.

```

17488 \cs_new:cpn { __fp_parse_prefix_):Nw } #1
17489 {
17490   \if_int_compare:w #1 = \c__fp_prec_comma_int
17491   \else:
17492     \if_int_compare:w #1 = \c__fp_prec_tuple_int
17493     \exp_after:wN \c__fp_empty_tuple_fp \exp:w
17494   \else:
17495     \__kernel_msg_expandable_error:nnn
17496     { kernel } { fp-missing-number } { } }
17497   \exp_after:wN \c_nan_fp \exp:w
17498   \fi:
17499   \exp_end_continue_f:w
17500 \fi:
17501 \__fp_parse_infix_after_paren:NN #1 )
17502 }

```

(End definition for __fp_parse_prefix_):Nw.)

28.5.2 Constants

Some words correspond to constant floating points. The floating point constant is left as a result of `__fp_parse_one:Nw` after expanding `__fp_parse_infix:NN`.

```

\__fp_parse_word_inf:N
\__fp_parse_word_nan:N
\__fp_parse_word_pi:N
\__fp_parse_word_deg:N
\__fp_parse_word_true:N
\__fp_parse_word_false:N
17503 \cs_set_protected:Npn \__fp_tmp:w #1 #2
17504 {
17505   \cs_new:cpn { __fp_parse_word_#1:N }
17506     { \exp_after:wN #2 \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN }
17507 }

```

```

17508 \__fp_tmp:w { inf } \c_inf_fp
17509 \__fp_tmp:w { nan } \c_nan_fp
17510 \__fp_tmp:w { pi } \c_pi_fp
17511 \__fp_tmp:w { deg } \c_one_degree_fp
17512 \__fp_tmp:w { true } \c_one_fp
17513 \__fp_tmp:w { false } \c_zero_fp

```

(End definition for `__fp_parse_word_inf:N` and others.)

Copies of `__fp_parse_word_...:N` commands, to allow arbitrary case as mandated by the standard.

```

\__fp_parse_caseless_inf:N
\__fp_parse_caseless_infinity:N
\__fp_parse_caseless_nan:N
17514 \cs_new_eq:NN \__fp_parse_caseless_inf:N \__fp_parse_word_inf:N
17515 \cs_new_eq:NN \__fp_parse_caseless_infinity:N \__fp_parse_word_inf:N
17516 \cs_new_eq:NN \__fp_parse_caseless_nan:N \__fp_parse_word_nan:N

```

(End definition for `__fp_parse_caseless_inf:N`, `__fp_parse_caseless_infinity:N`, and `__fp_parse_caseless_nan:N`.)

Dimension units are also floating point constants but their value is not stored as a floating point constant. We give the values explicitly here.

```

\__fp_parse_word_pt:N
\__fp_parse_word_in:N
\__fp_parse_word_pc:N
\__fp_parse_word_cm:N
\__fp_parse_word_mm:N
\__fp_parse_word_dd:N
\__fp_parse_word_cc:N
\__fp_parse_word_nd:N
\__fp_parse_word_nc:N
\__fp_parse_word_bp:N
\__fp_parse_word_sp:N
17517 \cs_set_protected:Npn \__fp_tmp:w #1 #2
17518 {
17519   \cs_new:cpn { __fp_parse_word_#1:N }
17520     {
17521       \__fp_exp_after_f:nw { \__fp_parse_infix:NN }
17522       \s__fp \__fp_chk:w 10 #2 ;
17523     }
17524 }
17525 \__fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
17526 \__fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
17527 \__fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
17528 \__fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
17529 \__fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
17530 \__fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
17531 \__fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
17532 \__fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
17533 \__fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
17534 \__fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
17535 \__fp_tmp:w {sp} { {-4} {1525} {8789} {0625} {0000} }

```

(End definition for `__fp_parse_word_pt:N` and others.)

The font-dependent units `em` and `ex` must be evaluated on the fly. We reuse an auxiliary of `\dim_to_fp:n`.

```

\__fp_parse_word_em:N
\__fp_parse_word_ex:N
17536 \tl_map_inline:nn { {em} {ex} }
17537 {

```

```

17538 \cs_new:cpn { __fp_parse_word_#1:N }
17539 {
17540   \exp_after:wN \__fp_from_dim_test:ww
17541   \exp_after:wN 0 \exp_after:wN ,
17542   \int_value:w \dim_to_decimal_in_sp:n { 1 #1 } \exp_after:wN ;
17543   \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN
17544 }
17545 }

```

(End definition for __fp_parse_word_em:N and __fp_parse_word_ex:N.)

28.5.3 Functions

```

\__fp_parse_unary_function:NNN
\__fp_parse_function:NNN
17546 \cs_new:Npn \__fp_parse_unary_function:NNN #1#2#3
17547 {
17548   \exp_after:wN \__fp_parse_apply_unary:NNNwN
17549   \exp_after:wN #3
17550   \exp_after:wN #2
17551   \exp_after:wN #1
17552   \exp:w
17553   \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
17554 }
17555 \cs_new:Npn \__fp_parse_function:NNN #1#2#3
17556 {
17557   \exp_after:wN \__fp_parse_apply_function:NNNwN
17558   \exp_after:wN #3
17559   \exp_after:wN #2
17560   \exp_after:wN #1
17561   \exp:w
17562   \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
17563 }

```

(End definition for __fp_parse_unary_function:NNN and __fp_parse_function:NNN.)

28.6 Main functions

__fp_parse:n Start an \exp:w expansion so that __fp_parse:n expands in two steps. The __fp_parse_operand:Nw function performs computations until reaching an operation with precedence \c__fp_prec_end_int or less, namely, the end of the expression. The marker \s__fp_expr_mark indicates that the next token is an already parsed version of an infix operator, and __fp_parse_infix_end:N has infinitely negative precedence. Finally, clean up a (well-defined) set of extra tokens and stop the initial expansion with \exp_end:.

```

17564 \cs_new:Npn \__fp_parse:n #1
17565 {
17566   \exp:w
17567   \exp_after:wN \__fp_parse_after:ww
17568   \exp:w
17569   \__fp_parse_operand:Nw \c__fp_prec_end_int
17570   \__fp_parse_expand:w #1
17571   \s__fp_expr_mark \__fp_parse_infix_end:N
17572   \s__fp_expr_stop

```

```

17573     \exp_end:
17574   }
17575   \cs_new:Npn \__fp_parse_after:ww
17576     #1@ \__fp_parse_infix_end:N \s__fp_expr_stop #2 { #2 #1 }
17577   \cs_new:Npn \__fp_parse_o:n #1
17578   {
17579     \exp:w
17580     \exp_after:wN \__fp_parse_after:ww
17581     \exp:w
17582     \__fp_parse_operand:Nw \c__fp_prec_end_int
17583     \__fp_parse_expand:w #1
17584     \s__fp_expr_mark \__fp_parse_infix_end:N
17585     \s__fp_expr_stop
17586   {
17587     \exp_end_continue_f:w
17588     \__fp_exp_after_any_f:nw { \exp_after:wN \exp_stop_f: }
17589   }
17590 }

```

(End definition for __fp_parse:n, __fp_parse_o:n, and __fp_parse_after:ww.)

__fp_parse_operand:Nw This is just a shorthand which sets up both __fp_parse_continue:NwN and __fp_parse_one:Nw with the same precedence. Note the trailing \exp:w.

```

17591 \cs_new:Npn \__fp_parse_operand:Nw #1
17592 {
17593   \exp_end_continue_f:w
17594   \exp_after:wN \__fp_parse_continue:NwN
17595   \exp_after:wN #1
17596   \exp:w \exp_end_continue_f:w
17597   \exp_after:wN \__fp_parse_one:Nw
17598   \exp_after:wN #1
17599   \exp:w
17600 }
17601 \cs_new:Npn \__fp_parse_continue:NwN #1 #2 @ #3 { #3 #1 #2 @ }

```

(End definition for __fp_parse_operand:Nw and __fp_parse_continue:NwN.)

_fp_parse_apply_binary:NwNwN Receives $\langle precedence \rangle \langle operand_1 \rangle @ \langle operation \rangle \langle operand_2 \rangle @ \langle infix command \rangle$. Builds the appropriate call to the $\langle operation \rangle$ #3, dispatching on both types. If the resulting control sequence does not exist, the operation is not allowed.

This is redefined in l3fp-extras.

```

17602 \cs_new:Npn \__fp_parse_apply_binary:NwNwN #1 #2#3@ #4 #5#6@ #7
17603 {
17604   \exp_after:wN \__fp_parse_continue:NwN
17605   \exp_after:wN #1
17606   \exp:w \exp_end_continue_f:w
17607   \exp_after:wN \__fp_parse_apply_binary_chk:NN
17608   \cs:w
17609   __fp
17610   \__fp_type_from_scan:N #2
17611   _#4
17612   \__fp_type_from_scan:N #5
17613   _o:ww
17614   \cs_end:

```



```

17615         #4
17616         #2#3 #5#6
17617         \exp:w \exp_end_continue_f:w #7 #1
17618     }
17619 \cs_new:Npn \__fp_parse_apply_binary_chk:NN #1#2
17620 {
17621     \if_meaning:w \scan_stop: #1
17622     \__fp_parse_apply_binary_error:NNN #2
17623     \fi:
17624     #1
17625 }
17626 \cs_new:Npn \__fp_parse_apply_binary_error:NNN #1#2#3
17627 {
17628     #2
17629     \__fp_invalid_operation_o:Nww #1
17630 }

```

(End definition for __fp_parse_apply_binary:NwNwN, __fp_parse_apply_binary_chk:NN, and __fp_parse_apply_binary_error:NNN.)

__fp_binary_type_o:Nww
__fp_binary_rev_type_o:Nww

Applies the operator #1 to its two arguments, dispatching according to their types, and expands once after the result. The rev version swaps its arguments before doing this.

```

17631 \cs_new:Npn \__fp_binary_type_o:Nww #1 #2#3 ; #4
17632 {
17633     \exp_after:wN \__fp_parse_apply_binary_chk:NN
17634     \cs:w
17635     __fp
17636     \__fp_type_from_scan:N #2
17637     _ #1
17638     \__fp_type_from_scan:N #4
17639     _o:ww
17640     \cs_end:
17641     #1
17642     #2 #3 ; #4
17643 }
17644 \cs_new:Npn \__fp_binary_rev_type_o:Nww #1 #2#3 ; #4#5 ;
17645 {
17646     \exp_after:wN \__fp_parse_apply_binary_chk:NN
17647     \cs:w
17648     __fp
17649     \__fp_type_from_scan:N #4
17650     _ #1
17651     \__fp_type_from_scan:N #2
17652     _o:ww
17653     \cs_end:
17654     #1
17655     #4 #5 ; #2 #3 ;
17656 }

```

(End definition for __fp_binary_type_o:Nww and __fp_binary_rev_type_o:Nww.)

28.7 Infix operators

__fp_parse_infix_after_operand:NwN

```

17657 \cs_new:Npn \__fp_parse_infix_after_operand:NwN #1 #2;
17658 {
17659   \__fp_exp_after_f:nw { \__fp_parse_infix:NN #1 }
17660   #2;
17661 }
17662 \cs_new:Npn \__fp_parse_infix:NN #1 #2
17663 {
17664   \if_catcode:w \scan_stop: \exp_not:N #2
17665   \if_int_compare:w
17666     \__fp_str_if_eq:nn { \s__fp_expr_mark } { \exp_not:N #2 }
17667     = 0 \exp_stop_f:
17668     \exp_after:wN \exp_after:wN
17669     \exp_after:wN \__fp_parse_infix_mark:NNN
17670   \else:
17671     \exp_after:wN \exp_after:wN
17672     \exp_after:wN \__fp_parse_infix_juxt:N
17673   \fi:
17674 \else:
17675   \if_int_compare:w
17676     \__fp_int_eval:w
17677     ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
17678     = 3 \exp_stop_f:
17679     \exp_after:wN \exp_after:wN
17680     \exp_after:wN \__fp_parse_infix_juxt:N
17681   \else:
17682     \exp_after:wN \__fp_parse_infix_check:NNN
17683     \cs:w
17684       __fp_parse_infix_ \token_to_str:N #2 :N
17685     \exp_after:wN \exp_after:wN \exp_after:wN
17686     \cs_end:
17687   \fi:
17688 \fi:
17689 #1
17690 #2
17691 }
17692 \cs_new:Npn \__fp_parse_infix_check:NNN #1#2#3
17693 {
17694   \if_meaning:w \scan_stop: #1
17695   \__kernel_msg_expandable_error:nnn
17696   { kernel } { fp-missing } { * }
17697   \exp_after:wN \__fp_parse_infix_mul:N
17698   \exp_after:wN #2
17699   \exp_after:wN #3
17700 \else:
17701   \exp_after:wN #1
17702   \exp_after:wN #2
17703   \exp:w \exp_after:wN \__fp_parse_expand:w
17704 \fi:
17705 }

```

(End definition for __fp_parse_infix_after_operand:NwN.)

__fp_parse_infix_after_paren:NN Variant of __fp_parse_infix:NN for use after a closing parenthesis. The only difference is that __fp_parse_infix_juxt:N is replaced by __fp_parse_infix_mul:N.

```

17706 \cs_new:Npn \__fp_parse_infix_after_paren:NN #1 #2
17707 {
17708   \if_catcode:w \scan_stop: \exp_not:N #2
17709   \if_int_compare:w
17710     \__fp_str_if_eq:nn { \s__fp_expr_mark } { \exp_not:N #2 }
17711     = 0 \exp_stop_f:
17712     \exp_after:wN \exp_after:wN
17713     \exp_after:wN \__fp_parse_infix_mark:NNN
17714   \else:
17715     \exp_after:wN \exp_after:wN
17716     \exp_after:wN \__fp_parse_infix_mul:N
17717   \fi:
17718 \else:
17719   \if_int_compare:w
17720     \__fp_int_eval:w
17721     ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
17722     = 3 \exp_stop_f:
17723     \exp_after:wN \exp_after:wN
17724     \exp_after:wN \__fp_parse_infix_mul:N
17725   \else:
17726     \exp_after:wN \__fp_parse_infix_check:NNN
17727   \cs:w
17728     __fp_parse_infix_ \token_to_str:N #2 :N
17729     \exp_after:wN \exp_after:wN \exp_after:wN
17730   \cs_end:
17731   \fi:
17732 \fi:
17733 #1
17734 #2
17735 }

```

(End definition for __fp_parse_infix_after_paren:NN.)

28.7.1 Closing parentheses and commas

__fp_parse_infix_mark:NNN As an infix operator, \s__fp_expr_mark means that the next token (#3) has already gone through __fp_parse_infix:NN and should be provided the precedence #1. The scan mark #2 is discarded.

```

17736 \cs_new:Npn \__fp_parse_infix_mark:NNN #1#2#3 { #3 #1 }

```

(End definition for __fp_parse_infix_mark:NNN.)

__fp_parse_infix_end:N This one is a little bit odd: force every previous operator to end, regardless of the precedence.

```

17737 \cs_new:Npn \__fp_parse_infix_end:N #1
17738 { @ \use_none:n \__fp_parse_infix_end:N }

```

(End definition for __fp_parse_infix_end:N.)

__fp_parse_infix_):N This is very similar to __fp_parse_infix_end:N, complaining about an extra closing parenthesis if the previous operator was the beginning of the expression, with precedence \c__fp_prec_end_int.

```

17739 \cs_set_protected:Npn \__fp_tmp:w #1
17740 {

```

```

17741 \cs_new:Npn #1 ##1
17742 {
17743   \if_int_compare:w ##1 > \c__fp_prec_end_int
17744     \exp_after:wN @
17745     \exp_after:wN \use_none:n
17746     \exp_after:wN #1
17747   \else:
17748     \__kernel_msg_expandable_error:nnn { kernel } { fp-extra } { } }
17749     \exp_after:wN \__fp_parse_infix:NN
17750     \exp_after:wN ##1
17751     \exp:w \exp_after:wN \__fp_parse_expand:w
17752   \fi:
17753 }
17754 }
17755 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_):N }

```

(End definition for __fp_parse_infix_):N.)

```

\__fp_parse_infix_,:N
\__fp_parse_infix_comma:w
\__fp_parse_apply_comma:NwNwN

```

As for other infix operations, if the previous operations has higher precedence the comma waits. Otherwise we call __fp_parse_operand:Nw to read more comma-delimited arguments that __fp_parse_infix_comma:w simply concatenates into a @-delimited array. The first comma in a tuple that is not a function argument is distinguished: in that case call __fp_parse_apply_comma:NwNwN whose job is to convert the first item of the tuple and an array of the remaining items into a tuple. In contrast to __fp_parse_apply_binary:NwNwN this function's operands are not single-object arrays.

```

17756 \cs_set_protected:Npn \__fp_tmp:w #1
17757 {
17758   \cs_new:Npn #1 ##1
17759   {
17760     \if_int_compare:w ##1 > \c__fp_prec_comma_int
17761       \exp_after:wN @
17762       \exp_after:wN \use_none:n
17763       \exp_after:wN #1
17764     \else:
17765       \if_int_compare:w ##1 < \c__fp_prec_comma_int
17766         \exp_after:wN @
17767         \exp_after:wN \__fp_parse_apply_comma:NwNwN
17768         \exp_after:wN ,
17769         \exp:w
17770       \else:
17771         \exp_after:wN \__fp_parse_infix_comma:w
17772         \exp:w
17773       \fi:
17774       \__fp_parse_operand:Nw \c__fp_prec_comma_int
17775       \exp_after:wN \__fp_parse_expand:w
17776     \fi:
17777   }
17778 }
17779 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_,:N }
17780 \cs_new:Npn \__fp_parse_infix_comma:w #1 @
17781 { #1 @ \use_none:n }
17782 \cs_new:Npn \__fp_parse_apply_comma:NwNwN #1 #2@ #3 #4@ #5
17783 {
17784   \exp_after:wN \__fp_parse_continue:NwN

```

```

17785 \exp_after:wN #1
17786 \exp:w \exp_end_continue_f:w
17787 \__fp_exp_after_tuple_f:nw { }
17788 \s__fp_tuple \__fp_tuple_chk:w { #2 #4 } ;
17789 #5 #1
17790 }

```

(End definition for __fp_parse_infix_.,:N, __fp_parse_infix_comma:w, and __fp_parse_apply_comma:NwNwN.)

28.7.2 Usual infix operators

As described in the “work plan”, each infix operator has an associated \..._infix... function, a computing function, and precedence, given as arguments to __fp_tmp:w. Using the general mechanism for arithmetic operations. The power operation must be associative in the opposite order from all others. For this, we use two distinct precedences.

```

\__fp_parse_infix_+:N
\__fp_parse_infix_-:N
\__fp_parse_infix_juxt:N
\__fp_parse_infix_/:N
\__fp_parse_infix_mul:N
\__fp_parse_infix_and:N
\__fp_parse_infix_or:N
\__fp_parse_infix_^:N
17791 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
17792 {
17793   \cs_new:Npn #1 ##1
17794   {
17795     \if_int_compare:w ##1 < #3
17796       \exp_after:wN @
17797       \exp_after:wN \__fp_parse_apply_binary:NwNwN
17798       \exp_after:wN #2
17799       \exp:w
17800       \__fp_parse_operand:Nw #4
17801       \exp_after:wN \__fp_parse_expand:w
17802     \else:
17803       \exp_after:wN @
17804       \exp_after:wN \use_none:n
17805       \exp_after:wN #1
17806     \fi:
17807   }
17808 }
17809 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_^:N } ^
17810 \c__fp_prec_hatii_int \c__fp_prec_hat_int
17811 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_juxt:N } *
17812 \c__fp_prec_juxt_int \c__fp_prec_juxt_int
17813 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_/:N } /
17814 \c__fp_prec_times_int \c__fp_prec_times_int
17815 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_mul:N } *
17816 \c__fp_prec_times_int \c__fp_prec_times_int
17817 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_-:N } -
17818 \c__fp_prec_plus_int \c__fp_prec_plus_int
17819 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_+:N } +
17820 \c__fp_prec_plus_int \c__fp_prec_plus_int
17821 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_and:N } &
17822 \c__fp_prec_and_int \c__fp_prec_and_int
17823 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_or:N } |
17824 \c__fp_prec_or_int \c__fp_prec_or_int

```

(End definition for __fp_parse_infix_+:N and others.)

28.7.3 Juxtaposition

`__fp_parse_infix_(:N` When an opening parenthesis appears where we expect an infix operator, we compute the product of the previous operand and the contents of the parentheses using `__fp_parse_infix_mul:N`.

```
17825 \cs_new:cpn { __fp_parse_infix_(:N } #1
17826 { \__fp_parse_infix_mul:N #1 ( }
```

(End definition for `__fp_parse_infix_(:N`.)

28.7.4 Multi-character cases

`__fp_parse_infix_*:N`

```
17827 \cs_set_protected:Npn \__fp_tmp:w #1
17828 {
17829   \cs_new:cpn { __fp_parse_infix_*:N } ##1##2
17830   {
17831     \if:w * \exp_not:N ##2
17832       \exp_after:wN #1
17833       \exp_after:wN ##1
17834     \else:
17835       \exp_after:wN \__fp_parse_infix_mul:N
17836       \exp_after:wN ##1
17837       \exp_after:wN ##2
17838     \fi:
17839   }
17840 }
17841 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix^:N }
```

(End definition for `__fp_parse_infix_*:N`.)

`__fp_parse_infix_|:Nw`

`__fp_parse_infix_&:Nw`

```
17842 \cs_set_protected:Npn \__fp_tmp:w #1#2#3
17843 {
17844   \cs_new:Npn #1 ##1##2
17845   {
17846     \if:w #2 \exp_not:N ##2
17847       \exp_after:wN #1
17848       \exp_after:wN ##1
17849       \exp:w \exp_after:wN \__fp_parse_expand:w
17850     \else:
17851       \exp_after:wN #3
17852       \exp_after:wN ##1
17853       \exp_after:wN ##2
17854     \fi:
17855   }
17856 }
17857 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_|:N } | \__fp_parse_infix_or:N
17858 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_&:N } & \__fp_parse_infix_and:N
```

(End definition for `__fp_parse_infix_|:Nw` and `__fp_parse_infix_&:Nw`.)

28.7.5 Ternary operator

```

__fp_parse_infix_?:N
__fp_parse_infix_:N
17859 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
17860 {
17861   \cs_new:Npn #1 ##1
17862   {
17863     \if_int_compare:w ##1 < \c__fp_prec_quest_int
17864     #4
17865     \exp_after:wN @
17866     \exp_after:wN #2
17867     \exp:w
17868     \__fp_parse_operand:Nw #3
17869     \exp_after:wN \__fp_parse_expand:w
17870   \else:
17871     \exp_after:wN @
17872     \exp_after:wN \use_none:n
17873     \exp_after:wN #1
17874   \fi:
17875 }
17876 }
17877 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_?:N }
17878 \__fp_ternary:NwwN \c__fp_prec_quest_int { }
17879 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_:N }
17880 \__fp_ternary_auxii:NwwN \c__fp_prec_colon_int
17881 {
17882   \__kernel_msg_expandable_error:nnnn
17883   { kernel } { fp-missing } { ? } { ~for~?: }
17884 }

```

(End definition for __fp_parse_infix_?:N and __fp_parse_infix_:N.)

28.7.6 Comparisons

```

__fp_parse_infix_<:N
__fp_parse_infix_=:N
__fp_parse_infix_>:N
__fp_parse_infix_!:N
__fp_parse_excl_error:
__fp_parse_compare:NNNNNNN
  __fp_parse_compare_auxi:NNNNNNN
  __fp_parse_compare_auxii:NNNNN
  __fp_parse_compare_end:NNNNw
  __fp_compare:wNNNNw
17885 \cs_new:cpn { __fp_parse_infix_<:N } #1
17886 { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 < }
17887 \cs_new:cpn { __fp_parse_infix_=:N } #1
17888 { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 = }
17889 \cs_new:cpn { __fp_parse_infix_>:N } #1
17890 { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 > }
17891 \cs_new:cpn { __fp_parse_infix_!:N } #1
17892 {
17893   \exp_after:wN \__fp_parse_compare:NNNNNNN
17894   \exp_after:wN #1
17895   \exp_after:wN 0
17896   \exp_after:wN 1
17897   \exp_after:wN 1
17898   \exp_after:wN 1
17899   \exp_after:wN 1
17900 }
17901 \cs_new:Npn \__fp_parse_excl_error:
17902 {
17903   \__kernel_msg_expandable_error:nnnn

```

```

17904         { kernel } { fp-missing } { = } { ~after~!. }
17905     }
17906 \cs_new:Npn \__fp_parse_compare:NNNNNNN #1
17907 {
17908     \if_int_compare:w #1 < \c__fp_prec_comp_int
17909         \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
17910         \exp_after:wN \__fp_parse_excl_error:
17911     \else:
17912         \exp_after:wN @
17913         \exp_after:wN \use_none:n
17914         \exp_after:wN \__fp_parse_compare:NNNNNNN
17915     \fi:
17916 }
17917 \cs_new:Npn \__fp_parse_compare_auxi:NNNNNNN #1#2#3#4#5#6#7
17918 {
17919     \if_case:w
17920         \__fp_int_eval:w \exp_after:wN ' \token_to_str:N #7 - '<
17921         \__fp_int_eval_end:
17922         \__fp_parse_compare_auxii:NNNNN #2#2#4#5#6
17923     \or: \__fp_parse_compare_auxii:NNNNN #2#3#2#5#6
17924     \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#2#6
17925     \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#5#2
17926     \else: #1 \__fp_parse_compare_end:NNNNw #3#4#5#6#7
17927     \fi:
17928 }
17929 \cs_new:Npn \__fp_parse_compare_auxii:NNNNN #1#2#3#4#5
17930 {
17931     \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
17932     \exp_after:wN \prg_do_nothing:
17933     \exp_after:wN #1
17934     \exp_after:wN #2
17935     \exp_after:wN #3
17936     \exp_after:wN #4
17937     \exp_after:wN #5
17938     \exp:w \exp_after:wN \__fp_parse_expand:w
17939 }
17940 \cs_new:Npn \__fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
17941 {
17942     \fi:
17943     \exp_after:wN @
17944     \exp_after:wN \__fp_parse_apply_compare:NwNNNNNNwN
17945     \exp_after:wN \c_one_fp
17946     \exp_after:wN #1
17947     \exp_after:wN #2
17948     \exp_after:wN #3
17949     \exp_after:wN #4
17950     \exp:w
17951     \__fp_parse_operand:Nw \c__fp_prec_comp_int \__fp_parse_expand:w #5
17952 }
17953 \cs_new:Npn \__fp_parse_apply_compare:NwNNNNNNwN
17954     #1 #2@ #3 #4#5#6#7 #8@ #9
17955 {
17956     \if_int_odd:w
17957         \if_meaning:w \c_zero_fp #3

```



```

17958         0
17959         \else:
17960             \if_case:w \_fp_compare_back_any:ww #8 #2 \exp_stop_f:
17961                 #5 \or: #6 \or: #7 \else: #4
17962             \fi:
17963         \fi:
17964         \exp_stop_f:
17965         \exp_after:wN \_fp_parse_apply_compare_aux:NNwN
17966         \exp_after:wN \c_one_fp
17967     \else:
17968         \exp_after:wN \_fp_parse_apply_compare_aux:NNwN
17969         \exp_after:wN \c_zero_fp
17970     \fi:
17971     #1 #8 #9
17972 }
17973 \cs_new:Npn \_fp_parse_apply_compare_aux:NNwN #1 #2 #3; #4
17974 {
17975     \if_meaning:w \_fp_parse_compare:NNNNNNN #4
17976         \exp_after:wN \_fp_parse_continue_compare:NNwNN
17977         \exp_after:wN #1
17978         \exp_after:wN #2
17979         \exp:w \exp_end_continue_f:w
17980         \_fp_exp_after_o:w #3;
17981         \exp:w \exp_end_continue_f:w
17982     \else:
17983         \exp_after:wN \_fp_parse_continue:NwN
17984         \exp_after:wN #2
17985         \exp:w \exp_end_continue_f:w
17986         \exp_after:wN #1
17987         \exp:w \exp_end_continue_f:w
17988     \fi:
17989     #4 #2
17990 }
17991 \cs_new:Npn \_fp_parse_continue_compare:NNwNN #1#2 #3@ #4#5
17992 { #4 #2 #3@ #1 }

```

(End definition for _fp_parse_infix_<:N and others.)

28.8 Tools for functions

_fp_parse_function_all_fp_o:fnw Followed by {<function name>} {<code>} <float array> @ this checks all floats are floating point numbers (no tuples).

```

17993 \cs_new:Npn \_fp_parse_function_all_fp_o:fnw #1#2#3 @
17994 {
17995     \_fp_array_if_all_fp:nTF {#3}
17996     { #2 #3 @ }
17997     {
17998         \_fp_error:nffn { fp-bad-args }
17999         {#1}
18000         { \fp_to_tl:n { \s__fp_tuple \_fp_tuple_chk:w {#3} ; } }
18001         { }
18002         \exp_after:wN \c_nan_fp
18003     }
18004 }

```

(End definition for `_fp_parse_function_all_fp_o:fnw`.)

`_fp_parse_function_one_two:nnw` This is followed by `{(function name)}{(code)}{float array} @`. It checks that the `{float array}` consists of one or two floating point numbers (not tuples), then leaves the `{code}` (if there is one float) or its tail (if there are two floats) followed by the `{float array}`. The `{code}` should start with a single token such as `_fp_atan_default:w` that deals with the single-float case.

The first `_fp_if_type_fp:NTwFw` test catches the case of no argument and the case of a tuple argument. The next one distinguishes the case of a single argument (no error, just add `\c_one_fp`) from a tuple second argument. Finally check there is no further argument.

```

18005 \cs_new:Npn \_fp_parse_function_one_two:nnw #1#2#3
18006   {
18007     \_fp_if_type_fp:NTwFw
18008     #3 { } \s__fp \_fp_parse_function_one_two_error_o:w \s__fp_stop
18009     \_fp_parse_function_one_two_aux:nnw {#1} {#2} #3
18010   }
18011 \cs_new:Npn \_fp_parse_function_one_two_error_o:w #1#2#3#4 @
18012   {
18013     \_fp_error:nffn { fp-bad-args }
18014     {#2}
18015     { \fp_to_tl:n { \s__fp_tuple \_fp_tuple_chk:w {#4} ; } }
18016     { }
18017     \exp_after:wN \c_nan_fp
18018   }
18019 \cs_new:Npn \_fp_parse_function_one_two_aux:nnw #1#2 #3; #4
18020   {
18021     \_fp_if_type_fp:NTwFw
18022     #4 { }
18023     \s__fp
18024     {
18025       \if_meaning:w @ #4
18026       \exp_after:wN \use_iv:nnnn
18027       \fi:
18028       \_fp_parse_function_one_two_error_o:w
18029     }
18030     \s__fp_stop
18031     \_fp_parse_function_one_two_auxii:nnw {#1} {#2} #3; #4
18032   }
18033 \cs_new:Npn \_fp_parse_function_one_two_auxii:nnw #1#2#3; #4; #5
18034   {
18035     \if_meaning:w @ #5 \else:
18036     \exp_after:wN \_fp_parse_function_one_two_error_o:w
18037     \fi:
18038     \use_ii:nn {#1} { \use_none:n #2 } #3; #4; #5
18039   }

```

(End definition for `_fp_parse_function_one_two:nnw` and others.)

`_fp_tuple_map_o:nw` Apply #1 to all items in the following tuple and expand once afterwards. The code #1 should itself expand once after its result.

```

18040 \cs_new:Npn \_fp_tuple_map_o:nw #1 \s__fp_tuple \_fp_tuple_chk:w #2 ;
18041   {

```

```

18042 \exp_after:wN \s__fp_tuple
18043 \exp_after:wN \__fp_tuple_chk:w
18044 \exp_after:wN {
18045   \exp:w \exp_end_continue_f:w
18046   \__fp_tuple_map_loop_o:nw {#1} #2
18047   { \s__fp \prg_break: } ;
18048   \prg_break_point:
18049   \exp_after:wN } \exp_after:wN ;
18050 }
18051 \cs_new:Npn \__fp_tuple_map_loop_o:nw #1#2#3 ;
18052 {
18053   \use_none:n #2
18054   #1 #2 #3 ;
18055   \exp:w \exp_end_continue_f:w
18056   \__fp_tuple_map_loop_o:nw {#1}
18057 }

```

(End definition for __fp_tuple_map_o:nw and __fp_tuple_map_loop_o:nw.)

__fp_tuple_mapthread_o:nww Apply #1 to pairs of items in the two following tuples and expand once afterwards.

```

\__fp_tuple_mapthread_loop_o:nw
18058 \cs_new:Npn \__fp_tuple_mapthread_o:nww #1
18059   \s__fp_tuple \__fp_tuple_chk:w #2 ;
18060   \s__fp_tuple \__fp_tuple_chk:w #3 ;
18061   {
18062     \exp_after:wN \s__fp_tuple
18063     \exp_after:wN \__fp_tuple_chk:w
18064     \exp_after:wN {
18065       \exp:w \exp_end_continue_f:w
18066       \__fp_tuple_mapthread_loop_o:nw {#1}
18067       #2 { \s__fp \prg_break: } ; @
18068       #3 { \s__fp \prg_break: } ;
18069       \prg_break_point:
18070       \exp_after:wN } \exp_after:wN ;
18071     }
18072   \cs_new:Npn \__fp_tuple_mapthread_loop_o:nw #1#2#3 ; #4 @ #5#6 ;
18073   {
18074     \use_none:n #2
18075     \use_none:n #5
18076     #1 #2 #3 ; #5 #6 ;
18077     \exp:w \exp_end_continue_f:w
18078     \__fp_tuple_mapthread_loop_o:nw {#1} #4 @
18079   }

```

(End definition for __fp_tuple_mapthread_o:nww and __fp_tuple_mapthread_loop_o:nw.)

28.9 Messages

```

18080 \__kernel_msg_new:nnn { kernel } { fp-deprecated }
18081 { '#1'~deprecated;~use~'#2' }
18082 \__kernel_msg_new:nnn { kernel } { unknown-fp-word }
18083 { Unknown~fp~word~#1. }
18084 \__kernel_msg_new:nnn { kernel } { fp-missing }
18085 { Missing~#1~inserted #2. }
18086 \__kernel_msg_new:nnn { kernel } { fp-extra }
18087 { Extra~#1~ignored. }

```

```

18088 \__kernel_msg_new:nnn { kernel } { fp-early-end }
18089 { Premature-end-in-fp-expression. }
18090 \__kernel_msg_new:nnn { kernel } { fp-after-e }
18091 { Cannot~use~#1 after~'e'. }
18092 \__kernel_msg_new:nnn { kernel } { fp-missing-number }
18093 { Missing-number~before~'#1'. }
18094 \__kernel_msg_new:nnn { kernel } { fp-unknown-symbol }
18095 { Unknown-symbol~#1~ignored. }
18096 \__kernel_msg_new:nnn { kernel } { fp-extra-comma }
18097 { Unexpected~comma~turned~to~nan~result. }
18098 \__kernel_msg_new:nnn { kernel } { fp-no-arg }
18099 { #1~got~no~argument;~used~nan. }
18100 \__kernel_msg_new:nnn { kernel } { fp-multi-arg }
18101 { #1~got~more~than~one~argument;~used~nan. }
18102 \__kernel_msg_new:nnn { kernel } { fp-num-args }
18103 { #1~expects~between~#2~and~#3~arguments. }
18104 \__kernel_msg_new:nnn { kernel } { fp-bad-args }
18105 { Arguments~in~#1#2~are~invalid. }
18106 \__kernel_msg_new:nnn { kernel } { fp-infty-pi }
18107 { Math-command~#1 is-not-an~fp }
18108 \cs_if_exist:cT { @unexpandable@protect }
18109 {
18110   \__kernel_msg_new:nnn { kernel } { fp-robust-cmd }
18111   { Robust~command~#1 invalid~in~fp-expression! }
18112 }
18113 </package>

```

29 l3fp-assign implementation

```

18114 <*package>
18115 <@@=fp>

```

29.1 Assigning values

\fp_new:N Floating point variables are initialized to be +0.

```

18116 \cs_new_protected:Npn \fp_new:N #1
18117 { \cs_new_eq:NN #1 \c_zero_fp }
18118 \cs_generate_variant:Nn \fp_new:N {c}

```

(End definition for \fp_new:N. This function is documented on page 202.)

\fp_set:Nn Simply use __fp_parse:n within various f-expanding assignments.

```

\fp_set:cn 18119 \cs_new_protected:Npn \fp_set:Nn #1#2
\fp_gset:Nn 18120 { \tl_set:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_gset:cn 18121 \cs_new_protected:Npn \fp_gset:Nn #1#2
\fp_const:Nn 18122 { \tl_gset:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_const:cn 18123 \cs_new_protected:Npn \fp_const:Nn #1#2
18124 { \tl_const:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
18125 \cs_generate_variant:Nn \fp_set:Nn {c}
18126 \cs_generate_variant:Nn \fp_gset:Nn {c}
18127 \cs_generate_variant:Nn \fp_const:Nn {c}

```

(End definition for \fp_set:Nn, \fp_gset:Nn, and \fp_const:Nn. These functions are documented on page 203.)

```

\fp_set_eq:NN Copying a floating point is the same as copying the underlying token list.
\fp_set_eq:cN 18128 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
\fp_set_eq:Nc 18129 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
\fp_set_eq:cc 18130 \cs_generate_variant:Nn \fp_set_eq:NN { c , Nc , cc }
\fp_gset_eq:NN 18131 \cs_generate_variant:Nn \fp_gset_eq:NN { c , Nc , cc }
\fp_gset_eq:cN (End definition for \fp_set_eq:NN and \fp_gset_eq:NN. These functions are documented on page 203.)
\fp_gset_eq:Nc
\fp_gset_eq:cc
\fp_zero:N Setting a floating point to zero: copy \c_zero_fp.
\fp_zero:c 18132 \cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }
\fp_gzero:N 18133 \cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }
\fp_gzero:c 18134 \cs_generate_variant:Nn \fp_zero:N { c }
18135 \cs_generate_variant:Nn \fp_gzero:N { c }

(End definition for \fp_zero:N and \fp_gzero:N. These functions are documented on page 202.)

\fp_zero_new:N Set the floating point to zero, or define it if needed.
\fp_zero_new:c 18136 \cs_new_protected:Npn \fp_zero_new:N #1
\fp_gzero_new:N 18137 { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
\fp_gzero_new:c 18138 \cs_new_protected:Npn \fp_gzero_new:N #1
18139 { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
18140 \cs_generate_variant:Nn \fp_zero_new:N { c }
18141 \cs_generate_variant:Nn \fp_gzero_new:N { c }

(End definition for \fp_zero_new:N and \fp_gzero_new:N. These functions are documented on page 203.)

```

29.2 Updating values

These match the equivalent functions in l3int and l3skip.

```

\fp_add:NN For the sake of error recovery we should not simply set #1 to #1 ± (#2): for instance, if #2
\fp_add:cN is 0)+2, the parsing error would be raised at the last closing parenthesis rather than at
\fp_gadd:NN the closing parenthesis in the user argument. Thus we evaluate #2 instead of just putting
\fp_gadd:cN parentheses. As an optimization we use \__fp_parse:n rather than \fp_eval:n, which
\fp_sub:NN would convert the result away from the internal representation and back.
\fp_sub:cN 18142 \cs_new_protected:Npn \fp_add:NN { \__fp_add:NNNn \fp_set:Nn + }
\fp_gsub:NN 18143 \cs_new_protected:Npn \fp_gadd:NN { \__fp_add:NNNn \fp_gset:Nn + }
\fp_gsub:cN 18144 \cs_new_protected:Npn \fp_sub:NN { \__fp_add:NNNn \fp_set:Nn - }
\__fp_add:NNNn 18145 \cs_new_protected:Npn \fp_gsub:NN { \__fp_add:NNNn \fp_gset:Nn - }
18146 \cs_new_protected:Npn \__fp_add:NNNn #1#2#3#4
18147 { #1 #3 { #3 #2 \__fp_parse:n {#4} } }
18148 \cs_generate_variant:Nn \fp_add:NN { c }
18149 \cs_generate_variant:Nn \fp_gadd:NN { c }
18150 \cs_generate_variant:Nn \fp_sub:NN { c }
18151 \cs_generate_variant:Nn \fp_gsub:NN { c }

(End definition for \fp_add:NN and others. These functions are documented on page 203.)

```

29.3 Showing values

`\fp_show:N` This shows the result of computing its argument by passing the right data to `\tl_show:n` or `\tl_log:n`.

`\fp_show:c`

`\fp_log:N` 18152 `\cs_new_protected:Npn \fp_show:N { __fp_show:NN \tl_show:n }`

`\fp_log:c` 18153 `\cs_generate_variant:Nn \fp_show:N { c }`

`__fp_show:NN` 18154 `\cs_new_protected:Npn \fp_log:N { __fp_show:NN \tl_log:n }`

18155 `\cs_generate_variant:Nn \fp_log:N { c }`

18156 `\cs_new_protected:Npn __fp_show:NN #1#2`

18157 `{`

18158 `__kernel_chk_defined:NT #2`

18159 `{ \exp_args:Nx #1 { \token_to_str:N #2 = \fp_to_tl:N #2 } }`

18160 `}`

(End definition for `\fp_show:N`, `\fp_log:N`, and `__fp_show:NN`. These functions are documented on page 210.)

`\fp_show:n` Use general tools.

`\fp_log:n` 18161 `\cs_new_protected:Npn \fp_show:n`

18162 `{ \msg_show_eval:Nn \fp_to_tl:n }`

18163 `\cs_new_protected:Npn \fp_log:n`

18164 `{ \msg_log_eval:Nn \fp_to_tl:n }`

(End definition for `\fp_show:n` and `\fp_log:n`. These functions are documented on page 210.)

29.4 Some useful constants and scratch variables

`\c_one_fp` Some constants.

`\c_e_fp` 18165 `\fp_const:Nn \c_e_fp { 2.718 2818 2845 9045 }`

18166 `\fp_const:Nn \c_one_fp { 1 }`

(End definition for `\c_one_fp` and `\c_e_fp`. These variables are documented on page 208.)

`\c_pi_fp` We simply round π to and $\pi/180$ to 16 significant digits.

`\c_one_degree_fp` 18167 `\fp_const:Nn \c_pi_fp { 3.141 5926 5358 9793 }`

18168 `\fp_const:Nn \c_one_degree_fp { 0.0 1745 3292 5199 4330 }`

(End definition for `\c_pi_fp` and `\c_one_degree_fp`. These variables are documented on page 209.)

`\l_tmpa_fp` Scratch variables are simply initialized there.

`\l_tmpb_fp` 18169 `\fp_new:N \l_tmpa_fp`

`\g_tmpa_fp` 18170 `\fp_new:N \l_tmpb_fp`

`\g_tmpb_fp` 18171 `\fp_new:N \g_tmpa_fp`

18172 `\fp_new:N \g_tmpb_fp`

(End definition for `\l_tmpa_fp` and others. These variables are documented on page 209.)

18173 `\</package>`

30 l3fp-logic Implementation

18174 $\langle *package \rangle$

18175 $\langle @@=fp \rangle$

$\backslash_fp_parse_word_max:N$
 $\backslash_fp_parse_word_min:N$

Those functions may receive a variable number of arguments.

18176 $\backslash cs_new:Npn \backslash_fp_parse_word_max:N$
 18177 $\{ \backslash_fp_parse_function:NNN \backslash_fp_minmax_o:Nw 2 \}$
 18178 $\backslash cs_new:Npn \backslash_fp_parse_word_min:N$
 18179 $\{ \backslash_fp_parse_function:NNN \backslash_fp_minmax_o:Nw 0 \}$

(End definition for $\backslash_fp_parse_word_max:N$ and $\backslash_fp_parse_word_min:N$.)

30.1 Syntax of internal functions

- $\backslash_fp_compare_npos:nwnw \{ \langle expo_1 \rangle \} \langle body_1 \rangle ; \{ \langle expo_2 \rangle \} \langle body_2 \rangle ;$
- $\backslash_fp_minmax_o:Nw \langle sign \rangle \langle floating\ point\ array \rangle$
- $\backslash_fp_not_o:w ? \langle floating\ point\ array \rangle$ (with one floating point number only)
- $\backslash_fp_ \& _o:ww \langle floating\ point \rangle \langle floating\ point \rangle$
- $\backslash_fp_ | _o:ww \langle floating\ point \rangle \langle floating\ point \rangle$
- $\backslash_fp_ternary:NwwN, \backslash_fp_ternary_auxi:NwwN, \backslash_fp_ternary_auxii:NwwN$ have to be understood.

30.2 Tests

$\backslash fp_if_exist_p:N$
 $\backslash fp_if_exist_p:c$
 $\backslash fp_if_exist:N \underline{TF}$
 $\backslash fp_if_exist:c \underline{TF}$

Copies of the cs functions defined in l3basics.

18180 $\backslash prg_new_eq_conditional:NNn \backslash fp_if_exist:N \backslash cs_if_exist:N \{ TF , T , F , p \}$
 18181 $\backslash prg_new_eq_conditional:NNn \backslash fp_if_exist:c \backslash cs_if_exist:c \{ TF , T , F , p \}$

(End definition for $\backslash fp_if_exist:N \underline{TF}$. This function is documented on page 205.)

$\backslash fp_if_nan_p:n$
 $\backslash fp_if_nan:n \underline{TF}$

Evaluate and check if the result is a floating point of the same kind as NaN.

18182 $\backslash prg_new_conditional:Npnn \backslash fp_if_nan:n \#1 \{ TF , T , F , p \}$
 18183 $\{$
 18184 $\quad \backslash if:w 3 \backslash exp_last_unbraced:Nf \backslash_fp_kind:w \{ \backslash_fp_parse:n \{ \#1 \} \}$
 18185 $\quad \backslash prg_return_true:$
 18186 $\quad \backslash else:$
 18187 $\quad \backslash prg_return_false:$
 18188 $\quad \backslash fi:$
 18189 $\}$

(End definition for $\backslash fp_if_nan:n \underline{TF}$. This function is documented on page 266.)

30.3 Comparison

`\fp_compare_p:n` Within floating point expressions, comparison operators are treated as operations, so we evaluate #1, then compare with ± 0 . Tuples are true.

`\fp_compare:nTF`

`__fp_compare_return:w`

```

18190 \prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }
18191 {
18192     \exp_after:wN \__fp_compare_return:w
18193     \exp:w \exp_end_continue_f:w \__fp_parse:n {#1}
18194 }
18195 \cs_new:Npn \__fp_compare_return:w #1#2#3;
18196 {
18197     \if_charcode:w 0
18198         \__fp_if_type_fp:NTwFw
18199         #1 { \__fp_use_i_delimit_by_s_stop:nw #3 \s__fp_stop }
18200         \s__fp 1 \s__fp_stop
18201     \prg_return_false:
18202 \else:
18203     \prg_return_true:
18204 \fi:
18205 }
```

(End definition for `\fp_compare:nTF` and `__fp_compare_return:w`. This function is documented on page 206.)

`\fp_compare_p:nNn`

`\fp_compare:nNnTF`

`__fp_compare_aux:wn`

Evaluate #1 and #3, using an auxiliary to expand both, and feed the two floating point numbers swapped to `__fp_compare_back_any:ww`, defined below. Compare the result with '`#2-'`', which is -1 for $<$, 0 for $=$, 1 for $>$ and 2 for $?$.

```

18206 \prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
18207 {
18208     \if_int_compare:w
18209         \exp_after:wN \__fp_compare_aux:wn
18210         \exp:w \exp_end_continue_f:w \__fp_parse:n {#1} {#3}
18211         = \__fp_int_eval:w '#2 - '=' \__fp_int_eval_end:
18212     \prg_return_true:
18213 \else:
18214     \prg_return_false:
18215 \fi:
18216 }
18217 \cs_new:Npn \__fp_compare_aux:wn #1; #2
18218 {
18219     \exp_after:wN \__fp_compare_back_any:ww
18220     \exp:w \exp_end_continue_f:w \__fp_parse:n {#2} #1;
18221 }
```

(End definition for `\fp_compare:nNnTF` and `__fp_compare_aux:wn`. This function is documented on page 205.)

`__fp_compare_back_any:ww`

`__fp_compare_back:ww`

`__fp_compare_nan:w`

`__fp_compare_back_any:ww` $\langle y \rangle$; $\langle x \rangle$;

Expands (in the same way as `\int_eval:n`) to -1 if $x < y$, 0 if $x = y$, 1 if $x > y$, and 2 otherwise (denoted as $x?y$). If either operand is `nan`, stop the comparison with `__fp_compare_nan:w` returning 2 . If x is negative, swap the outputs 1 and -1 (i.e., $>$ and $<$); we can henceforth assume that $x \geq 0$. If $y \geq 0$, and they have the same type, either they are normal and we compare them with `__fp_compare_npos:wnnw`, or they

are equal. If $y \geq 0$, but of a different type, the highest type is a larger number. Finally, if $y \leq 0$, then $x > y$, unless both are zero.

```

18222 \cs_new:Npn \__fp_compare_back_any:ww #1#2; #3
18223 {
18224   \__fp_if_type_fp:NTwFw
18225   #1 { \__fp_if_type_fp:NTwFw #3 \use_i:nn \s__fp \use_ii:nn \s__fp_stop }
18226   \s__fp \use_ii:nn \s__fp_stop
18227   \__fp_compare_back:ww
18228   {
18229     \cs:w
18230     __fp
18231     \__fp_type_from_scan:N #1
18232     _compare_back
18233     \__fp_type_from_scan:N #3
18234     :ww
18235     \cs_end:
18236   }
18237   #1#2 ; #3
18238 }
18239 \cs_new:Npn \__fp_compare_back:ww
18240   \s__fp \__fp_chk:w #1 #2 #3;
18241   \s__fp \__fp_chk:w #4 #5 #6;
18242 {
18243   \int_value:w
18244   \if_meaning:w 3 #1 \exp_after:wN \__fp_compare_nan:w \fi:
18245   \if_meaning:w 3 #4 \exp_after:wN \__fp_compare_nan:w \fi:
18246   \if_meaning:w 2 #5 - \fi:
18247   \if_meaning:w #2 #5
18248   \if_meaning:w #1 #4
18249   \if_meaning:w 1 #1
18250   \__fp_compare_npos:nwnw #6; #3;
18251   \else:
18252     0
18253   \fi:
18254   \else:
18255     \if_int_compare:w #4 < #1 - \fi: 1
18256   \fi:
18257   \else:
18258     \if_int_compare:w #1#4 = 0 \exp_stop_f:
18259     0
18260   \else:
18261     1
18262   \fi:
18263   \fi:
18264   \exp_stop_f:
18265 }
18266 \cs_new:Npn \__fp_compare_nan:w #1 \fi: \exp_stop_f: { 2 \exp_stop_f: }
```

(End definition for __fp_compare_back_any:ww, __fp_compare_back:ww, and __fp_compare_nan:w.)

__fp_compare_back_tuple:ww Tuple and floating point numbers are not comparable so return 2 in mixed cases or
 __fp_tuple_compare_back:ww when tuples have a different number of items. Otherwise compare pairs of items with
 __fp_tuple_compare_back_tuple:ww __fp_compare_back_any:ww and if any don't match return 2 (as \int_value:w 02
 __fp_tuple_compare_back_loop:w \exp_stop_f:).

```

18267 \cs_new:Npn \__fp_compare_back_tuple:ww #1; #2; { 2 }
18268 \cs_new:Npn \__fp_tuple_compare_back:ww #1; #2; { 2 }
18269 \cs_new:Npn \__fp_tuple_compare_back_tuple:ww
18270   \s__fp_tuple \__fp_tuple_chk:w #1;
18271   \s__fp_tuple \__fp_tuple_chk:w #2;
18272   {
18273     \int_compare:nNnTF { \__fp_array_count:n {#1} } =
18274       { \__fp_array_count:n {#2} }
18275     {
18276       \int_value:w 0
18277       \__fp_tuple_compare_back_loop:w
18278         #1 { \s__fp \prg_break: } ; @
18279         #2 { \s__fp \prg_break: } ;
18280       \prg_break_point:
18281       \exp_stop_f:
18282     }
18283     { 2 }
18284   }
18285 \cs_new:Npn \__fp_tuple_compare_back_loop:w #1#2 ; #3 @ #4#5 ;
18286 {
18287   \use_none:n #1
18288   \use_none:n #4
18289   \if_int_compare:w
18290     \__fp_compare_back_any:ww #1 #2 ; #4 #5 ; = 0 \exp_stop_f:
18291   \else:
18292     2 \exp_after:wN \prg_break:
18293   \fi:
18294   \__fp_tuple_compare_back_loop:w #3 @
18295 }

```

(End definition for __fp_compare_back_tuple:ww and others.)

__fp_compare_npos:nwnw
 __fp_compare_significand:nnnnnnnn

__fp_compare_npos:nwnw {<expo₁>} <body₁> ; {<expo₂>} <body₂> ;
 Within an \int_value:w ... \exp_stop_f: construction, this expands to 0 if the two numbers are equal, -1 if the first is smaller, and 1 if the first is bigger. First compare the exponents: the larger one denotes the larger number. If they are equal, we must compare significands. If both the first 8 digits and the next 8 digits coincide, the numbers are equal. If only the first 8 digits coincide, the next 8 decide. Otherwise, the first 8 digits are compared.

```

18296 \cs_new:Npn \__fp_compare_npos:nwnw #1#2; #3#4;
18297 {
18298   \if_int_compare:w #1 = #3 \exp_stop_f:
18299     \__fp_compare_significand:nnnnnnnn #2 #4
18300   \else:
18301     \if_int_compare:w #1 < #3 - \fi: 1
18302   \fi:
18303 }
18304 \cs_new:Npn \__fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
18305 {
18306   \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
18307   \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
18308     0
18309   \else:
18310     \if_int_compare:w #3#4 < #7#8 - \fi: 1

```

```

18311     \fi:
18312 \else:
18313     \if_int_compare:w #1#2 < #5#6 - \fi: 1
18314 \fi:
18315 }

```

(End definition for `_fp_compare_npos:nwnw` and `_fp_compare_significand:nnnnnnnn`.)

30.4 Floating point expression loops

`\fp_do_until:nn` These are quite easy given the above functions. The `do_until` and `do_while` versions execute the body, then test. The `until_do` and `while_do` do it the other way round.

```

18316 \cs_new:Npn \fp_do_until:nn #1#2
18317 {
18318     #2
18319     \fp_compare:nF {#1}
18320     { \fp_do_until:nn {#1} {#2} }
18321 }
18322 \cs_new:Npn \fp_do_while:nn #1#2
18323 {
18324     #2
18325     \fp_compare:nT {#1}
18326     { \fp_do_while:nn {#1} {#2} }
18327 }
18328 \cs_new:Npn \fp_until_do:nn #1#2
18329 {
18330     \fp_compare:nF {#1}
18331     {
18332         #2
18333         \fp_until_do:nn {#1} {#2}
18334     }
18335 }
18336 \cs_new:Npn \fp_while_do:nn #1#2
18337 {
18338     \fp_compare:nT {#1}
18339     {
18340         #2
18341         \fp_while_do:nn {#1} {#2}
18342     }
18343 }

```

(End definition for `\fp_do_until:nn` and others. These functions are documented on page 207.)

`\fp_do_until:nNnn` As above but not using the `nNn` syntax.

```

18344 \cs_new:Npn \fp_do_until:nNnn #1#2#3#4
18345 {
18346     #4
18347     \fp_compare:nNnF {#1} #2 {#3}
18348     { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
18349 }
18350 \cs_new:Npn \fp_do_while:nNnn #1#2#3#4
18351 {
18352     #4
18353     \fp_compare:nNnT {#1} #2 {#3}

```

```

18354     { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
18355   }
18356 \cs_new:Npn \fp_until_do:nNnn #1#2#3#4
18357 {
18358   \fp_compare:nNnF {#1} #2 {#3}
18359   {
18360     #4
18361     \fp_until_do:nNnn {#1} #2 {#3} {#4}
18362   }
18363 }
18364 \cs_new:Npn \fp_while_do:nNnn #1#2#3#4
18365 {
18366   \fp_compare:nNnT {#1} #2 {#3}
18367   {
18368     #4
18369     \fp_while_do:nNnn {#1} #2 {#3} {#4}
18370   }
18371 }

```

(End definition for `\fp_do_until:nNnn` and others. These functions are documented on page 206.)

\fp_step_function:nnnN

\fp_step_function:nnnc

`__fp_step:wwwN`

`__fp_step_fp:wwwN`

`__fp_step:NnnnnN`

`__fp_step:NfnnnN`

The approach here is somewhat similar to `\int_step_function:nnnN`. There are two subtleties: we use the internal parser `__fp_parse:n` to avoid converting back and forth from the internal representation; and (due to rounding) even a non-zero step does not guarantee that the loop counter increases.

```

18372 \cs_new:Npn \fp_step_function:nnnN #1#2#3
18373 {
18374   \exp_after:wN \__fp_step:wwwN
18375   \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#1}
18376   \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#2}
18377   \exp:w \exp_end_continue_f:w \__fp_parse:n {#3}
18378 }
18379 \cs_generate_variant:Nn \fp_step_function:nnnN { nnnc }
18380 % \end{macrocode}
18381 % Only floating point numbers (not tuples) are allowed arguments.
18382 % Only \enquote{normal} floating points (not $\pm 0$,
18383 % $\pm\texttt{inf}$, $\texttt{nan}$) can be used as step; if positive,
18384 % call \cs{__fp_step:NnnnnN} with argument |>| otherwise~|<|. This
18385 % function has one more argument than its integer counterpart, namely
18386 % the previous value, to catch the case where the loop has made no
18387 % progress. Conversion to decimal is done just before calling the
18388 % user's function.
18389 % \begin{macrocode}
18390 \cs_new:Npn \__fp_step:wwwN #1#2; #3#4; #5#6; #7
18391 {
18392   \__fp_if_type_fp:NTwFw #1 { } \s__fp \prg_break: \s__fp_stop
18393   \__fp_if_type_fp:NTwFw #3 { } \s__fp \prg_break: \s__fp_stop
18394   \__fp_if_type_fp:NTwFw #5 { } \s__fp \prg_break: \s__fp_stop
18395   \use_i:nnnn { \__fp_step_fp:wwwN #1#2; #3#4; #5#6; #7 }
18396   \prg_break_point:
18397   \use:n
18398   {
18399     \__fp_error:nfff { fp-step-tuple } { \fp_to_tl:n { #1#2 ; } }
18400     { \fp_to_tl:n { #3#4 ; } } { \fp_to_tl:n { #5#6 ; } }

```

```

18401     }
18402   }
18403   \cs_new:Npn \__fp_step_fp:wwwN #1 ; \s__fp \__fp_chk:w #2#3#4 ; #5; #6
18404   {
18405     \token_if_eq_meaning:NNTF #2 1
18406     {
18407       \token_if_eq_meaning:NNTF #3 0
18408       { \__fp_step:NnnnnN > }
18409       { \__fp_step:NnnnnN < }
18410     }
18411     {
18412       \token_if_eq_meaning:NNTF #2 0
18413       {
18414         \__kernel_msg_expandable_error:nnn { kernel }
18415         { zero-step } {#6}
18416       }
18417       {
18418         \__fp_error:nnfn { fp-bad-step } { }
18419         { \fp_to_tl:n { \s__fp \__fp_chk:w #2#3#4 ; } } {#6}
18420       }
18421       \use_none:nnnnn
18422     }
18423     { #1 ; } { \c_nan_fp } { \s__fp \__fp_chk:w #2#3#4 ; } { #5 ; } #6
18424   }
18425   \cs_new:Npn \__fp_step:NnnnnN #1#2#3#4#5#6
18426   {
18427     \fp_compare:nNnTF {#2} = {#3}
18428     {
18429       \__fp_error:nffn { fp-tiny-step }
18430       { \fp_to_tl:n {#3} } { \fp_to_tl:n {#4} } {#6}
18431     }
18432     {
18433       \fp_compare:nNnF {#2} #1 {#5}
18434       {
18435         \exp_args:Nf #6 { \__fp_to_decimal_dispatch:w #2 }
18436         \__fp_step:NfnnnnN
18437         #1 { \__fp_parse:n { #2 + #4 } } {#2} {#4} {#5} #6
18438       }
18439     }
18440   }
18441   \cs_generate_variant:Nn \__fp_step:NnnnnN { Nf }

```

(End definition for \fp_step_function:nnnN and others. This function is documented on page 208.)

\fp_step_inline:nnnn As for \int_step_inline:nnnn, create a global function and apply it, following up with
\fp_step_variable:nnnNn a break point.

```

\__fp_step:NNnnnnn
18442 \cs_new_protected:Npn \fp_step_inline:nnnn
18443 {
18444   \int_gincr:N \g__kernel_prg_map_int
18445   \exp_args:NNc \__fp_step:NNnnnn
18446   \cs_gset_protected:Npn
18447   { __fp_map_ \int_use:N \g__kernel_prg_map_int :w }
18448 }
18449 \cs_new_protected:Npn \fp_step_variable:nnnNn #1#2#3#4#5

```

```

18450 {
18451   \int_gincr:N \g__kernel_prg_map_int
18452   \exp_args:Nnc \__fp_step:NNnnnn
18453   \cs_gset_protected:Npx
18454   { \__fp_map_ \int_use:N \g__kernel_prg_map_int :w }
18455   {#1} {#2} {#3}
18456   {
18457     \tl_set:Nn \exp_not:N #4 {##1}
18458     \exp_not:n {#5}
18459   }
18460 }
18461 \cs_new_protected:Npn \__fp_step:NNnnnn #1#2#3#4#5#6
18462 {
18463   #1 #2 ##1 {#6}
18464   \fp_step_function:nnnN {#3} {#4} {#5} #2
18465   \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
18466 }

```

(End definition for `\fp_step_inline:nnnn`, `\fp_step_variable:nnnN`, and `__fp_step:NNnnnn`. These functions are documented on page 208.)

```

18467 \__kernel_msg_new:nnn { kernel } { fp-step-tuple }
18468 { Tuple~argument~in~fp_step...~{#1}{#2}{#3}. }
18469 \__kernel_msg_new:nnn { kernel } { fp-bad-step }
18470 { Invalid~step~size~#2~in~step~function~#3. }
18471 \__kernel_msg_new:nnn { kernel } { fp-tiny-step }
18472 { Tiny~step~size~(#{1}+#{2}=#{1})~in~step~function~#3. }

```

30.5 Extrema

```

\__fp_minmax_o:Nw
\__fp_minmax_aux_o:Nw

```

First check all operands are floating point numbers. The argument #1 is 2 to find the maximum of an array #2 of floating point numbers, and 0 to find the minimum. We read numbers sequentially, keeping track of the largest (smallest) number found so far. If numbers are equal (for instance ± 0), the first is kept. We append $-\infty$ (∞), for the case of an empty array. Since no number is smaller (larger) than that, this additional item only affects the maximum (minimum) in the case of `max()` and `min()` with no argument. The weird fp-like trailing marker breaks the loop correctly: see the precise definition of `__fp_minmax_loop:Nww`.

```

18473 \cs_new:Npn \__fp_minmax_o:Nw #1
18474 {
18475   \__fp_parse_function_all_fp_o:fnw
18476   { \token_if_eq_meaning:NNTF 0 #1 { min } { max } }
18477   { \__fp_minmax_aux_o:Nw #1 }
18478 }
18479 \cs_new:Npn \__fp_minmax_aux_o:Nw #1#2 @
18480 {
18481   \if_meaning:w 0 #1
18482     \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN +
18483   \else:
18484     \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN -
18485   \fi:
18486   #2
18487   \s__fp \__fp_chk:w 2 #1 \s__fp_exact ;
18488   \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;

```

```
18489 }
```

(End definition for `_fp_minmax_o:Nw` and `_fp_minmax_aux_o:Nw`.)

`_fp_minmax_loop:Nww`

The first argument is `-` or `+` to denote the case where the currently largest (smallest) number found (first floating point argument) should be replaced by the new number (second floating point argument). If the new number is `nan`, keep that as the extremum, unless that extremum is already a `nan`. Otherwise, compare the two numbers. If the new number is larger (in the case of `max`) or smaller (in the case of `min`), the test yields `true`, and we keep the second number as a new maximum; otherwise we keep the first number. Then loop.

```
18490 \cs_new:Npn \_fp_minmax_loop:Nww
18491   #1 \s__fp \_fp_chk:w #2#3; \s__fp \_fp_chk:w #4#5;
18492   {
18493     \if_meaning:w 3 #4
18494     \if_meaning:w 3 #2
18495       \_fp_minmax_auxi:ww
18496     \else:
18497       \_fp_minmax_auxii:ww
18498     \fi:
18499   \else:
18500     \if_int_compare:w
18501       \_fp_compare_back:ww
18502       \s__fp \_fp_chk:w #4#5;
18503       \s__fp \_fp_chk:w #2#3;
18504       = #1 1 \exp_stop_f:
18505       \_fp_minmax_auxii:ww
18506     \else:
18507       \_fp_minmax_auxi:ww
18508     \fi:
18509   \fi:
18510   \_fp_minmax_loop:Nww #1
18511   \s__fp \_fp_chk:w #2#3;
18512   \s__fp \_fp_chk:w #4#5;
18513 }
```

(End definition for `_fp_minmax_loop:Nww`.)

`_fp_minmax_auxi:ww`

Keep the first/second number, and remove the other.

`_fp_minmax_auxii:ww`

```
18514 \cs_new:Npn \_fp_minmax_auxi:ww #1 \fi: \fi: #2 \s__fp #3 ; \s__fp #4;
18515   { \fi: \fi: #2 \s__fp #3 ; }
18516 \cs_new:Npn \_fp_minmax_auxii:ww #1 \fi: \fi: #2 \s__fp #3 ;
18517   { \fi: \fi: #2 }
```

(End definition for `_fp_minmax_auxi:ww` and `_fp_minmax_auxii:ww`.)

`_fp_minmax_break_o:w`

This function is called from within an `\if_meaning:w` test. Skip to the end of the tests, close the current test with `\fi:`, clean up, and return the appropriate number with one post-expansion.

```
18518 \cs_new:Npn \_fp_minmax_break_o:w #1 \fi: \fi: #2 \s__fp #3; #4;
18519   { \fi: \_fp_exp_after_o:w \s__fp #3; }
```

(End definition for `_fp_minmax_break_o:w`.)

30.6 Boolean operations

`__fp_not_o:w` Return true or false, with two expansions, one to exit the conditional, and one to please `l3fp-parse`. The first argument is provided by `l3fp-parse` and is ignored.

```

18520 \cs_new:Npn \__fp_not_o:w #1 \s__fp \__fp_chk:w #2#3; @
18521 {
18522   \if_meaning:w 0 #2
18523     \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
18524   \else:
18525     \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
18526   \fi:
18527 }
18528 \cs_new:Npn \__fp_tuple_not_o:w #1 @ { \exp_after:wN \c_zero_fp }
```

(End definition for `__fp_not_o:w` and `__fp_tuple_not_o:w`.)

`__fp_&_o:ww` For `and`, if the first number is zero, return it (with the same sign). Otherwise, return the second one. For `or`, the logic is reversed: if the first number is non-zero, return it, otherwise return the second number: we achieve that by hi-jacking `__fp_&_o:ww`, inserting an extra argument, `\else:`, before `\s__fp`. In all cases, expand after the floating point number.

```

18529 \group_begin:
18530 \char_set_catcode_letter:N &
18531 \char_set_catcode_letter:N |
18532 \cs_new:Npn \__fp_&_o:ww #1 \s__fp \__fp_chk:w #2#3;
18533 {
18534   \if_meaning:w 0 #2 #1
18535     \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
18536   \fi:
18537   \__fp_exp_after_o:w
18538 }
18539 \cs_new:Npn \__fp_&_tuple_o:ww #1 \s__fp \__fp_chk:w #2#3;
18540 {
18541   \if_meaning:w 0 #2 #1
18542     \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
18543   \fi:
18544   \__fp_exp_after_tuple_o:w
18545 }
18546 \cs_new:Npn \__fp_tuple_&_o:ww #1; { \__fp_exp_after_o:w }
18547 \cs_new:Npn \__fp_tuple_&_tuple_o:ww #1; { \__fp_exp_after_tuple_o:w }
18548 \cs_new:Npn \__fp_|_o:ww { \__fp_&_o:ww \else: }
18549 \cs_new:Npn \__fp_|_tuple_o:ww { \__fp_&_tuple_o:ww \else: }
18550 \cs_new:Npn \__fp_tuple_|_o:ww #1; #2; { \__fp_exp_after_tuple_o:w #1; }
18551 \cs_new:Npn \__fp_tuple_|_tuple_o:ww #1; #2;
18552   { \__fp_exp_after_tuple_o:w #1; }
18553 \group_end:
18554 \cs_new:Npn \__fp_and_return:wNw #1; \fi: #2;
18555   { \fi: \__fp_exp_after_o:w #1; }
```

(End definition for `__fp_&_o:ww` and others.)

30.7 Ternary operator

`_fp_ternary:NwN`
`_fp_ternary_auxi:NwN`
`_fp_ternary_auxii:NwN`

The first function receives the test and the true branch of the `?:` ternary operator. It calls `_fp_ternary_auxii:NwN` if the test branch is a floating point number ± 0 , and otherwise calls `_fp_ternary_auxi:NwN`. These functions select one of their two arguments.

```

18556 \cs_new:Npn \_fp_ternary:NwN #1 #2#3@ #4@ #5
18557 {
18558   \if_meaning:w \_fp_parse_infix_:N #5
18559   \if_charcode:w 0
18560     \_fp_if_type_fp:NTwFw
18561     #2 { \use_i:nn \_fp_use_i_delimit_by_s_stop:nw #3 \s__fp_stop }
18562     \s__fp 1 \s__fp_stop
18563     \exp_after:wN \exp_after:wN \exp_after:wN \_fp_ternary_auxii:NwN
18564   \else:
18565     \exp_after:wN \exp_after:wN \exp_after:wN \_fp_ternary_auxi:NwN
18566   \fi:
18567   \exp_after:wN #1
18568   \exp:w \exp_end_continue_f:w
18569   \_fp_exp_after_array_f:w #4 \s__fp_expr_stop
18570   \exp_after:wN @
18571   \exp:w
18572   \_fp_parse_operand:Nw \c__fp_prec_colon_int
18573   \_fp_parse_expand:w
18574 \else:
18575   \__kernel_msg_expandable_error:nnnn
18576   { kernel } { fp-missing } { : } { ~for~?: }
18577   \exp_after:wN \_fp_parse_continue:NwN
18578   \exp_after:wN #1
18579   \exp:w \exp_end_continue_f:w
18580   \_fp_exp_after_array_f:w #4 \s__fp_expr_stop
18581   \exp_after:wN #5
18582   \exp_after:wN #1
18583 \fi:
18584 }
18585 \cs_new:Npn \_fp_ternary_auxi:NwN #1#2@#3@#4
18586 {
18587   \exp_after:wN \_fp_parse_continue:NwN
18588   \exp_after:wN #1
18589   \exp:w \exp_end_continue_f:w
18590   \_fp_exp_after_array_f:w #2 \s__fp_expr_stop
18591   #4 #1
18592 }
18593 \cs_new:Npn \_fp_ternary_auxii:NwN #1#2@#3@#4
18594 {
18595   \exp_after:wN \_fp_parse_continue:NwN
18596   \exp_after:wN #1
18597   \exp:w \exp_end_continue_f:w
18598   \_fp_exp_after_array_f:w #3 \s__fp_expr_stop
18599   #4 #1
18600 }

```

(End definition for `_fp_ternary:NwN`, `_fp_ternary_auxi:NwN`, and `_fp_ternary_auxii:NwN`.)

18601 `\</package>`

31 l3fp-basics Implementation

```
18602 ⟨*package⟩
18603 ⟨@@=fp⟩
```

The `l3fp-basics` module implements addition, subtraction, multiplication, and division of two floating points, and the absolute value and sign-changing operations on one floating point. All operations implemented in this module yield the outcome of rounding the infinitely precise result of the operation to the nearest floating point.

Some algorithms used below end up being quite similar to some described in “What Every Computer Scientist Should Know About Floating Point Arithmetic”, by David Goldberg, which can be found at <http://cr.yp.to/2005-590/goldberg.pdf>.

Unary functions.

```

__fp_parse_word_abs:N
__fp_parse_word_logb:N
__fp_parse_word_sign:N
__fp_parse_word_sqrt:N
18604 \cs_new:Npn \__fp_parse_word_abs:N
18605   { \__fp_parse_unary_function:NNN \__fp_set_sign_o:w 0 }
18606 \cs_new:Npn \__fp_parse_word_logb:N
18607   { \__fp_parse_unary_function:NNN \__fp_logb_o:w ? }
18608 \cs_new:Npn \__fp_parse_word_sign:N
18609   { \__fp_parse_unary_function:NNN \__fp_sign_o:w ? }
18610 \cs_new:Npn \__fp_parse_word_sqrt:N
18611   { \__fp_parse_unary_function:NNN \__fp_sqrt_o:w ? }
```

(End definition for `__fp_parse_word_abs:N` and others.)

31.1 Addition and subtraction

We define here two functions, `__fp_-_o:ww` and `__fp+_o:ww`, which perform the subtraction and addition of their two floating point operands, and expand the tokens following the result once.

A more obscure function, `__fp_add_big_i_o:wNww`, is used in `l3fp-expo`.

The logic goes as follows:

- `__fp_-_o:ww` calls `__fp+_o:ww` to do the work, with the sign of the second operand flipped;
- `__fp+_o:ww` dispatches depending on the type of floating point, calling specialized auxiliaries;
- in all cases except summing two normal floating point numbers, we return one or the other operands depending on the signs, or detect an invalid operation in the case of $\infty - \infty$;
- for normal floating point numbers, compare the signs;
- to add two floating point numbers of the same sign or of opposite signs, shift the significand of the smaller one to match the bigger one, perform the addition or subtraction of significands, check for a carry, round, and pack using the `__fp-basics_pack_...` functions.

The trickiest part is to round correctly when adding or subtracting normal floating point numbers.

31.1.1 Sign, exponent, and special numbers

`__fp_-_o:ww` The `__fp+_o:ww` auxiliary has a hook: it takes one argument between the first `\s__fp` and `__fp_chk:w`, which is applied to the sign of the second operand. Positioning the hook there means that `__fp+_o:ww` can still perform the sanity check that it was followed by `\s__fp`.

```
18612 \cs_new:cpx { __fp_-_o:ww } \s__fp
18613 {
18614     \exp_not:c { __fp+_o:ww }
18615     \exp_not:n { \s__fp \__fp_neg_sign:N }
18616 }
```

(End definition for `__fp_-_o:ww`.)

`__fp+_o:ww` This function is either called directly with an empty #1 to compute an addition, or it is called by `__fp_-_o:ww` with `__fp_neg_sign:N` as #1 to compute a subtraction, in which case the second operand's sign should be changed. If the *<types>* #2 and #4 are the same, dispatch to case #2 (0, 1, 2, or 3), where we call specialized functions: thanks to `\int_value:w`, those receive the tweaked *<sign₂>* (expansion of #1#5) as an argument. If the *<types>* are distinct, the result is simply the floating point number with the highest *<type>*. Since case 3 (used for two nan) also picks the first operand, we can also use it when *<type₁>* is greater than *<type₂>*. Also note that we don't need to worry about *<sign₂>* in that case since the second operand is discarded.

```
18617 \cs_new:cpn { __fp+_o:ww }
18618     \s__fp #1 \__fp_chk:w #2 #3 ; \s__fp \__fp_chk:w #4 #5
18619 {
18620     \if_case:w
18621         \if_meaning:w #2 #4
18622             #2
18623         \else:
18624             \if_int_compare:w #2 > #4 \exp_stop_f:
18625                 3
18626             \else:
18627                 4
18628             \fi:
18629         \fi:
18630     \exp_stop_f:
18631         \exp_after:wN \__fp_add_zeros_o:Nww \int_value:w
18632     \or:     \exp_after:wN \__fp_add_normal_o:Nww \int_value:w
18633     \or:     \exp_after:wN \__fp_add_inf_o:Nww \int_value:w
18634     \or:     \__fp_case_return_i_o:ww
18635     \else: \exp_after:wN \__fp_add_return_ii_o:Nww \int_value:w
18636     \fi:
18637     #1 #5
18638     \s__fp \__fp_chk:w #2 #3 ;
18639     \s__fp \__fp_chk:w #4 #5
18640 }
```

(End definition for `__fp+_o:ww`.)

`__fp_add_return_ii_o:Nww` Ignore the first operand, and return the second, but using the sign #1 rather than #4. As usual, expand after the floating point.

```
18641 \cs_new:Npn \__fp_add_return_ii_o:Nww #1 #2 ; \s__fp \__fp_chk:w #3 #4
18642     { \__fp_exp_after_o:w \s__fp \__fp_chk:w #3 #1 }
```

(End definition for _fp_add_return_ii_o:Nww.)

_fp_add_zeros_o:Nww Adding two zeros yields \c_zero_fp, except if both zeros were -0 .

```

18643 \cs_new:Npn \_fp_add_zeros_o:Nww #1 \s__fp \_fp_chk:w 0 #2
18644 {
18645   \if_int_compare:w #2 #1 = 20 \exp_stop_f:
18646     \exp_after:wN \_fp_add_return_ii_o:Nww
18647   \else:
18648     \_fp_case_return_i_o:ww
18649   \fi:
18650   #1
18651   \s__fp \_fp_chk:w 0 #2
18652 }

```

(End definition for _fp_add_zeros_o:Nww.)

_fp_add_inf_o:Nww If both infinities have the same sign, just return that infinity, otherwise, it is an invalid operation. We find out if that invalid operation is an addition or a subtraction by testing whether the tweaked $\langle sign_2 \rangle$ (#1) and the $\langle sign_2 \rangle$ (#4) are identical.

```

18653 \cs_new:Npn \_fp_add_inf_o:Nww
18654   #1 \s__fp \_fp_chk:w 2 #2 #3; \s__fp \_fp_chk:w 2 #4
18655 {
18656   \if_meaning:w #1 #2
18657     \_fp_case_return_i_o:ww
18658   \else:
18659     \_fp_case_use:nw
18660     {
18661       \exp_last_unbraced:Nf \_fp_invalid_operation_o:Nww
18662       { \token_if_eq_meaning:NNTF #1 #4 + - }
18663     }
18664   \fi:
18665   \s__fp \_fp_chk:w 2 #2 #3;
18666   \s__fp \_fp_chk:w 2 #4
18667 }

```

(End definition for _fp_add_inf_o:Nww.)

_fp_add_normal_o:Nww _fp_add_normal_o:Nww $\langle sign_2 \rangle$ \s__fp _fp_chk:w 1 $\langle sign_1 \rangle$ $\langle exp_1 \rangle$ $\langle body_1 \rangle$; \s__fp _fp_chk:w 1 $\langle initial\ sign_2 \rangle$ $\langle exp_2 \rangle$ $\langle body_2 \rangle$;

We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.

```

18668 \cs_new:Npn \_fp_add_normal_o:Nww #1 \s__fp \_fp_chk:w 1 #2
18669 {
18670   \if_meaning:w #1#2
18671     \exp_after:wN \_fp_add_npos_o:NnwNnw
18672   \else:
18673     \exp_after:wN \_fp_sub_npos_o:NnwNnw
18674   \fi:
18675   #2
18676 }

```

(End definition for _fp_add_normal_o:Nww.)

31.1.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

`__fp_add_npos_o:NnwNnw` `__fp_add_npos_o:NnwNnw <sign1> <exp1> <body1> ; \s__fp __fp_chk:w 1
<initial sign2> <exp2> <body2> ;`

Since we are doing an addition, the final sign is $\langle sign_1 \rangle$. Start an `__fp_int_eval:w`, responsible for computing the exponent: the result, and the $\langle final\ sign \rangle$ are then given to `__fp_sanitize:Nw` which checks for overflow. The exponent is computed as the largest exponent #2 or #5, incremented if there is a carry. To add the significands, we decimate the smaller number by the difference between the exponents. This is done by `__fp_add_big_i:wNnw` or `__fp_add_big_ii:wNnw`. We need to bring the final sign with us in the midst of the calculation to round properly at the end.

```
18677 \cs_new:Npn \__fp_add_npos_o:NnwNnw #1#2#3 ; \s__fp \__fp_chk:w 1 #4 #5
18678 {
18679   \exp_after:wN \__fp_sanitize:Nw
18680   \exp_after:wN #1
18681   \int_value:w \__fp_int_eval:w
18682   \if_int_compare:w #2 > #5 \exp_stop_f:
18683     #2
18684   \exp_after:wN \__fp_add_big_i_o:wNnw \int_value:w -
18685   \else:
18686     #5
18687   \exp_after:wN \__fp_add_big_ii_o:wNnw \int_value:w
18688   \fi:
18689   \__fp_int_eval:w #5 - #2 ; #1 #3;
18690 }
```

(End definition for `__fp_add_npos_o:NnwNnw`.)

`__fp_add_big_i_o:wNnw` `__fp_add_big_i_o:wNnw <shift> ; <final sign> <body1> ; <body2> ;`
`__fp_add_big_ii_o:wNnw` Used in l3fp-expo. Shift the significand of the small number, then add with `__fp_add_significand_o:NnnwnnnnN`.

```
18691 \cs_new:Npn \__fp_add_big_i_o:wNnw #1; #2 #3; #4;
18692 {
18693   \__fp_decimate:Nnnnnn {#1}
18694   \__fp_add_significand_o:NnnwnnnnN
18695   #4
18696   #3
18697   #2
18698 }
18699 \cs_new:Npn \__fp_add_big_ii_o:wNnw #1; #2 #3; #4;
18700 {
18701   \__fp_decimate:Nnnnnn {#1}
18702   \__fp_add_significand_o:NnnwnnnnN
18703   #3
18704   #4
18705   #2
18706 }
```

(End definition for `__fp_add_big_i_o:wNnw` and `__fp_add_big_ii_o:wNnw`.)

```

\__fp_add_significand_o:NnnwnnnnN \__fp_add_significand_o:NnnwnnnnN <rounding digit> {\langle Y'_1 \rangle} {\langle Y'_2 \rangle}
\__fp_add_significand_pack:NNNNNNN <extra-digits> ; {\langle X_1 \rangle} {\langle X_2 \rangle} {\langle X_3 \rangle} {\langle X_4 \rangle} <final sign>
\__fp_add_significand_test_o:N

```

To round properly, we must know at which digit the rounding should occur. This requires to know whether the addition produces an overall carry or not. Thus, we do the computation now and check for a carry, then go back and do the rounding. The rounding may cause a carry in very rare cases such as $0.99\dots95 \rightarrow 1.00\dots0$, but this situation always give an exact power of 10, for which it is easy to correct the result at the end.

```

18707 \cs_new:Npn \__fp_add_significand_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
18708 {
18709   \exp_after:wN \__fp_add_significand_test_o:N
18710   \int_value:w \__fp_int_eval:w 1#5#6 + #2
18711   \exp_after:wN \__fp_add_significand_pack:NNNNNNN
18712   \int_value:w \__fp_int_eval:w 1#7#8 + #3 ; #1
18713 }
18714 \cs_new:Npn \__fp_add_significand_pack:NNNNNNN #1 #2#3#4#5#6#7
18715 {
18716   \if_meaning:w 2 #1
18717     + 1
18718   \fi:
18719   ; #2 #3 #4 #5 #6 #7 ;
18720 }
18721 \cs_new:Npn \__fp_add_significand_test_o:N #1
18722 {
18723   \if_meaning:w 2 #1
18724     \exp_after:wN \__fp_add_significand_carry_o:wwwNN
18725   \else:
18726     \exp_after:wN \__fp_add_significand_no_carry_o:wwwNN
18727   \fi:
18728 }

```

(End definition for `__fp_add_significand_o:NnnwnnnnN`, `__fp_add_significand_pack:NNNNNNN`, and `__fp_add_significand_test_o:N`.)

```

\__fp_add_significand_no_carry_o:wwwNN \__fp_add_significand_no_carry_o:wwwNN <8d> ; <6d> ; <2d> ; <rounding
digit> <sign>

```

If there's no carry, grab all the digits again and round. The packing function `__fp_basics_pack_high:NNNNNw` takes care of the case where rounding brings a carry.

```

18729 \cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
18730   #1; #2; #3#4 ; #5#6
18731 {
18732   \exp_after:wN \__fp_basics_pack_high:NNNNNw
18733   \int_value:w \__fp_int_eval:w 1 #1
18734   \exp_after:wN \__fp_basics_pack_low:NNNNNw
18735   \int_value:w \__fp_int_eval:w 1 #2 #3#4
18736   + \__fp_round:NNN #6 #4 #5
18737   \exp_after:wN ;
18738 }

```

(End definition for `__fp_add_significand_no_carry_o:wwwNN`.)

```

\__fp_add_significand_carry_o:wwwNN \__fp_add_significand_carry_o:wwwNN <8d> ; <6d> ; <2d> ; <rounding
digit> <sign>

```

The case where there is a carry is very similar. Rounding can even raise the first digit from 1 to 2, but we don't care.

```

18739 \cs_new:Npn \__fp_add_significand_carry_o:wwwNN
18740   #1; #2; #3#4; #5#6
18741   {
18742     + 1
18743     \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw
18744     \int_value:w \__fp_int_eval:w 1 1 #1
18745     \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
18746     \int_value:w \__fp_int_eval:w 1 #2#3 +
18747     \exp_after:wN \__fp_round:NNN
18748     \exp_after:wN #6
18749     \exp_after:wN #3
18750     \int_value:w \__fp_round_digit:Nw #4 #5 ;
18751     \exp_after:wN ;
18752   }

```

(End definition for __fp_add_significand_carry_o:wwwNN.)

31.1.3 Absolute subtraction

```

\__fp_sub_npos_o:NnwNnw \__fp_sub_npos_o:NnwNnw <sign1> <exp1> <body1> ; \s__fp \__fp_chk:w 1
\__fp_sub_eq_o:Nwnnw <initial sign2> <exp2> <body2> ;
\__fp_sub_npos_ii_o:Nwnnw

```

Rounding properly in some modes requires to know what the sign of the result will be. Thus, we start by comparing the exponents and significands. If the numbers coincide, return zero. If the second number is larger, swap the numbers and call __fp_sub_npos_i_o:Nwnnw with the opposite of $\langle sign_1 \rangle$.

```

18753 \cs_new:Npn \__fp_sub_npos_o:NnwNnw #1#2#3; \s__fp \__fp_chk:w 1 #4#5#6;
18754   {
18755     \if_case:w \__fp_compare_npos:nwnw {#2} #3; {#5} #6; \exp_stop_f:
18756     \exp_after:wN \__fp_sub_eq_o:Nwnnw
18757     \or:
18758     \exp_after:wN \__fp_sub_npos_i_o:Nwnnw
18759     \else:
18760     \exp_after:wN \__fp_sub_npos_ii_o:Nwnnw
18761     \fi:
18762     #1 {#2} #3; {#5} #6;
18763   }
18764 \cs_new:Npn \__fp_sub_eq_o:Nwnnw #1#2; #3; { \exp_after:wN \c_zero_fp }
18765 \cs_new:Npn \__fp_sub_npos_ii_o:Nwnnw #1 #2; #3;
18766   {
18767     \exp_after:wN \__fp_sub_npos_i_o:Nwnnw
18768     \int_value:w \__fp_neg_sign:N #1
18769     #3; #2;
18770   }

```

(End definition for __fp_sub_npos_o:NnwNnw, __fp_sub_eq_o:Nwnnw, and __fp_sub_npos_ii_o:Nwnnw.)

```

\__fp_sub_npos_i_o:Nwnnw

```

After the computation is done, __fp_sanitize:Nw checks for overflow/underflow. It expects the $\langle final\ sign \rangle$ and the $\langle exponent \rangle$ (delimited by ;). Start an integer expression for the exponent, which starts with the exponent of the largest number, and may be decreased if the two numbers are very close. If the two numbers have the same exponent, call the `near` auxiliary. Otherwise, decimate y , then call the `far` auxiliary to evaluate

the difference between the two significands. Note that we decimate by 1 less than one could expect.

```

18771 \cs_new:Npn \__fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
18772 {
18773   \exp_after:wN \__fp_sanitizew
18774   \exp_after:wN #1
18775   \int_value:w \__fp_int_eval:w
18776   #2
18777   \if_int_compare:w #2 = #4 \exp_stop_f:
18778     \exp_after:wN \__fp_sub_back_near_o:nnnnnnnnN
18779   \else:
18780     \exp_after:wN \__fp_decimate:nNnnnn \exp_after:wN
18781     { \int_value:w \__fp_int_eval:w #2 - #4 - 1 \exp_after:wN }
18782     \exp_after:wN \__fp_sub_back_far_o:NnnwnnnnnN
18783   \fi:
18784   #5
18785   #3
18786   #1
18787 }

```

(End definition for __fp_sub_npos_i_o:Nnwnw.)

```

\__fp_sub_back_near_o:nnnnnnnnN    \__fp_sub_back_near_o:nnnnnnnnN {<Y1>} {<Y2>} {<Y3>} {<Y4>} {<X1>}
\__fp_sub_back_near_pack:NNNNNNw   {<X2>} {<X3>} {<X4>} {<final sign>}
\__fp_sub_back_near_after:wNNNNw

```

In this case, the subtraction is exact, so we discard the *<final sign>* #9. The very large shifts of 10^9 and $1.1 \cdot 10^9$ are unnecessary here, but allow the auxiliaries to be reused later. Each integer expression produces a 10 digit result. If the resulting 16 digits start with a 0, then we need to shift the group, padding with trailing zeros.

```

18788 \cs_new:Npn \__fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
18789 {
18790   \exp_after:wN \__fp_sub_back_near_after:wNNNNw
18791   \int_value:w \__fp_int_eval:w 10#5#6 - #1#2 - 11
18792   \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
18793   \int_value:w \__fp_int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
18794 }
18795 \cs_new:Npn \__fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
18796 { + #1#2 ; {#3#4#5#6} {#7} ; }
18797 \cs_new:Npn \__fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 ;
18798 {
18799   \if_meaning:w 0 #1
18800     \exp_after:wN \__fp_sub_back_shift:wnnnn
18801   \fi:
18802   ; {#1#2#3#4} {#5}
18803 }

```

(End definition for __fp_sub_back_near_o:nnnnnnnnN, __fp_sub_back_near_pack:NNNNNNw, and __fp_sub_back_near_after:wNNNNw.)

```

\__fp_sub_back_shift:wnnnn    \__fp_sub_back_shift:wnnnn ; {<Z1>} {<Z2>} {<Z3>} {<Z4>} ;
\__fp_sub_back_shift_ii:ww    This function is called with <Z1> ≤ 999. Act with \number to trim leading zeros from
\__fp_sub_back_shift_iii:NNNNNNw <Z1> <Z2> (we don't do all four blocks at once, since non-zero blocks would then overflow
\__fp_sub_back_shift_iv:nnnnw  TEX's integers). If the first two blocks are zero, the auxiliary receives an empty #1 and

```

trims #2#30 from leading zeros, yielding a total shift between 7 and 16 to the exponent. Otherwise we get the shift from #1 alone, yielding a result between 1 and 6. Once the

exponent is taken care of, trim leading zeros from #1#2#3 (when #1 is empty, the space before #2#3 is ignored), get four blocks of 4 digits and finally clean up. Trailing zeros are added so that digits can be grabbed safely.

```

18804 \cs_new:Npn \__fp_sub_back_shift:wnnnn ; #1#2
18805 {
18806   \exp_after:wN \__fp_sub_back_shift_ii:ww
18807   \int_value:w #1 #2 0 ;
18808 }
18809 \cs_new:Npn \__fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
18810 {
18811   \if_meaning:w @ #1 @
18812   - 7
18813   - \exp_after:wN \use_i:nnn
18814     \exp_after:wN \__fp_sub_back_shift_iii:NNNNNNNNw
18815     \int_value:w #2#3 0 ~ 123456789;
18816   \else:
18817     - \__fp_sub_back_shift_iii:NNNNNNNNw #1 123456789;
18818   \fi:
18819   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
18820   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
18821   \exp_after:wN \__fp_sub_back_shift_iv:nnnnw
18822   \exp_after:wN ;
18823   \int_value:w
18824   #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
18825 }
18826 \cs_new:Npn \__fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
18827 \cs_new:Npn \__fp_sub_back_shift_iv:nnnnw #1 ; #2 ; { ; #1 ; }

```

(End definition for __fp_sub_back_shift:wnnnn and others.)

__fp_sub_back_far_o:NnnwnnnnN $\langle \text{rounding} \rangle \{ \langle Y'_1 \rangle \} \{ \langle Y'_2 \rangle \}$
 $\langle \text{extra-digits} \rangle ; \{ \langle X_1 \rangle \} \{ \langle X_2 \rangle \} \{ \langle X_3 \rangle \} \{ \langle X_4 \rangle \} \langle \text{final sign} \rangle$

If the difference is greater than $10^{\langle expo_x \rangle}$, call the `very_far` auxiliary. If the result is less than $10^{\langle expo_x \rangle}$, call the `not_far` auxiliary. If it is too close a call to know yet, namely if $1 \langle Y'_1 \rangle \langle Y'_2 \rangle = \langle X_1 \rangle \langle X_2 \rangle \langle X_3 \rangle \langle X_4 \rangle 0$, then call the `quite_far` auxiliary. We use the odd combination of space and semi-colon delimiters to allow the `not_far` auxiliary to grab each piece individually, the `very_far` auxiliary to use `__fp_pack_eight:wNNNNNNNN`, and the `quite_far` to ignore the significands easily (using the `; delimiter`).

```

18828 \cs_new:Npn \__fp_sub_back_far_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
18829 {
18830   \if_case:w
18831     \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
18832     \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
18833     0
18834     \else:
18835       \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: 1
18836       \fi:
18837     \else:
18838       \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: 1
18839       \fi:
18840     \exp_stop_f:
18841     \exp_after:wN \__fp_sub_back_quite_far_o:wwNN
18842   \or: \exp_after:wN \__fp_sub_back_very_far_o:wwwNN

```

```

18843 \else: \exp_after:wN \__fp_sub_back_not_far_o:wwwNN
18844 \fi:
18845 #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1
18846 }

```

(End definition for __fp_sub_back_far_o:NnnwnnnN.)

__fp_sub_back_quite_far_o:wwNN
__fp_sub_back_quite_far_ii:NN

The easiest case is when $x - y$ is extremely close to a power of 10, namely the first digit of x is 1, and all others vanish when subtracting y . Then the *rounding* #3 and the *final sign* #4 control whether we get 1 or 0.9999999999999999. In the usual round-to-nearest mode, we get 1 whenever the *rounding* digit is less than or equal to 5 (remember that the *rounding* digit is only equal to 5 if there was no further non-zero digit).

```

18847 \cs_new:Npn \__fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
18848 {
18849 \exp_after:wN \__fp_sub_back_quite_far_ii:NN
18850 \exp_after:wN #3
18851 \exp_after:wN #4
18852 }
18853 \cs_new:Npn \__fp_sub_back_quite_far_ii:NN #1#2
18854 {
18855 \if_case:w \__fp_round_neg:NNN #2 0 #1
18856 \exp_after:wN \use_i:nn
18857 \else:
18858 \exp_after:wN \use_ii:nn
18859 \fi:
18860 { ; {1000} {0000} {0000} {0000} ; }
18861 { - 1 ; {9999} {9999} {9999} {9999} ; }
18862 }

```

(End definition for __fp_sub_back_quite_far_o:wwNN and __fp_sub_back_quite_far_ii:NN.)

__fp_sub_back_not_far_o:wwwNN

In the present case, x and y have different exponents, but y is large enough that $x - y$ has a smaller exponent than x . Decrement the exponent (with -1). Then proceed in a way similar to the *near* auxiliaries seen earlier, but multiplying x by 10 (#30 and #40 below), and with the added quirk that the *rounding* digit has to be taken into account. Namely, we may have to decrease the result by one unit if __fp_round_neg:NNN returns 1. This function expects the *final sign* #6, the last digit of 1100000000+#40-#2, and the *rounding* digit. Instead of redoing the computation for the second argument, we note that __fp_round_neg:NNN only cares about its parity, which is identical to that of the last digit of #2.

```

18863 \cs_new:Npn \__fp_sub_back_not_far_o:wwwNN #1 ~ #2; #3 ~ #4; #5#6
18864 {
18865 - 1
18866 \exp_after:wN \__fp_sub_back_near_after:wNNNNw
18867 \int_value:w \__fp_int_eval:w 1#30 - #1 - 11
18868 \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
18869 \int_value:w \__fp_int_eval:w 11 0000 0000 + #40 - #2
18870 - \exp_after:wN \__fp_round_neg:NNN
18871 \exp_after:wN #6
18872 \use_none:nnnnnnn #2 #5
18873 \exp_after:wN ;
18874 }

```

(End definition for __fp_sub_back_not_far_o:wwwNN.)

_fp_sub_back_very_far_o:wwwNN
_fp_sub_back_very_far_ii_o:nnNwwNN

The case where $x - y$ and x have the same exponent is a bit more tricky, mostly because it cannot reuse the same auxiliaries. Shift the y significand by adding a leading 0. Then the logic is similar to the `not_far` functions above. Rounding is a bit more complicated: we have two *rounding* digits #3 and #6 (from the decimation, and from the new shift) to take into account, and getting the parity of the main result requires a computation. The first `\int_value:w` triggers the second one because the number is unfinished; we can thus not use 0 in place of 2 there.

```

18875 \cs_new:Npn \__fp_sub_back_very_far_o:wwwNN #1#2#3#4#5#6#7
18876 {
18877   \__fp_pack_eight:wNNNNNNNN
18878   \__fp_sub_back_very_far_ii_o:nnNwwNN
18879   { 0 #1#2#3 #4#5#6#7 }
18880   ;
18881 }
18882 \cs_new:Npn \__fp_sub_back_very_far_ii_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
18883 {
18884   \exp_after:wN \__fp_basics_pack_high:NNNNw
18885   \int_value:w \__fp_int_eval:w 1#4 - #1 - 1
18886   \exp_after:wN \__fp_basics_pack_low:NNNNw
18887   \int_value:w \__fp_int_eval:w 2#5 - #2
18888   - \exp_after:wN \__fp_round_neg:NNN
18889   \exp_after:wN #7
18890   \int_value:w
18891   \if_int_odd:w \__fp_int_eval:w #5 - #2 \__fp_int_eval_end:
18892   1 \else: 2 \fi:
18893   \int_value:w \__fp_round_digit:Nw #3 #6 ;
18894   \exp_after:wN ;
18895 }

```

(End definition for `__fp_sub_back_very_far_o:wwwNN` and `__fp_sub_back_very_far_ii_o:nnNwwNN`.)

31.2 Multiplication

31.2.1 Signs, and special numbers

_fp*_o:ww

We go through an auxiliary, which is common with `_fp/_o:ww`. The first argument is the operation, used for the invalid operation exception. The second is inserted in a formula to dispatch cases slightly differently between multiplication and division. The third is the operation for normal floating points. The fourth is there for extra cases needed in `_fp/_o:ww`.

```

18896 \cs_new:cpn { \_fp*_o:ww }
18897 {
18898   \__fp_mul_cases_o:NnNww
18899   *
18900   { - 2 + }
18901   \__fp_mul_npos_o:Nww
18902   { }
18903 }

```

(End definition for `_fp*_o:ww`.)

_fp_mul_cases_o:nNnnww

Split into 10 cases (12 for division). If both numbers are normal, go to case 0 (same sign) or case 1 (opposite signs): in both cases, call `__fp_mul_npos_o:Nww` to do the work. If

the first operand is `nan`, go to case 2, in which the second operand is discarded; if the second operand is `nan`, go to case 3, in which the first operand is discarded (note the weird interaction with the final test on signs). Then we separate the case where the first number is normal and the second is zero: this goes to cases 4 and 5 for multiplication, 10 and 11 for division. Otherwise, we do a computation which dispatches the products $0 \times 0 = 0 \times 1 = 1 \times 0 = 0$ to case 4 or 5 depending on the combined sign, the products $0 \times \infty$ and $\infty \times 0$ to case 6 or 7 (invalid operation), and the products $1 \times \infty = \infty \times 1 = \infty \times \infty = \infty$ to cases 8 and 9. Note that the code for these two cases (which return $\pm\infty$) is inserted as argument #4, because it differs in the case of divisions.

```

18904 \cs_new:Npn \__fp_mul_cases_o:NnNnw
18905   #1#2#3#4 \s__fp \__fp_chk:w #5#6#7; \s__fp \__fp_chk:w #8#9
18906   {
18907     \if_case:w \__fp_int_eval:w
18908       \if_int_compare:w #5 #8 = 11 ~
18909         1
18910       \else:
18911         \if_meaning:w 3 #8
18912         3
18913       \else:
18914         \if_meaning:w 3 #5
18915         2
18916       \else:
18917         \if_int_compare:w #5 #8 = 10 ~
18918         9 #2 - 2
18919       \else:
18920         (#5 #2 #8) / 2 * 2 + 7
18921       \fi:
18922     \fi:
18923   \fi:
18924   \fi:
18925   \if_meaning:w #6 #9 - 1 \fi:
18926   \__fp_int_eval_end:
18927   \__fp_case_use:nw { #3 0 }
18928   \or: \__fp_case_use:nw { #3 2 }
18929   \or: \__fp_case_return_i_o:ww
18930   \or: \__fp_case_return_ii_o:ww
18931   \or: \__fp_case_return_o:Nww \c_zero_fp
18932   \or: \__fp_case_return_o:Nww \c_minus_zero_fp
18933   \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
18934   \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
18935   \or: \__fp_case_return_o:Nww \c_inf_fp
18936   \or: \__fp_case_return_o:Nww \c_minus_inf_fp
18937   #4
18938   \fi:
18939   \s__fp \__fp_chk:w #5 #6 #7;
18940   \s__fp \__fp_chk:w #8 #9
18941 }

```

(End definition for `__fp_mul_cases_o:nNnnnw`.)

31.2.2 Absolute multiplication

In this subsection, we perform the multiplication of two positive normal numbers.

```

\__fp_mul_npos_o:Nww \__fp_mul_npos_o:Nww <final sign> \s__fp \__fp_chk:w 1 <sign1> {<exp1>}
<body1> ; \s__fp \__fp_chk:w 1 <sign2> {<exp2>} <body2> ;

```

After the computation, `__fp_sanitize:Nw` checks for overflow or underflow. As we did for addition, `__fp_int_eval:w` computes the exponent, catching any shift coming from the computation in the significand. The *<final sign>* is needed to do the rounding properly in the significand computation. We setup the post-expansion here, triggered by `__fp_mul_significand_o:nnnnNnnnn`.

This is also used in `l3fp-convert`.

```

18942 \cs_new:Npn \__fp_mul_npos_o:Nww
18943 #1 \s__fp \__fp_chk:w #2 #3 #4 #5 ; \s__fp \__fp_chk:w #6 #7 #8 #9 ;
18944 {
18945   \exp_after:wN \__fp_sanitize:Nw
18946   \exp_after:wN #1
18947   \int_value:w \__fp_int_eval:w
18948     #4 + #8
18949   \__fp_mul_significand_o:nnnnNnnnn #5 #1 #9
18950 }

```

(End definition for `__fp_mul_npos_o:Nww`.)

```

\__fp_mul_significand_o:nnnnNnnnn \__fp_mul_significand_o:nnnnNnnnn {<X1>} {<X2>} {<X3>} {<X4>} <sign>
\__fp_mul_significand_drop:NNNNNw {<Y1>} {<Y2>} {<Y3>} {<Y4>}
\__fp_mul_significand_keep:NNNNNw

```

Note the three semicolons at the end of the definition. One is for the last `__fp_mul_significand_drop:NNNNNw`; one is for `__fp_round_digit:Nw` later on; and one, preceded by `\exp_after:wN`, which is correctly expanded (within an `__fp_int_eval:w`), is used by `__fp_basics_pack_low:NNNNNw`.

The product of two 16 digit integers has 31 or 32 digits, but it is impossible to know which one before computing. The place where we round depends on that number of digits, and may depend on all digits until the last in some rare cases. The approach is thus to compute the 5 first blocks of 4 digits (the first one is between 100 and 9999 inclusive), and a compact version of the remaining 3 blocks. Afterwards, the number of digits is known, and we can do the rounding within yet another set of `__fp_int_eval:w`.

```

18951 \cs_new:Npn \__fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
18952 {
18953   \exp_after:wN \__fp_mul_significand_test_f:NNN
18954   \exp_after:wN #5
18955   \int_value:w \__fp_int_eval:w 99990000 + #1*#6 +
18956   \exp_after:wN \__fp_mul_significand_keep:NNNNNw
18957   \int_value:w \__fp_int_eval:w 99990000 + #1*#7 + #2*#6 +
18958   \exp_after:wN \__fp_mul_significand_keep:NNNNNw
18959   \int_value:w \__fp_int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
18960   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
18961   \int_value:w \__fp_int_eval:w 99990000 + #1*#9 + #2*#8 +
18962   #3*#7 + #4*#6 +
18963   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
18964   \int_value:w \__fp_int_eval:w 99990000 + #2*#9 + #3*#8 +
18965   #4*#7 +
18966   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
18967   \int_value:w \__fp_int_eval:w 99990000 + #3*#9 + #4*#8 +
18968   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
18969   \int_value:w \__fp_int_eval:w 100000000 + #4*#9 ;
18970 ; \exp_after:wN ;

```

```

18971 }
18972 \cs_new:Npn \__fp_mul_significand_drop:NNNNw #1#2#3#4#5 #6;
18973 { #1#2#3#4#5 ; + #6 }
18974 \cs_new:Npn \__fp_mul_significand_keep:NNNNw #1#2#3#4#5 #6;
18975 { #1#2#3#4#5 ; #6 ; }

```

(End definition for __fp_mul_significand_o:nnnnNnnnn, __fp_mul_significand_drop:NNNNw, and __fp_mul_significand_keep:NNNNw.)

```

\__fp_mul_significand_test_f:NNN \__fp_mul_significand_test_f:NNN <sign> 1 <digits 1-8> ; <digits 9-12> ;
<digits 13-16> ; + <digits 17-20> + <digits 21-24> + <digits 25-28> + <digits
29-32> ; \exp_after:wN ;

```

If the $\langle \text{digit } 1 \rangle$ is non-zero, then for rounding we only care about the digits 16 and 17, and whether further digits are zero or not (check for exact ties). On the other hand, if $\langle \text{digit } 1 \rangle$ is zero, we care about digits 17 and 18, and whether further digits are zero.

```

18976 \cs_new:Npn \__fp_mul_significand_test_f:NNN #1 #2 #3
18977 {
18978   \if_meaning:w 0 #3
18979     \exp_after:wN \__fp_mul_significand_small_f:NNwwwN
18980   \else:
18981     \exp_after:wN \__fp_mul_significand_large_f:NwwNNNN
18982   \fi:
18983   #1 #3
18984 }

```

(End definition for __fp_mul_significand_test_f:NNN.)

__fp_mul_significand_large_f:NwwNNNN In this branch, $\langle \text{digit } 1 \rangle$ is non-zero. The result is thus $\langle \text{digits } 1-16 \rangle$, plus some rounding which depends on the digits 16, 17, and whether all subsequent digits are zero or not. Here, __fp_round_digit:Nw takes digits 17 and further (as an integer expression), and replaces it by a $\langle \text{rounding digit} \rangle$, suitable for __fp_round:NNN.

```

18985 \cs_new:Npn \__fp_mul_significand_large_f:NwwNNNN #1 #2; #3; #4#5#6#7; +
18986 {
18987   \exp_after:wN \__fp_basics_pack_high:NNNNw
18988   \int_value:w \__fp_int_eval:w 1#2
18989   \exp_after:wN \__fp_basics_pack_low:NNNNw
18990   \int_value:w \__fp_int_eval:w 1#3#4#5#6#7
18991   + \exp_after:wN \__fp_round:NNN
18992   \exp_after:wN #1
18993   \exp_after:wN #7
18994   \int_value:w \__fp_round_digit:Nw
18995 }

```

(End definition for __fp_mul_significand_large_f:NwwNNNN.)

__fp_mul_significand_small_f:NNwwwN In this branch, $\langle \text{digit } 1 \rangle$ is zero. Our result is thus $\langle \text{digits } 2-17 \rangle$, plus some rounding which depends on the digits 17, 18, and whether all subsequent digits are zero or not. The 8 digits 1#3 are followed, after expansion of the small_pack auxiliary, by the next digit, to form a 9 digit number.

```

18996 \cs_new:Npn \__fp_mul_significand_small_f:NNwwwN #1 #2#3; #4#5; #6; + #7
18997 {
18998   - 1
18999   \exp_after:wN \__fp_basics_pack_high:NNNNw
19000   \int_value:w \__fp_int_eval:w 1#3#4

```

```

19001      \exp_after:wN \__fp_basics_pack_low:NNNNw
19002      \int_value:w \__fp_int_eval:w 1#5#6#7
19003      + \exp_after:wN \__fp_round:NNN
19004      \exp_after:wN #1
19005      \exp_after:wN #7
19006      \int_value:w \__fp_round_digit:Nw
19007  }

```

(End definition for __fp_mul_significand_small_f:NNwwN.)

31.3 Division

31.3.1 Signs, and special numbers

Time is now ripe to tackle the hardest of the four elementary operations: division.

__fp/_o:ww Filtering special floating point is very similar to what we did for multiplications, with a few variations. Invalid operation exceptions display / rather than *. In the formula for dispatch, we replace - 2 + by -. The case of normal numbers is treated using __fp_div_npos_o:Nww rather than __fp_mul_npos_o:Nww. There are two additional cases: if the first operand is normal and the second is a zero, then the division by zero exception is raised: cases 10 and 11 of the \if_case:w construction in __fp_mul_cases_o:NnNww are provided as the fourth argument here.

```

19008 \cs_new:cpn { __fp/_o:ww }
19009 {
19010   \__fp_mul_cases_o:NnNww
19011   /
19012   { - }
19013   \__fp_div_npos_o:Nww
19014   {
19015     \or:
19016     \__fp_case_use:nw
19017     { \__fp_division_by_zero_o:NNww \c_inf_fp / }
19018     \or:
19019     \__fp_case_use:nw
19020     { \__fp_division_by_zero_o:NNww \c_minus_inf_fp / }
19021   }
19022 }

```

(End definition for __fp/_o:ww.)

```

\__fp_div_npos_o:Nww \__fp_div_npos_o:Nww <final sign> \s__fp \__fp_chk:w 1 <sign_A> {<exp A>}
{<A_1>} {<A_2>} {<A_3>} {<A_4>} ; \s__fp \__fp_chk:w 1 <sign_Z> {<exp Z>}
{<Z_1>} {<Z_2>} {<Z_3>} {<Z_4>} ;

```

We want to compute A/Z . As for multiplication, __fp_sanitize:Nw checks for overflow or underflow; we provide it with the $\langle final\ sign \rangle$, and an integer expression in which we compute the exponent. We set up the arguments of __fp_div_significand_i_o:wnnw, namely an integer $\langle y \rangle$ obtained by adding 1 to the first 5 digits of Z (explanation given soon below), then the four $\{<A_i>\}$, then the four $\{<Z_i>\}$, a semi-colon, and the $\langle final\ sign \rangle$, used for rounding at the end.

```

19023 \cs_new:Npn \__fp_div_npos_o:Nww
19024   #1 \s__fp \__fp_chk:w 1 #2 #3 #4 ; \s__fp \__fp_chk:w 1 #5 #6 #7#8#9;
19025 {

```

```

19026     \exp_after:wN \__fp_sanitize:Nw
19027     \exp_after:wN #1
19028     \int_value:w \__fp_int_eval:w
19029         #3 - #6
19030     \exp_after:wN \__fp_div_significand_i_o:wnnw
19031     \int_value:w \__fp_int_eval:w #7 \use_i:nnnn #8 + 1 ;
19032         #4
19033         {#7}{#8}#9 ;
19034         #1
19035     }

```

(End definition for `__fp_div_npos_o:Nww`.)

31.3.2 Work plan

In this subsection, we explain how to avoid overflowing $\text{T}_{\text{E}}\text{X}$'s integers when performing the division of two positive normal numbers.

We are given two numbers, $A = 0.A_1A_2A_3A_4$ and $Z = 0.Z_1Z_2Z_3Z_4$, in blocks of 4 digits, and we know that the first digits of A_1 and of Z_1 are non-zero. To compute A/Z , we proceed as follows.

- Find an integer $Q_A \simeq 10^4 A/Z$.
- Replace A by $B = 10^4 A - Q_A Z$.
- Find an integer $Q_B \simeq 10^4 B/Z$.
- Replace B by $C = 10^4 B - Q_B Z$.
- Find an integer $Q_C \simeq 10^4 C/Z$.
- Replace C by $D = 10^4 C - Q_C Z$.
- Find an integer $Q_D \simeq 10^4 D/Z$.
- Consider $E = 10^4 D - Q_D Z$, and ensure correct rounding.

The result is then $Q = 10^{-4}Q_A + 10^{-8}Q_B + 10^{-12}Q_C + 10^{-16}Q_D + \text{rounding}$. Since the Q_i are integers, B , C , D , and E are all exact multiples of 10^{-16} , in other words, computing with 16 digits after the decimal separator yields exact results. The problem is the risk of overflow: in general B , C , D , and E may be greater than 1.

Unfortunately, things are not as easy as they seem. In particular, we want all intermediate steps to be positive, since negative results would require extra calculations at the end. This requires that $Q_A \leq 10^4 A/Z$ etc. A reasonable attempt would be to define Q_A as

$$\text{\int_eval:n} \left\{ \frac{A_1 A_2}{Z_1 + 1} - 1 \right\} \leq 10^4 \frac{A}{Z}$$

Subtracting 1 at the end takes care of the fact that $\varepsilon\text{-T}_{\text{E}}\text{X}$'s `__fp_int_eval:w` rounds divisions instead of truncating (really, $1/2$ would be sufficient, but we work with integers). We add 1 to Z_1 because $Z_1 \leq 10^4 Z < Z_1 + 1$ and we need Q_A to be an underestimate. However, we are now underestimating Q_A too much: it can be wrong by up to 100, for instance when $Z = 0.1$ and $A \simeq 1$. Then B could take values up to 10 (maybe more), and a few steps down the line, we would run into arithmetic overflow, since $\text{T}_{\text{E}}\text{X}$ can only handle integers less than roughly $2 \cdot 10^9$.

A better formula is to take

$$Q_A = \backslash\text{int_eval:n}\left\{\frac{10 \cdot A_1 A_2}{\lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1} - 1\right\}.$$

This is always less than $10^9 A / (10^5 Z)$, as we wanted. In words, we take the 5 first digits of Z into account, and the 8 first digits of A , using 0 as a 9-th digit rather than the true digit for efficiency reasons. We shall prove that using this formula to define all the Q_i avoids any overflow. For convenience, let us denote

$$y = \lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1,$$

so that, taking into account the fact that $\varepsilon\text{-TeX}$ rounds ties away from zero,

$$\begin{aligned} Q_A &= \left\lfloor \frac{A_1 A_2 0}{y} - \frac{1}{2} \right\rfloor \\ &> \frac{A_1 A_2 0}{y} - \frac{3}{2}. \end{aligned}$$

Note that $10^4 < y \leq 10^5$, and $999 \leq Q_A \leq 99989$. Also note that this formula does not cause an overflow as long as $A < (2^{31} - 1)/10^9 \simeq 2.147 \dots$, since the numerator involves an integer slightly smaller than $10^9 A$.

Let us bound B :

$$\begin{aligned} 10^5 B &= A_1 A_2 0 + 10 \cdot 0.A_3 A_4 - 10 \cdot Z_1.Z_2 Z_3 Z_4 \cdot Q_A \\ &< A_1 A_2 0 \cdot \left(1 - 10 \cdot \frac{Z_1.Z_2 Z_3 Z_4}{y}\right) + \frac{3}{2} \cdot 10 \cdot Z_1.Z_2 Z_3 Z_4 + 10 \\ &\leq \frac{A_1 A_2 0 \cdot (y - 10 \cdot Z_1.Z_2 Z_3 Z_4)}{y} + \frac{3}{2} y + 10 \\ &\leq \frac{A_1 A_2 0 \cdot 1}{y} + \frac{3}{2} y + 10 \leq \frac{10^9 A}{y} + 1.6 \cdot y. \end{aligned}$$

At the last step, we hide 10 into the second term for later convenience. The same reasoning yields

$$\begin{aligned} 10^5 B &< 10^9 A / y + 1.6y, \\ 10^5 C &< 10^9 B / y + 1.6y, \\ 10^5 D &< 10^9 C / y + 1.6y, \\ 10^5 E &< 10^9 D / y + 1.6y. \end{aligned}$$

The goal is now to prove that none of B , C , D , and E can go beyond $(2^{31} - 1)/10^9 = 2.147 \dots$.

Combining the various inequalities together with $A < 1$, we get

$$\begin{aligned} 10^5 B &< 10^9 / y + 1.6y, \\ 10^5 C &< 10^{13} / y^2 + 1.6(y + 10^4), \\ 10^5 D &< 10^{17} / y^3 + 1.6(y + 10^4 + 10^8 / y), \\ 10^5 E &< 10^{21} / y^4 + 1.6(y + 10^4 + 10^8 / y + 10^{12} / y^2). \end{aligned}$$

All of those bounds are convex functions of y (since every power of y involved is convex, and the coefficients are positive), and thus maximal at one of the end-points of the allowed range $10^4 < y \leq 10^5$. Thus,

$$\begin{aligned} 10^5 B &< \max(1.16 \cdot 10^5, 1.7 \cdot 10^5), \\ 10^5 C &< \max(1.32 \cdot 10^5, 1.77 \cdot 10^5), \\ 10^5 D &< \max(1.48 \cdot 10^5, 1.777 \cdot 10^5), \\ 10^5 E &< \max(1.64 \cdot 10^5, 1.7777 \cdot 10^5). \end{aligned}$$

All of those bounds are less than $2.147 \cdot 10^5$, and we are thus within $\text{T}_{\text{E}}\text{X}$'s bounds in all cases!

We later need to have a bound on the Q_i . Their definitions imply that $Q_A < 10^9 A/y - 1/2 < 10^5 A$ and similarly for the other Q_i . Thus, all of them are less than 177770.

The last step is to ensure correct rounding. We have

$$A/Z = \sum_{i=1}^4 (10^{-4i} Q_i) + 10^{-16} E/Z$$

exactly. Furthermore, we know that the result is in $[0.1, 10)$, hence will be rounded to a multiple of 10^{-16} or of 10^{-15} , so we only need to know the integer part of E/Z , and a “rounding” digit encoding the rest. Equivalently, we need to find the integer part of $2E/Z$, and determine whether it was an exact integer or not (this serves to detect ties). Since

$$\frac{2E}{Z} = 2 \frac{10^5 E}{10^5 Z} \leq 2 \frac{10^5 E}{10^4} < 36,$$

this integer part is between 0 and 35 inclusive. We let $\varepsilon\text{-T}_{\text{E}}\text{X}$ round

$$P = \backslash\text{int_eval:n} \left\{ \frac{2 \cdot E_1 E_2}{Z_1 Z_2} \right\},$$

which differs from $2E/Z$ by at most

$$\frac{1}{2} + 2 \left| \frac{E}{Z} - \frac{E}{10^{-8} Z_1 Z_2} \right| + 2 \left| \frac{10^8 E - E_1 E_2}{Z_1 Z_2} \right| < 1,$$

($1/2$ comes from $\varepsilon\text{-T}_{\text{E}}\text{X}$'s rounding) because each absolute value is less than 10^{-7} . Thus P is either the correct integer part, or is off by 1; furthermore, if $2E/Z$ is an integer, $P = 2E/Z$. We will check the sign of $2E - PZ$. If it is negative, then $E/Z \in ((P-1)/2, P/2)$. If it is zero, then $E/Z = P/2$. If it is positive, then $E/Z \in (P/2, (P+1)/2)$. In each case, we know how to round to an integer, depending on the parity of P , and the rounding mode.

31.3.3 Implementing the significand division

`_fp_div_significand_i_o:wnnw`

`_fp_div_significand_i_o:wnnw <y> ; {<A1>} {<A2>} {<A3>} {<A4>}
{<Z1>} {<Z2>} {<Z3>} {<Z4>} ; <sign>`

Compute $10^6 + Q_A$ (a 7 digit number thanks to the shift), unbrace $\langle A_1 \rangle$ and $\langle A_2 \rangle$, and prepare the $\langle continuation \rangle$ arguments for 4 consecutive calls to `_fp_div_significand_calc:wnnnnnnn`. Each of these calls needs $\langle y \rangle$ (**#1**), and it turns out that

we need post-expansion there, hence the `\int_value:w`. Here, `#4` is six brace groups, which give the six first n-type arguments of the `calc` function.

```

19036 \cs_new:Npn \__fp_div_significand_i_o:wnnw #1 ; #2#3 #4 ;
19037 {
19038   \exp_after:wN \__fp_div_significand_test_o:w
19039   \int_value:w \__fp_int_eval:w
19040   \exp_after:wN \__fp_div_significand_calc:wnnnnnnnn
19041   \int_value:w \__fp_int_eval:w 999999 + #2 #3 0 / #1 ;
19042   #2 #3 ;
19043   #4
19044   { \exp_after:wN \__fp_div_significand_ii:wnn \int_value:w #1 }
19045   { \exp_after:wN \__fp_div_significand_ii:wnn \int_value:w #1 }
19046   { \exp_after:wN \__fp_div_significand_ii:wnn \int_value:w #1 }
19047   { \exp_after:wN \__fp_div_significand_iii:wnnnnnn \int_value:w #1 }
19048 }

```

(End definition for `__fp_div_significand_i_o:wnnw`.)

```

\__fp_div_significand_calc:wnnnnnnnn \__fp_div_significand_calc:wnnnnnnnn <106 + QA> ; <A1> <A2> ; {<A3>}
\__fp_div_significand_calc_i:wnnnnnnnn {<A4>} {<Z1>} {<Z2>} {<Z3>} {<Z4>} {<continuation>}
\__fp_div_significand_calc_ii:wnnnnnnnn expands to
<106 + QA> <continuation> ; <B1> <B2> ; {<B3>} {<B4>} {<Z1>} {<Z2>} {<Z3>}
{<Z4>}

```

where $B = 10^4 A - Q_A \cdot Z$. This function is also used to compute C , D , E (with the input shifted accordingly), and is used in `l3fp-expo`.

We know that $0 < Q_A < 1.8 \cdot 10^5$, so the product of Q_A with each Z_i is within $\text{T}_{\text{E}}\text{X}$'s bounds. However, it is a little bit too large for our purposes: we would not be able to use the usual trick of adding a large power of 10 to ensure that the number of digits is fixed.

The bound on Q_A , implies that $10^6 + Q_A$ starts with the digit 1, followed by 0 or 1. We test, and call different auxiliaries for the two cases. An earlier implementation did the tests within the computation, but since we added a `<continuation>`, this is not possible because the macro has 9 parameters.

The result we want is then (the overall power of 10 is arbitrary):

$$\begin{aligned}
& 10^{-4}(\#2 - \#1 \cdot \#5 - 10 \cdot \langle i \rangle \cdot \#5\#6) + 10^{-8}(\#3 - \#1 \cdot \#6 - 10 \cdot \langle i \rangle \cdot \#7) \\
& + 10^{-12}(\#4 - \#1 \cdot \#7 - 10 \cdot \langle i \rangle \cdot \#8) + 10^{-16}(-\#1 \cdot \#8),
\end{aligned}$$

where $\langle i \rangle$ stands for the 10^5 digit of Q_A , which is 0 or 1, and $\#1$, $\#2$, *etc.* are the parameters of either auxiliary. The factors of 10 come from the fact that $Q_A = 10 \cdot 10^4 \cdot \langle i \rangle + \#1$. As usual, to combine all the terms, we need to choose some shifts which must ensure that the number of digits of the second, third, and fourth terms are each fixed. Here, the positive contributions are at most 10^8 and the negative contributions can go up to 10^9 . Indeed, for the auxiliary with $\langle i \rangle = 1$, $\#1$ is at most 80000, leading to contributions of at worst $-8 \cdot 10^8 4$, while the other negative term is very small $< 10^6$ (except in the first expression, where we don't care about the number of digits); for the auxiliary with $\langle i \rangle = 0$, $\#1$ can go up to 99999, but there is no other negative term. Hence, a good choice is $2 \cdot 10^9$, which produces totals in the range $[10^9, 2.1 \cdot 10^9]$. We are flirting with $\text{T}_{\text{E}}\text{X}$'s limits once more.

```

19049 \cs_new:Npn \__fp_div_significand_calc:wnnnnnnnn #1#

```

```

19050 {
19051   \if_meaning:w 1 #1
19052   \exp_after:wN \__fp_div_significand_calc_i:wwnnnnnnnn
19053   \else:
19054     \exp_after:wN \__fp_div_significand_calc_ii:wwnnnnnnnn
19055   \fi:
19056 }
19057 \cs_new:Npn \__fp_div_significand_calc_i:wwnnnnnnnn
19058 #1; #2;#3#4 #5#6#7#8 #9
19059 {
19060   1 1 #1
19061   #9 \exp_after:wN ;
19062   \int_value:w \__fp_int_eval:w \c__fp_Bigg_leading_shift_int
19063   + #2 - #1 * #5 - #5#60
19064   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
19065   \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
19066   + #3 - #1 * #6 - #70
19067   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
19068   \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
19069   + #4 - #1 * #7 - #80
19070   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
19071   \int_value:w \__fp_int_eval:w \c__fp_Bigg_trailing_shift_int
19072   - #1 * #8 ;
19073   {#5}{#6}{#7}{#8}
19074 }
19075 \cs_new:Npn \__fp_div_significand_calc_ii:wwnnnnnnnn
19076 #1; #2;#3#4 #5#6#7#8 #9
19077 {
19078   1 0 #1
19079   #9 \exp_after:wN ;
19080   \int_value:w \__fp_int_eval:w \c__fp_Bigg_leading_shift_int
19081   + #2 - #1 * #5
19082   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
19083   \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
19084   + #3 - #1 * #6
19085   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
19086   \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
19087   + #4 - #1 * #7
19088   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
19089   \int_value:w \__fp_int_eval:w \c__fp_Bigg_trailing_shift_int
19090   - #1 * #8 ;
19091   {#5}{#6}{#7}{#8}
19092 }

```

(End definition for __fp_div_significand_calc:wwnnnnnnnn, __fp_div_significand_calc_i:wwnnnnnnnn, and __fp_div_significand_calc_ii:wwnnnnnnnn.)

__fp_div_significand_ii:wwn $\langle y \rangle$; $\langle B_1 \rangle$; $\{\langle B_2 \rangle\}$ $\{\langle B_3 \rangle\}$ $\{\langle B_4 \rangle\}$ $\{\langle Z_1 \rangle\}$ $\{\langle Z_2 \rangle\}$ $\{\langle Z_3 \rangle\}$ $\{\langle Z_4 \rangle\}$ $\langle continuations \rangle$ $\langle sign \rangle$

Compute Q_B by evaluating $\langle B_1 \rangle \langle B_2 \rangle 0 / y - 1$. The result is output to the left, in an __fp_int_eval:w which we start now. Once that is evaluated (and the other Q_i also, since later expansions are triggered by this one), a packing auxiliary takes care of placing the digits of Q_B in an appropriate way for the final addition to obtain Q . This auxiliary is also used to compute Q_C and Q_D with the inputs C and D instead of B .

```

19093 \cs_new:Npn \__fp_div_significand_ii:wwn #1; #2;#3
19094 {
19095   \exp_after:wN \__fp_div_significand_pack:NNN
19096   \int_value:w \__fp_int_eval:w
19097   \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
19098   \int_value:w \__fp_int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
19099 }

```

(End definition for __fp_div_significand_ii:wwn.)

```

\__fp_div_significand_iii:wwnnnnn <y> ; <E1> ; {<E2>} {<E3>} {<E4>}
{<Z1>} {<Z2>} {<Z3>} {<Z4>} <sign>

```

We compute $P \simeq 2E/Z$ by rounding $2E_1E_2/Z_1Z_2$. Note the first 0, which multiplies Q_D by 10: we later add (roughly) $5 \cdot P$, which amounts to adding $P/2 \simeq E/Z$ to Q_D , the appropriate correction from a hypothetical Q_E .

```

19100 \cs_new:Npn \__fp_div_significand_iii:wwnnnnn #1; #2;#3#4#5 #6#7
19101 {
19102   0
19103   \exp_after:wN \__fp_div_significand_iv:wwnnnnnnn
19104   \int_value:w \__fp_int_eval:w ( 2 * #2 #3 ) / #6 #7 ; % <- P
19105   #2 ; {#3} {#4} {#5}
19106   {#6} {#7}
19107 }

```

(End definition for __fp_div_significand_iii:wwnnnnn.)

```

\__fp_div_significand_iv:wwnnnnnnn <P> ; <E1> ; {<E2>} {<E3>} {<E4>}
\__fp_div_significand_v:NNw {<Z1>} {<Z2>} {<Z3>} {<Z4>} <sign>
\__fp_div_significand_vi:Nw

```

This adds to the current expression ($10^7 + 10 \cdot Q_D$) a contribution of $5 \cdot P + \text{sign}(T)$ with $T = 2E - PZ$. This amounts to adding $P/2$ to Q_D , with an extra *rounding* digit. This *rounding* digit is 0 or 5 if T does not contribute, *i.e.*, if $0 = T = 2E - PZ$, in other words if $10^{16}A/Z$ is an integer or half-integer. Otherwise it is in the appropriate range, $[1, 4]$ or $[6, 9]$. This is precise enough for rounding purposes (in any mode).

It seems an overkill to compute T exactly as I do here, but I see no faster way right now.

Once more, we need to be careful and show that the calculation $\#1 \cdot \#6\#7$ below does not cause an overflow: naively, P can be up to 35, and $\#6\#7$ up to 10^8 , but both cannot happen simultaneously. To show that things are fine, we split in two (non-disjoint) cases.

- For $P < 10$, the product obeys $P \cdot \#6\#7 < 10^8 \cdot P < 10^9$.
- For large $P \geq 3$, the rounding error on P , which is at most 1, is less than a factor of 2, hence $P \leq 4E/Z$. Also, $\#6\#7 \leq 10^8 \cdot Z$, hence $P \cdot \#6\#7 \leq 4E \cdot 10^8 < 10^9$.

Both inequalities could be made tighter if needed.

Note however that $P \cdot \#8\#9$ may overflow, since the two factors are now independent, and the result may reach $3.5 \cdot 10^9$. Thus we compute the two lower levels separately. The rest is standard, except that we use $+$ as a separator (ending integer expressions explicitly). T is negative if the first character is $-$, it is positive if the first character is neither 0 nor $-$. It is also positive if the first character is 0 and second argument of $__fp_div_significand_vi:Nw$, a sum of several terms, is also zero. Otherwise, there was an exact agreement: $T = 0$.

```

19108 \cs_new:Npn \__fp_div_significand_iv:wwnnnnnnn #1; #2;#3#4#5 #6#7#8#9
19109 {
19110   + 5 * #1
19111   \exp_after:wN \__fp_div_significand_vi:Nw
19112   \int_value:w \__fp_int_eval:w -20 + 2*#2#3 - #1*#6#7 +
19113   \exp_after:wN \__fp_div_significand_v:NN
19114   \int_value:w \__fp_int_eval:w 199980 + 2*#4 - #1*#8 +
19115   \exp_after:wN \__fp_div_significand_v:NN
19116   \int_value:w \__fp_int_eval:w 200000 + 2*#5 - #1*#9 ;
19117 }
19118 \cs_new:Npn \__fp_div_significand_v:NN #1#2 { #1#2 \__fp_int_eval_end: + }
19119 \cs_new:Npn \__fp_div_significand_vi:Nw #1#2;
19120 {
19121   \if_meaning:w 0 #1
19122     \if_int_compare:w \__fp_int_eval:w #2 > 0 + 1 \fi:
19123   \else:
19124     \if_meaning:w - #1 - \else: + \fi: 1
19125   \fi:
19126   ;
19127 }

```

(End definition for __fp_div_significand_iv:wwnnnnnnn, __fp_div_significand_v:NNw, and __fp_div_significand_vi:Nw.)

__fp_div_significand_pack:NNN At this stage, we are in the following situation: \TeX is in the process of expanding several integer expressions, thus functions at the bottom expand before those above.

$$\begin{aligned} & _ _ \text{fp_div_significand_test_o:w } 10^6 + Q_A _ _ \text{fp_div_significand_} \\ & \text{pack:NNN } 10^6 + Q_B _ _ \text{fp_div_significand_pack:NNN } 10^6 + Q_C _ _ \text{fp_} \\ & \text{div_significand_pack:NNN } 10^7 + 10 \cdot Q_D + 5 \cdot P + \varepsilon ; \langle \text{sign} \rangle \end{aligned}$$

Here, $\varepsilon = \text{sign}(T)$ is 0 in case $2E = PZ$, 1 in case $2E > PZ$, which means that P was the correct value, but not with an exact quotient, and -1 if $2E < PZ$, *i.e.*, P was an overestimate. The packing function we define now does nothing special: it removes the 10^6 and carries two digits (for the 10^5 's and the 10^4 's).

```

19128 \cs_new:Npn \__fp_div_significand_pack:NNN 1 #1 #2 { + #1 #2 ; }

```

(End definition for __fp_div_significand_pack:NNN.)

_ _ \text{fp_div_significand_test_o:w } 1 0 \langle 5d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 5d \rangle ; \langle \text{sign} \rangle

The reason we know that the first two digits are 1 and 0 is that the final result is known to be between 0.1 (inclusive) and 10, hence \widetilde{Q}_A (the tilde denoting the contribution from the other Q_i) is at most 99999, and $10^6 + \widetilde{Q}_A = 10 \dots$.

It is now time to round. This depends on how many digits the final result will have.

```

19129 \cs_new:Npn \__fp_div_significand_test_o:w 10 #1
19130 {
19131   \if_meaning:w 0 #1
19132     \exp_after:wN \__fp_div_significand_small_o:wwwNNNNwN
19133   \else:
19134     \exp_after:wN \__fp_div_significand_large_o:wwwNNNNwN
19135   \fi:
19136   #1
19137 }

```

(End definition for __fp_div_significand_test_o:w.)

```

\__fp_div_significand_small_o:wwwNNNNwN \__fp_div_significand_small_o:wwwNNNNwN 0 <4d> ; <4d> ; <4d> ; <5d>
; <final sign>

```

Standard use of the functions `__fp_basics_pack_low:NNNNw` and `__fp_basics_pack_high:NNNNw`. We finally get to use the *<final sign>* which has been sitting there for a while.

```

19138 \cs_new:Npn \__fp_div_significand_small_o:wwwNNNNwN
19139 0 #1; #2; #3; #4#5#6#7#8; #9
19140 {
19141 \exp_after:wN \__fp_basics_pack_high:NNNNw
19142 \int_value:w \__fp_int_eval:w 1 #1#2
19143 \exp_after:wN \__fp_basics_pack_low:NNNNw
19144 \int_value:w \__fp_int_eval:w 1 #3#4#5#6#7
19145 + \__fp_round:NNN #9 #7 #8
19146 \exp_after:wN ;
19147 }

```

(End definition for `__fp_div_significand_small_o:wwwNNNNwN`.)

```

\__fp_div_significand_large_o:wwwNNNNwN \__fp_div_significand_large_o:wwwNNNNwN <5d> ; <4d> ; <4d> ; <5d> ;
<sign>

```

We know that the final result cannot reach 10, hence `1#1#2`, together with contributions from the level below, cannot reach $2 \cdot 10^9$. For rounding, we build the *<rounding digit>* from the last two of our 18 digits.

```

19148 \cs_new:Npn \__fp_div_significand_large_o:wwwNNNNwN
19149 #1; #2; #3; #4#5#6#7#8; #9
19150 {
19151 + 1
19152 \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNw
19153 \int_value:w \__fp_int_eval:w 1 #1 #2
19154 \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
19155 \int_value:w \__fp_int_eval:w 1 #3 #4 #5 #6 +
19156 \exp_after:wN \__fp_round:NNN
19157 \exp_after:wN #9
19158 \exp_after:wN #6
19159 \int_value:w \__fp_round_digit:Nw #7 #8 ;
19160 \exp_after:wN ;
19161 }

```

(End definition for `__fp_div_significand_large_o:wwwNNNNwN`.)

31.4 Square root

`__fp_sqrt_o:w` Zeros are unchanged: $\sqrt{-0} = -0$ and $\sqrt{+0} = +0$. Negative numbers (other than -0) have no real square root. Positive infinity, and `nan`, are unchanged. Finally, for normal positive numbers, there is some work to do.

```

19162 \cs_new:Npn \__fp_sqrt_o:w #1 \s_fp \__fp_chk:w #2#3#4; @
19163 {
19164 \if_meaning:w 0 #2 \__fp_case_return_same_o:w \fi:
19165 \if_meaning:w 2 #3
19166 \__fp_case_use:nw { \__fp_invalid_operation_o:nw { sqrt } }
19167 \fi:
19168 \if_meaning:w 1 #2 \else: \__fp_case_return_same_o:w \fi:
19169 \__fp_sqrt_npos_o:w

```

```

19170   \s__fp \__fp_chk:w #2 #3 #4;
19171   }

```

(End definition for __fp_sqrt_o:w.)

```

\__fp_sqrt_npos_o:w
\__fp_sqrt_npos_auxi_o:wNnnN
\__fp_sqrt_npos_auxii_o:wNNNNNNNNN

```

Prepare __fp_sanitize:Nw to receive the final sign 0 (the result is always positive) and the exponent, equal to half of the exponent #1 of the argument. If the exponent #1 is even, find a first approximation of the square root of the significand $10^8 a_1 + a_2 = 10^8 \#2\#3 + \#4\#5$ through Newton's method, starting at $x = 57234133 \simeq 10^{7.75}$. Otherwise, first shift the significand of the argument by one digit, getting $a'_1 \in [10^6, 10^7)$ instead of $[10^7, 10^8)$, then use Newton's method starting at $17782794 \simeq 10^{7.25}$.

```

19172 \cs_new:Npn \__fp_sqrt_npos_o:w \s__fp \__fp_chk:w 1 0 #1#2#3#4#5;
19173 {
19174   \exp_after:wN \__fp_sanitize:Nw
19175   \exp_after:wN 0
19176   \int_value:w \__fp_int_eval:w
19177   \if_int_odd:w #1 \exp_stop_f:
19178     \exp_after:wN \__fp_sqrt_npos_auxi_o:wNnnN
19179     \fi:
19180     #1 / 2
19181     \__fp_sqrt_Newton_o:wN 56234133; 0; {#2#3} {#4#5} 0
19182   }
19183 \cs_new:Npn \__fp_sqrt_npos_auxi_o:wNnnN #1 / 2 #2; 0; #3#4#5
19184 {
19185   ( #1 + 1 ) / 2
19186   \__fp_pack_eight:wNNNNNNNNN
19187   \__fp_sqrt_npos_auxii_o:wNNNNNNNNN
19188   ;
19189   0 #3 #4
19190 }
19191 \cs_new:Npn \__fp_sqrt_npos_auxii_o:wNNNNNNNNN #1; #2#3#4#5#6#7#8#9
19192 { \__fp_sqrt_Newton_o:wN 17782794; 0; {#1} {#2#3#4#5#6#7#8#9} }

```

(End definition for __fp_sqrt_npos_o:w, __fp_sqrt_npos_auxi_o:wNnnN, and __fp_sqrt_npos_auxii_o:wNNNNNNNNN.)

```
\__fp_sqrt_Newton_o:wN
```

Newton's method maps $x \mapsto [(x + [10^8 a_1/x])/2]$ in each iteration, where $[b/c]$ denotes $\varepsilon\text{-TeX}$'s division. This division rounds the real number b/c to the closest integer, rounding ties away from zero, hence when c is even, $b/c - 1/2 + 1/c \leq [b/c] \leq b/c + 1/2$ and when c is odd, $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2 - 1/(2c)$. For all c , $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2$.

Let us prove that the method converges when implemented with $\varepsilon\text{-TeX}$ integer division, for any $10^6 \leq a_1 < 10^8$ and starting value $10^6 \leq x < 10^8$. Using the inequalities above and the arithmetic-geometric inequality $(x+t)/2 \geq \sqrt{xt}$ for $t = 10^8 a_1/x$, we find

$$x' = \left\lceil \frac{x + [10^8 a_1/x]}{2} \right\rceil \geq \frac{x + 10^8 a_1/x - 1/2 + 1/(2x)}{2} \geq \sqrt{10^8 a_1} - \frac{1}{4} + \frac{1}{4x}.$$

After any step of iteration, we thus have $\delta = x - \sqrt{10^8 a_1} \geq -0.25 + 0.25 \cdot 10^{-8}$. The new difference $\delta' = x' - \sqrt{10^8 a_1}$ after one step is bounded above as

$$x' - \sqrt{10^8 a_1} \leq \frac{x + 10^8 a_1/x + 1/2}{2} + \frac{1}{2} - \sqrt{10^8 a_1} \leq \frac{\delta}{2} \frac{\delta}{\sqrt{10^8 a_1} + \delta} + \frac{3}{4}.$$

For $\delta > 3/2$, this last expression is $\leq \delta/2 + 3/4 < \delta$, hence δ decreases at each step: since all x are integers, δ must reach a value $-1/4 < \delta \leq 3/2$. In this range of values, we get $\delta' \leq \frac{3}{4} \frac{3}{2\sqrt{10^8 a_1}} + \frac{3}{4} \leq 0.75 + 1.125 \cdot 10^{-7}$. We deduce that the difference $\delta = x - \sqrt{10^8 a_1}$ eventually reaches a value in the interval $[-0.25 + 0.25 \cdot 10^{-8}, 0.75 + 11.25 \cdot 10^{-8}]$, whose width is $1 + 11 \cdot 10^{-8}$. The corresponding interval for x may contain two integers, hence x might oscillate between those two values.

However, the fact that $x \mapsto x - 1$ and $x - 1 \mapsto x$ puts stronger constraints, which are not compatible: the first implies

$$x + [10^8 a_1 / x] \leq 2x - 2$$

hence $10^8 a_1 / x \leq x - 3/2$, while the second implies

$$x - 1 + [10^8 a_1 / (x - 1)] \geq 2x - 1$$

hence $10^8 a_1 / (x - 1) \geq x - 1/2$. Combining the two inequalities yields $x^2 - 3x/2 \geq 10^8 a_1 \geq x - 3x/2 + 1/2$, which cannot hold. Therefore, the iteration always converges to a single integer x . To stop the iteration when two consecutive results are equal, the function `_fp_sqrt_Newton_o:wnn` receives the newly computed result as #1, the previous result as #2, and a_1 as #3. Note that ε -TeX combines the computation of a multiplication and a following division, thus avoiding overflow in `#3 * 100000000 / #1`. In any case, the result is within $[10^7, 10^8]$.

```

19193 \cs_new:Npn \_fp_sqrt_Newton_o:wnn #1; #2; #3
19194 {
19195   \if_int_compare:w #1 = #2 \exp_stop_f:
19196     \exp_after:wN \_fp_sqrt_auxi_o:NNNNwnnnN
19197     \int_value:w \_fp_int_eval:w 9999 9999 +
19198     \exp_after:wN \_fp_use_none_until_s:w
19199   \fi:
19200   \exp_after:wN \_fp_sqrt_Newton_o:wnn
19201   \int_value:w \_fp_int_eval:w (#1 + #3 * 1 0000 0000 / #1) / 2 ;
19202   #1; {#3}
19203 }

```

(End definition for `_fp_sqrt_Newton_o:wnn`.)

`_fp_sqrt_auxi_o:NNNNwnnnN` This function is followed by $10^8 + x - 1$, which has 9 digits starting with 1, then ; $\{a_1\} \{a_2\} \{a'\}$. Here, $x \simeq \sqrt{10^8 a_1}$ and we want to estimate the square root of $a = 10^{-8} a_1 + 10^{-16} a_2 + 10^{-17} a'$. We set up an initial underestimate

$$y = (x - 1)10^{-8} + 0.2499998875 \cdot 10^{-8} \lesssim \sqrt{a}.$$

From the inequalities shown earlier, we know that $y \leq \sqrt{10^{-8} a_1} \leq \sqrt{a}$ and that $\sqrt{10^{-8} a_1} \leq y + 10^{-8} + 11 \cdot 10^{-16}$ hence (using $0.1 \leq y \leq \sqrt{a} \leq 1$)

$$a - y^2 \leq 10^{-8} a_1 + 10^{-8} - y^2 \leq (y + 10^{-8} + 11 \cdot 10^{-16})^2 - y^2 + 10^{-8} < 3.2 \cdot 10^{-8},$$

and $\sqrt{a} - y = (a - y^2) / (\sqrt{a} + y) \leq 16 \cdot 10^{-8}$. Next, `_fp_sqrt_auxii_o:NNnnnnnnN` is called several times to get closer and closer underestimates of \sqrt{a} . By construction, the underestimates y are always increasing, $a - y^2 < 3.2 \cdot 10^{-8}$ for all. Also, $y < 1$.

```

19204 \cs_new:Npn \_fp_sqrt_auxi_o:NNNNwnnnN 1 #1#2#3#4#5;
19205 {

```

```

19206 \__fp_sqrt_auxii_o:NnnnnnnnN
19207 \__fp_sqrt_auxiii_o:wnnnnnnnn
19208 {#1#2#3#4} {#5} {2499} {9988} {7500}
19209 }

```

(End definition for __fp_sqrt_auxi_o:NNNNwnnnN.)

__fp_sqrt_auxii_o:NnnnnnnnN

This receives a continuation function #1, then five blocks of 4 digits for y , then two 8-digit blocks and a single digit for a . A common estimate of $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y)$ is $(a - y^2)/(2y)$, which leads to alternating overestimates and underestimates. We tweak this, to only work with underestimates (no need then to worry about signs in the computation). Each step finds the largest integer $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then computes the integer (with ε -TeX's rounding division)

$$10^{4j}z = \left[\left(\lfloor 10^{4j}(a - y^2) \rfloor - 257 \right) \cdot (0.5 \cdot 10^8) \right] / \lfloor 10^8 y + 1 \rfloor.$$

The choice of j ensures that $10^{4j}z < 2 \cdot 10^8 \cdot 0.5 \cdot 10^8 / 10^7 = 10^9$, thus $10^9 + 10^{4j}z$ has exactly 10 digits, does not overflow TeX's integer range, and starts with 1. Incidentally, since all $a - y^2 \leq 3.2 \cdot 10^{-8}$, we know that $j \geq 3$.

Let us show that z is an underestimate of $\sqrt{a} - y$. On the one hand, $\sqrt{a} - y \leq 16 \cdot 10^{-8}$ because this holds for the initial y and values of y can only increase. On the other hand, the choice of j implies that $\sqrt{a} - y \leq 5(\sqrt{a} + y)(\sqrt{a} - y) = 5(a - y^2) < 10^{9-4j}$. For $j = 3$, the first bound is better, while for larger j , the second bound is better. For all $j \in [3, 6]$, we find $\sqrt{a} - y < 16 \cdot 10^{-2j}$. From this, we deduce that

$$10^{4j}(\sqrt{a} - y) = \frac{10^{4j}(a - y^2 - (\sqrt{a} - y)^2)}{2y} \geq \frac{\lfloor 10^{4j}(a - y^2) \rfloor - 257}{2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor} + \frac{1}{2}$$

where we have replaced the bound $10^{4j}(16 \cdot 10^{-2j}) = 256$ by 257 and extracted the corresponding term $1/(2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor) \geq 1/2$. Given that ε -TeX's integer division obeys $\lfloor b/c \rfloor \leq b/c + 1/2$, we deduce that $10^{4j}z \leq 10^{4j}(\sqrt{a} - y)$, hence $y + z \leq \sqrt{a}$ is an underestimate of \sqrt{a} , as claimed. One implementation detail: because the computation involves $-4*4 - 2*3*5 - 2*2*6$ which may be as low as $-5 \cdot 10^8$, we need to use the `pack_big` functions, and the big shifts.

```

19210 \cs_new:Npn \__fp_sqrt_auxii_o:NnnnnnnnN #1 #2#3#4#5#6 #7#8#9
19211 {
19212   \exp_after:wN #1
19213   \int_value:w \__fp_int_eval:w \c__fp_big_leading_shift_int
19214   + #7 - #2 * #2
19215   \exp_after:wN \__fp_pack_big:NNNNNNw
19216   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
19217   - 2 * #2 * #3
19218   \exp_after:wN \__fp_pack_big:NNNNNNw
19219   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
19220   + #8 - #3 * #3 - 2 * #2 * #4
19221   \exp_after:wN \__fp_pack_big:NNNNNNw
19222   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
19223   - 2 * #3 * #4 - 2 * #2 * #5
19224   \exp_after:wN \__fp_pack_big:NNNNNNw
19225   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
19226   + #9 000 0000 - #4 * #4 - 2 * #3 * #5 - 2 * #2 * #6
19227   \exp_after:wN \__fp_pack_big:NNNNNNw

```

```

19228      \int_value:w \_fp_int_eval:w \c\_fp\_big\_middle\_shift\_int
19229      - 2 * #4 * #5 - 2 * #3 * #6
19230      \exp_after:wN \_fp\_pack\_big:NNNNNNw
19231      \int_value:w \_fp_int_eval:w \c\_fp\_big\_middle\_shift\_int
19232      - #5 * #5 - 2 * #4 * #6
19233      \exp_after:wN \_fp\_pack\_big:NNNNNNw
19234      \int_value:w \_fp_int_eval:w
19235      \c\_fp\_big\_middle\_shift\_int
19236      - 2 * #5 * #6
19237      \exp_after:wN \_fp\_pack\_big:NNNNNNw
19238      \int_value:w \_fp_int_eval:w
19239      \c\_fp\_big\_trailing\_shift\_int
19240      - #6 * #6 ;
19241      % (
19242      - 257 ) * 5000 0000 / (#2#3 + 1) + 10 0000 0000 ;
19243      {#2}{#3}{#4}{#5}{#6} {#7}{#8}#9
19244      }

```

(End definition for _fp_sqrt_auxii_o:NnnnnnnnnN.)

```

\_fp\_sqrt\_auxiii\_o:wnnnnnnnnn
\_fp\_sqrt\_auxiv\_o:NNNNNNw
\_fp\_sqrt\_auxv\_o:NNNNNNw
\_fp\_sqrt\_auxvi\_o:NNNNNNw
\_fp\_sqrt\_auxvii\_o:NNNNNNw

```

We receive here the difference $a - y^2 = d = \sum_i d_i \cdot 10^{-4i}$, as $\langle d_2 \rangle$; $\{\langle d_3 \rangle\} \dots \{\langle d_{10} \rangle\}$, where each block has 4 digits, except $\langle d_2 \rangle$. This function finds the largest $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then leaves an open parenthesis and the integer $\lfloor 10^{4j}(a - y^2) \rfloor$ in an integer expression. The closing parenthesis is provided by the caller _fp_sqrt_auxii_o:NnnnnnnnnN, which completes the expression

$$10^{4j}z = \left[(\lfloor 10^{4j}(a - y^2) \rfloor - 257) \cdot (0.5 \cdot 10^8) / \lfloor 10^8 y + 1 \rfloor \right]$$

for an estimate of $10^{4j}(\sqrt{a} - y)$. If $d_2 \geq 2$, $j = 3$ and the **auxiv** auxiliary receives $10^{12}z$. If $d_2 \leq 1$ but $10^4 d_2 + d_3 \geq 2$, $j = 4$ and the **auxv** auxiliary is called, and receives $10^{16}z$, and so on. In all those cases, the **auxviii** auxiliary is set up to add z to y , then go back to the **auxii** step with continuation **auxiii** (the function we are currently describing). The maximum value of j is 6, regardless of whether $10^{12}d_2 + 10^8 d_3 + 10^4 d_4 + d_5 \geq 1$. In this last case, we detect when $10^{24}z < 10^7$, which essentially means $\sqrt{a} - y \lesssim 10^{-17}$: once this threshold is reached, there is enough information to find the correctly rounded \sqrt{a} with only one more call to _fp_sqrt_auxii_o:NnnnnnnnnN. Note that the iteration cannot be stuck before reaching $j = 6$, because for $j < 6$, one has $2 \cdot 10^8 \leq 10^{4(j+1)}(a - y^2)$, hence

$$10^{4j}z \geq \frac{(20000 - 257)(0.5 \cdot 10^8)}{\lfloor 10^8 y + 1 \rfloor} \geq (20000 - 257) \cdot 0.5 > 0.$$

```

19245 \cs_new:Npn \_fp\_sqrt\_auxiii\_o:wnnnnnnnnn
19246   #1; #2#3#4#5#6#7#8#9
19247   {
19248     \if_int_compare:w #1 > 1 \exp_stop_f:
19249     \exp_after:wN \_fp\_sqrt\_auxiv\_o:NNNNNNw
19250     \int_value:w \_fp_int_eval:w (#1#2 %)
19251   \else:
19252     \if_int_compare:w #1#2 > 1 \exp_stop_f:
19253     \exp_after:wN \_fp\_sqrt\_auxv\_o:NNNNNNw
19254     \int_value:w \_fp_int_eval:w (#1#2#3 %)
19255   \else:
19256     \if_int_compare:w #1#2#3 > 1 \exp_stop_f:
19257     \exp_after:wN \_fp\_sqrt\_auxvi\_o:NNNNNNw

```

```

19258         \int_value:w \__fp_int_eval:w (#1#2#3#4 %)
19259     \else:
19260         \exp_after:wN \__fp_sqrt_auxvii_o:NNNNNw
19261         \int_value:w \__fp_int_eval:w (#1#2#3#4#5 %)
19262     \fi:
19263 \fi:
19264 \fi:
19265 }
19266 \cs_new:Npn \__fp_sqrt_auxiv_o:NNNNNw #1#2#3#4#5#6;
19267 { \__fp_sqrt_auxviii_o:nnnnnnnn {#1#2#3#4#5#6} {00000000} }
19268 \cs_new:Npn \__fp_sqrt_auxv_o:NNNNNw #1#2#3#4#5#6;
19269 { \__fp_sqrt_auxviii_o:nnnnnnnn {000#1#2#3#4#5} {#60000} }
19270 \cs_new:Npn \__fp_sqrt_auxvi_o:NNNNNw #1#2#3#4#5#6;
19271 { \__fp_sqrt_auxviii_o:nnnnnnnn {0000000#1} {#2#3#4#5#6} }
19272 \cs_new:Npn \__fp_sqrt_auxvii_o:NNNNNw #1#2#3#4#5#6;
19273 {
19274     \if_int_compare:w #1#2 = 0 \exp_stop_f:
19275     \exp_after:wN \__fp_sqrt_auxx_o:Nnnnnnnnn
19276     \fi:
19277     \__fp_sqrt_auxviii_o:nnnnnnnn {00000000} {000#1#2#3#4#5}
19278 }

```

(End definition for `__fp_sqrt_auxiii_o:wnnnnnnnn` and others.)

`__fp_sqrt_auxviii_o:nnnnnnnn` Simply add the two 8-digit blocks of z , aligned to the last four of the five 4-digit blocks of y , then call the `auxii` auxiliary to evaluate $y'^2 = (y + z)^2$.

```

19279 \cs_new:Npn \__fp_sqrt_auxviii_o:nnnnnnnn #1#2 #3#4#5#6#7
19280 {
19281     \exp_after:wN \__fp_sqrt_auxix_o:wnwnw
19282     \int_value:w \__fp_int_eval:w #3
19283     \exp_after:wN \__fp_basics_pack_low:NNNNNw
19284     \int_value:w \__fp_int_eval:w #1 + 1#4#5
19285     \exp_after:wN \__fp_basics_pack_low:NNNNNw
19286     \int_value:w \__fp_int_eval:w #2 + 1#6#7 ;
19287 }
19288 \cs_new:Npn \__fp_sqrt_auxix_o:wnwnw #1; #2#3; #4#5;
19289 {
19290     \__fp_sqrt_auxii_o:NnnnnnnnnN
19291     \__fp_sqrt_auxiii_o:wnnnnnnnnn {#1}{#2}{#3}{#4}{#5}
19292 }

```

(End definition for `__fp_sqrt_auxviii_o:nnnnnnnn` and `__fp_sqrt_auxix_o:wnwnw`.)

`__fp_sqrt_auxx_o:Nnnnnnnnn` At this stage, $j = 6$ and $10^{24}z < 10^7$, hence
`__fp_sqrt_auxxi_o:wnnnN`

$$10^7 + 1/2 > 10^{24}z + 1/2 \geq (10^{24}(a - y^2) - 258) \cdot (0.5 \cdot 10^8) / (10^8y + 1),$$

then $10^{24}(a - y^2) - 258 < 2(10^7 + 1/2)(y + 10^{-8})$, and

$$10^{24}(a - y^2) < (10^7 + 1290.5)(1 + 10^{-8}/y)(2y) < (10^7 + 1290.5)(1 + 10^{-7})(y + \sqrt{a}),$$

which finally implies $0 \leq \sqrt{a} - y < 0.2 \cdot 10^{-16}$. In particular, y is an underestimate of \sqrt{a} and $y + 0.5 \cdot 10^{-16}$ is a (strict) overestimate. There is at exactly one multiple m of $0.5 \cdot 10^{-16}$ in the interval $[y, y + 0.5 \cdot 10^{-16})$. If $m^2 > a$, then the square root is inexact and

is obtained by rounding $m - \epsilon$ to a multiple of 10^{-16} (the precise shift $0 < \epsilon < 0.5 \cdot 10^{-16}$ is irrelevant for rounding). If $m^2 = a$ then the square root is exactly m , and there is no rounding. If $m^2 < a$ then we round $m + \epsilon$. For now, discard a few irrelevant arguments #1, #2, #3, and find the multiple of $0.5 \cdot 10^{-16}$ within $[y, y + 0.5 \cdot 10^{-16})$; rather, only the last 4 digits #8 of y are considered, and we do not perform any carry yet. The `auxxi` auxiliary sets up `auxii` with a continuation function `auxxii` instead of `auxiii` as before. To prevent `auxii` from giving a negative results $a - m^2$, we compute $a + 10^{-16} - m^2$ instead, always positive since $m < \sqrt{a} + 0.5 \cdot 10^{-16}$ and $a \leq 1 - 10^{-16}$.

```

19293 \cs_new:Npn \__fp_sqrt_auxx_o:Nnnnnnnn #1#2#3 #4#5#6#7#8
19294 {
19295   \exp_after:wN \__fp_sqrt_auxxi_o:wwnnN
19296   \int_value:w \__fp_int_eval:w
19297     (#8 + 2499) / 5000 * 5000 ;
19298   {#4} {#5} {#6} {#7} ;
19299 }
19300 \cs_new:Npn \__fp_sqrt_auxxi_o:wwnnN #1; #2; #3#4#5
19301 {
19302   \__fp_sqrt_auxii_o:NnnnnnnnN
19303   \__fp_sqrt_auxxii_o:nnnnnnnnw
19304   #2 {#1}
19305   {#3} { #4 + 1 } #5
19306 }

```

(End definition for `__fp_sqrt_auxx_o:Nnnnnnnn` and `__fp_sqrt_auxxi_o:wwnnN`.)

`__fp_sqrt_auxxii_o:nnnnnnnnw`
`__fp_sqrt_auxxiii_o:w`

The difference $0 \leq a + 10^{-16} - m^2 \leq 10^{-16} + (\sqrt{a} - m)(\sqrt{a} + m) \leq 2 \cdot 10^{-16}$ was just computed: its first 8 digits vanish, as do the next four, #1, and most of the following four, #2. The guess m is an overestimate if $a + 10^{-16} - m^2 < 10^{-16}$, that is, #1#2 vanishes. Otherwise it is an underestimate, unless $a + 10^{-16} - m^2 = 10^{-16}$ exactly. For an underestimate, call the `auxxiv` function with argument 9998. For an exact result call it with 9999, and for an overestimate call it with 10000.

```

19307 \cs_new:Npn \__fp_sqrt_auxxii_o:nnnnnnnnw 0; #1#2#3#4#5#6#7#8 #9;
19308 {
19309   \if_int_compare:w #1#2 > 0 \exp_stop_f:
19310   \if_int_compare:w #1#2 = 1 \exp_stop_f:
19311   \if_int_compare:w #3#4 = 0 \exp_stop_f:
19312   \if_int_compare:w #5#6 = 0 \exp_stop_f:
19313   \if_int_compare:w #7#8 = 0 \exp_stop_f:
19314     \__fp_sqrt_auxxiii_o:w
19315   \fi:
19316   \fi:
19317   \fi:
19318   \fi:
19319   \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnN
19320   \int_value:w 9998
19321 \else:
19322   \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnN
19323   \int_value:w 10000
19324 \fi:
19325 ;
19326 }
19327 \cs_new:Npn \__fp_sqrt_auxxiii_o:w \fi: \fi: \fi: \fi: #1 \fi: ;
19328 {

```

```

19329      \fi: \fi: \fi: \fi: \fi:
19330      \__fp_sqrt_auxxiv_o:wnnnnnnnnN 9999 ;
19331  }

```

(End definition for __fp_sqrt_auxxii_o:nnnnnnnnw and __fp_sqrt_auxxiii_o:w.)

__fp_sqrt_auxxiv_o:wnnnnnnnnN This receives 9998, 9999 or 10000 as #1 when m is an underestimate, exact, or an overestimate, respectively. Then comes m as five blocks of 4 digits, but where the last block #6 may be 0, 5000, or 10000. In the latter case, we need to add a carry, unless m is an overestimate (#1 is then 10000). Then comes a as three arguments. Rounding is done by __fp_round:NNN, whose first argument is the final sign 0 (square roots are positive). We fake its second argument. It should be the last digit kept, but this is only used when ties are “rounded to even”, and only when the result is exactly half-way between two representable numbers rational square roots of numbers with 16 significant digits have: this situation never arises for the square root, as any exact square root of a 16 digit number has at most 8 significant digits. Finally, the last argument is the next digit, possibly shifted by 1 when there are further nonzero digits. This is achieved by __fp_round_digit:Nw, which receives (after removal of the 10000’s digit) one of 0000, 0001, 4999, 5000, 5001, or 9999, which it converts to 0, 1, 4, 5, 6, and 9, respectively.

```

19332 \cs_new:Npn \__fp_sqrt_auxxiv_o:wnnnnnnnnN #1; #2#3#4#5#6 #7#8#9
19333 {
19334   \exp_after:wN \__fp_basics_pack_high:NNNNNw
19335   \int_value:w \__fp_int_eval:w 1 0000 0000 + #2#3
19336   \exp_after:wN \__fp_basics_pack_low:NNNNNw
19337   \int_value:w \__fp_int_eval:w 1 0000 0000
19338   + #4#5
19339   \if_int_compare:w #6 > #1 \exp_stop_f: + 1 \fi:
19340   + \exp_after:wN \__fp_round:NNN
19341   \exp_after:wN 0
19342   \exp_after:wN 0
19343   \int_value:w
19344   \exp_after:wN \use_i:nn
19345   \exp_after:wN \__fp_round_digit:Nw
19346   \int_value:w \__fp_int_eval:w #6 + 19999 - #1 ;
19347   \exp_after:wN ;
19348 }

```

(End definition for __fp_sqrt_auxxiv_o:wnnnnnnnnN.)

31.5 About the sign and exponent

__fp_logb_o:w The exponent of a normal number is its *exponent* minus one.
__fp_logb_aux_o:w

```

19349 \cs_new:Npn \__fp_logb_o:w ? \s__fp \__fp_chk:w #1#2; @
19350 {
19351   \if_case:w #1 \exp_stop_f:
19352   \__fp_case_use:nw
19353   { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { logb } }
19354   \or: \exp_after:wN \__fp_logb_aux_o:w
19355   \or: \__fp_case_return_o:Nw \c_inf_fp
19356   \else: \__fp_case_return_same_o:w
19357   \fi:
19358   \s__fp \__fp_chk:w #1 #2;
19359 }

```

```

19360 \cs_new:Npn \__fp_logb_aux_o:w \s__fp \__fp_chk:w #1 #2 #3 #4 ;
19361 {
19362   \exp_after:wN \__fp_parse:n \exp_after:wN
19363   { \int_value:w \int_eval:w #3 - 1 \exp_after:wN }
19364 }

```

(End definition for __fp_logb_o:w and __fp_logb_aux_o:w.)

```

\__fp_sign_o:w Find the sign of the floating point: nan, +0, -0, +1 or -1.
\__fp_sign_aux_o:w
19365 \cs_new:Npn \__fp_sign_o:w ? \s__fp \__fp_chk:w #1#2; @
19366 {
19367   \if_case:w #1 \exp_stop_f:
19368     \__fp_case_return_same_o:w
19369   \or: \exp_after:wN \__fp_sign_aux_o:w
19370   \or: \exp_after:wN \__fp_sign_aux_o:w
19371   \else: \__fp_case_return_same_o:w
19372   \fi:
19373   \s__fp \__fp_chk:w #1 #2;
19374 }
19375 \cs_new:Npn \__fp_sign_aux_o:w \s__fp \__fp_chk:w #1 #2 #3 ;
19376 { \exp_after:wN \__fp_set_sign_o:w \exp_after:wN #2 \c_one_fp @ }

```

(End definition for __fp_sign_o:w and __fp_sign_aux_o:w.)

__fp_set_sign_o:w This function is used for the unary minus and for `abs`. It leaves the sign of `nan` invariant, turns negative numbers (sign 2) to positive numbers (sign 0) and positive numbers (sign 0) to positive or negative numbers depending on #1. It also expands after itself in the input stream, just like __fp_+_o:ww.

```

19377 \cs_new:Npn \__fp_set_sign_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
19378 {
19379   \exp_after:wN \__fp_exp_after_o:w
19380   \exp_after:wN \s__fp
19381   \exp_after:wN \__fp_chk:w
19382   \exp_after:wN #2
19383   \int_value:w
19384   \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
19385   #4;
19386 }

```

(End definition for __fp_set_sign_o:w.)

31.6 Operations on tuples

__fp_tuple_set_sign_o:w Two cases: `abs(<tuple>)` for which #1 is 0 (invalid for tuples) and `-<tuple>` for which #1 is 2. In that case, map over all items in the tuple an auxiliary that dispatches to the type-appropriate sign-flipping function.

```

\__fp_tuple_set_sign_o:w
\__fp_tuple_set_sign_aux_o:Nnw
\__fp_tuple_set_sign_aux_o:w
19387 \cs_new:Npn \__fp_tuple_set_sign_o:w #1
19388 {
19389   \if_meaning:w 2 #1
19390     \exp_after:wN \__fp_tuple_set_sign_aux_o:Nnw
19391   \fi:
19392   \__fp_invalid_operation_o:nw { abs }
19393 }
19394 \cs_new:Npn \__fp_tuple_set_sign_aux_o:Nnw #1#2#3 @

```

```

19395 { \_fp_tuple_map_o:nw \_fp_tuple_set_sign_aux_o:w #3 }
19396 \cs_new:Npn \_fp_tuple_set_sign_aux_o:w #1#2 ;
19397 {
19398   \_fp_change_func_type:NNN #1 \_fp_set_sign_o:w
19399   \_fp_parse_apply_unary_error:NNw
19400   2 #1 #2 ; @
19401 }

```

(End definition for `_fp_tuple_set_sign_o:w`, `_fp_tuple_set_sign_aux_o:Nnw`, and `_fp_tuple_set_sign_aux_o:w`.)

`_fp*_tuple_o:ww` For $\langle number \rangle * \langle tuple \rangle$ and $\langle tuple \rangle * \langle number \rangle$ and $\langle tuple \rangle / \langle number \rangle$, loop through the `_fp_tuple*_o:ww` $\langle tuple \rangle$ some code that multiplies or divides by the appropriate $\langle number \rangle$. Importantly `_fp_tuple/_o:ww` we need to dispatch according to the type, and we make sure to apply the operator in the correct order.

```

19402 \cs_new:cpn { \_fp*_tuple_o:ww } #1 ;
19403 { \_fp_tuple_map_o:nw { \_fp_binary_type_o:Nww * #1 ; } }
19404 \cs_new:cpn { \_fp_tuple*_o:ww } #1 ; #2 ;
19405 { \_fp_tuple_map_o:nw { \_fp_binary_rev_type_o:Nww * #2 ; } #1 ; }
19406 \cs_new:cpn { \_fp_tuple/_o:ww } #1 ; #2 ;
19407 { \_fp_tuple_map_o:nw { \_fp_binary_rev_type_o:Nww / #2 ; } #1 ; }

```

(End definition for `_fp*_tuple_o:ww`, `_fp_tuple*_o:ww`, and `_fp_tuple/_o:ww`.)

`_fp_tuple+_tuple_o:ww` Check the two tuples have the same number of items and map through these a helper `_fp_tuple-_tuple_o:ww` that dispatches appropriately depending on the types. This means $(1,2) + ((1,1),2)$ gives $(\text{nan},4)$.

```

19408 \cs_set_protected:Npn \_fp_tmp:w #1
19409 {
19410   \cs_new:cpn { \_fp_tuple_#1_tuple_o:ww }
19411   \s_fp_tuple \_fp_tuple_chk:w ##1 ;
19412   \s_fp_tuple \_fp_tuple_chk:w ##2 ;
19413   {
19414     \int_compare:nNnTF
19415     { \_fp_array_count:n {##1} } = { \_fp_array_count:n {##2} }
19416     { \_fp_tuple_mapthread_o:nww { \_fp_binary_type_o:Nww #1 } }
19417     { \_fp_invalid_operation_o:nww #1 }
19418     \s_fp_tuple \_fp_tuple_chk:w {##1} ;
19419     \s_fp_tuple \_fp_tuple_chk:w {##2} ;
19420   }
19421 }
19422 \_fp_tmp:w +
19423 \_fp_tmp:w -

```

(End definition for `_fp_tuple+_tuple_o:ww` and `_fp_tuple-_tuple_o:ww`.)

```

19424 \</package>

```

32 l3fp-extended implementation

```

19425 (*package)
19426 @@=fp)

```


32.1 Description of fixed point numbers

This module provides a few functions to manipulate positive floating point numbers with extended precision (24 digits), but mostly provides functions for fixed-point numbers with this precision (24 digits). Those are used in the computation of Taylor series for the logarithm, exponential, and trigonometric functions. Since we eventually only care about the 16 first digits of the final result, some of the calculations are not performed with the full 24-digit precision. In other words, the last two blocks of each fixed point number may be wrong as long as the error is small enough to be rounded away when converting back to a floating point number. The fixed point numbers are expressed as

$$\{\langle a_1 \rangle\} \{\langle a_2 \rangle\} \{\langle a_3 \rangle\} \{\langle a_4 \rangle\} \{\langle a_5 \rangle\} \{\langle a_6 \rangle\} ;$$

where each $\langle a_i \rangle$ is exactly 4 digits (ranging from 0000 to 9999), except $\langle a_1 \rangle$, which may be any “not-too-large” non-negative integer, with or without leading zeros. Here, “not-too-large” depends on the specific function (see the corresponding comments for details). Checking for overflow is the responsibility of the code calling those functions. The fixed point number a corresponding to the representation above is $a = \sum_{i=1}^6 \langle a_i \rangle \cdot 10^{-4i}$.

Most functions we define here have the form

```
\__fp_fixed_⟨calculation⟩:wnn ⟨operand1⟩ ; ⟨operand2⟩ ; {⟨continuation⟩}
```

They perform the $\langle calculation \rangle$ on the two $\langle operands \rangle$, then feed the result (6 brace groups followed by a semicolon) to the $\langle continuation \rangle$, responsible for the next step of the calculation. Some functions only accept an N-type $\langle continuation \rangle$. This allows constructions such as

```
\__fp_fixed_add:wnn ⟨X1⟩ ; ⟨X2⟩ ;
\__fp_fixed_mul:wnn ⟨X3⟩ ;
\__fp_fixed_add:wnn ⟨X4⟩ ;
```

to compute $(X_1 + X_2) \cdot X_3 + X_4$. This turns out to be very appropriate for computing continued fractions and Taylor series.

At the end of the calculation, the result is turned back to a floating point number using `__fp_fixed_to_float_o:wn`. This function has to change the exponent of the floating point number: it must be used after starting an integer expression for the overall exponent of the result.

32.2 Helpers for numbers with extended precision

`\c__fp_one_fixed_t1` The fixed-point number 1, used in `l3fp-expo`.

```
19427 \tl_const:Nn \c__fp_one_fixed_t1
19428 { {10000} {0000} {0000} {0000} {0000} {0000} ; }
```

(End definition for `\c__fp_one_fixed_t1`.)

`__fp_fixed_continue:wn` This function simply calls the next function.

```
19429 \cs_new:Npn \__fp_fixed_continue:wn #1; #2 { #2 #1; }
```

(End definition for `__fp_fixed_continue:wn`.)

`__fp_fixed_add_one:wn`

`__fp_fixed_add_one:wn <a> ; <continuation>`

This function adds 1 to the fixed point $\langle a \rangle$, by changing a_1 to $10000 + a_1$, then calls the $\langle continuation \rangle$. This requires $a_1 + 10000 < 2^{31}$.

```

19430 \cs_new:Npn \__fp_fixed_add_one:wn #1#2; #3
19431 {
19432   \exp_after:wn #3 \exp_after:wn
19433   { \int_value:w \__fp_int_eval:w \c__fp_myriad_int + #1 } #2 ;
19434 }

```

(End definition for `__fp_fixed_add_one:wn`.)

`__fp_fixed_div_myriad:wn`

Divide a fixed point number by 10000. This is a little bit more subtle than just removing the last group and adding a leading group of zeros: the first group #1 may have any number of digits, and we must split #1 into the new first group and a second group of exactly 4 digits. The choice of shifts allows #1 to be in the range $[0, 5 \cdot 10^8 - 1]$.

```

19435 \cs_new:Npn \__fp_fixed_div_myriad:wn #1#2#3#4#5#6;
19436 {
19437   \exp_after:wn \__fp_fixed_mul_after:wnn
19438   \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
19439   \exp_after:wn \__fp_pack:NNNNNw
19440   \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
19441   + #1 ; {#2}{#3}{#4}{#5};
19442 }

```

(End definition for `__fp_fixed_div_myriad:wn`.)

`__fp_fixed_mul_after:wnn`

The fixed point operations which involve multiplication end by calling this auxiliary. It braces the last block of digits, and places the $\langle continuation \rangle$ #3 in front.

```

19443 \cs_new:Npn \__fp_fixed_mul_after:wnn #1; #2; #3 { #3 {#1} #2; }

```

(End definition for `__fp_fixed_mul_after:wnn`.)

32.3 Multiplying a fixed point number by a short one

`__fp_fixed_mul_short:wnn`

```

\__fp_fixed_mul_short:wnn
{ \langle a_1 \rangle \langle a_2 \rangle \langle a_3 \rangle \langle a_4 \rangle \langle a_5 \rangle \langle a_6 \rangle ;
  \langle b_0 \rangle \langle b_1 \rangle \langle b_2 \rangle ; \langle continuation \rangle }

```

Computes the product $c = ab$ of $a = \sum_i \langle a_i \rangle 10^{-4i}$ and $b = \sum_i \langle b_i \rangle 10^{-4i}$, rounds it to the closest multiple of 10^{-24} , and leaves $\langle continuation \rangle \{ \langle c_1 \rangle \} \dots \{ \langle c_6 \rangle \}$; in the input stream, where each of the $\langle c_i \rangle$ are blocks of 4 digits, except $\langle c_1 \rangle$, which is any \TeX integer. Note that indices for $\langle b \rangle$ start at 0: for instance a second operand of $\{0001\}\{0000\}\{0000\}$ leaves the first operand unchanged (rather than dividing it by 10^4 , as `__fp_fixed_mul:wnn` would).

```

19444 \cs_new:Npn \__fp_fixed_mul_short:wnn #1#2#3#4#5#6; #7#8#9;
19445 {
19446   \exp_after:wn \__fp_fixed_mul_after:wnn
19447   \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
19448   + #1*#7
19449   \exp_after:wn \__fp_pack:NNNNNw
19450   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
19451   + #1*#8 + #2*#7
19452   \exp_after:wn \__fp_pack:NNNNNw
19453   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int

```

```

19454      + #1*#9 + #2*#8 + #3*#7
19455      \exp_after:wN \__fp_pack:NNNNNw
19456      \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
19457      + #2*#9 + #3*#8 + #4*#7
19458      \exp_after:wN \__fp_pack:NNNNNw
19459      \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
19460      + #3*#9 + #4*#8 + #5*#7
19461      \exp_after:wN \__fp_pack:NNNNNw
19462      \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
19463      + #4*#9 + #5*#8 + #6*#7
19464      + ( #5*#9 + #6*#8 + #6*#9 / \c__fp_myriad_int )
19465      / \c__fp_myriad_int ; ;
19466  }

```

(End definition for __fp_fixed_mul_short:wnn.)

32.4 Dividing a fixed point number by a small integer

```

\__fp_fixed_div_int:wnN \__fp_fixed_div_int:wnN <a> ; <n> ; <continuation>

```

Divides the fixed point number $\langle a \rangle$ by the (small) integer $0 < \langle n \rangle < 10^4$ and feeds the result to the $\langle continuation \rangle$. There is no bound on a_1 .

```

\__fp_fixed_div_int_auxi:wnn

```

The arguments of the **i** auxiliary are 1: one of the a_i , 2: n , 3: the **ii** or the **iii** auxiliary. It computes a (somewhat tight) lower bound Q_i for the ratio a_i/n .

```

\__fp_fixed_div_int_pack:Nw

```

The **ii** auxiliary receives Q_i , n , and a_i as arguments. It adds Q_i to a surrounding integer expression, and starts a new one with the initial value 9999, which ensures that the result of this expression has 5 digits. The auxiliary also computes $a_i - n \cdot Q_i$, placing the result in front of the 4 digits of a_{i+1} . The resulting $a'_{i+1} = 10^4(a_i - n \cdot Q_i) + a_{i+1}$ serves as the first argument for a new call to the **i** auxiliary.

```

\__fp_fixed_div_int_after:Nw

```

When the **iii** auxiliary is called, the situation looks like this:

```

\__fp_fixed_div_int_after:Nw <continuation>
-1 + Q1
\__fp_fixed_div_int_pack:Nw 9999 + Q2
\__fp_fixed_div_int_pack:Nw 9999 + Q3
\__fp_fixed_div_int_pack:Nw 9999 + Q4
\__fp_fixed_div_int_pack:Nw 9999 + Q5
\__fp_fixed_div_int_pack:Nw 9999
\__fp_fixed_div_int_auxii:wnn Q6 ; {<n>} {<a6

```

where expansion is happening from the last line up. The **iii** auxiliary adds $Q_6 + 2 \simeq a_6/n + 1$ to the last 9999, giving the integer closest to $10000 + a_6/n$.

Each **pack** auxiliary receives 5 digits followed by a semicolon. The first digit is added as a carry to the integer expression above, and the 4 other digits are braced. Each call to the **pack** auxiliary thus produces one brace group. The last brace group is produced by the **after** auxiliary, which places the $\langle continuation \rangle$ as appropriate.

```

19467 \cs_new:Npn \__fp_fixed_div_int:wnN #1#2#3#4#5#6 ; #7 ; #8
19468 {
19469   \exp_after:wN \__fp_fixed_div_int_after:Nw
19470   \exp_after:wN #8
19471   \int_value:w \__fp_int_eval:w - 1
19472   \__fp_fixed_div_int:wnN
19473   #1; {#7} \__fp_fixed_div_int_auxi:wnn

```

```

19474      #2; {#7} \__fp_fixed_div_int_auxi:wnn
19475      #3; {#7} \__fp_fixed_div_int_auxi:wnn
19476      #4; {#7} \__fp_fixed_div_int_auxi:wnn
19477      #5; {#7} \__fp_fixed_div_int_auxi:wnn
19478      #6; {#7} \__fp_fixed_div_int_auxii:wnn ;
19479    }
19480 \cs_new:Npn \__fp_fixed_div_int:wnN #1; #2 #3
19481 {
19482   \exp_after:wN #3
19483   \int_value:w \__fp_int_eval:w #1 / #2 - 1 ;
19484   {#2}
19485   {#1}
19486 }
19487 \cs_new:Npn \__fp_fixed_div_int_auxi:wnn #1; #2 #3
19488 {
19489   + #1
19490   \exp_after:wN \__fp_fixed_div_int_pack:Nw
19491   \int_value:w \__fp_int_eval:w 9999
19492   \exp_after:wN \__fp_fixed_div_int:wnN
19493   \int_value:w \__fp_int_eval:w #3 - #1*#2 \__fp_int_eval_end:
19494 }
19495 \cs_new:Npn \__fp_fixed_div_int_auxii:wnn #1; #2 #3 { + #1 + 2 ; }
19496 \cs_new:Npn \__fp_fixed_div_int_pack:Nw #1 #2; { + #1; {#2} }
19497 \cs_new:Npn \__fp_fixed_div_int_after:Nw #1 #2; { #1 {#2} }

```

(End definition for __fp_fixed_div_int:wnN and others.)

32.5 Adding and subtracting fixed points

`__fp_fixed_add:wnn` `__fp_fixed_add:wnn <a> ; ; {<continuation>}`
`__fp_fixed_sub:wnn` Computes $a + b$ (resp. $a - b$) and feeds the result to the $\langle continuation \rangle$. This function
`__fp_fixed_add:Nnnnnwnn` requires $0 \leq a_1, b_1 \leq 114748$, its result must be positive (this happens automatically for
`__fp_fixed_add:nnNnnwnn` addition) and its first group must have at most 5 digits: $(a \pm b)_1 < 100000$. The two
`__fp_fixed_add_pack:NNNNNwn` functions only differ by a sign, hence use a common auxiliary. It would be nice to grab
`__fp_fixed_add_after:NNNNNwn` the 12 brace groups in one go; only 9 parameters are allowed. Start by grabbing the sign,
 a_1, \dots, a_4 , the rest of a , and b_1 and b_2 . The second auxiliary receives the rest of a , the
the sign multiplying b , the rest of b , and the $\langle continuation \rangle$ as arguments. After going down
through the various level, we go back up, packing digits and bringing the $\langle continuation \rangle$
(#8, then #7) from the end of the argument list to its start.

```

19498 \cs_new:Npn \__fp_fixed_add:wnn { \__fp_fixed_add:Nnnnnwnn + }
19499 \cs_new:Npn \__fp_fixed_sub:wnn { \__fp_fixed_add:Nnnnnwnn - }
19500 \cs_new:Npn \__fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6; #7#8
19501 {
19502   \exp_after:wN \__fp_fixed_add_after:NNNNNwn
19503   \int_value:w \__fp_int_eval:w 9 9999 9998 + #2#3 #1 #7#8
19504   \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
19505   \int_value:w \__fp_int_eval:w 1 9999 9998 + #4#5
19506   \__fp_fixed_add:nnNnnwn #6 #1
19507 }
19508 \cs_new:Npn \__fp_fixed_add:nnNnnwn #1#2 #3 #4#5 #6#7 ; #8
19509 {
19510   #3 #4#5
19511   \exp_after:wN \__fp_fixed_add_pack:NNNNNwn

```

```

19512 \int_value:w \__fp_int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
19513 }
19514 \cs_new:Npn \__fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6; #7
19515 { + #1 ; {#7} {#2#3#4#5} {#6} }
19516 \cs_new:Npn \__fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6; #7
19517 { #7 {#1#2#3#4#5} {#6} }

```

(End definition for `__fp_fixed_add:wnn` and others.)

32.6 Multiplying fixed points

```

\__fp_fixed_mul:wnn
\__fp_fixed_mul:nnnnnnnw

```

`__fp_fixed_mul:wnn` $\langle a \rangle$; $\langle b \rangle$; $\{\langle continuation \rangle\}$

Computes $a \times b$ and feeds the result to $\langle continuation \rangle$. This function requires $0 \leq a_1, b_1 < 10000$. Once more, we need to play around the limit of 9 arguments for $\text{T}_{\text{E}}\text{X}$ macros. Note that we don't need to obtain an exact rounding, contrarily to the `*` operator, so things could be harder. We wish to perform carries in

$$\begin{aligned}
a \times b = & a_1 \cdot b_1 \cdot 10^{-8} \\
& + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
& + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1) \cdot 10^{-16} \\
& + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
& + \left(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \right. \\
& \quad \left. + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \right. \\
& \quad \left. + a_1 \cdot b_5 + a_5 \cdot b_1 \right) \cdot 10^{-24} + O(10^{-24}),
\end{aligned}$$

where the $O(10^{-24})$ stands for terms which are at most $5 \cdot 10^{-24}$; ignoring those leads to an error of at most 5 ulp. Note how the first 15 terms only depend on a_1, \dots, a_4 and b_1, \dots, b_4 , while the last 6 terms only depend on a_1, a_2, a_5, a_6 , and the corresponding parts of b . Hence, the first function grabs a_1, \dots, a_4 , the rest of a , and b_1, \dots, b_4 , and writes the 15 first terms of the expression, including a left parenthesis for the fraction. The `i` auxiliary receives $a_5, a_6, b_1, b_2, a_1, a_2, b_5, b_6$ and finally the $\langle continuation \rangle$ as arguments. It writes the end of the expression, including the right parenthesis and the denominator of the fraction. The $\langle continuation \rangle$ is finally placed in front of the 6 brace groups by `__fp_fixed_mul_after:wnn`.

```

19518 \cs_new:Npn \__fp_fixed_mul:wnn #1#2#3#4 #5; #6#7#8#9
19519 {
19520   \exp_after:wN \__fp_fixed_mul_after:wnn
19521   \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
19522   \exp_after:wN \__fp_pack:NNNNNw
19523   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
19524   + #1*#6
19525   \exp_after:wN \__fp_pack:NNNNNw
19526   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
19527   + #1*#7 + #2*#6
19528   \exp_after:wN \__fp_pack:NNNNNw
19529   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
19530   + #1*#8 + #2*#7 + #3*#6
19531   \exp_after:wN \__fp_pack:NNNNNw

```

```

19532         \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
19533         + #1*#9 + #2*#8 + #3*#7 + #4*#6
19534         \exp_after:wN \__fp_pack:NNNNNw
19535         \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
19536         + #2*#9 + #3*#8 + #4*#7
19537         + ( #3*#9 + #4*#8
19538         + \__fp_fixed_mul:nnnnnnnw #5 {#6}{#7} {#1}{#2}
19539     }
19540 \cs_new:Npn \__fp_fixed_mul:nnnnnnnw #1#2 #3#4 #5#6 #7#8 ;
19541 {
19542     #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c__fp_myriad_int
19543     + #1*#3 + #5*#7 ; ;
19544 }

```

(End definition for `__fp_fixed_mul:wnn` and `__fp_fixed_mul:nnnnnnnw`.)

32.7 Combining product and sum of fixed points

```

\__fp_fixed_mul_add:wwwn
\__fp_fixed_mul_sub_back:wwwn
\__fp_fixed_one_minus_mul:wnn

```

Sometimes called FMA (fused multiply-add), these functions compute $a \times b + c$, $c - a \times b$, and $1 - a \times b$ and feed the result to the `\langle continuation \rangle`. Those functions require $0 \leq a_1, b_1, c_1 \leq 10000$. Since those functions are at the heart of the computation of Taylor expansions, we over-optimize them a bit, and in particular we do not factor out the common parts of the three functions.

For definiteness, consider the task of computing $a \times b + c$. We perform carries in

$$\begin{aligned}
 a \times b + c = & (a_1 \cdot b_1 + c_1 c_2) \cdot 10^{-8} \\
 & + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
 & + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4) \cdot 10^{-16} \\
 & + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
 & + \left(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \right. \\
 & \quad \left. + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \right. \\
 & \quad \left. + a_1 \cdot b_5 + a_5 \cdot b_1 + c_5 c_6 \right) \cdot 10^{-24} + O(10^{-24}),
 \end{aligned}$$

where $c_1 c_2$, $c_3 c_4$, $c_5 c_6$ denote the 8-digit number obtained by juxtaposing the two blocks of digits of c , and \cdot denotes multiplication. The task is obviously tough because we have 18 brace groups in front of us.

Each of the three function starts the first two levels (the first, corresponding to 10^{-4} , is empty), with $c_1 c_2$ in the first level, calls the `i` auxiliary with arguments described later, and adds a trailing $+ c_5 c_6 ; \{\langle continuation \rangle\}$; . The $+ c_5 c_6$ piece, which is omitted for `__fp_fixed_one_minus_mul:wnn`, is taken in the integer expression for the 10^{-24} level.

```

19545 \cs_new:Npn \__fp_fixed_mul_add:wwwn #1; #2; #3#4#5#6#7#8;
19546 {
19547     \exp_after:wN \__fp_fixed_mul_after:wwn
19548     \int_value:w \__fp_int_eval:w \c__fp_big_leading_shift_int
19549     \exp_after:wN \__fp_pack_big:NNNNNNw

```

```

19550      \int_value:w \_fp_int_eval:w \c\_fp_big_middle_shift_int + #3 #4
19551      \_fp_fixed_mul_add:Nwnnnwnnn +
19552      + #5 #6 ; #2 ; #1 ; #2 ; +
19553      + #7 #8 ; ;
19554    }
19555 \cs_new:Npn \_fp_fixed_mul_sub_back:wwn #1; #2; #3#4#5#6#7#8;
19556 {
19557   \exp_after:wN \_fp_fixed_mul_after:wwn
19558   \int_value:w \_fp_int_eval:w \c\_fp_big_leading_shift_int
19559   \exp_after:wN \_fp_pack_big:NNNNNNw
19560   \int_value:w \_fp_int_eval:w \c\_fp_big_middle_shift_int + #3 #4
19561   \_fp_fixed_mul_add:Nwnnnwnnn -
19562   + #5 #6 ; #2 ; #1 ; #2 ; -
19563   + #7 #8 ; ;
19564 }
19565 \cs_new:Npn \_fp_fixed_one_minus_mul:wwn #1; #2;
19566 {
19567   \exp_after:wN \_fp_fixed_mul_after:wwn
19568   \int_value:w \_fp_int_eval:w \c\_fp_big_leading_shift_int
19569   \exp_after:wN \_fp_pack_big:NNNNNNw
19570   \int_value:w \_fp_int_eval:w \c\_fp_big_middle_shift_int +
19571   1 0000 0000
19572   \_fp_fixed_mul_add:Nwnnnwnnn -
19573   ; #2 ; #1 ; #2 ; -
19574   ; ;
19575 }

```

(End definition for `_fp_fixed_mul_add:wwn`, `_fp_fixed_mul_sub_back:wwn`, and `_fp_fixed_mul_one_minus_mul:wwn`.)

```

\_fp_fixed_mul_add:Nwnnnwnnn
      \_fp_fixed_mul_add:Nwnnnwnnn <op> + <c3> <c4> ;
      <b> ; <a> ; <b> ; <op>
      + <c5> <c6> ;

```

Here, `<op>` is either `+` or `-`. Arguments `#3`, `#4`, `#5` are $\langle b_1 \rangle$, $\langle b_2 \rangle$, $\langle b_3 \rangle$; arguments `#7`, `#8`, `#9` are $\langle a_1 \rangle$, $\langle a_2 \rangle$, $\langle a_3 \rangle$. We can build three levels: $a_1 \cdot b_1$ for 10^{-8} , $(a_1 \cdot b_2 + a_2 \cdot b_1)$ for 10^{-12} , and $(a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4)$ for 10^{-16} . The a - b products use the sign `#1`. Note that `#2` is empty for `_fp_fixed_one_minus_mul:wwn`. We call the `ii` auxiliary for levels 10^{-20} and 10^{-24} , keeping the pieces of $\langle a \rangle$ we've read, but not $\langle b \rangle$, since there is another copy later in the input stream.

```

19576 \cs_new:Npn \_fp_fixed_mul_add:Nwnnnwnnn #1 #2; #3#4#5#6; #7#8#9
19577 {
19578   #1 #7*#3
19579   \exp_after:wN \_fp_pack_big:NNNNNNw
19580   \int_value:w \_fp_int_eval:w \c\_fp_big_middle_shift_int
19581   #1 #7*#4 #1 #8*#3
19582   \exp_after:wN \_fp_pack_big:NNNNNNw
19583   \int_value:w \_fp_int_eval:w \c\_fp_big_middle_shift_int
19584   #1 #7*#5 #1 #8*#4 #1 #9*#3 #2
19585   \exp_after:wN \_fp_pack_big:NNNNNNw
19586   \int_value:w \_fp_int_eval:w \c\_fp_big_middle_shift_int
19587   #1 \_fp_fixed_mul_add:nnnnwnnn {#7}{#8}{#9}
19588 }

```

(End definition for `_fp_fixed_mul_add:Nwnnnwnnn`.)

_fp_fixed_mul_add:nnnnwnnnn

_fp_fixed_mul_add:nnnnwnnnn $\langle a \rangle$; $\langle b \rangle$; $\langle op \rangle$
 $+ \langle c_5 \rangle \langle c_6 \rangle$;

Level 10^{-20} is $(a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1)$, multiplied by the sign, which was inserted by the *i* auxiliary. Then we prepare level 10^{-24} . We don't have access to all parts of $\langle a \rangle$ and $\langle b \rangle$ needed to make all products. Instead, we prepare the partial expressions

$$b_1 + a_4 \cdot b_2 + a_3 \cdot b_3 + a_2 \cdot b_4 + a_1$$

$$b_2 + a_4 \cdot b_3 + a_3 \cdot b_4 + a_2.$$

Obviously, those expressions make no mathematical sense: we complete them with $a_5 \cdot$ and $\cdot b_5$, and with $a_6 \cdot b_1 + a_5 \cdot$ and $\cdot b_5 + a_1 \cdot b_6$, and of course with the trailing $+ c_5 c_6$. To do all this, we keep a_1, a_5, a_6 , and the corresponding pieces of $\langle b \rangle$.

```

19589 \cs_new:Npn \_fp_fixed_mul_add:nnnnwnnnn #1#2#3#4#5; #6#7#8#9
19590 {
19591   ( #1*#9 + #2*#8 + #3*#7 + #4*#6 )
19592   \exp_after:wN \_fp_pack_big:NNNNNNw
19593   \int_value:w \_fp_int_eval:w \c_fp_big_trailing_shift_int
19594   \_fp_fixed_mul_add:nnnnwnnwN
19595   { #6 + #4*#7 + #3*#8 + #2*#9 + #1 }
19596   { #7 + #4*#8 + #3*#9 + #2 }
19597   {#1} #5;
19598   {#6}
19599 }
```

(End definition for _fp_fixed_mul_add:nnnnwnnnn.)

_fp_fixed_mul_add:nnnnwnnwN

_fp_fixed_mul_add:nnnnwnnwN $\{\langle partial_1 \rangle\} \{\langle partial_2 \rangle\}$
 $\{\langle a_1 \rangle\} \{\langle a_5 \rangle\} \{\langle a_6 \rangle\}$; $\{\langle b_1 \rangle\} \{\langle b_5 \rangle\} \{\langle b_6 \rangle\}$;
 $\langle op \rangle + \langle c_5 \rangle \langle c_6 \rangle$;

Complete the $\langle partial_1 \rangle$ and $\langle partial_2 \rangle$ expressions as explained for the *ii* auxiliary. The second one is divided by 10000: this is the carry from level 10^{-28} . The trailing $+ c_5 c_6$ is taken into the expression for level 10^{-24} . Note that the total of level 10^{-24} is in the interval $[-5 \cdot 10^8, 6 \cdot 10^8]$ (give or take a couple of 10000), hence adding it to the shift gives a 10-digit number, as expected by the packing auxiliaries. See *l3fp-aux* for the definition of the shifts and packing auxiliaries.

```

19600 \cs_new:Npn \_fp_fixed_mul_add:nnnnwnnwN #1#2 #3#4#5; #6#7#8; #9
19601 {
19602   #9 (#4* #1 *#7)
19603   #9 (#5*#6+#4* #2 *#7+#3*#8) / \c_fp_myriad_int
19604 }
```

(End definition for _fp_fixed_mul_add:nnnnwnnwN.)

32.8 Extended-precision floating point numbers

In this section we manipulate floating point numbers with roughly 24 significant figures (“extended-precision” numbers, in short, “ep”), which take the form of an integer exponent, followed by a comma, then six groups of digits, ending with a semicolon. The first group of digit may be any non-negative integer, while other groups of digits have 4 digits. In other words, an extended-precision number is an exponent ending in a comma, then a fixed point number. The corresponding value is $0.\langle digits \rangle \cdot 10^{\langle exponent \rangle}$. This convention differs from floating points.

`__fp_ep_to_fixed:wwn` Converts an extended-precision number with an exponent at most 4 and a first block less than 10^8 to a fixed point number whose first block has 12 digits, hopefully starting with many zeros.
`__fp_ep_to_fixed_auxi:www`
`__fp_ep_to_fixed_auxii:nnnnnnnwn`

```

19605 \cs_new:Npn __fp_ep_to_fixed:wwn #1,#2
19606 {
19607   \exp_after:wN __fp_ep_to_fixed_auxi:www
19608   \int_value:w __fp_int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
19609   \exp:w \exp_end_continue_f:w
19610   \prg_replicate:nn { 4 - \int_max:nn {#1} { -32 } } { 0 } ;
19611 }
19612 \cs_new:Npn __fp_ep_to_fixed_auxi:www 1#1; #2; #3#4#5#6#7;
19613 {
19614   __fp_pack_eight:wnnnnnnnn
19615   __fp_pack_twice_four:wnnnnnnnn
19616   __fp_pack_twice_four:wnnnnnnnn
19617   __fp_pack_twice_four:wnnnnnnnn
19618   __fp_ep_to_fixed_auxii:nnnnnnnwn ;
19619   #2 #1#3#4#5#6#7 0000 !
19620 }
19621 \cs_new:Npn __fp_ep_to_fixed_auxii:nnnnnnnwn #1#2#3#4#5#6#7; #8! #9
19622 { #9 {#1#2}{#3}{#4}{#5}{#6}{#7}; }

```

(End definition for `__fp_ep_to_fixed:wwn`, `__fp_ep_to_fixed_auxi:www`, and `__fp_ep_to_fixed_auxii:nnnnnnnwn`.)

`__fp_ep_to_ep:wwN` Normalize an extended-precision number. More precisely, leading zeros are removed from the mantissa of the argument, decreasing its exponent as appropriate. Then the digits are packed into 6 groups of 4 (discarding any remaining digit, not rounding). Finally, the continuation #8 is placed before the resulting exponent-mantissa pair. The input exponent may in fact be given as an integer expression. The `loop` auxiliary grabs a digit: if it is 0, decrement the exponent and continue looping, and otherwise call the `end` auxiliary, which places all digits in the right order (the digit that was not 0, and any remaining digits), followed by some 0, then packs them up neatly in $3 \times 2 = 6$ blocks of four. At the end of the day, remove with `__fp_use_i:ww` any digit that did not make it in the final mantissa (typically only zeros, unless the original first block has more than 4 digits).
`__fp_ep_to_ep_loop:N`
`__fp_ep_to_ep_end:www`
`__fp_ep_to_ep_zero:ww`

```

19623 \cs_new:Npn __fp_ep_to_ep:wwN #1,#2#3#4#5#6#7; #8
19624 {
19625   \exp_after:wN #8
19626   \int_value:w __fp_int_eval:w #1 + 4
19627   \exp_after:wN \use_i:nn
19628   \exp_after:wN __fp_ep_to_ep_loop:N
19629   \int_value:w __fp_int_eval:w 1 0000 0000 + #2 __fp_int_eval_end:
19630   #3#4#5#6#7 ; ; !
19631 }
19632 \cs_new:Npn __fp_ep_to_ep_loop:N #1
19633 {
19634   \if_meaning:w 0 #1
19635   - 1
19636   \else:
19637     __fp_ep_to_ep_end:www #1
19638   \fi:
19639   __fp_ep_to_ep_loop:N

```

```

19640 }
19641 \cs_new:Npn \__fp_ep_to_ep_end:www
19642 #1 \fi: \__fp_ep_to_ep_loop:N #2; #3!
19643 {
19644   \fi:
19645   \if_meaning:w ; #1
19646     - 2 * \c_fp_max_exponent_int
19647     \__fp_ep_to_ep_zero:ww
19648   \fi:
19649   \__fp_pack_twice_four:wNNNNNNNN
19650   \__fp_pack_twice_four:wNNNNNNNN
19651   \__fp_pack_twice_four:wNNNNNNNN
19652   \__fp_use_i:ww , ;
19653   #1 #2 0000 0000 0000 0000 0000 0000 ;
19654 }
19655 \cs_new:Npn \__fp_ep_to_ep_zero:ww \fi: #1; #2; #3;
19656 { \fi: , {1000}{0000}{0000}{0000}{0000}{0000} ; }

```

(End definition for __fp_ep_to_ep:wwN and others.)

__fp_ep_compare:www
__fp_ep_compare_aux:www

In l3fp-trig we need to compare two extended-precision numbers. This is based on the same function for positive floating point numbers, with an extra test if comparing only 16 decimals is not enough to distinguish the numbers. Note that this function only works if the numbers are normalized so that their first block is in [1000,9999].

```

19657 \cs_new:Npn \__fp_ep_compare:www #1,#2#3#4#5#6#7;
19658 { \__fp_ep_compare_aux:www {#1}{#2}{#3}{#4}{#5}; #6#7; }
19659 \cs_new:Npn \__fp_ep_compare_aux:www #1;#2;#3;#4#5#6#7#8#9;
19660 {
19661   \if_case:w
19662     \__fp_compare_npos:nwnw #1; {#3}{#4}{#5}{#6}{#7}; \exp_stop_f:
19663     \if_int_compare:w #2 = #8#9 \exp_stop_f:
19664       0
19665     \else:
19666       \if_int_compare:w #2 < #8#9 - \fi: 1
19667     \fi:
19668   \or: 1
19669   \else: -1
19670   \fi:
19671 }

```

(End definition for __fp_ep_compare:www and __fp_ep_compare_aux:www.)

__fp_ep_mul:wwwN
__fp_ep_mul_raw:wwwN

Multiply two extended-precision numbers: first normalize them to avoid losing too much precision, then multiply the mantissas #2 and #4 as fixed point numbers, and sum the exponents #1 and #3. The result's first block is in [100,9999].

```

19672 \cs_new:Npn \__fp_ep_mul:wwwN #1,#2; #3,#4;
19673 {
19674   \__fp_ep_to_ep:wwN #3,#4;
19675   \__fp_fixed_continue:wn
19676   {
19677     \__fp_ep_to_ep:wwN #1,#2;
19678     \__fp_ep_mul_raw:wwwN
19679   }
19680   \__fp_fixed_continue:wn

```

```

19681 }
19682 \cs_new:Npn \__fp_ep_mul_raw:wwwN #1,#2; #3,#4; #5
19683 {
19684   \__fp_fixed_mul:wn #2; #4;
19685   { \exp_after:wN #5 \int_value:w \__fp_int_eval:w #1 + #3 , }
19686 }

```

(End definition for `__fp_ep_mul:wwwN` and `__fp_ep_mul_raw:wwwN`.)

32.9 Dividing extended-precision numbers

Divisions of extended-precision numbers are difficult to perform with exact rounding: the technique used in `l3fp-basics` for 16-digit floating point numbers does not generalize easily to 24-digit numbers. Thankfully, there is no need for exact rounding.

Let us call $\langle n \rangle$ the numerator and $\langle d \rangle$ the denominator. After a simple normalization step, we can assume that $\langle n \rangle \in [0.1, 1)$ and $\langle d \rangle \in [0.1, 1)$, and compute $\langle n \rangle / (10 \langle d \rangle) \in (0.01, 1)$. In terms of the 6 blocks of digits $\langle n_1 \rangle \cdots \langle n_6 \rangle$ and the 6 blocks $\langle d_1 \rangle \cdots \langle d_6 \rangle$, the condition translates to $\langle n_1 \rangle, \langle d_1 \rangle \in [1000, 9999]$.

We first find an integer estimate $a \simeq 10^8 / \langle d \rangle$ by computing

$$\alpha = \left\lceil \frac{10^9}{\langle d_1 \rangle + 1} \right\rceil$$

$$\beta = \left\lfloor \frac{10^9}{\langle d_1 \rangle} \right\rfloor$$

$$a = 10^3 \alpha + (\beta - \alpha) \cdot \left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) - 1250,$$

where $\left\lceil \cdot \right\rceil$ denotes ε -TEX's rounding division, which rounds ties away from zero. The idea is to interpolate between $10^3 \alpha$ and $10^3 \beta$ with a parameter $\langle d_2 \rangle / 10^4$, so that when $\langle d_2 \rangle = 0$ one gets $a = 10^3 \beta - 1250 \simeq 10^{12} / \langle d_1 \rangle \simeq 10^8 / \langle d \rangle$, while when $\langle d_2 \rangle = 9999$ one gets $a = 10^3 \alpha - 1250 \simeq 10^{12} / (\langle d_1 \rangle + 1) \simeq 10^8 / \langle d \rangle$. The shift by 1250 helps to ensure that a is an underestimate of the correct value. We shall prove that

$$1 - 1.755 \cdot 10^{-5} < \frac{\langle d \rangle a}{10^8} < 1.$$

We can then compute the inverse of $\langle d \rangle a / 10^8 = 1 - \epsilon$ using the relation $1 / (1 - \epsilon) \simeq (1 + \epsilon)(1 + \epsilon^2) + \epsilon^4$, which is correct up to a relative error of $\epsilon^5 < 1.6 \cdot 10^{-24}$. This allows us to find the desired ratio as

$$\frac{\langle n \rangle}{\langle d \rangle} = \frac{\langle n \rangle a}{10^8} ((1 + \epsilon)(1 + \epsilon^2) + \epsilon^4).$$

Let us prove the upper bound first (multiplied by 10^{15}). Note that $10^7 \langle d \rangle < 10^3 \langle d_1 \rangle + 10^{-1}(\langle d_2 \rangle + 1)$, and that ε -TEX's division $\left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor$ underestimates $10^{-1}(\langle d_2 \rangle + 1)$ by 0.5 at

most, as can be checked for each possible last digit of $\langle d_2 \rangle$. Then,

$$10^7 \langle d \rangle a < \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \left(\left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) \beta + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \alpha - 1250 \right) \quad (1)$$

$$< \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \quad (2)$$

$$\left(\left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) \left(\frac{10^9}{\langle d_1 \rangle} + \frac{1}{2} \right) + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \left(\frac{10^9}{\langle d_1 \rangle + 1} + \frac{1}{2} \right) - 1250 \right) \quad (3)$$

$$< \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 750 \right) \quad (4)$$

We recognize a quadratic polynomial in $[\langle d_2 \rangle / 10]$ with a negative leading coefficient: this polynomial is bounded above, according to $([\langle d_2 \rangle / 10] + a)(b - c[\langle d_2 \rangle / 10]) \leq (b + ca)^2 / (4c)$. Hence,

$$10^7 \langle d \rangle a < \frac{10^{15}}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} \left(\langle d_1 \rangle + \frac{1}{2} + \frac{1}{4} 10^{-3} - \frac{3}{8} \cdot 10^{-9} \langle d_1 \rangle (\langle d_1 \rangle + 1) \right)^2$$

Since $\langle d_1 \rangle$ takes integer values within $[1000, 9999]$, it is a simple programming exercise to check that the squared expression is always less than $\langle d_1 \rangle (\langle d_1 \rangle + 1)$, hence $10^7 \langle d \rangle a < 10^{15}$. The upper bound is proven. We also find that $\frac{3}{8}$ can be replaced by slightly smaller numbers, but nothing less than $0.374563\dots$, and going back through the derivation of the upper bound, we find that 1250 is as small a shift as we can obtain without breaking the bound.

Now, the lower bound. The same computation as for the upper bound implies

$$10^7 \langle d \rangle a > \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor - \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 1750 \right)$$

This time, we want to find the minimum of this quadratic polynomial. Since the leading coefficient is still negative, the minimum is reached for one of the extreme values $[y/10] = 0$ or $[y/10] = 100$, and we easily check the bound for those values.

We have proven that the algorithm gives us a precise enough answer. Incidentally, the upper bound that we derived tells us that $a < 10^8 / \langle d \rangle \leq 10^9$, hence we can compute a safely as a $\text{T}_{\text{E}}\text{X}$ integer, and even add 10^9 to it to ease grabbing of all the digits. The lower bound implies $10^8 - 1755 < a$, which we do not care about.

`_fp_ep_div:wwwn` Compute the ratio of two extended-precision numbers. The result is an extended-precision number whose first block lies in the range $[100, 9999]$, and is placed after the $\langle continuation \rangle$ once we are done. First normalize the inputs so that both first block lie in $[1000, 9999]$, then call `_fp_ep_div_esti:wwwn $\langle denominator \rangle$ $\langle numerator \rangle$` , responsible for estimating the inverse of the denominator.

```

19687 \cs_new:Npn \_fp\_ep\_div:wwwn #1,#2; #3,#4;
19688 {
19689   \_fp\_ep\_to\_ep:wwN #1,#2;
19690   \_fp\_fixed\_continue:wn
19691   {
19692     \_fp\_ep\_to\_ep:wwN #3,#4;
19693     \_fp\_ep\_div\_esti:wwwn
19694   }
19695 }
```

(End definition for _fp_ep_div:wwwn.)

_fp_ep_div_esti:wwwn
_fp_ep_div_estii:wwnnwn
_fp_ep_div_estiii:NNNNwwwn

The **esti** function evaluates $\alpha = 10^9 / (\langle d_1 \rangle + 1)$, which is used twice in the expression for a , and combines the exponents **#1** and **#4** (with a shift by 1 because we later compute $\langle n \rangle / (10 \langle d \rangle)$). Then the **estii** function evaluates $10^9 + a$, and puts the exponent **#2** after the continuation **#7**: from there on we can forget exponents and focus on the mantissa. The **estiii** function multiplies the denominator **#7** by $10^{-8}a$ (obtained as a split into the single digit **#1** and two blocks of 4 digits, **#2#3#4#5** and **#6**). The result $10^{-8}a \langle d \rangle = (1 - \epsilon)$, and a partially packed $10^{-9}a$ (as a block of four digits, and five individual digits, not packed by lack of available macro parameters here) are passed to **_fp_ep_div_epsilon:wnNNNNn**, which computes $10^{-9}a / (1 - \epsilon)$, that is, $1 / (10 \langle d \rangle)$ and we finally multiply this by the numerator **#8**.

```

19696 \cs_new:Npn \_fp_ep_div_esti:wwwn #1,#2#3; #4,
19697 {
19698   \exp_after:wN \_fp_ep_div_estii:wwnnwn
19699   \int_value:w \_fp_int_eval:w 10 0000 0000 / ( #2 + 1 )
19700   \exp_after:wN ;
19701   \int_value:w \_fp_int_eval:w #4 - #1 + 1 ,
19702   {#2} #3;
19703 }
19704 \cs_new:Npn \_fp_ep_div_estii:wwnnwn #1; #2,#3#4#5; #6; #7
19705 {
19706   \exp_after:wN \_fp_ep_div_estiii:NNNNwwwn
19707   \int_value:w \_fp_int_eval:w 10 0000 0000 - 1750
19708   + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
19709   {#3}{#4}#5; #6; { #7 #2, }
19710 }
19711 \cs_new:Npn \_fp_ep_div_estiii:NNNNwwwn 1#1#2#3#4#5#6; #7;
19712 {
19713   \_fp_fixed_mul_short:wwn #7; {#1}{#2#3#4#5}{#6};
19714   \_fp_ep_div_epsilon:wnNNNNn {#1#2#3#4}#5#6
19715   \_fp_fixed_mul:wwn
19716 }

```

(End definition for _fp_ep_div_esti:wwwn, _fp_ep_div_estii:wwnnwn, and _fp_ep_div_estiii:NNNNwwwn.)

_fp_ep_div_epsilon:wnNNNNn
_fp_ep_div_eps_pack:NNNNw
_fp_ep_div_epsii:wwnnNNNNn

The bounds shown above imply that the **epsi** function's first operand is $(1 - \epsilon)$ with $\epsilon \in [0, 1.755 \cdot 10^{-5}]$. The **epsi** function computes ϵ as $1 - (1 - \epsilon)$. Since $\epsilon < 10^{-4}$, its first block vanishes and there is no need to explicitly use **#1** (which is 9999). Then **epsii** evaluates $10^{-9}a / (1 - \epsilon)$ as $(1 + \epsilon^2)(1 + \epsilon)(10^{-9}a\epsilon) + 10^{-9}a$. Importantly, we compute $10^{-9}a\epsilon$ before multiplying it with the rest, rather than multiplying by ϵ and then $10^{-9}a$, as this second option loses more precision. Also, the combination of **short_mul** and **div_myriad** is both faster and more precise than a simple **mul**.

```

19717 \cs_new:Npn \_fp_ep_div_epsilon:wnNNNNn #1#2#3#4#5#6;
19718 {
19719   \exp_after:wN \_fp_ep_div_epsii:wwnnNNNNn
19720   \int_value:w \_fp_int_eval:w 1 9998 - #2
19721   \exp_after:wN \_fp_ep_div_eps_pack:NNNNw
19722   \int_value:w \_fp_int_eval:w 1 9999 9998 - #3#4
19723   \exp_after:wN \_fp_ep_div_eps_pack:NNNNw
19724   \int_value:w \_fp_int_eval:w 2 0000 0000 - #5#6 ; ;
19725 }

```

```

19726 \cs_new:Npn \__fp_ep_div_eps_pack:NNNNNw #1#2#3#4#5#6;
19727 { + #1 ; {#2#3#4#5} {#6} }
19728 \cs_new:Npn \__fp_ep_div_epsii:wnNNNNNn 1#1; #2; #3#4#5#6#7#8
19729 {
19730   \__fp_fixed_mul:wnn {0000}{#1}#2; {0000}{#1}#2;
19731   \__fp_fixed_add_one:wN
19732   \__fp_fixed_mul:wnn {10000} {#1} #2 ;
19733   {
19734     \__fp_fixed_mul_short:wnn {0000}{#1}#2; {#3}{#4#5#6#7}{#8000};
19735     \__fp_fixed_div_myriad:wn
19736     \__fp_fixed_mul:wnn
19737   }
19738   \__fp_fixed_add:wnn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000};
19739 }

```

(End definition for `__fp_ep_div_epsilon:wnNNNNNn`, `__fp_ep_div_eps_pack:NNNNNw`, and `__fp_ep_div_epsii:wnNNNNNn`.)

32.10 Inverse square root of extended precision numbers

The idea here is similar to division. Normalize the input, multiplying by powers of 100 until we have $x \in [0.01, 1)$. Then find an integer approximation $r \in [101, 1003]$ of $10^2/\sqrt{x}$, as the fixed point of iterations of the Newton method: essentially $r \mapsto (r + 10^8/(x_1 r))/2$, starting from a guess that optimizes the number of steps before convergence. In fact, just as there is a slight shift when computing divisions to ensure that some inequalities hold, we replace 10^8 by a slightly larger number which ensures that $r^2 x \geq 10^4$. This also causes $r \in [101, 1003]$. Another correction to the above is that the input is actually normalized to $[0.1, 1)$, and we use either 10^8 or 10^9 in the Newton method, depending on the parity of the exponent. Skipping those technical hurdles, once we have the approximation r , we set $y = 10^{-4}r^2x$ (or rather, the correct power of 10 to get $y \simeq 1$) and compute $y^{-1/2}$ through another application of Newton's method. This time, the starting value is $z = 1$, each step maps $z \mapsto z(1.5 - 0.5yz^2)$, and we perform a fixed number of steps. Our final result combines r with $y^{-1/2}$ as $x^{-1/2} = 10^{-2}ry^{-1/2}$.

```

\__fp_ep_isqrt:wnn
\__fp_ep_isqrt_aux:wnn
\__fp_ep_isqrt_auxii:wnnnwnn

```

First normalize the input, then check the parity of the exponent `#1`. If it is even, the result's exponent will be `-#1/2`, otherwise it will be `(#1 - 1)/2` (except in the case where the input was an exact power of 100). The `auxii` function receives as `#1` the result's exponent just computed, as `#2` the starting value for the iteration giving r (the values 168 and 535 lead to the least number of iterations before convergence, on average), as `#3` and `#4` one empty argument and one 0, depending on the parity of the original exponent, as `#5` and `#6` the normalized mantissa (`#5` $\in [1000, 9999]$), and as `#7` the continuation. It sets up the iteration giving r : the `esti` function thus receives the initial two guesses `#2` and 0, an approximation `#5` of 10^4x (its first block of digits), and the empty/zero arguments `#3` and `#4`, followed by the mantissa and an altered continuation where we have stored the result's exponent.

```

19740 \cs_new:Npn \__fp_ep_isqrt:wnn #1,#2;
19741 {
19742   \__fp_ep_to_ep:wnN #1,#2;
19743   \__fp_ep_isqrt_auxi:wnn
19744 }
19745 \cs_new:Npn \__fp_ep_isqrt_auxi:wnn #1,
19746 {

```

```

19747 \exp_after:wN \_fp_ep_isqrt_auxii:wwnnwn
19748 \int_value:w \_fp_int_eval:w
19749 \int_if_odd:nTF {#1}
19750 { (1 - #1) / 2 , 535 , { 0 } { } }
19751 { 1 - #1 / 2 , 168 , { } { 0 } }
19752 }
19753 \cs_new:Npn \_fp_ep_isqrt_auxii:wwnnwn #1, #2, #3#4 #5#6; #7
19754 {
19755 \_fp_ep_isqrt_esti:wwnnwn #2, 0, #5, {#3} {#4}
19756 {#5} #6 ; { #7 #1 , }
19757 }

```

(End definition for _fp_ep_isqrt:wn, _fp_ep_isqrt_aux:wn, and _fp_ep_isqrt_auxii:wwnnwn.)

```

\_fp_ep_isqrt_esti:wwnnwn
\_fp_ep_isqrt_estii:wwnnwn
\_fp_ep_isqrt_estiii:NNNNNwww

```

If the last two approximations gave the same result, we are done: call the `esti` function to clean up. Otherwise, evaluate $(\langle prev \rangle + 1.005 \cdot 10^8 \text{ or } 9 / (\langle prev \rangle \cdot x)) / 2$, as the next approximation: omitting the 1.005 factor, this would be Newton's method. We can check by brute force that if `#4` is empty (the original exponent was even), the process computes an integer slightly larger than $100/\sqrt{x}$, while if `#4` is 0 (the original exponent was odd), the result is an integer slightly larger than $100/\sqrt{x/10}$. Once we are done, we evaluate $100r^2/2$ or $10r^2/2$ (when the exponent is even or odd, respectively) and feed that to `estiii`. This third auxiliary finds $y_{\text{even}}/2 = 10^{-4}r^2x/2$ or $y_{\text{odd}}/2 = 10^{-5}r^2x/2$ (again, depending on earlier parity). A simple program shows that $y \in [1, 1.0201]$. The number $y/2$ is fed to `_fp_ep_isqrt_epsilon:wn`, which computes $1/\sqrt{y}$, and we finally multiply the result by r .

```

19758 \cs_new:Npn \_fp_ep_isqrt_esti:wwnnwn #1, #2, #3, #4
19759 {
19760 \if_int_compare:w #1 = #2 \exp_stop_f:
19761 \exp_after:wN \_fp_ep_isqrt_estii:wwnnwn
19762 \fi:
19763 \exp_after:wN \_fp_ep_isqrt_esti:wwnnwn
19764 \int_value:w \_fp_int_eval:w
19765 (#1 + 1 0050 0000 #4 / (#1 * #3)) / 2 ,
19766 #1, #3, {#4}
19767 }
19768 \cs_new:Npn \_fp_ep_isqrt_estii:wwnnwn #1, #2, #3, #4#5
19769 {
19770 \exp_after:wN \_fp_ep_isqrt_estiii:NNNNNwww
19771 \int_value:w \_fp_int_eval:w 1000 0000 + #2 * #2 #5 * 5
19772 \exp_after:wN , \int_value:w \_fp_int_eval:w 10000 + #2 ;
19773 }
19774 \cs_new:Npn \_fp_ep_isqrt_estiii:NNNNNwww 1#1#2#3#4#5#6, 1#7#8; #9;
19775 {
19776 \_fp_fixed_mul_short:wn #9; {#1} {#2#3#4#5} {#600} ;
19777 \_fp_ep_isqrt_epsilon:wn
19778 \_fp_fixed_mul_short:wn {#7} {#80} {0000} ;
19779 }

```

(End definition for _fp_ep_isqrt_esti:wwnnwn, _fp_ep_isqrt_estii:wwnnwn, and _fp_ep_isqrt_estiii:NNNNNwww.)

```

\_fp_ep_isqrt_epsilon:wn
\_fp_ep_isqrt_epsilonii:wn

```

Here, we receive a fixed point number $y/2$ with $y \in [1, 1.0201]$. Starting from $z = 1$ we iterate $z \mapsto z(3/2 - z^2y/2)$. In fact, we start from the first iteration $z = 3/2 - y/2$ to avoid useless multiplications. The `epsii` auxiliary receives z as `#1` and y as `#2`.

```

19780 \cs_new:Npn \__fp_ep_isqrt_epsi:wwN #1;
19781 {
19782   \__fp_fixed_sub:wwn {15000}{0000}{0000}{0000}{0000}{0000}; #1;
19783   \__fp_ep_isqrt_epsi:wwN #1;
19784   \__fp_ep_isqrt_epsi:wwN #1;
19785   \__fp_ep_isqrt_epsi:wwN #1;
19786 }
19787 \cs_new:Npn \__fp_ep_isqrt_epsi:wwN #1; #2;
19788 {
19789   \__fp_fixed_mul:wwn #1; #1;
19790   \__fp_fixed_mul_sub_back:wwwn #2;
19791   {15000}{0000}{0000}{0000}{0000}{0000};
19792   \__fp_fixed_mul:wwn #1;
19793 }

```

(End definition for __fp_ep_isqrt_epsi:wwN and __fp_ep_isqrt_epsi:wwN.)

32.11 Converting from fixed point to floating point

After computing Taylor series, we wish to convert the result from extended precision (with or without an exponent) to the public floating point format. The functions here should be called within an integer expression for the overall exponent of the floating point.

__fp_ep_to_float_o:wwN
 __fp_ep_inv_to_float_o:wwN

An extended-precision number is simply a comma-delimited exponent followed by a fixed point number. Leave the exponent in the current integer expression then convert the fixed point number.

```

19794 \cs_new:Npn \__fp_ep_to_float_o:wwN #1,
19795 { + \__fp_int_eval:w #1 \__fp_fixed_to_float_o:wwN }
19796 \cs_new:Npn \__fp_ep_inv_to_float_o:wwN #1,#2;
19797 {
19798   \__fp_ep_div:wwwn 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1,#2;
19799   \__fp_ep_to_float_o:wwN
19800 }

```

(End definition for __fp_ep_to_float_o:wwN and __fp_ep_inv_to_float_o:wwN.)

__fp_fixed_inv_to_float_o:wwN

Another function which reduces to converting an extended precision number to a float.

```

19801 \cs_new:Npn \__fp_fixed_inv_to_float_o:wwN
19802 { \__fp_ep_inv_to_float_o:wwN 0, }

```

(End definition for __fp_fixed_inv_to_float_o:wwN.)

__fp_fixed_to_float_rad_o:wwN

Converts the fixed point number #1 from degrees to radians then to a floating point number. This could perhaps remain in l3fp-trig.

```

19803 \cs_new:Npn \__fp_fixed_to_float_rad_o:wwN #1;
19804 {
19805   \__fp_fixed_mul:wwn #1; {5729}{5779}{5130}{8232}{0876}{7981};
19806   { \__fp_ep_to_float_o:wwN 2, }
19807 }

```

(End definition for __fp_fixed_to_float_rad_o:wwN.)


```

\__fp_fixed_to_float_o:wN      ... \__fp_int_eval:w <exponent> \__fp_fixed_to_float_o:wN {\langle a_1 \rangle} {\langle a_2 \rangle} {\langle a_3 \rangle}
\__fp_fixed_to_float_o:Nw      {\langle a_4 \rangle} {\langle a_5 \rangle} {\langle a_6 \rangle} ; <sign>
                                yields

```

```

                                <exponent'> ; {\langle a'_1 \rangle} {\langle a'_2 \rangle} {\langle a'_3 \rangle} {\langle a'_4 \rangle} ;

```

And the `to_fixed` version gives six brace groups instead of 4, ensuring that $1000 \leq \langle a'_1 \rangle \leq 9999$. At this stage, we know that $\langle a_1 \rangle$ is positive (otherwise, it is sign of an error before), and we assume that it is less than 10^8 .⁹

```

19808 \cs_new:Npn \__fp_fixed_to_float_o:Nw #1#2;
19809 { \__fp_fixed_to_float_o:wN #2; #1 }
19810 \cs_new:Npn \__fp_fixed_to_float_o:wN #1#2#3#4#5#6; #7
19811 { % for the 8-digit-at-the-start thing
19812   + \__fp_int_eval:w \c__fp_block_int
19813   \exp_after:wN \exp_after:wN
19814   \exp_after:wN \__fp_fixed_to_loop:N
19815   \exp_after:wN \use_none:n
19816   \int_value:w \__fp_int_eval:w
19817   1 0000 0000 + #1 \exp_after:wN \__fp_use_none_stop_f:n
19818   \int_value:w 1#2 \exp_after:wN \__fp_use_none_stop_f:n
19819   \int_value:w 1#3#4 \exp_after:wN \__fp_use_none_stop_f:n
19820   \int_value:w 1#5#6
19821   \exp_after:wN ;
19822   \exp_after:wN ;
19823 }
19824 \cs_new:Npn \__fp_fixed_to_loop:N #1
19825 {
19826   \if_meaning:w 0 #1
19827   - 1
19828   \exp_after:wN \__fp_fixed_to_loop:N
19829   \else:
19830   \exp_after:wN \__fp_fixed_to_loop_end:w
19831   \exp_after:wN #1
19832   \fi:
19833 }
19834 \cs_new:Npn \__fp_fixed_to_loop_end:w #1 #2 ;
19835 {
19836   \if_meaning:w ; #1
19837   \exp_after:wN \__fp_fixed_to_float_zero:w
19838   \else:
19839   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
19840   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
19841   \exp_after:wN \__fp_fixed_to_float_pack:ww
19842   \exp_after:wN ;
19843   \fi:
19844   #1 #2 0000 0000 0000 0000 ;
19845 }
19846 \cs_new:Npn \__fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
19847 {
19848   - 2 * \c__fp_max_exponent_int ;
19849   {0000} {0000} {0000} {0000} ;
19850 }

```

⁹Bruno: I must double check this assumption.

```

19851 \cs_new:Npn \__fp_fixed_to_float_pack:ww #1 ; #2#3 ; ;
19852 {
19853   \if_int_compare:w #2 > 4 \exp_stop_f:
19854     \exp_after:wN \__fp_fixed_to_float_round_up:wnnnnw
19855   \fi:
19856   ; #1 ;
19857 }
19858 \cs_new:Npn \__fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
19859 {
19860   \exp_after:wN \__fp_basics_pack_high:NNNNNw
19861   \int_value:w \__fp_int_eval:w 1 #1#2
19862   \exp_after:wN \__fp_basics_pack_low:NNNNNw
19863   \int_value:w \__fp_int_eval:w 1 #3#4 + 1 ;
19864 }

```

(End definition for __fp_fixed_to_float_o:wN and __fp_fixed_to_float_o:Nw.)

```

19865 \</package>

```

33 l3fp-expo implementation

```

19866 \*package>
19867 \@@=fp>

```

Unary functions.

```

\__fp_parse_word_exp:N
\__fp_parse_word_ln:N
\__fp_parse_word_fact:N
19868 \cs_new:Npn \__fp_parse_word_exp:N
19869 { \__fp_parse_unary_function:NNN \__fp_exp_o:w ? }
19870 \cs_new:Npn \__fp_parse_word_ln:N
19871 { \__fp_parse_unary_function:NNN \__fp_ln_o:w ? }
19872 \cs_new:Npn \__fp_parse_word_fact:N
19873 { \__fp_parse_unary_function:NNN \__fp_fact_o:w ? }

```

(End definition for __fp_parse_word_exp:N, __fp_parse_word_ln:N, and __fp_parse_word_fact:N.)

33.1 Logarithm

33.1.1 Work plan

As for many other functions, we filter out special cases in __fp_ln_o:w. Then __fp_ln_npos_o:w receives a positive normal number, which we write in the form $a \cdot 10^b$ with $a \in [0.1, 1)$.

The rest of this section is actually not in sync with the code. Or is the code not in sync with the section? In the current code, $c \in [1, 10]$ is such that $0.7 \leq ac < 1.4$.

We are given a positive normal number, of the form $a \cdot 10^b$ with $a \in [0.1, 1)$. To compute its logarithm, we find a small integer $5 \leq c < 50$ such that $0.91 \leq ac/5 < 1.1$, and use the relation

$$\ln(a \cdot 10^b) = b \cdot \ln(10) - \ln(c/5) + \ln(ac/5).$$

The logarithms $\ln(10)$ and $\ln(c/5)$ are looked up in a table. The last term is computed using the following Taylor series of \ln near 1:

$$\ln\left(\frac{ac}{5}\right) = \ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + t^2 \left(\frac{1}{3} + t^2 \left(\frac{1}{5} + t^2 \left(\frac{1}{7} + t^2 \left(\frac{1}{9} + \dots\right)\right)\right)\right)\right)$$

where $t = 1 - 10/(ac + 5)$. We can now see one reason for the choice of $ac \sim 5$: then $ac + 5 = 10(1 - \epsilon)$ with $-0.05 < \epsilon \leq 0.045$, hence

$$t = \frac{\epsilon}{1 - \epsilon} = \epsilon(1 + \epsilon)(1 + \epsilon^2)(1 + \epsilon^4) \dots,$$

is not too difficult to compute.

33.1.2 Some constants

A few values of the logarithm as extended fixed point numbers. Those are needed in the implementation. It turns out that we don't need the value of $\ln(5)$.

```
\c__fp_ln_i_fixed_t1
\c__fp_ln_ii_fixed_t1
\c__fp_ln_iii_fixed_t1
\c__fp_ln_iv_fixed_t1
\c__fp_ln_vi_fixed_t1
\c__fp_ln_vii_fixed_t1
\c__fp_ln_viii_fixed_t1
\c__fp_ln_ix_fixed_t1
\c__fp_ln_x_fixed_t1
19874 \tl_const:Nn \c__fp_ln_i_fixed_t1 { {0000}{0000}{0000}{0000}{0000}{0000};}
19875 \tl_const:Nn \c__fp_ln_ii_fixed_t1 { {6931}{4718}{0559}{9453}{0941}{7232};}
19876 \tl_const:Nn \c__fp_ln_iii_fixed_t1 { {10986}{1228}{8668}{1096}{9139}{5245};}
19877 \tl_const:Nn \c__fp_ln_iv_fixed_t1 { {13862}{9436}{1119}{8906}{1883}{4464};}
19878 \tl_const:Nn \c__fp_ln_vi_fixed_t1 { {17917}{5946}{9228}{0550}{0081}{2477};}
19879 \tl_const:Nn \c__fp_ln_vii_fixed_t1 { {19459}{1014}{9055}{3133}{0510}{5353};}
19880 \tl_const:Nn \c__fp_ln_viii_fixed_t1 { {20794}{4154}{1679}{8359}{2825}{1696};}
19881 \tl_const:Nn \c__fp_ln_ix_fixed_t1 { {21972}{2457}{7336}{2193}{8279}{0490};}
19882 \tl_const:Nn \c__fp_ln_x_fixed_t1 { {23025}{8509}{2994}{0456}{8401}{7991};}
```

(End definition for `\c__fp_ln_i_fixed_t1` and others.)

33.1.3 Sign, exponent, and special numbers

`__fp_ln_o:w` The logarithm of negative numbers (including $-\infty$ and -0) raises the “invalid” exception. The logarithm of $+0$ is $-\infty$, raising a division by zero exception. The logarithm of $+\infty$ or a `nan` is itself. Positive normal numbers call `__fp_ln_npos_o:w`.

```
19883 \cs_new:Npn \__fp_ln_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
19884 {
19885   \if_meaning:w 2 #3
19886     \__fp_case_use:nw { \__fp_invalid_operation_o:nw { ln } }
19887   \fi:
19888   \if_case:w #2 \exp_stop_f:
19889     \__fp_case_use:nw
19890     { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
19891   \or:
19892   \else:
19893     \__fp_case_return_same_o:w
19894   \fi:
19895   \__fp_ln_npos_o:w \s__fp \__fp_chk:w #2#3#4;
19896 }
```

(End definition for `__fp_ln_o:w`.)

33.1.4 Absolute ln

`__fp_ln_npos_o:w` We catch the case of a significand very close to 0.1 or to 1. In all other cases, the final result is at least 10^{-4} , and then an error of $0.5 \cdot 10^{-20}$ is acceptable.

```
19897 \cs_new:Npn \__fp_ln_npos_o:w \s__fp \__fp_chk:w 10#1#2#3;
19898 { %^^A todo: ln(1) should be "exact zero", not "underflow"
19899   \exp_after:wN \__fp_sanitize:Nw
19900   \int_value:w % for the overall sign
```

```

19901     \if_int_compare:w #1 < 1 \exp_stop_f:
19902         2
19903     \else:
19904         0
19905     \fi:
19906     \exp_after:wN \exp_stop_f:
19907     \int_value:w \__fp_int_eval:w % for the exponent
19908         \__fp_ln_significand:NNNNnnnnN #2#3
19909         \__fp_ln_exponent:wn {#1}
19910 }

```

(End definition for __fp_ln_npos_o:w.)

__fp_ln_significand:NNNNnnnnN __fp_ln_significand:NNNNnnnnN $\langle X_1 \rangle$ $\{\langle X_2 \rangle\}$ $\{\langle X_3 \rangle\}$ $\{\langle X_4 \rangle\}$ $\langle continuation \rangle$
This function expands to

$\langle continuation \rangle$ $\{\langle Y_1 \rangle\}$ $\{\langle Y_2 \rangle\}$ $\{\langle Y_3 \rangle\}$ $\{\langle Y_4 \rangle\}$ $\{\langle Y_5 \rangle\}$ $\{\langle Y_6 \rangle\}$;

where $Y = -\ln(X)$ as an extended fixed point.

```

19911 \cs_new:Npn \__fp_ln_significand:NNNNnnnnN #1#2#3#4
19912 {
19913     \exp_after:wN \__fp_ln_x_ii:wnnnnn
19914     \int_value:w
19915     \if_case:w #1 \exp_stop_f:
19916     \or:
19917         \if_int_compare:w #2 < 4 \exp_stop_f:
19918             \__fp_int_eval:w 10 - #2
19919     \else:
19920         6
19921     \fi:
19922     \or: 4
19923     \or: 3
19924     \or: 2
19925     \or: 2
19926     \or: 2
19927     \else: 1
19928     \fi:
19929     ; { #1 #2 #3 #4 }
19930 }

```

(End definition for __fp_ln_significand:NNNNnnnnN.)

__fp_ln_x_ii:wnnnnn We have thus found $c \in [1, 10]$ such that $0.7 \leq ac < 1.4$ in all cases. Compute $1 + x = 1 + ac \in [1.7, 2.4)$.

```

19931 \cs_new:Npn \__fp_ln_x_ii:wnnnnn #1; #2#3#4#5
19932 {
19933     \exp_after:wN \__fp_ln_div_after:Nw
19934     \cs:w c__fp_ln_ \__fp_int_to_roman:w #1 _fixed_tl \exp_after:wN \cs_end:
19935     \int_value:w
19936     \exp_after:wN \__fp_ln_x_iv:wnnnnnnnnn
19937     \int_value:w \__fp_int_eval:w
19938     \exp_after:wN \__fp_ln_x_iii_var:NNNNNw
19939     \int_value:w \__fp_int_eval:w 9999 9990 + #1*#2#3 +
19940     \exp_after:wN \__fp_ln_x_iii:NNNNNNw
19941     \int_value:w \__fp_int_eval:w 10 0000 0000 + #1*#4#5 ;

```

```

19942     {20000} {0000} {0000} {0000}
19943   } %^A todo: reoptimize (a generalization attempt failed).
19944 \cs_new:Npn \__fp_ln_x_iii:NNNNNNw #1#2 #3#4#5#6 #7;
19945   { #1#2; {#3#4#5#6} {#7} }
19946 \cs_new:Npn \__fp_ln_x_iii_var:NNNNNNw #1 #2#3#4#5 #6;
19947   {
19948     #1#2#3#4#5 + 1 ;
19949     {#1#2#3#4#5} {#6}
19950   }

```

The Taylor series to be used is expressed in terms of $t = (x - 1)/(x + 1) = 1 - 2/(x + 1)$. We now compute the quotient with extended precision, reusing some code from `__fp_/_o:ww`. Note that $1 + x$ is known exactly.

To reuse notations from `l3fp-basics`, we want to compute A/Z with $A = 2$ and $Z = x + 1$. In `l3fp-basics`, we considered the case where both A and Z are arbitrary, in the range $[0.1, 1)$, and we had to monitor the growth of the sequence of remainders A , B , C , etc. to ensure that no overflow occurred during the computation of the next quotient. The main source of risk was our choice to define the quotient as roughly $10^9 \cdot A/10^5 \cdot Z$: then A was bound to be below $2.147 \dots$, and this limit was never far.

In our case, we can simply work with $10^8 \cdot A$ and $10^4 \cdot Z$, because our reason to work with higher powers has gone: we needed the integer $y \simeq 10^5 \cdot Z$ to be at least 10^4 , and now, the definition $y \simeq 10^4 \cdot Z$ suffices.

Let us thus define $y = \lfloor 10^4 \cdot Z \rfloor + 1 \in (1.7 \cdot 10^4, 2.4 \cdot 10^4]$, and

$$Q_1 = \left\lfloor \frac{\lfloor 10^8 \cdot A \rfloor}{y} - \frac{1}{2} \right\rfloor.$$

(The $1/2$ comes from how ε -TeX rounds.) As for division, it is easy to see that $Q_1 \leq 10^4 A/Z$, *i.e.*, Q_1 is an underestimate.

Exactly as we did for division, we set $B = 10^4 A - Q_1 Z$. Then

$$\begin{aligned}
10^4 B &\leq A_1 A_2 \cdot A_3 A_4 - \left(\frac{A_1 A_2}{y} - \frac{3}{2} \right) 10^4 Z \\
&\leq A_1 A_2 \left(1 - \frac{10^4 Z}{y} \right) + 1 + \frac{3}{2} y \\
&\leq 10^8 \frac{A}{y} + 1 + \frac{3}{2} y
\end{aligned}$$

In the same way, and using $1.7 \cdot 10^4 \leq y \leq 2.4 \cdot 10^4$, and convexity, we get

$$\begin{aligned}
10^4 A &= 2 \cdot 10^4 \\
10^4 B &\leq 10^8 \frac{A}{y} + 1.6y \leq 4.7 \cdot 10^4 \\
10^4 C &\leq 10^8 \frac{B}{y} + 1.6y \leq 5.8 \cdot 10^4 \\
10^4 D &\leq 10^8 \frac{C}{y} + 1.6y \leq 6.3 \cdot 10^4 \\
10^4 E &\leq 10^8 \frac{D}{y} + 1.6y \leq 6.5 \cdot 10^4 \\
10^4 F &\leq 10^8 \frac{E}{y} + 1.6y \leq 6.6 \cdot 10^4
\end{aligned}$$

Note that we compute more steps than for division: since t is not the end result, we need to know it with more accuracy (on the other hand, the ending is much simpler, as we don't need an exact rounding for transcendental functions, but just a faithful rounding).

`__fp_ln_x_iv:wnnnnnnnn <1 or 2> <8d> ; {<4d>} {<4d>} <fixed-t1>`

The number is x . Compute y by adding 1 to the five first digits.

```

19951 \cs_new:Npn __fp_ln_x_iv:wnnnnnnnn #1; #2#3#4#5 #6#7#8#9
19952 {
19953   \exp_after:wN __fp_div_significand_pack:NNN
19954   \int_value:w __fp_int_eval:w
19955   __fp_ln_div_i:w #1 ;
19956   #6 #7 ; {#8} {#9}
19957   {#2} {#3} {#4} {#5}
19958   { \exp_after:wN __fp_ln_div_ii:wnn \int_value:w #1 }
19959   { \exp_after:wN __fp_ln_div_ii:wnn \int_value:w #1 }
19960   { \exp_after:wN __fp_ln_div_ii:wnn \int_value:w #1 }
19961   { \exp_after:wN __fp_ln_div_ii:wnn \int_value:w #1 }
19962   { \exp_after:wN __fp_ln_div_vi:wnn \int_value:w #1 }
19963 }
19964 \cs_new:Npn __fp_ln_div_i:w #1;
19965 {
19966   \exp_after:wN __fp_div_significand_calc:wnnnnnnnn
19967   \int_value:w __fp_int_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
19968 }
19969 \cs_new:Npn __fp_ln_div_ii:wnn #1; #2;#3 % y; B1;B2 <- for k=1
19970 {
19971   \exp_after:wN __fp_div_significand_pack:NNN
19972   \int_value:w __fp_int_eval:w
19973   \exp_after:wN __fp_div_significand_calc:wnnnnnnnn
19974   \int_value:w __fp_int_eval:w 999999 + #2 #3 / #1 ; % Q2
19975   #2 #3 ;
19976 }
19977 \cs_new:Npn __fp_ln_div_vi:wnn #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
19978 {
19979   \exp_after:wN __fp_div_significand_pack:NNN

```

```

19980     \int_value:w \_fp_int_eval:w 1000000 + #2 #3 / #1 ; % Q6
19981   }

```

We now have essentially

```

\_fp_ln_div_after:Nw <fixed t1>
\_fp_div_significand_pack:NNN 106 + Q1
\_fp_div_significand_pack:NNN 106 + Q2
\_fp_div_significand_pack:NNN 106 + Q3
\_fp_div_significand_pack:NNN 106 + Q4
\_fp_div_significand_pack:NNN 106 + Q5
\_fp_div_significand_pack:NNN 106 + Q6 ;
<exponent> ; <continuation>

```

where $\langle \text{fixed } t1 \rangle$ holds the logarithm of a number in $[1, 10]$, and $\langle \text{exponent} \rangle$ is the exponent. Also, the expansion is done backwards. Then `_fp_div_significand_pack:NNN` puts things in the correct order to add the Q_i together and put semicolons between each piece. Once those have been expanded, we get

```

\_fp_ln_div_after:Nw <fixed-t1> <1d> ; <4d> ; <4d> ;
<4d> ; <4d> ; <4d> ; <4d> ; <exponent> ;

```

Just as with division, we know that the first two digits are 1 and 0 because of bounds on the final result of the division $2/(x+1)$, which is between roughly 0.8 and 1.2. We then compute $1 - 2/(x+1)$, after testing whether $2/(x+1)$ is greater than or smaller than 1.

```

19982 \cs_new:Npn \_fp_ln_div_after:Nw #1#2;
19983 {
19984   \if_meaning:w 0 #2
19985     \exp_after:wN \_fp_ln_t_small:Nw
19986   \else:
19987     \exp_after:wN \_fp_ln_t_large:NNw
19988     \exp_after:wN -
19989   \fi:
19990   #1
19991 }
19992 \cs_new:Npn \_fp_ln_t_small:Nw #1 #2; #3; #4; #5; #6; #7;
19993 {
19994   \exp_after:wN \_fp_ln_t_large:NNw
19995   \exp_after:wN + % <sign>
19996   \exp_after:wN #1
19997   \int_value:w \_fp_int_eval:w 9999 - #2 \exp_after:wN ;
19998   \int_value:w \_fp_int_eval:w 9999 - #3 \exp_after:wN ;
19999   \int_value:w \_fp_int_eval:w 9999 - #4 \exp_after:wN ;
20000   \int_value:w \_fp_int_eval:w 9999 - #5 \exp_after:wN ;
20001   \int_value:w \_fp_int_eval:w 9999 - #6 \exp_after:wN ;
20002   \int_value:w \_fp_int_eval:w 1 0000 - #7 ;
20003 }

\_fp_ln_t_large:NNw <sign> <fixed t1>
<t123456

```

Compute the square t^2 , and keep t at the end with its sign. We know that $t < 0.1765$, so every piece has at most 4 digits. However, since we were not careful in `_fp_ln_t_small:w`, they can have less than 4 digits.

```

20004 \cs_new:Npn \__fp_ln_t_large:NNw #1 #2 #3; #4; #5; #6; #7; #8;
20005 {
20006   \exp_after:wN \__fp_ln_square_t_after:w
20007   \int_value:w \__fp_int_eval:w 9999 0000 + #3*#3
20008   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
20009   \int_value:w \__fp_int_eval:w 9999 0000 + 2*#3*#4
20010   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
20011   \int_value:w \__fp_int_eval:w 9999 0000 + 2*#3*#5 + #4*#4
20012   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
20013   \int_value:w \__fp_int_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
20014   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
20015   \int_value:w \__fp_int_eval:w
20016     1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
20017     + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
20018     % ; ; ;
20019   \exp_after:wN \__fp_ln_twice_t_after:w
20020   \int_value:w \__fp_int_eval:w -1 + 2*#3
20021   \exp_after:wN \__fp_ln_twice_t_pack:Nw
20022   \int_value:w \__fp_int_eval:w 9999 + 2*#4
20023   \exp_after:wN \__fp_ln_twice_t_pack:Nw
20024   \int_value:w \__fp_int_eval:w 9999 + 2*#5
20025   \exp_after:wN \__fp_ln_twice_t_pack:Nw
20026   \int_value:w \__fp_int_eval:w 9999 + 2*#6
20027   \exp_after:wN \__fp_ln_twice_t_pack:Nw
20028   \int_value:w \__fp_int_eval:w 9999 + 2*#7
20029   \exp_after:wN \__fp_ln_twice_t_pack:Nw
20030   \int_value:w \__fp_int_eval:w 10000 + 2*#8 ; ;
20031   { \__fp_ln_c:NwNw #1 }
20032   #2
20033 }
20034 \cs_new:Npn \__fp_ln_twice_t_pack:Nw #1 #2; { + #1 ; {#2} }
20035 \cs_new:Npn \__fp_ln_twice_t_after:w #1; { ; ; ; {#1} }
20036 \cs_new:Npn \__fp_ln_square_t_pack:NNNNNw #1 #2#3#4#5 #6;
20037   { + #1#2#3#4#5 ; {#6} }
20038 \cs_new:Npn \__fp_ln_square_t_after:w 1 0 #1#2#3 #4;
20039   { \__fp_ln_Taylor:wwNw {0#1#2#3} {#4} }

```

(End definition for __fp_ln_x_ii:wnnnn.)

__fp_ln_Taylor:wwNw Denoting $T = t^2$, we get

```

\__fp_ln_Taylor:wwNw
{ \langle T_1 \rangle } { \langle T_2 \rangle } { \langle T_3 \rangle } { \langle T_4 \rangle } { \langle T_5 \rangle } { \langle T_6 \rangle } ; ;
{ \langle (2t)_1 \rangle } { \langle (2t)_2 \rangle } { \langle (2t)_3 \rangle } { \langle (2t)_4 \rangle } { \langle (2t)_5 \rangle } { \langle (2t)_6 \rangle } ;
{ \__fp_ln_c:NwNw \langle sign \rangle }
\langle fixed t1 \rangle \langle exponent \rangle ; \langle continuation \rangle

```

And we want to compute

$$\ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + T \left(\frac{1}{3} + T \left(\frac{1}{5} + T \left(\frac{1}{7} + T \left(\frac{1}{9} + \cdots \right) \right) \right) \right) \right)$$

The process looks as follows


```

\loop 5; A;
\div_int 5; 1.0; \add A; \mul T; {\loop \eval 5-2;}
\add 0.2; A; \mul T; {\loop \eval 5-2;}
\mul B; T; {\loop 3;}
\loop 3; C;

```

This uses the routine for dividing a number by a small integer ($< 10^4$).

```

20040 \cs_new:Npn \__fp_ln_Taylor:wwNw
20041   { \__fp_ln_Taylor_loop:www 21 ; {0000}{0000}{0000}{0000}{0000}{0000} ; }
20042 \cs_new:Npn \__fp_ln_Taylor_loop:www #1; #2; #3;
20043   {
20044     \if_int_compare:w #1 = 1 \exp_stop_f:
20045     \__fp_ln_Taylor_break:w
20046     \fi:
20047     \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl #1;
20048     \__fp_fixed_add:wwn #2;
20049     \__fp_fixed_mul:wwn #3;
20050     {
20051       \exp_after:wN \__fp_ln_Taylor_loop:www
20052       \int_value:w \__fp_int_eval:w #1 - 2 ;
20053     }
20054     #3;
20055   }
20056 \cs_new:Npn \__fp_ln_Taylor_break:w \fi: #1 \__fp_fixed_add:wwn #2#3; #4 ;;
20057   {
20058     \fi:
20059     \exp_after:wN \__fp_fixed_mul:wwn
20060     \exp_after:wN { \int_value:w \__fp_int_eval:w 10000 + #2 } #3;
20061   }

```

(End definition for $\backslash_fp_ln_Taylor:wwNw$.)

```

\__fp_ln_c:NwNw <sign>
{\langle r_1 \rangle} {\langle r_2 \rangle} {\langle r_3 \rangle} {\langle r_4 \rangle} {\langle r_5 \rangle} {\langle r_6 \rangle} ;
<fixed tl> <exponent> ; <continuation>

```

We are now reduced to finding $\ln(c)$ and $\langle exponent \rangle \ln(10)$ in a table, and adding it to the mixture. The first step is to get $\ln(c) - \ln(x) = -\ln(a)$, then we get $\mathbf{b} \ln(10)$ and add or subtract.

For now, $\ln(x)$ is given as $\cdot 10^0$. Unless both the exponent is 1 and $c = 1$, we shift to working in units of $\cdot 10^4$, since the final result is at least $\ln(10/7) \simeq 0.35$.

```

20062 \cs_new:Npn \__fp_ln_c:NwNw #1 #2; #3
20063   {
20064     \if_meaning:w + #1
20065     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_sub:wwn
20066     \else:
20067     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_add:wwn
20068     \fi:
20069     #3 #2 ;
20070   }

```

(End definition for $\backslash_fp_ln_c:NwNw$.)

```

    \_fp_ln_exponent:wn
    {<s1>} {<s2>} {<s3>} {<s4>} {<s5>} {<s6>} ;
    {<exponent>}

```

Compute $\langle exponent \rangle$ times $\ln(10)$. Apart from the cases where $\langle exponent \rangle$ is 0 or 1, the result is necessarily at least $\ln(10) \simeq 2.3$ in magnitude. We can thus drop the least significant 4 digits. In the case of a very large (positive or negative) exponent, we can (and we need to) drop 4 additional digits, since the result is of order 10^4 . Naively, one would think that in both cases we can drop 4 more digits than we do, but that would be slightly too tight for rounding to happen correctly. Besides, we already have addition and subtraction for 24 digits fixed point numbers.

```

20071 \cs_new:Npn \_fp_ln_exponent:wn #1; #2
20072 {
20073   \if_case:w #2 \exp_stop_f:
20074   0 \_fp_case_return:nw { \_fp_fixed_to_float_o:Nw 2 }
20075   \or:
20076   \exp_after:wN \_fp_ln_exponent_one:ww \int_value:w
20077   \else:
20078   \if_int_compare:w #2 > 0 \exp_stop_f:
20079   \exp_after:wN \_fp_ln_exponent_small:NNww
20080   \exp_after:wN 0
20081   \exp_after:wN \_fp_fixed_sub:wwn \int_value:w
20082   \else:
20083   \exp_after:wN \_fp_ln_exponent_small:NNww
20084   \exp_after:wN 2
20085   \exp_after:wN \_fp_fixed_add:wwn \int_value:w -
20086   \fi:
20087   \fi:
20088   #2; #1;
20089 }

```

Now we painfully write all the cases.¹⁰ No overflow nor underflow can happen, except when computing $\ln(1)$.

```

20090 \cs_new:Npn \_fp_ln_exponent_one:ww 1; #1;
20091 {
20092   0
20093   \exp_after:wN \_fp_fixed_sub:wwn \c__fp_ln_x_fixed_tl #1;
20094   \_fp_fixed_to_float_o:wN 0
20095 }

```

For small exponents, we just drop one block of digits, and set the exponent of the log to 4 (minus any shift coming from leading zeros in the conversion from fixed point to floating point). Note that here the exponent has been made positive.

```

20096 \cs_new:Npn \_fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
20097 {
20098   4
20099   \exp_after:wN \_fp_fixed_mul:wwn
20100   \c__fp_ln_x_fixed_tl
20101   {#3}{0000}{0000}{0000}{0000}{0000} ;
20102   #2
20103   {0000}{#4}{#5}{#6}{#7}{#8};
20104   \_fp_fixed_to_float_o:wN #1
20105 }

```

¹⁰Bruno: do rounding.

(End definition for _fp_ln_exponent:wn.)

33.2 Exponential

33.2.1 Sign, exponent, and special numbers

_fp_exp_o:w

```
20106 \cs_new:Npn \_fp_exp_o:w #1 \s__fp \_fp_chk:w #2#3#4; @
20107 {
20108   \if_case:w #2 \exp_stop_f:
20109     \_fp_case_return_o:Nw \c_one_fp
20110   \or:
20111     \exp_after:wN \_fp_exp_normal_o:w
20112   \or:
20113     \if_meaning:w 0 #3
20114       \exp_after:wN \_fp_case_return_o:Nw
20115       \exp_after:wN \c_inf_fp
20116     \else:
20117       \exp_after:wN \_fp_case_return_o:Nw
20118       \exp_after:wN \c_zero_fp
20119     \fi:
20120   \or:
20121     \_fp_case_return_same_o:w
20122   \fi:
20123   \s__fp \_fp_chk:w #2#3#4;
20124 }
```

(End definition for _fp_exp_o:w.)

_fp_exp_normal_o:w

_fp_exp_pos_o:NNwnw

_fp_exp_overflow:NN

```
20125 \cs_new:Npn \_fp_exp_normal_o:w \s__fp \_fp_chk:w 1#1
20126 {
20127   \if_meaning:w 0 #1
20128     \_fp_exp_pos_o:NNwnw + \_fp_fixed_to_float_o:wN
20129   \else:
20130     \_fp_exp_pos_o:NNwnw - \_fp_fixed_inv_to_float_o:wN
20131   \fi:
20132 }
20133 \cs_new:Npn \_fp_exp_pos_o:NNwnw #1#2#3 \fi: #4#5;
20134 {
20135   \fi:
20136   \if_int_compare:w #4 > \c__fp_max_exp_exponent_int
20137     \token_if_eq_charcode:NNTF + #1
20138     { \_fp_exp_overflow:NN \_fp_overflow:w \c_inf_fp }
20139     { \_fp_exp_overflow:NN \_fp_underflow:w \c_zero_fp }
20140   \exp:w
20141   \else:
20142     \exp_after:wN \_fp_sanitize:Nw
20143     \exp_after:wN 0
20144     \int_value:w #1 \_fp_int_eval:w
20145     \if_int_compare:w #4 < 0 \exp_stop_f:
20146       \exp_after:wN \use_i:nn
20147     \else:
20148       \exp_after:wN \use_ii:nn
```

```

20149     \fi:
20150     {
20151         0
20152         \__fp_decimate:nNnnnn { - #4 }
20153         \__fp_exp_Taylor:Nnnwn
20154     }
20155     {
20156         \__fp_decimate:nNnnnn { \c__fp_prec_int - #4 }
20157         \__fp_exp_pos_large:NnnNwn
20158     }
20159     #5
20160     {#4}
20161     #1 #2 0
20162     \exp:w
20163     \fi:
20164     \exp_after:wN \exp_end:
20165 }
20166 \cs_new:Npn \__fp_exp_overflow:NN #1#2
20167 {
20168     \exp_after:wN \exp_after:wN
20169     \exp_after:wN #1
20170     \exp_after:wN #2
20171 }

```

(End definition for __fp_exp_normal_o:w, __fp_exp_pos_o:Nnnwn, and __fp_exp_overflow:NN.)

```

\__fp_exp_Taylor:Nnnwn
\__fp_exp_Taylor_loop:www
\__fp_exp_Taylor_break:Nww

```

This function is called for numbers in the range $[10^{-9}, 10^{-1}]$. We compute 10 terms of the Taylor series. The first argument is irrelevant (rounding digit used by some other functions). The next three arguments, at least 16 digits, delimited by a semicolon, form a fixed point number, so we pack it in blocks of 4 digits.

```

20172 \cs_new:Npn \__fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
20173 {
20174     #6
20175     \__fp_pack_twice_four:wNNNNNNNN
20176     \__fp_pack_twice_four:wNNNNNNNN
20177     \__fp_pack_twice_four:wNNNNNNNN
20178     \__fp_exp_Taylor_ii:ww
20179     ; #2#3#4 0000 0000 ;
20180 }
20181 \cs_new:Npn \__fp_exp_Taylor_ii:ww #1; #2;
20182 { \__fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s__fp_stop }
20183 \cs_new:Npn \__fp_exp_Taylor_loop:www #1; #2; #3;
20184 {
20185     \if_int_compare:w #1 = 1 \exp_stop_f:
20186     \exp_after:wN \__fp_exp_Taylor_break:Nww
20187     \fi:
20188     \__fp_fixed_div_int:wwN #3 ; #1 ;
20189     \__fp_fixed_add_one:wN
20190     \__fp_fixed_mul:wwN #2 ;
20191     {
20192         \exp_after:wN \__fp_exp_Taylor_loop:www
20193         \int_value:w \__fp_int_eval:w #1 - 1 ;
20194         #2 ;
20195     }

```

```

20196     }
20197 \cs_new:Npn \__fp_exp_Taylor_break:Nww #1 #2; #3 \s__fp_stop
20198 { \__fp_fixed_add_one:wN #2 ; }

```

(End definition for `__fp_exp_Taylor:Nnnwn`, `__fp_exp_Taylor_loop:www`, and `__fp_exp_Taylor_break:Nww`.)

`\c__fp_exp_intarray` The integer array has $6 \times 9 \times 4 = 216$ items encoding the values of $\exp(j \times 10^i)$ for $j = 1, \dots, 9$ and $i = -1, \dots, 4$. Each value is expressed as $\simeq 10^p \times 0.m_1m_2m_3$ with three 8-digit blocks m_1, m_2, m_3 and an integer exponent p (one more than the scientific exponent), and these are stored in the integer array as four items: $p, 10^8 + m_1, 10^8 + m_2, 10^8 + m_3$. The various exponentials are stored in increasing order of $j \times 10^i$.

Storing this data in an integer array makes it slightly harder to access (slower, too), but uses 16 bytes of memory per exponential stored, while storing as tokens used around 40 tokens; tokens have an especially large footprint in Unicode-aware engines.

```

20199 \intarray_const_from_clist:Nn \c__fp_exp_intarray
20200 {
20201     1 , 1 1105 1709 , 1 1807 5647 , 1 6248 1171 ,
20202     1 , 1 1221 4027 , 1 5816 0169 , 1 8339 2107 ,
20203     1 , 1 1349 8588 , 1 0757 6003 , 1 1039 8374 ,
20204     1 , 1 1491 8246 , 1 9764 1270 , 1 3178 2485 ,
20205     1 , 1 1648 7212 , 1 7070 0128 , 1 1468 4865 ,
20206     1 , 1 1822 1188 , 1 0039 0508 , 1 9748 7537 ,
20207     1 , 1 2013 7527 , 1 0747 0476 , 1 5216 2455 ,
20208     1 , 1 2225 5409 , 1 2849 2467 , 1 6045 7954 ,
20209     1 , 1 2459 6031 , 1 1115 6949 , 1 6638 0013 ,
20210     1 , 1 2718 2818 , 1 2845 9045 , 1 2353 6029 ,
20211     1 , 1 7389 0560 , 1 9893 0650 , 1 2272 3043 ,
20212     2 , 1 2008 5536 , 1 9231 8766 , 1 7740 9285 ,
20213     2 , 1 5459 8150 , 1 0331 4423 , 1 9078 1103 ,
20214     3 , 1 1484 1315 , 1 9102 5766 , 1 0342 1116 ,
20215     3 , 1 4034 2879 , 1 3492 7351 , 1 2260 8387 ,
20216     4 , 1 1096 6331 , 1 5842 8458 , 1 5992 6372 ,
20217     4 , 1 2980 9579 , 1 8704 1728 , 1 2747 4359 ,
20218     4 , 1 8103 0839 , 1 2757 5384 , 1 0077 1000 ,
20219     5 , 1 2202 6465 , 1 7948 0671 , 1 6516 9579 ,
20220     9 , 1 4851 6519 , 1 5409 7902 , 1 7796 9107 ,
20221     14 , 1 1068 6474 , 1 5815 2446 , 1 2146 9905 ,
20222     18 , 1 2353 8526 , 1 6837 0199 , 1 8540 7900 ,
20223     22 , 1 5184 7055 , 1 2858 7072 , 1 4640 8745 ,
20224     27 , 1 1142 0073 , 1 8981 5684 , 1 2836 6296 ,
20225     31 , 1 2515 4386 , 1 7091 9167 , 1 0062 6578 ,
20226     35 , 1 5540 6223 , 1 8439 3510 , 1 0525 7117 ,
20227     40 , 1 1220 4032 , 1 9431 7840 , 1 8020 0271 ,
20228     44 , 1 2688 1171 , 1 4181 6135 , 1 4484 1263 ,
20229     87 , 1 7225 9737 , 1 6812 5749 , 1 2581 7748 ,
20230     131 , 1 1942 4263 , 1 9524 1255 , 1 9365 8421 ,
20231     174 , 1 5221 4696 , 1 8976 4143 , 1 9505 8876 ,
20232     218 , 1 1403 5922 , 1 1785 2837 , 1 4107 3977 ,
20233     261 , 1 3773 0203 , 1 0092 9939 , 1 8234 0143 ,
20234     305 , 1 1014 2320 , 1 5473 5004 , 1 5094 5533 ,
20235     348 , 1 2726 3745 , 1 7211 2566 , 1 5673 6478 ,
20236     391 , 1 7328 8142 , 1 2230 7421 , 1 7051 8866 ,
20237     435 , 1 1970 0711 , 1 1401 7046 , 1 9938 8888 ,

```

```

20238      869 , 1 3881 1801 , 1 9428 4368 , 1 5764 8232 ,
20239      1303 , 1 7646 2009 , 1 8905 4704 , 1 8893 1073 ,
20240      1738 , 1 1506 3559 , 1 7005 0524 , 1 9009 7592 ,
20241      2172 , 1 2967 6283 , 1 8402 3667 , 1 0689 6630 ,
20242      2606 , 1 5846 4389 , 1 5650 2114 , 1 7278 5046 ,
20243      3041 , 1 1151 7900 , 1 5080 6878 , 1 2914 4154 ,
20244      3475 , 1 2269 1083 , 1 0850 6857 , 1 8724 4002 ,
20245      3909 , 1 4470 3047 , 1 3316 5442 , 1 6408 6591 ,
20246      4343 , 1 8806 8182 , 1 2566 2921 , 1 5872 6150 ,
20247      8686 , 1 7756 0047 , 1 2598 6861 , 1 0458 3204 ,
20248      13029 , 1 6830 5723 , 1 7791 4884 , 1 1932 7351 ,
20249      17372 , 1 6015 5609 , 1 3095 3052 , 1 3494 7574 ,
20250      21715 , 1 5297 7951 , 1 6443 0315 , 1 3251 3576 ,
20251      26058 , 1 4665 6719 , 1 0099 3379 , 1 5527 2929 ,
20252      30401 , 1 4108 9724 , 1 3326 3186 , 1 5271 5665 ,
20253      34744 , 1 3618 6973 , 1 3140 0875 , 1 3856 4102 ,
20254      39087 , 1 3186 9209 , 1 6113 3900 , 1 6705 9685 ,
20255      }

```

(End definition for \c_fp_exp_intarray.)

_fp_exp_pos_large:NnnNwn The first two arguments are irrelevant (a rounding digit, and a brace group with 8 zeros).
_fp_exp_large_after:wnn The third argument is the integer part of our number, then we have the decimal part
 _fp_exp_large:NwN delimited by a semicolon, and finally the exponent, in the range [0,5]. Remove leading
 _fp_exp_intarray:w zeros from the integer part: putting #4 in there too ensures that an integer part of 0 is
 _fp_exp_intarray_aux:w also removed. Then read digits one by one, looking up $\exp(\langle digit \rangle \cdot 10^{\langle exponent \rangle})$ in a table,
and multiplying that to the current total. The loop is done by _fp_exp_large:NwN,
whose #1 is the $\langle exponent \rangle$, #2 is the current mantissa, and #3 is the $\langle digit \rangle$. At the end,
_fp_exp_large_after:wnn moves on to the Taylor series, eventually multiplied with
the mantissa that we have just computed.

```

20256 \cs_new:Npn \_fp\_exp\_pos\_large:NnnNwn #1#2#3 #4#5; #6
20257 {
20258   \exp_after:wN \exp_after:wN \exp_after:wN \_fp\_exp\_large:NwN
20259   \exp_after:wN \exp_after:wN \exp_after:wN #6
20260   \exp_after:wN \c\_fp\_one\_fixed\_tl
20261   \int_value:w #3 #4 \exp_stop_f:
20262   #5 00000 ;
20263 }
20264 \cs_new:Npn \_fp\_exp\_large:NwN #1#2; #3
20265 {
20266   \if_case:w #3 ~
20267     \exp_after:wN \_fp\_fixed\_continue:wn
20268   \else:
20269     \exp_after:wN \_fp\_exp\_intarray:w
20270     \int_value:w \_fp\_int\_eval:w 36 * #1 + 4 * #3 \exp_after:wN ;
20271   \fi:
20272   #2;
20273   {
20274     \if_meaning:w 0 #1
20275       \exp_after:wN \_fp\_exp\_large\_after:wnn
20276     \else:
20277       \exp_after:wN \_fp\_exp\_large:NwN
20278       \int_value:w \_fp\_int\_eval:w #1 - 1 \exp_after:wN \scan_stop:
20279     \fi:

```

```

20280     }
20281   }
20282   \cs_new:Npn \__fp_exp_intarray:w #1 ;
20283   {
20284     +
20285     \__kernel_intarray_item:Nn \c__fp_exp_intarray
20286     { \__fp_int_eval:w #1 - 3 \scan_stop: }
20287     \exp_after:wN \use_i:nnn
20288     \exp_after:wN \__fp_fixed_mul:wwn
20289     \int_value:w 0
20290     \exp_after:wN \__fp_exp_intarray_aux:w
20291     \int_value:w \__kernel_intarray_item:Nn
20292     \c__fp_exp_intarray { \__fp_int_eval:w #1 - 2 }
20293     \exp_after:wN \__fp_exp_intarray_aux:w
20294     \int_value:w \__kernel_intarray_item:Nn
20295     \c__fp_exp_intarray { \__fp_int_eval:w #1 - 1 }
20296     \exp_after:wN \__fp_exp_intarray_aux:w
20297     \int_value:w \__kernel_intarray_item:Nn \c__fp_exp_intarray {#1} ; ;
20298   }
20299   \cs_new:Npn \__fp_exp_intarray_aux:w 1 #1#2#3#4#5 ; { ; {#1#2#3#4} {#5} }
20300   \cs_new:Npn \__fp_exp_large_after:wwn #1; #2; #3
20301   {
20302     \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2; { } #3
20303     \__fp_fixed_mul:wwn #1;
20304   }

```

(End definition for `__fp_exp_pos_large:NnnNwn` and others.)

33.3 Power

Raising a number a to a power b leads to many distinct situations.

a^b	$-\infty$	$(-\infty, -0)$	$-\text{integer}$	± 0	$+\text{integer}$	$(0, \infty)$	$+\infty$	NaN
$+\infty$	$+0$		$+0$	$+1$	$+\infty$		$+\infty$	NaN
$(1, \infty)$	$+0$		$+ a ^b$	$+1$	$+ a ^b$		$+\infty$	NaN
$+1$	$+1$		$+1$	$+1$	$+1$		$+1$	$+1$
$(0, 1)$	$+\infty$		$+ a ^b$	$+1$	$+ a ^b$		$+0$	NaN
$+0$	$+\infty$		$+\infty$	$+1$	$+0$		$+0$	NaN
-0	$+\infty$	NaN	$(-1)^b \infty$	$+1$	$(-1)^b 0$	$+0$	$+0$	NaN
$(-1, 0)$	$+\infty$	NaN	$(-1)^b a ^b$	$+1$	$(-1)^b a ^b$	NaN	$+0$	NaN
-1	$+1$	NaN	$(-1)^b$	$+1$	$(-1)^b$	NaN	$+1$	NaN
$(-\infty, -1)$	$+0$	NaN	$(-1)^b a ^b$	$+1$	$(-1)^b a ^b$	NaN	$+\infty$	NaN
$-\infty$	$+0$	$+0$	$(-1)^b 0$	$+1$	$(-1)^b \infty$	NaN	$+\infty$	NaN
NaN	NaN	NaN	NaN	$+1$	NaN	NaN	NaN	NaN

We distinguished in this table the cases of finite (positive or negative) integer exponents, as $(-1)^b$ is defined in that case. One peculiarity of this operation is that $\text{NaN}^0 = 1^{\text{NaN}} = 1$, because this relation is obeyed for any number, even $\pm\infty$.

`__fp_~_o:ww` We cram most of the tests into a single function to save csnames. First treat the case $b = 0$: $a^0 = 1$ for any a , even nan. Then test the sign of a .

- If it is positive, and a is a normal number, call `__fp_pow_normal_o:ww` followed by the two `fp` a and b . For $a = +0$ or $+\infty$, call `__fp_pow_zero_or_inf:ww` instead, to return either $+0$ or $+\infty$ as appropriate.
- If a is a `nan`, then skip to the next semicolon (which happens to be conveniently the end of b) and return `nan`.
- Finally, if a is negative, compute a^b (`__fp_pow_normal_o:ww` which ignores the sign of its first operand), and keep an extra copy of a and b (the second brace group, containing $\{ b a \}$, is inserted between a and b). Then do some tests to find the final sign of the result if it exists.

```

20305 \cs_new:cpn { __fp_ \iow_char:N \^_ _o:ww }
20306   \s__fp \__fp_chk:w #1#2#3; \s__fp \__fp_chk:w #4#5#6;
20307   {
20308     \if_meaning:w 0 #4
20309     \__fp_case_return_o:Nw \c_one_fp
20310     \fi:
20311     \if_case:w #2 \exp_stop_f:
20312     \exp_after:wN \use_i:nn
20313     \or:
20314     \__fp_case_return_o:Nw \c_nan_fp
20315     \else:
20316     \exp_after:wN \__fp_pow_neg:www
20317     \exp:w \exp_end_continue_f:w \exp_after:wN \use:nn
20318     \fi:
20319     {
20320       \if_meaning:w 1 #1
20321       \exp_after:wN \__fp_pow_normal_o:ww
20322       \else:
20323       \exp_after:wN \__fp_pow_zero_or_inf:ww
20324       \fi:
20325       \s__fp \__fp_chk:w #1#2#3;
20326     }
20327     { \s__fp \__fp_chk:w #4#5#6; \s__fp \__fp_chk:w #1#2#3; }
20328     \s__fp \__fp_chk:w #4#5#6;
20329   }

```

(End definition for `__fp_~_o:ww`.)

`__fp_pow_zero_or_inf:ww` Raising -0 or $-\infty$ to `nan` yields `nan`. For other powers, the result is $+0$ if 0 is raised to a positive power or ∞ to a negative power, and $+\infty$ otherwise. Thus, if the type of a and the sign of b coincide, the result is 0 , since those conveniently take the same possible values, 0 and 2 . Otherwise, either $a = \pm\infty$ and $b > 0$ and the result is $+\infty$, or $a = \pm 0$ with $b < 0$ and we have a division by zero unless $b = -\infty$.

```

20330 \cs_new:Npn \__fp_pow_zero_or_inf:ww
20331   \s__fp \__fp_chk:w #1#2; \s__fp \__fp_chk:w #3#4
20332   {
20333     \if_meaning:w 1 #4
20334     \__fp_case_return_same_o:w
20335     \fi:
20336     \if_meaning:w #1 #4
20337     \__fp_case_return_o:Nw \c_zero_fp
20338     \fi:

```



```

20339 \if_meaning:w 2 #1
20340 \__fp_case_return_o:Nw \c_inf_fp
20341 \fi:
20342 \if_meaning:w 2 #3
20343 \__fp_case_return_o:Nw \c_inf_fp
20344 \else:
20345 \__fp_case_use:nw
20346 {
20347 \__fp_division_by_zero_o:NNww \c_inf_fp ^
20348 \s__fp \__fp_chk:w #1 #2 ;
20349 }
20350 \fi:
20351 \s__fp \__fp_chk:w #3#4
20352 }

```

(End definition for `__fp_pow_zero_or_inf:ww`.)

`__fp_pow_normal_o:ww` We have in front of us a , and $b \neq 0$, we know that a is a normal number, and we wish to compute $|a|^b$. If $|a| = 1$, we return 1, unless $a = -1$ and b is `nan`. Indeed, returning 1 at this point would wrongly raise “invalid” when the sign is considered. If $|a| \neq 1$, test the type of b :

- 0 Impossible, we already filtered $b = \pm 0$.
- 1 Call `__fp_pow_npos_o:Nww`.
- 2 Return $+\infty$ or $+0$ depending on the sign of b and whether the exponent of a is positive or not.
- 3 Return b .

```

20353 \cs_new:Npn \__fp_pow_normal_o:ww
20354 \s__fp \__fp_chk:w 1 #1#2#3; \s__fp \__fp_chk:w #4#5
20355 {
20356 \if_int_compare:w \__fp_str_if_eq:nn { #2 #3 }
20357 { 1 {1000} {0000} {0000} {0000} } = 0 \exp_stop_f:
20358 \if_int_compare:w #4 #1 = 32 \exp_stop_f:
20359 \exp_after:wN \__fp_case_return_ii_o:ww
20360 \fi:
20361 \__fp_case_return_o:Nww \c_one_fp
20362 \fi:
20363 \if_case:w #4 \exp_stop_f:
20364 \or:
20365 \exp_after:wN \__fp_pow_npos_o:Nww
20366 \exp_after:wN #5
20367 \or:
20368 \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
20369 \if_int_compare:w #2 > 0 \exp_stop_f:
20370 \exp_after:wN \__fp_case_return_o:Nww
20371 \exp_after:wN \c_inf_fp
20372 \else:
20373 \exp_after:wN \__fp_case_return_o:Nww
20374 \exp_after:wN \c_zero_fp
20375 \fi:
20376 \or:

```

```

20377     \__fp_case_return_ii_o:ww
20378 \fi:
20379 \s__fp \__fp_chk:w 1 #1 {#2} #3 ;
20380 \s__fp \__fp_chk:w #4 #5
20381 }

```

(End definition for __fp_pow_normal_o:ww.)

__fp_pow_npos_o:Nww We now know that $a \neq \pm 1$ is a normal number, and b is a normal number too. We want to compute $|a|^b = (|x| \cdot 10^n)^{y \cdot 10^p} = \exp((\ln|x| + n \ln(10)) \cdot y \cdot 10^p) = \exp(z)$. To compute the exponential accurately, we need to know the digits of z up to the 16-th position. Since the exponential of 10^5 is infinite, we only need at most 21 digits, hence the fixed point result of __fp_ln_o:w is precise enough for our needs. Start an integer expression for the decimal exponent of $e^{|z|}$. If z is negative, negate that decimal exponent, and prepare to take the inverse when converting from the fixed point to the floating point result.

```

20382 \cs_new:Npn \__fp_pow_npos_o:Nww #1 \s__fp \__fp_chk:w 1#2#3
20383 {
20384   \exp_after:wN \__fp_sanitize:Nw
20385   \exp_after:wN 0
20386   \int_value:w
20387   \if:w #1 \if_int_compare:w #3 > 0 \exp_stop_f: 0 \else: 2 \fi:
20388   \exp_after:wN \__fp_pow_npos_aux:NNnww
20389   \exp_after:wN +
20390   \exp_after:wN \__fp_fixed_to_float_o:wN
20391   \else:
20392   \exp_after:wN \__fp_pow_npos_aux:NNnww
20393   \exp_after:wN -
20394   \exp_after:wN \__fp_fixed_inv_to_float_o:wN
20395   \fi:
20396   {#3}
20397 }

```

(End definition for __fp_pow_npos_o:Nww.)

__fp_pow_npos_aux:NNnww The first argument is the conversion function from fixed point to float. Then comes an exponent and the 4 brace groups of x , followed by b . Compute $-\ln(x)$.

```

20398 \cs_new:Npn \__fp_pow_npos_aux:NNnww #1#2#3#4#5; \s__fp \__fp_chk:w 1#6#7#8;
20399 {
20400   #1
20401   \__fp_int_eval:w
20402   \__fp_ln_significand:NNNNnnnnN #4#5
20403   \__fp_pow_exponent:wnN {#3}
20404   \__fp_fixed_mul:wwn #8 {0000}{0000} ;
20405   \__fp_pow_B:wwN #7;
20406   #1 #2 0 % fixed_to_float_o:wN
20407 }
20408 \cs_new:Npn \__fp_pow_exponent:wnN #1; #2
20409 {
20410   \if_int_compare:w #2 > 0 \exp_stop_f:
20411   \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % n\ln(10) - (-\ln(x))
20412   \exp_after:wN +
20413   \else:
20414   \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % -(\ln|\ln(10) + (-\ln(x)))
20415   \exp_after:wN -

```

```

20416     \fi:
20417     #2; #1;
20418 }
20419 \cs_new:Npn \__fp_pow_exponent:Nwnnnnnw #1#2; #3#4#5#6#7#8;
20420 { %^A todo: use that in ln.
20421   \exp_after:wN \__fp_fixed_mul_after:wwn
20422   \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
20423   \exp_after:wN \__fp_pack:NNNNNw
20424   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
20425   #1#2*23025 - #1 #3
20426   \exp_after:wN \__fp_pack:NNNNNw
20427   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
20428   #1 #2*8509 - #1 #4
20429   \exp_after:wN \__fp_pack:NNNNNw
20430   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
20431   #1 #2*2994 - #1 #5
20432   \exp_after:wN \__fp_pack:NNNNNw
20433   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
20434   #1 #2*0456 - #1 #6
20435   \exp_after:wN \__fp_pack:NNNNNw
20436   \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
20437   #1 #2*8401 - #1 #7
20438   #1 ( #2*7991 - #8 ) / 1 0000 ; ;
20439 }
20440 \cs_new:Npn \__fp_pow_B:wwN #1#2#3#4#5#6; #7;
20441 {
20442   \if_int_compare:w #7 < 0 \exp_stop_f:
20443     \exp_after:wN \__fp_pow_C_neg:w \int_value:w -
20444   \else:
20445     \if_int_compare:w #7 < 22 \exp_stop_f:
20446       \exp_after:wN \__fp_pow_C_pos:w \int_value:w
20447     \else:
20448       \exp_after:wN \__fp_pow_C_overflow:w \int_value:w
20449     \fi:
20450   \fi:
20451   #7 \exp_after:wN ;
20452   \int_value:w \__fp_int_eval:w 10 0000 + #1 \__fp_int_eval_end:
20453   #2#3#4#5#6 0000 0000 0000 0000 0000 0000 ; %^A todo: how many 0?
20454 }
20455 \cs_new:Npn \__fp_pow_C_overflow:w #1; #2; #3
20456 {
20457   + 2 * \c__fp_max_exponent_int
20458   \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl
20459 }
20460 \cs_new:Npn \__fp_pow_C_neg:w #1 ; 1
20461 {
20462   \exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w
20463   \prg_replicate:nn {#1} {0}
20464 }
20465 \cs_new:Npn \__fp_pow_C_pos:w #1; 1
20466 { \__fp_pow_C_pos_loop:wN #1; }
20467 \cs_new:Npn \__fp_pow_C_pos_loop:wN #1; #2
20468 {
20469   \if_meaning:w 0 #1

```

```

20470     \exp_after:wN \__fp_pow_C_pack:w
20471     \exp_after:wN #2
20472   \else:
20473     \if_meaning:w 0 #2
20474     \exp_after:wN \__fp_pow_C_pos_loop:wN \int_value:w
20475   \else:
20476     \exp_after:wN \__fp_pow_C_overflow:w \int_value:w
20477   \fi:
20478   \__fp_int_eval:w #1 - 1 \exp_after:wN ;
20479 \fi:
20480 }
20481 \cs_new:Npn \__fp_pow_C_pack:w
20482 {
20483   \exp_after:wN \__fp_exp_large:NwN
20484   \exp_after:wN 5
20485   \c__fp_one_fixed_tl
20486 }

```

(End definition for __fp_pow_npos_aux:NNnww.)

__fp_pow_neg:www
__fp_pow_neg_aux:wNN

This function is followed by three floating point numbers: a^b , $a \in [-\infty, -0]$, and b . If b is an even integer (case -1), $a^b = a^b$. If b is an odd integer (case 0), $a^b = -a^b$, obtained by a call to __fp_pow_neg_aux:wNN. Otherwise, the sign is undefined. This is invalid, unless a^b turns out to be $+0$ or nan , in which case we return that as a^b . In particular, since the underflow detection occurs before __fp_pow_neg:www is called, $(-0.1)**(12345.67)$ gives $+0$ rather than complaining that the sign is not defined.

```

20487 \cs_new:Npn \__fp_pow_neg:www \s__fp \__fp_chk:w #1#2; #3; #4;
20488 {
20489   \if_case:w \__fp_pow_neg_case:w #4 ;
20490     \exp_after:wN \__fp_pow_neg_aux:wNN
20491   \or:
20492     \if_int_compare:w \__fp_int_eval:w #1 / 2 = 1 \exp_stop_f:
20493     \__fp_invalid_operation_o:Nww ^ #3; #4;
20494     \exp:w \exp_end_continue_f:w
20495     \exp_after:wN \exp_after:wN
20496     \exp_after:wN \__fp_use_none_until_s:w
20497   \fi:
20498 \fi:
20499   \__fp_exp_after_o:w
20500   \s__fp \__fp_chk:w #1#2;
20501 }
20502 \cs_new:Npn \__fp_pow_neg_aux:wNN #1 \s__fp \__fp_chk:w #2#3
20503 {
20504   \exp_after:wN \__fp_exp_after_o:w
20505   \exp_after:wN \s__fp
20506   \exp_after:wN \__fp_chk:w
20507   \exp_after:wN #2
20508   \int_value:w \__fp_int_eval:w 2 - #3 \__fp_int_eval_end:
20509 }

```

(End definition for __fp_pow_neg:www and __fp_pow_neg_aux:wNN.)

__fp_pow_neg_case:w
__fp_pow_neg_case_aux:nnnnn
__fp_pow_neg_case_aux:Nnnw

This function expects a floating point number, and determines its “parity”. It should be used after \if_case:w or in an integer expression. It gives -1 if the number is an

even integer, 0 if the number is an odd integer, and 1 otherwise. Zeros and $\pm\infty$ are even (because very large finite floating points are even), while `nan` is a non-integer. The sign of normal numbers is irrelevant to parity. After `__fp_decimate:nNnnnn` the argument #1 of `__fp_pow_neg_case_aux:Nnnw` is a rounding digit, 0 if and only if the number was an integer, and #3 is the 8 least significant digits of that integer.

```

20510 \cs_new:Npn \__fp_pow_neg_case:w \s__fp \__fp_chk:w #1#2#3;
20511 {
20512   \if_case:w #1 \exp_stop_f:
20513     -1
20514   \or:   \__fp_pow_neg_case_aux:nnnnn #3
20515   \or:   -1
20516   \else: 1
20517   \fi:
20518   \exp_stop_f:
20519 }
20520 \cs_new:Npn \__fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
20521 {
20522   \if_int_compare:w #1 > \c__fp_prec_int
20523     -1
20524   \else:
20525     \__fp_decimate:nNnnnn { \c__fp_prec_int - #1 }
20526     \__fp_pow_neg_case_aux:Nnnw
20527     {#2} {#3} {#4} {#5}
20528   \fi:
20529 }
20530 \cs_new:Npn \__fp_pow_neg_case_aux:Nnnw #1#2#3#4 ;
20531 {
20532   \if_meaning:w 0 #1
20533     \if_int_odd:w #3 \exp_stop_f:
20534     0
20535   \else:
20536     -1
20537   \fi:
20538   \else:
20539     1
20540   \fi:
20541 }

```

(End definition for `__fp_pow_neg_case:w`, `__fp_pow_neg_case_aux:nnnnn`, and `__fp_pow_neg_case_aux:Nnnw`.)

33.4 Factorial

`\c__fp_fact_max_arg_int` The maximum integer whose factorial fits in the exponent range is 3248, as $3249! \sim 10^{10000.8}$

```

20542 \int_const:Nn \c__fp_fact_max_arg_int { 3248 }

```

(End definition for `\c__fp_fact_max_arg_int`.)

`__fp_fact_o:w` First detect ± 0 and $+\infty$ and `nan`. Then note that factorial of anything with a negative sign (except -0) is undefined. Then call `__fp_small_int:wTF` to get an integer as the argument, and start a loop. This is not the most efficient way of computing the factorial,

but it works all right. Of course we work with 24 digits instead of 16. It is easy to check that computing factorials with this precision is enough.

```

20543 \cs_new:Npn \__fp_fact_o:w #1 \s_fp \__fp_chk:w #2#3#4; @
20544 {
20545   \if_case:w #2 \exp_stop_f:
20546     \__fp_case_return_o:Nw \c_one_fp
20547   \or:
20548   \or:
20549     \if_meaning:w 0 #3
20550     \exp_after:wN \__fp_case_return_same_o:w
20551   \fi:
20552   \or:
20553     \__fp_case_return_same_o:w
20554   \fi:
20555   \if_meaning:w 2 #3
20556     \__fp_case_use:nw { \__fp_invalid_operation_o:fw { fact } }
20557   \fi:
20558   \__fp_fact_pos_o:w
20559   \s_fp \__fp_chk:w #2 #3 #4 ;
20560 }

```

(End definition for __fp_fact_o:w.)

__fp_fact_pos_o:w Then check the input is an integer, and call __fp_facorial_int_o:n with that int as
 __fp_fact_int_o:w an argument. If it's too big the factorial overflows. Otherwise call __fp_sanitize:Nw
 with a positive sign marker 0 and an integer expression that will mop up any exponent
 in the calculation.

```

20561 \cs_new:Npn \__fp_fact_pos_o:w #1;
20562 {
20563   \__fp_small_int:wTF #1;
20564   { \__fp_fact_int_o:n }
20565   { \__fp_invalid_operation_o:fw { fact } #1; }
20566 }
20567 \cs_new:Npn \__fp_fact_int_o:n #1
20568 {
20569   \if_int_compare:w #1 > \c__fp_fact_max_arg_int
20570     \__fp_case_return:nw
20571     {
20572       \exp_after:wN \exp_after:wN \exp_after:wN \__fp_overflow:w
20573       \exp_after:wN \c_inf_fp
20574     }
20575   \fi:
20576   \exp_after:wN \__fp_sanitize:Nw
20577   \exp_after:wN 0
20578   \int_value:w \__fp_int_eval:w
20579   \__fp_fact_loop_o:w #1 . 4 , { 1 } { } { } { } { } { } { } ;
20580 }

```

(End definition for __fp_fact_pos_o:w and __fp_fact_int_o:w.)

__fp_fact_loop_o:w The loop receives an integer #1 whose factorial we want to compute, which we progres-
 sively decrement, and the result so far as an extended-precision number #2 in the form
 $\langle \text{exponent} \rangle, \langle \text{mantissa} \rangle$; The loop goes in steps of two because we compute $\#1 \cdot \#1 - 1$
 as an integer expression (it must fit since #1 is at most 3248), then multiply with the

result so far. We don't need to fill in most of the mantissa with zeros because `__fp_ep_mul:wwwn` first normalizes the extended precision number to avoid loss of precision. When reaching a small enough number simply use a table of factorials less than 10^8 . This limit is chosen because the normalization step cannot deal with larger integers.

```

20581 \cs_new:Npn \__fp_fact_loop_o:w #1 . #2 ;
20582 {
20583   \if_int_compare:w #1 < 12 \exp_stop_f:
20584     \__fp_fact_small_o:w #1
20585   \fi:
20586   \exp_after:wN \__fp_ep_mul:wwwn
20587   \exp_after:wN 4 \exp_after:wN ,
20588   \exp_after:wN { \int_value:w \__fp_int_eval:w #1 * (#1 - 1) }
20589   { } { } { } { } { } { } ;
20590   #2 ;
20591   {
20592     \exp_after:wN \__fp_fact_loop_o:w
20593     \int_value:w \__fp_int_eval:w #1 - 2 .
20594   }
20595 }
20596 \cs_new:Npn \__fp_fact_small_o:w #1 \fi: #2 ; #3 ; #4
20597 {
20598   \fi:
20599   \exp_after:wN \__fp_ep_mul:wwwn
20600   \exp_after:wN 4 \exp_after:wN ,
20601   \exp_after:wN
20602   {
20603     \int_value:w
20604     \if_case:w #1 \exp_stop_f:
20605       1 \or: 1 \or: 2 \or: 6 \or: 24 \or: 120 \or: 720 \or: 5040
20606       \or: 40320 \or: 362880 \or: 3628800 \or: 39916800
20607     \fi:
20608     } { } { } { } { } { } { } ;
20609     #3 ;
20610     \__fp_ep_to_float_o:wwN 0
20611   }

```

(End definition for `__fp_fact_loop_o:w`.)

```

20612 \</package>

```

34 l3fp-trig Implementation

```

20613 \*package>
20614 \@@=fp>

```

Unary functions.

```

\__fp_parse_word_acos:N
\__fp_parse_word_acosd:N
\__fp_parse_word_acsc:N
\__fp_parse_word_acscd:N
\__fp_parse_word_asec:N
\__fp_parse_word_asecd:N
\__fp_parse_word_asin:N
\__fp_parse_word_asind:N
\__fp_parse_word_cos:N
\__fp_parse_word_cosd:N
\__fp_parse_word_cot:N
\__fp_parse_word_cotd:N
\__fp_parse_word_csc:N
\__fp_parse_word_cscd:N
\__fp_parse_word_sec:N
\__fp_parse_word_secd:N
\__fp_parse_word_sin:N
\__fp_parse_word_sind:N

```

```

20615 \tl_map_inline:nn
20616 {
20617   {acos} {acsc} {asec} {asin}
20618   {cos} {cot} {csc} {sec} {sin} {tan}
20619 }
20620 {
20621   \cs_new:cpx { __fp_parse_word_#1:N }

```

```

20622     {
20623         \exp_not:N \__fp_parse_unary_function:NNN
20624         \exp_not:c { __fp_#1_o:w }
20625         \exp_not:N \use_i:nn
20626     }
20627 \cs_new:cpx { __fp_parse_word_#1d:N }
20628 {
20629     \exp_not:N \__fp_parse_unary_function:NNN
20630     \exp_not:c { __fp_#1_o:w }
20631     \exp_not:N \use_ii:nn
20632 }
20633 }

```

(End definition for `__fp_parse_word_acos:N` and others.)

```

\__fp_parse_word_acot:N Those functions may receive a variable number of arguments.
\__fp_parse_word_acotd:N
\__fp_parse_word_atan:N
\__fp_parse_word_atand:N
20634 \cs_new:Npn \__fp_parse_word_acot:N
20635 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_i:nn }
20636 \cs_new:Npn \__fp_parse_word_acotd:N
20637 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_ii:nn }
20638 \cs_new:Npn \__fp_parse_word_atan:N
20639 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_i:nn }
20640 \cs_new:Npn \__fp_parse_word_atand:N
20641 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_ii:nn }

```

(End definition for `__fp_parse_word_acot:N` and others.)

34.1 Direct trigonometric functions

The approach for all trigonometric functions (sine, cosine, tangent, cotangent, cosecant, and secant), with arguments given in radians or in degrees, is the same.

- Filter out special cases (± 0 , $\pm \infty$ and NaN).
- Keep the sign for later, and work with the absolute value $|x|$ of the argument.
- Small numbers ($|x| < 1$ in radians, $|x| < 10$ in degrees) are converted to fixed point numbers (and to radians if $|x|$ is in degrees).
- For larger numbers, we need argument reduction. Subtract a multiple of $\pi/2$ (in degrees, 90) to bring the number to the range to $[0, \pi/2)$ (in degrees, $[0, 90)$).
- Reduce further to $[0, \pi/4]$ (in degrees, $[0, 45]$) using $\sin x = \cos(\pi/2 - x)$, and when working in degrees, convert to radians.
- Use the appropriate power series depending on the octant $\lfloor \frac{x}{\pi/4} \rfloor \bmod 8$ (in degrees, the same formula with $\pi/4 \rightarrow 45$), the sign, and the function to compute.

34.1.1 Filtering special cases

`__fp_sin_o:w` This function, and its analogs for `cos`, `csc`, `sec`, `tan`, and `cot` instead of `sin`, are followed either by `\use_i:nn` and a float in radians or by `\use_ii:nn` and a float in degrees. The sine of ± 0 or NaN is the same float. The sine of $\pm \infty$ raises an invalid operation exception with the appropriate function name. Otherwise, call the `trig` function to perform argument reduction and if necessary convert the reduced argument to radians.

Then, `__fp_sin_series_o:NNwww` is called to compute the Taylor series: this function receives a sign `#3`, an initial octant of 0, and the function `__fp_ep_to_float_o:wwN` which converts the result of the series to a floating point directly rather than taking its inverse, since $\sin(x) = \#3 \sin|x|$.

```

20642 \cs_new:Npn \__fp_sin_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
20643 {
20644   \if_case:w #2 \exp_stop_f:
20645     \__fp_case_return_same_o:w
20646   \or: \__fp_case_use:nw
20647     {
20648       \__fp_trig:NNNNwn #1 \__fp_sin_series_o:NNwww
20649       \__fp_ep_to_float_o:wwN #3 0
20650     }
20651   \or: \__fp_case_use:nw
20652     { \__fp_invalid_operation_o:fw { #1 { sin } { sind } } }
20653   \else: \__fp_case_return_same_o:w
20654   \fi:
20655   \s__fp \__fp_chk:w #2 #3 #4;
20656 }

```

(End definition for `__fp_sin_o:w`.)

`__fp_cos_o:w` The cosine of ± 0 is 1. The cosine of $\pm\infty$ raises an invalid operation exception. The cosine of NaN is itself. Otherwise, the `trig` function reduces the argument to at most half a right-angle and converts if necessary to radians. We then call the same series as for sine, but using a positive sign 0 regardless of the sign of x , and with an initial octant of 2, because $\cos(x) = +\sin(\pi/2 + |x|)$.

```

20657 \cs_new:Npn \__fp_cos_o:w #1 \s__fp \__fp_chk:w #2#3; @
20658 {
20659   \if_case:w #2 \exp_stop_f:
20660     \__fp_case_return_o:Nw \c_one_fp
20661   \or: \__fp_case_use:nw
20662     {
20663       \__fp_trig:NNNNwn #1 \__fp_sin_series_o:NNwww
20664       \__fp_ep_to_float_o:wwN 0 2
20665     }
20666   \or: \__fp_case_use:nw
20667     { \__fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
20668   \else: \__fp_case_return_same_o:w
20669   \fi:
20670   \s__fp \__fp_chk:w #2 #3;
20671 }

```

(End definition for `__fp_cos_o:w`.)

`__fp_csc_o:w` The cosecant of ± 0 is $\pm\infty$ with the same sign, with a division by zero exception (see `__fp_cot_zero_o:Nfw` defined below), which requires the function name. The cosecant of $\pm\infty$ raises an invalid operation exception. The cosecant of NaN is itself. Otherwise, the `trig` function performs the argument reduction, and converts if necessary to radians before calling the same series as for sine, using the sign `#3`, a starting octant of 0, and inverting during the conversion from the fixed point sine to the floating point result, because $\csc(x) = \#3(\sin|x|)^{-1}$.

```

20672 \cs_new:Npn \__fp_csc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @

```

```

20673 {
20674     \if_case:w #2 \exp_stop_f:
20675         \__fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
20676     \or: \__fp_case_use:nw
20677         {
20678             \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
20679             \__fp_ep_inv_to_float_o:wwN #3 0
20680         }
20681     \or: \__fp_case_use:nw
20682         { \__fp_invalid_operation_o:fw { #1 { csc } { cscd } } }
20683     \else: \__fp_case_return_same_o:w
20684     \fi:
20685     \s__fp \__fp_chk:w #2 #3 #4;
20686 }

```

(End definition for __fp_csc_o:w.)

__fp_sec_o:w The secant of ± 0 is 1. The secant of $\pm\infty$ raises an invalid operation exception. The secant of NaN is itself. Otherwise, the `trig` function reduces the argument and turns it to radians before calling the same series as for sine, using a positive sign 0, a starting octant of 2, and inverting upon conversion, because $\sec(x) = +1/\sin(\pi/2 + |x|)$.

```

20687 \cs_new:Npn \__fp_sec_o:w #1 \s__fp \__fp_chk:w #2#3; @
20688 {
20689     \if_case:w #2 \exp_stop_f:
20690         \__fp_case_return_o:Nw \c_one_fp
20691     \or: \__fp_case_use:nw
20692         {
20693             \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
20694             \__fp_ep_inv_to_float_o:wwN 0 2
20695         }
20696     \or: \__fp_case_use:nw
20697         { \__fp_invalid_operation_o:fw { #1 { sec } { secd } } }
20698     \else: \__fp_case_return_same_o:w
20699     \fi:
20700     \s__fp \__fp_chk:w #2 #3;
20701 }

```

(End definition for __fp_sec_o:w.)

__fp_tan_o:w The tangent of ± 0 or NaN is the same floating point number. The tangent of $\pm\infty$ raises an invalid operation exception. Once more, the `trig` function does the argument reduction step and conversion to radians before calling `__fp_tan_series_o:NNwww`, with a sign #3 and an initial octant of 1 (this shift is somewhat arbitrary). See `__fp_cot_o:w` for an explanation of the 0 argument.

```

20702 \cs_new:Npn \__fp_tan_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
20703 {
20704     \if_case:w #2 \exp_stop_f:
20705         \__fp_case_return_same_o:w
20706     \or: \__fp_case_use:nw
20707         {
20708             \__fp_trig:NNNNNwn #1
20709             \__fp_tan_series_o:NNwww 0 #3 1
20710         }
20711     \or: \__fp_case_use:nw

```

```

20712         { \__fp_invalid_operation_o:fw { #1 { tan } { tand } } }
20713     \else: \__fp_case_return_same_o:w
20714     \fi:
20715     \s__fp \__fp_chk:w #2 #3 #4;
20716 }

```

(End definition for __fp_tan_o:w.)

__fp_cot_o:w
__fp_cot_zero_o:Nfw

The cotangent of ± 0 is $\pm\infty$ with the same sign, with a division by zero exception (see __fp_cot_zero_o:Nfw. The cotangent of $\pm\infty$ raises an invalid operation exception. The cotangent of NaN is itself. We use $\cot x = -\tan(\pi/2 + x)$, and the initial octant for the tangent was chosen to be 1, so the octant here starts at 3. The change in sign is obtained by feeding __fp_tan_series_o:NNwww two signs rather than just the sign of the argument: the first of those indicates whether we compute tangent or cotangent. Those signs are eventually combined.

```

20717 \cs_new:Npn \__fp_cot_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
20718 {
20719     \if_case:w #2 \exp_stop_f:
20720         \__fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
20721     \or: \__fp_case_use:nw
20722         {
20723             \__fp_trig:NNNNwn #1
20724             \__fp_tan_series_o:NNwww 2 #3 3
20725         }
20726     \or: \__fp_case_use:nw
20727         { \__fp_invalid_operation_o:fw { #1 { cot } { cotd } } }
20728     \else: \__fp_case_return_same_o:w
20729     \fi:
20730     \s__fp \__fp_chk:w #2 #3 #4;
20731 }
20732 \cs_new:Npn \__fp_cot_zero_o:Nfw #1#2#3 \fi:
20733 {
20734     \fi:
20735     \token_if_eq_meaning:NNTF 0 #1
20736     { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_inf_fp }
20737     { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_minus_inf_fp }
20738     {#2}
20739 }

```

(End definition for __fp_cot_o:w and __fp_cot_zero_o:Nfw.)

34.1.2 Distinguishing small and large arguments

__fp_trig:NNNNwn

The first argument is \use_i:nn if the operand is in radians and \use_ii:nn if it is in degrees. Arguments #2 to #5 control what trigonometric function we compute, and #6 to #8 are pieces of a normal floating point number. Call the _series function #2, with arguments #3, either a conversion function (__fp_ep_to_float_o:wN or __fp_ep_inv_to_float_o:wN) or a sign 0 or 2 when computing tangent or cotangent; #4, a sign 0 or 2; the octant, computed in an integer expression starting with #5 and stopped by a period; and a fixed point number obtained from the floating point number by argument reduction (if necessary) and conversion to radians (if necessary). Any argument reduction adjusts the octant accordingly by leaving a (positive) shift into its integer expression. Let us explain the integer comparison. Two of the four \exp_after:wN are expanded, the

expansion hits the test, which is true if the float is at least 1 when working in radians, and at least 10 when working in degrees. Then one of the remaining `\exp_after:wN` hits `#1`, which picks the `trig` or `trigd` function in whichever branch of the conditional was taken. The final `\exp_after:wN` closes the conditional. At the end of the day, a number is `large` if it is ≥ 1 in radians or ≥ 10 in degrees, and `small` otherwise. All four `trig/trigd` auxiliaries receive the operand as an extended-precision number.

```

20740 \cs_new:Npn \__fp_trig:NNNNwn #1#2#3#4#5 \s__fp \__fp_chk:w 1#6#7#8;
20741 {
20742   \exp_after:wN #2
20743   \exp_after:wN #3
20744   \exp_after:wN #4
20745   \int_value:w \__fp_int_eval:w #5
20746   \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
20747   \if_int_compare:w #7 > #1 0 1 \exp_stop_f:
20748     #1 \__fp_trig_large:ww \__fp_trigd_large:ww
20749   \else:
20750     #1 \__fp_trig_small:ww \__fp_trigd_small:ww
20751   \fi:
20752   #7,#8{0000}{0000};
20753 }

```

(End definition for `__fp_trig:NNNNwn`.)

34.1.3 Small arguments

`__fp_trig_small:ww` This receives a small extended-precision number in radians and converts it to a fixed point number. Some trailing digits may be lost in the conversion, so we keep the original floating point number around: when computing sine or tangent (or their inverses), the last step is to multiply by the floating point number (as an extended-precision number) rather than the fixed point number. The period serves to end the integer expression for the octant.

```

20754 \cs_new:Npn \__fp_trig_small:ww #1,#2;
20755 { \__fp_ep_to_fixed:wwn #1,#2; . #1,#2; }

```

(End definition for `__fp_trig_small:ww`.)

`__fp_trigd_small:ww` Convert the extended-precision number to radians, then call `__fp_trig_small:ww` to massage it in the form appropriate for the `_series` auxiliary.

```

20756 \cs_new:Npn \__fp_trigd_small:ww #1,#2;
20757 {
20758   \__fp_ep_mul_raw:wwwN
20759   -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;
20760   \__fp_trig_small:ww
20761 }

```

(End definition for `__fp_trigd_small:ww`.)

34.1.4 Argument reduction in degrees

`__fp_trigd_large:ww` Note that $25 \times 360 = 9000$, so $10^{k+1} \equiv 10^k \pmod{360}$ for $k \geq 3$. When the exponent `#1` is very large, we can thus safely replace it by 22 (or even 19). We turn the floating point number into a fixed point number with two blocks of 8 digits followed by five blocks of 4 digits. The original float is $100 \times \langle block_1 \rangle \cdots \langle block_3 \rangle . \langle block_4 \rangle \cdots \langle block_7 \rangle$, or is equal to

`__fp_trigd_large_auxi:nnnnwNNNN`

`__fp_trigd_large_auxii:wNw`

`__fp_trigd_large_auxiii:www`

it modulo 360 if the exponent #1 is very large. The first auxiliary finds $\langle block_1 \rangle + \langle block_2 \rangle$ (mod 9), a single digit, and prepends it to the 4 digits of $\langle block_3 \rangle$. It also unpacks $\langle block_4 \rangle$ and grabs the 4 digits of $\langle block_7 \rangle$. The second auxiliary grabs the $\langle block_3 \rangle$ plus any contribution from the first two blocks as #1, the first digit of $\langle block_4 \rangle$ (just after the decimal point in hundreds of degrees) as #2, and the three other digits as #3. It finds the quotient and remainder of #1#2 modulo 9, adds twice the quotient to the integer expression for the octant, and places the remainder (between 0 and 8) before #3 to form a new $\langle block_4 \rangle$. The resulting fixed point number is $x \in [0, 0.9]$. If $x \geq 0.45$, we add 1 to the octant and feed $0.9 - x$ with an exponent of 2 (to compensate the fact that we are working in units of hundreds of degrees rather than degrees) to `_fp_trigd_small:ww`. Otherwise, we feed it x with an exponent of 2. The third auxiliary also discards digits which were not packed into the various $\langle blocks \rangle$. Since the original exponent #1 is at least 2, those are all 0 and no precision is lost (#6 and #7 are four 0 each).

```

20762 \cs_new:Npn \_fp_trigd_large:ww #1, #2#3#4#5#6#7;
20763 {
20764   \exp_after:wN \_fp_pack_eight:wNNNNNNNN
20765   \exp_after:wN \_fp_pack_eight:wNNNNNNNN
20766   \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
20767   \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
20768   \exp_after:wN \_fp_trigd_large_auxi:nnnnwNNNN
20769   \exp_after:wN ;
20770   \exp:w \exp_end_continue_f:w
20771   \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
20772   #2#3#4#5#6#7 0000 0000 0000 !
20773 }
20774 \cs_new:Npn \_fp_trigd_large_auxi:nnnnwNNNN #1#2#3#4#5; #6#7#8#9
20775 {
20776   \exp_after:wN \_fp_trigd_large_auxii:wNw
20777   \int_value:w \_fp_int_eval:w #1 + #2
20778   - (#1 + #2 - 4) / 9 * 9 \_fp_int_eval_end:
20779   #3;
20780   #4; #5{#6#7#8#9};
20781 }
20782 \cs_new:Npn \_fp_trigd_large_auxii:wNw #1; #2#3;
20783 {
20784   + (#1#2 - 4) / 9 * 2
20785   \exp_after:wN \_fp_trigd_large_auxiii:www
20786   \int_value:w \_fp_int_eval:w #1#2
20787   - (#1#2 - 4) / 9 * 9 \_fp_int_eval_end: #3 ;
20788 }
20789 \cs_new:Npn \_fp_trigd_large_auxiii:www #1; #2; #3!
20790 {
20791   \if_int_compare:w #1 < 4500 \exp_stop_f:
20792   \exp_after:wN \_fp_use_i_until_s:nw
20793   \exp_after:wN \_fp_fixed_continue:wn
20794   \else:
20795     + 1
20796   \fi:
20797   \_fp_fixed_sub:wwn {9000}{0000}{0000}{0000}{0000}{0000};
20798   {#1}#2{0000}{0000};
20799   { \_fp_trigd_small:ww 2, }
20800 }

```

(End definition for `_fp_trigd_large:ww` and others.)

34.1.5 Argument reduction in radians

Arguments greater or equal to 1 need to be reduced to a range where we only need a few terms of the Taylor series. We reduce to the range $[0, 2\pi]$ by subtracting multiples of 2π , then to the smaller range $[0, \pi/2]$ by subtracting multiples of $\pi/2$ (keeping track of how many times $\pi/2$ is subtracted), then to $[0, \pi/4]$ by mapping $x \rightarrow \pi/2 - x$ if appropriate. When the argument is very large, say, 10^{100} , an equally large multiple of 2π must be subtracted, hence we must work with a very good approximation of 2π in order to get a sensible remainder modulo 2π .

Specifically, we multiply the argument by an approximation of $1/(2\pi)$ with 10048 digits, then discard the integer part of the result, keeping 52 digits of the fractional part. From the fractional part of $x/(2\pi)$ we deduce the octant (quotient of the first three digits by 125). We then multiply by 8 or -8 (the latter when the octant is odd), ignore any integer part (related to the octant), and convert the fractional part to an extended precision number, before multiplying by $\pi/4$ to convert back to a value in radians in $[0, \pi/4]$.

It is possible to prove that given the precision of floating points and their range of exponents, the 52 digits may start at most with 24 zeros. The 5 last digits are affected by carries from computations which are not done, hence we are left with at least $52 - 24 - 5 = 23$ significant digits, enough to round correctly up to $0.6 \cdot \text{ulp}$ in all cases.

`\c_fp_trig_intarray` This integer array stores blocks of 8 decimals of $10^{-16}/(2\pi)$. Each entry is 10^8 plus an 8 digit number storing 8 decimals. In total we store 10112 decimals of $10^{-16}/(2\pi)$. The number of decimals we really need is the maximum exponent plus the number of digits we later need, 52, plus 12 ($4 - 1$ groups of 4 digits). The memory footprint (1/2 byte per digit) is the same as an earlier method of storing the data as a control sequence name, but the major advantage is that we can unpack specific subsets of the digits without unpacking the 10112 decimals.

```
20801 \intarray_const_from_clist:Nn \c\_fp_trig_intarray
20802 {
20803     100000000, 100000000, 115915494, 130918953, 135768883, 176337251,
20804     143620344, 159645740, 145644874, 176673440, 158896797, 163422653,
20805     150901138, 102766253, 108595607, 128427267, 157958036, 189291184,
20806     161145786, 152877967, 141073169, 198392292, 139966937, 140907757,
20807     130777463, 196925307, 168871739, 128962173, 197661693, 136239024,
20808     117236290, 111832380, 111422269, 197557159, 140461890, 108690267,
20809     139561204, 189410936, 193784408, 155287230, 199946443, 140024867,
20810     123477394, 159610898, 132309678, 130749061, 166986462, 180469944,
20811     186521878, 181574786, 156696424, 110389958, 174139348, 160998386,
20812     180991999, 162442875, 158517117, 188584311, 117518767, 116054654,
20813     175369880, 109739460, 136475933, 137680593, 102494496, 163530532,
20814     171567755, 103220324, 177781639, 171660229, 146748119, 159816584,
20815     106060168, 103035998, 113391198, 174988327, 186654435, 127975507,
20816     100162406, 177564388, 184957131, 108801221, 199376147, 168137776,
20817     147378906, 133068046, 145797848, 117613124, 127314069, 196077502,
20818     145002977, 159857089, 105690279, 167851315, 125210016, 131774602,
20819     109248116, 106240561, 145620314, 164840892, 148459191, 143521157,
20820     154075562, 100871526, 160680221, 171591407, 157474582, 172259774,
20821     162853998, 175155329, 139081398, 117724093, 158254797, 107332871,
20822     190406999, 175907657, 170784934, 170393589, 182808717, 134256403,
```

20823	166895116,	162545705,	194332763,	112686500,	126122717,	197115321,
20824	112599504,	138667945,	103762556,	108363171,	116952597,	158128224,
20825	194162333,	143145106,	112353687,	185631136,	136692167,	114206974,
20826	169601292,	150578336,	105311960,	185945098,	139556718,	170995474,
20827	165104316,	123815517,	158083944,	129799709,	199505254,	138756612,
20828	194458833,	106846050,	178529151,	151410404,	189298850,	163881607,
20829	176196993,	107341038,	199957869,	118905980,	193737772,	106187543,
20830	122271893,	101366255,	126123878,	103875388,	181106814,	106765434,
20831	108282785,	126933426,	179955607,	107903860,	160352738,	199624512,
20832	159957492,	176297023,	159409558,	143011648,	129641185,	157771240,
20833	157544494,	157021789,	176979240,	194903272,	194770216,	164960356,
20834	153181535,	144003840,	168987471,	176915887,	163190966,	150696440,
20835	147769706,	187683656,	177810477,	197954503,	153395758,	130188183,
20836	186879377,	166124814,	195305996,	155802190,	183598751,	103512712,
20837	190432315,	180498719,	168687775,	194656634,	162210342,	104440855,
20838	149785037,	192738694,	129353661,	193778292,	187359378,	143470323,
20839	102371458,	137923557,	111863634,	119294601,	183182291,	196416500,
20840	187830793,	131353497,	179099745,	186492902,	167450609,	189368909,
20841	145883050,	133703053,	180547312,	132158094,	131976760,	132283131,
20842	141898097,	149822438,	133517435,	169898475,	101039500,	168388003,
20843	197867235,	199608024,	100273901,	108749548,	154787923,	156826113,
20844	199489032,	168997427,	108349611,	149208289,	103776784,	174303550,
20845	145684560,	183671479,	130845672,	133270354,	185392556,	120208683,
20846	193240995,	162211753,	131839402,	109707935,	170774965,	149880868,
20847	160663609,	168661967,	103747454,	121028312,	119251846,	122483499,
20848	111611495,	166556037,	196967613,	199312829,	196077608,	127799010,
20849	107830360,	102338272,	198790854,	102387615,	157445430,	192601191,
20850	100543379,	198389046,	154921248,	129516070,	172853005,	122721023,
20851	160175233,	113173179,	175931105,	103281551,	109373913,	163964530,
20852	157926071,	180083617,	195487672,	146459804,	173977292,	144810920,
20853	109371257,	186918332,	189588628,	139904358,	168666639,	175673445,
20854	114095036,	137327191,	174311388,	106638307,	125923027,	159734506,
20855	105482127,	178037065,	133778303,	121709877,	134966568,	149080032,
20856	169885067,	141791464,	168350828,	116168533,	114336160,	173099514,
20857	198531198,	119733758,	144420984,	116559541,	152250643,	139431286,
20858	144403838,	183561508,	179771645,	101706470,	167518774,	156059160,
20859	187168578,	157939226,	123475633,	117111329,	198655941,	159689071,
20860	198506887,	144230057,	151919770,	156900382,	118392562,	120338742,
20861	135362568,	108354156,	151729710,	188117217,	195936832,	156488518,
20862	174997487,	108553116,	159830610,	113921445,	144601614,	188452770,
20863	125114110,	170248521,	173974510,	138667364,	103872860,	109967489,
20864	131735618,	112071174,	104788993,	168886556,	192307848,	150230570,
20865	157144063,	163863202,	136852010,	174100574,	185922811,	115721968,
20866	100397824,	175953001,	166958522,	112303464,	118773650,	143546764,
20867	164565659,	171901123,	108476709,	193097085,	191283646,	166919177,
20868	169387914,	133315566,	150669813,	121641521,	100895711,	172862384,
20869	126070678,	145176011,	113450800,	169947684,	122356989,	162488051,
20870	157759809,	153397080,	185475059,	175362656,	149034394,	145420581,
20871	178864356,	183042000,	131509559,	147434392,	152544850,	167491429,
20872	108647514,	142303321,	133245695,	111634945,	167753939,	142403609,
20873	105438335,	152829243,	142203494,	184366151,	146632286,	102477666,
20874	166049531,	140657343,	157553014,	109082798,	180914786,	169343492,
20875	127376026,	134997829,	195701816,	119643212,	133140475,	176289748,
20876	140828911,	174097478,	126378991,	181699939,	148749771,	151989818,

20877	172666294,	160183053,	195832752,	109236350,	168538892,	128468247,
20878	125997252,	183007668,	156937583,	165972291,	198244297,	147406163,
20879	181831139,	158306744,	134851692,	185973832,	137392662,	140243450,
20880	119978099,	140402189,	161348342,	173613676,	144991382,	171541660,
20881	163424829,	136374185,	106122610,	186132119,	198633462,	184709941,
20882	183994274,	129559156,	128333990,	148038211,	175011612,	111667205,
20883	119125793,	103552929,	124113440,	131161341,	112495318,	138592695,
20884	184904438,	146807849,	109739828,	108855297,	104515305,	139914009,
20885	188698840,	188365483,	166522246,	168624087,	125401404,	100911787,
20886	142122045,	123075334,	173972538,	114940388,	141905868,	142311594,
20887	163227443,	139066125,	116239310,	162831953,	123883392,	113153455,
20888	163815117,	152035108,	174595582,	101123754,	135976815,	153401874,
20889	107394340,	136339780,	138817210,	104531691,	182951948,	179591767,
20890	139541778,	179243527,	161740724,	160593916,	102732282,	187946819,
20891	136491289,	149714953,	143255272,	135916592,	198072479,	198580612,
20892	169007332,	118844526,	179433504,	155801952,	149256630,	162048766,
20893	116134365,	133992028,	175452085,	155344144,	109905129,	182727454,
20894	165911813,	122232840,	151166615,	165070983,	175574337,	129548631,
20895	120411217,	116380915,	160616116,	157320000,	183306114,	160618128,
20896	103262586,	195951602,	146321661,	138576614,	180471993,	127077713,
20897	116441201,	159496011,	106328305,	120759583,	148503050,	179095584,
20898	198298218,	167402898,	138551383,	123957020,	180763975,	150429225,
20899	198476470,	171016426,	197438450,	143091658,	164528360,	132493360,
20900	143546572,	137557916,	113663241,	120457809,	196971566,	134022158,
20901	180545794,	131328278,	100552461,	132088901,	187421210,	192448910,
20902	141005215,	149680971,	113720754,	100571096,	134066431,	135745439,
20903	191597694,	135788920,	179342561,	177830222,	137011486,	142492523,
20904	192487287,	113132021,	176673607,	156645598,	127260957,	141566023,
20905	143787436,	129132109,	174858971,	150713073,	191040726,	143541417,
20906	197057222,	165479803,	181512759,	157912400,	125344680,	148220261,
20907	173422990,	101020483,	106246303,	137964746,	178190501,	181183037,
20908	151538028,	179523433,	141955021,	135689770,	191290561,	143178787,
20909	192086205,	174499925,	178975690,	118492103,	124206471,	138519113,
20910	188147564,	102097605,	154895793,	178514140,	141453051,	151583964,
20911	128232654,	106020603,	131189158,	165702720,	186250269,	191639375,
20912	115278873,	160608114,	155694842,	110322407,	177272742,	116513642,
20913	134366992,	171634030,	194053074,	180652685,	109301658,	192136921,
20914	141431293,	171341061,	157153714,	106203978,	147618426,	150297807,
20915	186062669,	169960809,	118422347,	163350477,	146719017,	145045144,
20916	161663828,	146208240,	186735951,	102371302,	190444377,	194085350,
20917	134454426,	133413062,	163074595,	113830310,	122931469,	134466832,
20918	185176632,	182415152,	110179422,	164439571,	181217170,	121756492,
20919	119644493,	196532222,	118765848,	182445119,	109401340,	150443213,
20920	198586286,	121083179,	139396084,	143898019,	114787389,	177233102,
20921	186310131,	148695521,	126205182,	178063494,	157118662,	177825659,
20922	188310053,	151552316,	165984394,	109022180,	163144545,	121212978,
20923	197344714,	188741258,	126822386,	102360271,	109981191,	152056882,
20924	134723983,	158013366,	106837863,	128867928,	161973236,	172536066,
20925	185216856,	132011948,	197807339,	158419190,	166595838,	167852941,
20926	124187182,	117279875,	106103946,	106481958,	157456200,	160892122,
20927	184163943,	173846549,	158993202,	184812364,	133466119,	170732430,
20928	195458590,	173361878,	162906318,	150165106,	126757685,	112163575,
20929	188696307,	145199922,	100107766,	176830946,	198149756,	122682434,
20930	179367131,	108412102,	119520899,	148191244,	140487511,	171059184,

20931	141399078,	189455775,	118462161,	190415309,	134543802,	180893862,
20932	180732375,	178615267,	179711433,	123241969,	185780563,	176301808,
20933	184386640,	160717536,	183213626,	129671224,	126094285,	140110963,
20934	121826276,	151201170,	122552929,	128965559,	146082049,	138409069,
20935	107606920,	103954646,	119164002,	115673360,	117909631,	187289199,
20936	186343410,	186903200,	157966371,	103128612,	135698881,	176403642,
20937	152540837,	109810814,	183519031,	121318624,	172281810,	150845123,
20938	169019064,	166322359,	138872454,	163073727,	128087898,	130041018,
20939	194859136,	173742589,	141812405,	167291912,	138003306,	134499821,
20940	196315803,	186381054,	124578934,	150084553,	128031351,	118843410,
20941	107373060,	159565443,	173624887,	171292628,	198074235,	139074061,
20942	178690578,	144431052,	174262641,	176783005,	182214864,	162289361,
20943	192966929,	192033046,	169332843,	181580535,	164864073,	118444059,
20944	195496893,	153773183,	167266131,	130108623,	158802128,	180432893,
20945	144562140,	147978945,	142337360,	158506327,	104399819,	132635916,
20946	168734194,	136567839,	101281912,	120281622,	195003330,	112236091,
20947	185875592,	101959081,	122415367,	194990954,	148881099,	175891989,
20948	108115811,	163538891,	163394029,	123722049,	184837522,	142362091,
20949	100834097,	156679171,	100841679,	157022331,	178971071,	102928884,
20950	189701309,	195339954,	124415335,	106062584,	139214524,	133864640,
20951	134324406,	157317477,	155340540,	144810061,	177612569,	108474646,
20952	114329765,	143900008,	138265211,	145210162,	136643111,	197987319,
20953	102751191,	144121361,	169620456,	193602633,	161023559,	162140467,
20954	102901215,	167964187,	135746835,	187317233,	110047459,	163339773,
20955	124770449,	118885134,	141536376,	100915375,	164267438,	145016622,
20956	113937193,	106748706,	128815954,	164819775,	119220771,	102367432,
20957	189062690,	170911791,	194127762,	112245117,	123546771,	115640433,
20958	135772061,	166615646,	174474627,	130562291,	133320309,	153340551,
20959	138417181,	194605321,	150142632,	180008795,	151813296,	175497284,
20960	167018836,	157425342,	150169942,	131069156,	134310662,	160434122,
20961	105213831,	158797111,	150754540,	163290657,	102484886,	148697402,
20962	187203725,	198692811,	149360627,	140384233,	128749423,	132178578,
20963	177507355,	171857043,	178737969,	134023369,	102911446,	196144864,
20964	197697194,	134527467,	144296030,	189437192,	154052665,	188907106,
20965	162062575,	150993037,	199766583,	167936112,	181374511,	104971506,
20966	115378374,	135795558,	167972129,	135876446,	130937572,	103221320,
20967	124605656,	161129971,	131027586,	191128460,	143251843,	143269155,
20968	129284585,	173495971,	150425653,	199302112,	118494723,	121323805,
20969	116549802,	190991967,	168151180,	122483192,	151273721,	199792134,
20970	133106764,	121874844,	126215985,	112167639,	167793529,	182985195,
20971	185453921,	106957880,	158685312,	132775454,	133229161,	198905318,
20972	190537253,	191582222,	192325972,	178133427,	181825606,	148823337,
20973	160719681,	101448145,	131983362,	137910767,	112550175,	128826351,
20974	183649210,	135725874,	110356573,	189469487,	154446940,	118175923,
20975	106093708,	128146501,	185742532,	149692127,	164624247,	183221076,
20976	154737505,	168198834,	156410354,	158027261,	125228550,	131543250,
20977	139591848,	191898263,	104987591,	115406321,	103542638,	190012837,
20978	142615518,	178773183,	175862355,	117537850,	169565995,	170028011,
20979	158412588,	170150030,	117025916,	174630208,	142412449,	112839238,
20980	105257725,	114737141,	123102301,	172563968,	130555358,	132628403,
20981	183638157,	168682846,	143304568,	105994018,	170010719,	152092970,
20982	117799058,	132164175,	179868116,	158654714,	177489647,	116547948,
20983	183121404,	131836079,	184431405,	157311793,	149677763,	173989893,
20984	102277656,	107058530,	140837477,	152640947,	143507039,	152145247,

```

20985 101683884, 107090870, 161471944, 137225650, 128231458, 172995869,
20986 173831689, 171268519, 139042297, 111072135, 107569780, 137262545,
20987 181410950, 138270388, 198736451, 162848201, 180468288, 120582913,
20988 153390138, 135649144, 130040157, 106509887, 192671541, 174507066,
20989 186888783, 143805558, 135011967, 145862340, 180595327, 124727843,
20990 182925939, 157715840, 136885940, 198993925, 152416883, 178793572,
20991 179679516, 154076673, 192703125, 164187609, 162190243, 104699348,
20992 159891990, 160012977, 174692145, 132970421, 167781726, 115178506,
20993 153008552, 155999794, 102099694, 155431545, 127458567, 104403686,
20994 168042864, 184045128, 181182309, 179349696, 127218364, 192935516,
20995 120298724, 169583299, 148193297, 183358034, 159023227, 105261254,
20996 121144370, 184359584, 194433836, 138388317, 175184116, 108817112,
20997 151279233, 137457721, 193398208, 119005406, 132929377, 175306906,
20998 160741530, 149976826, 147124407, 176881724, 186734216, 185881509,
20999 191334220, 175930947, 117385515, 193408089, 157124410, 163472089,
21000 131949128, 180783576, 131158294, 100549708, 191802336, 165960770,
21001 170927599, 101052702, 181508688, 197828549, 143403726, 142729262,
21002 110348701, 139928688, 153550062, 106151434, 130786653, 196085995,
21003 100587149, 139141652, 106530207, 100852656, 124074703, 166073660,
21004 153338052, 163766757, 120188394, 197277047, 122215363, 138511354,
21005 183463624, 161985542, 159938719, 133367482, 104220974, 149956672,
21006 170250544, 164232439, 157506869, 159133019, 137469191, 142980999,
21007 134242305, 150172665, 121209241, 145596259, 160554427, 159095199,
21008 168243130, 184279693, 171132070, 121049823, 123819574, 171759855,
21009 119501864, 163094029, 175943631, 194450091, 191506160, 149228764,
21010 132319212, 197034460, 193584259, 126727638, 168143633, 109856853,
21011 127860243, 132141052, 133076065, 188414958, 158718197, 107124299,
21012 159592267, 181172796, 144388537, 196763139, 127431422, 179531145,
21013 100064922, 112650013, 132686230, 121550837,
21014 }

```

(End definition for \c__fp_trig_intarray.)

```

\__fp_trig_large:ww The exponent #1 is between 1 and 10000. We wish to look up decimals  $10^{\#1-16}/(2\pi)$ 
\__fp_trig_large_auxi:w starting from the digit #1 + 1. Since they are stored in batches of 8, compute  $\lfloor \#1/8 \rfloor$ 
\__fp_trig_large_auxii:w and fetch blocks of 8 digits starting there. The numbering of items in \c__fp_trig_
\__fp_trig_large_auxiii:w intarray starts at 1, so the block  $\lfloor \#1/8 \rfloor + 1$  contains the digit we want, at one of the
eight positions. Each call to \int_value:w \__kernel_intarray_item:Nn expands the
next, until being stopped by \__fp_trig_large_auxiii:w using \exp_stop_f:. Once
all these blocks are unpacked, the \exp_stop_f: and 0 to 7 digits are removed by \use_
none:n...n. Finally, \__fp_trig_large_auxii:w packs 64 digits (there are between 65
and 72 at this point) into groups of 4 and the auxv auxiliary is called.

```

```

21015 \cs_new:Npn \__fp_trig_large:ww #1, #2#3#4#5#6;
21016 {
21017   \exp_after:wN \__fp_trig_large_auxi:w
21018   \int_value:w \__fp_int_eval:w (#1 - 4) / 8 \exp_after:wN ,
21019   \int_value:w #1 , ;
21020   {#2}{#3}{#4}{#5} ;
21021 }
21022 \cs_new:Npn \__fp_trig_large_auxi:w #1, #2,
21023 {
21024   \exp_after:wN \exp_after:wN
21025   \exp_after:wN \__fp_trig_large_auxii:w
21026   \cs:w

```

```

21027     use_none:n \prg_replicate:nn { #2 - #1 * 8 } { n }
21028     \exp_after:wN
21029   \cs_end:
21030   \int_value:w
21031   \__kernel_intarray_item:Nn \c__fp_trig_intarray
21032     { \__fp_int_eval:w #1 + 1 \scan_stop: }
21033   \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
21034   \__kernel_intarray_item:Nn \c__fp_trig_intarray
21035     { \__fp_int_eval:w #1 + 2 \scan_stop: }
21036   \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
21037   \__kernel_intarray_item:Nn \c__fp_trig_intarray
21038     { \__fp_int_eval:w #1 + 3 \scan_stop: }
21039   \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
21040   \__kernel_intarray_item:Nn \c__fp_trig_intarray
21041     { \__fp_int_eval:w #1 + 4 \scan_stop: }
21042   \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
21043   \__kernel_intarray_item:Nn \c__fp_trig_intarray
21044     { \__fp_int_eval:w #1 + 5 \scan_stop: }
21045   \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
21046   \__kernel_intarray_item:Nn \c__fp_trig_intarray
21047     { \__fp_int_eval:w #1 + 6 \scan_stop: }
21048   \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
21049   \__kernel_intarray_item:Nn \c__fp_trig_intarray
21050     { \__fp_int_eval:w #1 + 7 \scan_stop: }
21051   \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
21052   \__kernel_intarray_item:Nn \c__fp_trig_intarray
21053     { \__fp_int_eval:w #1 + 8 \scan_stop: }
21054   \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
21055   \__kernel_intarray_item:Nn \c__fp_trig_intarray
21056     { \__fp_int_eval:w #1 + 9 \scan_stop: }
21057   \exp_stop_f:
21058 }
21059 \cs_new:Npn \__fp_trig_large_auxii:w
21060 {
21061   \__fp_pack_twice_four:wNNNNNNNN \__fp_pack_twice_four:wNNNNNNNN
21062   \__fp_pack_twice_four:wNNNNNNNN \__fp_pack_twice_four:wNNNNNNNN
21063   \__fp_pack_twice_four:wNNNNNNNN \__fp_pack_twice_four:wNNNNNNNN
21064   \__fp_pack_twice_four:wNNNNNNNN \__fp_pack_twice_four:wNNNNNNNN
21065   \__fp_trig_large_auxv:www ;
21066 }
21067 \cs_new:Npn \__fp_trig_large_auxiii:w 1 { \exp_stop_f: }

```

(End definition for __fp_trig_large:ww and others.)

__fp_trig_large_auxv:www
 __fp_trig_large_auxvi:wNNNNNNNN
 __fp_trig_large_pack:NNNNNw

First come the first 64 digits of the fractional part of $10^{#1-16}/(2\pi)$, arranged in 16 blocks of 4, and ending with a semicolon. Then a few more digits of the same fractional part, ending with a semicolon, then 4 blocks of 4 digits holding the significand of the original argument. Multiply the 16-digit significand with the 64-digit fractional part: the `auxvi` auxiliary receives the significand as `#2#3#4#5` and 16 digits of the fractional part as `#6#7#8#9`, and computes one step of the usual ladder of pack functions we use for multiplication (see *e.g.*, `__fp_fixed_mul:wwn`), then discards one block of the fractional part to set things up for the next step of the ladder. We perform 13 such steps, replacing the last `middle` shift by the appropriate `trailing` shift, then discard the significand and remaining 3 blocks from the fractional part, as there are not enough digits to compute

any more step in the ladder. The last semicolon closes the ladder, and we return control to the `auxvii` auxiliary.

```

21068 \cs_new:Npn \__fp_trig_large_auxv:www #1; #2; #3;
21069 {
21070   \exp_after:wN \__fp_use_i_until_s:nw
21071   \exp_after:wN \__fp_trig_large_auxvii:w
21072   \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
21073   \prg_replicate:nn { 13 }
21074   { \__fp_trig_large_auxvi:wnnnnnnnn }
21075   + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
21076   \__fp_use_i_until_s:nw
21077   ; #3 #1 ; ;
21078 }
21079 \cs_new:Npn \__fp_trig_large_auxvi:wnnnnnnnn #1; #2#3#4#5#6#7#8#9
21080 {
21081   \exp_after:wN \__fp_trig_large_pack:NNNNNw
21082   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
21083   + #2*#9 + #3*#8 + #4*#7 + #5*#6
21084   #1; {#2}{#3}{#4}{#5} {#7}{#8}{#9}
21085 }
21086 \cs_new:Npn \__fp_trig_large_pack:NNNNNw #1#2#3#4#5#6;
21087 { + #1#2#3#4#5 ; #6 }

```

(End definition for `__fp_trig_large_auxv:www`, `__fp_trig_large_auxvi:wnnnnnnnn`, and `__fp_trig_large_pack:NNNNNw`.)

The `auxvii` auxiliary is followed by 52 digits and a semicolon. We find the octant as the integer part of 8 times what follows, or equivalently as the integer part of $\#1\#2\#3/125$, and add it to the surrounding integer expression for the octant. We then compute 8 times the 52-digit number, with a minus sign if the octant is odd. Again, the last `middle` shift is converted to a `trailing` shift. Any integer part (including negative values which come up when the octant is odd) is discarded by `__fp_use_i_until_s:nw`. The resulting fractional part should then be converted to radians by multiplying by $2\pi/8$, but first, build an extended precision number by abusing `__fp_ep_to_ep_loop:N` with the appropriate trailing markers. Finally, `__fp_trig_small:ww` sets up the argument for the functions which compute the Taylor series.

```

21088 \cs_new:Npn \__fp_trig_large_auxvii:w #1#2#3
21089 {
21090   \exp_after:wN \__fp_trig_large_auxviii:ww
21091   \int_value:w \__fp_int_eval:w (#1#2#3 - 62) / 125 ;
21092   #1#2#3
21093 }
21094 \cs_new:Npn \__fp_trig_large_auxviii:ww #1;
21095 {
21096   + #1
21097   \if_int_odd:w #1 \exp_stop_f:
21098     \exp_after:wN \__fp_trig_large_auxix:Nw
21099     \exp_after:wN -
21100   \else:
21101     \exp_after:wN \__fp_trig_large_auxix:Nw
21102     \exp_after:wN +
21103   \fi:
21104 }

```

```

21105 \cs_new:Npn \__fp_trig_large_auxix:Nw
21106 {
21107   \exp_after:wN \__fp_use_i_until_s:nw
21108   \exp_after:wN \__fp_trig_large_auxxi:w
21109   \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
21110   \prg_replicate:nn { 13 }
21111   { \__fp_trig_large_auxx:wNNNNN }
21112   + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
21113   ;
21114 }
21115 \cs_new:Npn \__fp_trig_large_auxx:wNNNNN #1; #2 #3#4#5#6
21116 {
21117   \exp_after:wN \__fp_trig_large_pack:NNNNNw
21118   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
21119   #2 8 * #3#4#5#6
21120   #1; #2
21121 }
21122 \cs_new:Npn \__fp_trig_large_auxxi:w #1;
21123 {
21124   \exp_after:wN \__fp_ep_mul_raw:wwwN
21125   \int_value:w \__fp_int_eval:w 0 \__fp_ep_to_ep_loop:N #1 ; ; !
21126   0,{7853}{9816}{3397}{4483}{0961}{5661};
21127   \__fp_trig_small:ww
21128 }

```

(End definition for __fp_trig_large_auxvii:w and others.)

34.1.6 Computing the power series

__fp_sin_series_o:NNwww Here we receive a conversion function __fp_ep_to_float_o:wwN or __fp_ep_inv_to_float_o:wwN, a *sign* (0 or 2), a (non-negative) *octant* delimited by a dot, a *fixed point* number delimited by a semicolon, and an extended-precision number. The auxiliary receives:

- the conversion function #1;
- the final sign, which depends on the octant #3 and the sign #2;
- the octant #3, which controls the series we use;
- the square #4 * #4 of the argument as a fixed point number, computed with __fp_fixed_mul:wwn;
- the number itself as an extended-precision number.

If the octant is in {1, 2, 5, 6, ...}, we are near an extremum of the function and we use the series

$$\cos(x) = 1 - x^2 \left(\frac{1}{2!} - x^2 \left(\frac{1}{4!} - x^2 \left(\dots \right) \right) \right).$$

Otherwise, the series

$$\sin(x) = x \left(1 - x^2 \left(\frac{1}{3!} - x^2 \left(\frac{1}{5!} - x^2 \left(\dots \right) \right) \right) \right)$$

is used. Finally, the extended-precision number is converted to a floating point number with the given sign, and `__fp_sanitizew` checks for overflow and underflow.

```

21129 \cs_new:Npn \__fp_sin_series_o:NNwww #1#2#3. #4;
21130 {
21131   \__fp_fixed_mul:wwn #4; #4;
21132   {
21133     \exp_after:wN \__fp_sin_series_aux_o:NNwww
21134     \exp_after:wN #1
21135     \int_value:w
21136     \if_int_odd:w \__fp_int_eval:w (#3 + 2) / 4 \__fp_int_eval_end:
21137       #2
21138     \else:
21139       \if_meaning:w #2 0 2 \else: 0 \fi:
21140     \fi:
21141     {#3}
21142   }
21143 }
21144 \cs_new:Npn \__fp_sin_series_aux_o:NNwww #1#2#3 #4; #5,#6;
21145 {
21146   \if_int_odd:w \__fp_int_eval:w #3 / 2 \__fp_int_eval_end:
21147     \exp_after:wN \use_i:nn
21148   \else:
21149     \exp_after:wN \use_ii:nn
21150   \fi:
21151   { % 1/18!
21152     \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0001}{5619}{2070};
21153     #4;{0000}{0000}{0000}{0477}{9477}{3324};
21154     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0011}{4707}{4559}{7730};
21155     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{2087}{6756}{9878}{6810};
21156     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0027}{5573}{1922}{3985}{8907};
21157     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{2480}{1587}{3015}{8730}{1587};
21158     \__fp_fixed_mul_sub_back:wwwn #4;{0013}{8888}{8888}{8888}{8888}{8889};
21159     \__fp_fixed_mul_sub_back:wwwn #4;{0416}{6666}{6666}{6666}{6666}{6667};
21160     \__fp_fixed_mul_sub_back:wwwn #4;{5000}{0000}{0000}{0000}{0000}{0000};
21161     \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
21162     { \__fp_fixed_continue:wn 0, }
21163   }
21164   { % 1/17!
21165     \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0028}{1145}{7254};
21166     #4;{0000}{0000}{0000}{7647}{1637}{3182};
21167     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0160}{5904}{3836}{8216};
21168     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0002}{5052}{1083}{8544}{1719};
21169     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0275}{5731}{9223}{9858}{9065};
21170     \__fp_fixed_mul_sub_back:wwwn #4;{0001}{9841}{2698}{4126}{9841}{2698};
21171     \__fp_fixed_mul_sub_back:wwwn #4;{0083}{3333}{3333}{3333}{3333}{3333};
21172     \__fp_fixed_mul_sub_back:wwwn #4;{1666}{6666}{6666}{6666}{6666}{6667};
21173     \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
21174     { \__fp_ep_mul:wwwn 0, } #5,#6;
21175   }
21176   {
21177     \exp_after:wN \__fp_sanitizew
21178     \exp_after:wN #2
21179     \int_value:w \__fp_int_eval:w #1
21180   }

```

```

21181     #2
21182 }

```

(End definition for `_fp_sin_series_o:NNwww` and `_fp_sin_series_aux_o:NNwww`.)

```

\_fp_tan_series_o:NNwww
\_fp_tan_series_aux_o:Nnwww

```

Contrarily to `_fp_sin_series_o:NNwww` which received a conversion auxiliary as **#1**, here, **#1** is 0 for tangent and 2 for cotangent. Consider first the case of the tangent. The octant **#3** starts at 1, which means that it is 1 or 2 for $|x| \in [0, \pi/2]$, it is 3 or 4 for $|x| \in [\pi/2, \pi]$, and so on: the intervals on which $\tan|x| \geq 0$ coincide with those for which $\lfloor (\#3 + 1)/2 \rfloor$ is odd. We also have to take into account the original sign of x to get the sign of the final result; it is straightforward to check that the first `\int_value:w` expansion produces 0 for a positive final result, and 2 otherwise. A similar story holds for $\cot(x)$.

The auxiliary receives the sign, the octant, the square of the (reduced) input, and the (reduced) input (an extended-precision number) as arguments. It then computes the numerator and denominator of

$$\tan(x) \simeq \frac{x(1 - x^2(a_1 - x^2(a_2 - x^2(a_3 - x^2(a_4 - x^2a_5))))))}{1 - x^2(b_1 - x^2(b_2 - x^2(b_3 - x^2(b_4 - x^2b_5)))}.$$

The ratio is computed by `_fp_ep_div:wwwn`, then converted to a floating point number. For octants **#3** (really, quadrants) next to a pole of the functions, the fixed point numerator and denominator are exchanged before computing the ratio. Note that this `\if_int_odd:w` test relies on the fact that the octant is at least 1.

```

21183 \cs_new:Npn \_fp_tan_series_o:NNwww #1#2#3. #4;
21184 {
21185   \_fp_fixed_mul:wwn #4; #4;
21186   {
21187     \exp_after:wN \_fp_tan_series_aux_o:Nnwww
21188     \int_value:w
21189     \if_int_odd:w \_fp_int_eval:w #3 / 2 \_fp_int_eval_end:
21190     \exp_after:wN \reverse_if:N
21191     \fi:
21192     \if_meaning:w #1#2 2 \else: 0 \fi:
21193     {#3}
21194   }
21195 }
21196 \cs_new:Npn \_fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5;
21197 {
21198   \_fp_fixed_mul_sub_back:wwwn {0000}{0000}{1527}{3493}{0856}{7059};
21199   #3; {0000}{0159}{6080}{0274}{5257}{6472};
21200   \_fp_fixed_mul_sub_back:wwwn #3; {0002}{4571}{2320}{0157}{2558}{8481};
21201   \_fp_fixed_mul_sub_back:wwwn #3; {0115}{5830}{7533}{5397}{3168}{2147};
21202   \_fp_fixed_mul_sub_back:wwwn #3; {1929}{8245}{6140}{3508}{7719}{2982};
21203   \_fp_fixed_mul_sub_back:wwwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
21204   { \_fp_ep_mul:wwwn 0, } #4,#5;
21205   {
21206     \_fp_fixed_mul_sub_back:wwwn {0000}{0007}{0258}{0681}{9408}{4706};
21207     #3; {0000}{2343}{7175}{1399}{6151}{7670};
21208     \_fp_fixed_mul_sub_back:wwwn #3; {0019}{2638}{4588}{9232}{8861}{3691};
21209     \_fp_fixed_mul_sub_back:wwwn #3; {0536}{6357}{0691}{4344}{6852}{4252};
21210     \_fp_fixed_mul_sub_back:wwwn #3; {5263}{1578}{9473}{6842}{1052}{6315};
21211     \_fp_fixed_mul_sub_back:wwwn #3; {10000}{0000}{0000}{0000}{0000}{0000};

```

```

21212     {
21213         \reverse_if:N \if_int_odd:w
21214         \__fp_int_eval:w (#2 - 1) / 2 \__fp_int_eval_end:
21215         \exp_after:wN \__fp_reverse_args:Nww
21216         \fi:
21217         \__fp_ep_div:wwwn 0,
21218     }
21219 }
21220 {
21221     \exp_after:wN \__fp_sanitize:Nw
21222     \exp_after:wN #1
21223     \int_value:w \__fp_int_eval:w \__fp_ep_to_float_o:wwN
21224 }
21225 #1
21226 }

```

(End definition for `__fp_tan_series_o:NNwww` and `__fp_tan_series_aux_o:Nnwww`.)

34.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arcsecant, and arcsecant) are based on a function often denoted `atan2`. This function is accessed directly by feeding two arguments to arctangent, and is defined by $\text{atan}(y, x) = \text{atan}(y/x)$ for generic y and x . Its advantages over the conventional arctangent is that it takes values in $[-\pi, \pi]$ rather than $[-\pi/2, \pi/2]$, and that it is better behaved in boundary cases. Other inverse trigonometric functions are expressed in terms of `atan` as

$$\arccos x = \text{atan}(\sqrt{1 - x^2}, x) \quad (5)$$

$$\arcsin x = \text{atan}(x, \sqrt{1 - x^2}) \quad (6)$$

$$\text{asec } x = \text{atan}(\sqrt{x^2 - 1}, 1) \quad (7)$$

$$\text{acsc } x = \text{atan}(1, \sqrt{x^2 - 1}) \quad (8)$$

$$\text{atan } x = \text{atan}(x, 1) \quad (9)$$

$$\text{acot } x = \text{atan}(1, x). \quad (10)$$

Rather than introducing a new function, `atan2`, the arctangent function `atan` is overloaded: it can take one or two arguments. In the comments below, following many texts, we call the first argument y and the second x , because $\text{atan}(y, x) = \text{atan}(y/x)$ is the angular coordinate of the point (x, y) .

As for direct trigonometric functions, the first step in computing $\text{atan}(y, x)$ is argument reduction. The sign of y gives that of the result. We distinguish eight regions where the point $(x, |y|)$ can lie, of angular size roughly $\pi/8$, characterized by their “octant”, between 0 and 7 included. In each region, we compute an arctangent as a Taylor series, then shift this arctangent by the appropriate multiple of $\pi/4$ and sign to get the result. Here is a list of octants, and how we compute the arctangent (we assume $y > 0$; otherwise replace y by $-y$ below):

0 $0 < |y| < 0.41421x$, then $\text{atan } \frac{|y|}{x}$ is given by a nicely convergent Taylor series;

1 $0 < 0.41421x < |y| < x$, then $\text{atan } \frac{|y|}{x} = \frac{\pi}{4} - \text{atan } \frac{x - |y|}{x + |y|}$;

- 2 $0 < 0.41421|y| < x < |y|$, then $\operatorname{atan} \frac{|y|}{x} = \frac{\pi}{4} + \operatorname{atan} \frac{-x+|y|}{x+|y|}$;
- 3 $0 < x < 0.41421|y|$, then $\operatorname{atan} \frac{|y|}{x} = \frac{\pi}{2} - \operatorname{atan} \frac{x}{|y|}$;
- 4 $0 < -x < 0.41421|y|$, then $\operatorname{atan} \frac{|y|}{x} = \frac{\pi}{2} + \operatorname{atan} \frac{-x}{|y|}$;
- 5 $0 < 0.41421|y| < -x < |y|$, then $\operatorname{atan} \frac{|y|}{x} = \frac{3\pi}{4} - \operatorname{atan} \frac{x+|y|}{-x+|y|}$;
- 6 $0 < -0.41421x < |y| < -x$, then $\operatorname{atan} \frac{|y|}{x} = \frac{3\pi}{4} + \operatorname{atan} \frac{-x-|y|}{-x+|y|}$;
- 7 $0 < |y| < -0.41421x$, then $\operatorname{atan} \frac{|y|}{x} = \pi - \operatorname{atan} \frac{|y|}{-x}$.

In the following, we denote by z the ratio among $|\frac{y}{x}|$, $|\frac{x}{y}|$, $|\frac{x+y}{x-y}|$, $|\frac{x-y}{x+y}|$ which appears in the right-hand side above.

34.2.1 Arctangent and arccotangent

`__fp_atan_o:Nw` The parsing step manipulates `atan` and `acot` like `min` and `max`, reading in an array of operands, but also leaves `\use_i:nn` or `\use_ii:nn` depending on whether the result should be given in radians or in degrees. The helper `__fp_parse_function_one_two:nnw` checks that the operand is one or two floating point numbers (not tuples) and leaves its second argument or its tail accordingly (its first argument is used for error messages). More precisely if we are given a single floating point number `__fp_atan_default:w` places `\c_one_fp` (expanded) after it; otherwise `__fp_atan_default:w` is omitted by `__fp_parse_function_one_two:nnw`.

```

21227 \cs_new:Npn __fp_atan_o:Nw #1
21228 {
21229   __fp_parse_function_one_two:nnw
21230   { #1 { atan } { atand } }
21231   { __fp_atan_default:w __fp_atanii_o:Nww #1 }
21232 }
21233 \cs_new:Npn __fp_acot_o:Nw #1
21234 {
21235   __fp_parse_function_one_two:nnw
21236   { #1 { acot } { acotd } }
21237   { __fp_atan_default:w __fp_acotii_o:Nww #1 }
21238 }
21239 \cs_new:Npx __fp_atan_default:w #1#2#3 @ { #1 #2 #3 \c_one_fp @ }

```

(End definition for `__fp_atan_o:Nw`, `__fp_acot_o:Nw`, and `__fp_atan_default:w`.)

`__fp_atanii_o:Nww` If either operand is `nan`, we return it. If both are normal, we call `__fp_atan_normal_o:NNnwNnw`. If both are zero or both infinity, we call `__fp_atan_inf_o:NNNw` with argument 2, leading to a result among $\{\pm\pi/4, \pm3\pi/4\}$ (in degrees, $\{\pm45, \pm135\}$). Otherwise, one is much bigger than the other, and we call `__fp_atan_inf_o:NNNw` with either an argument of 4, leading to the values $\pm\pi/2$ (in degrees, ±90), or 0, leading to $\{\pm0, \pm\pi\}$ (in degrees, $\{\pm0, \pm180\}$). Since `acot(x,y) = atan(y,x)`, `__fp_acotii_o:ww` simply reverses its two arguments.

```

21240 \cs_new:Npn __fp_atanii_o:Nww
21241   #1 \s__fp __fp_chk:w #2#3#4; \s__fp __fp_chk:w #5 #6 @
21242   {
21243     \if_meaning:w 3 #2 __fp_case_return_i_o:ww \fi:

```

```

21244 \if_meaning:w 3 #5 \__fp_case_return_ii_o:ww \fi:
21245 \if_case:w
21246   \if_meaning:w #2 #5
21247     \if_meaning:w 1 #2 10 \else: 0 \fi:
21248   \else:
21249     \if_int_compare:w #2 > #5 \exp_stop_f: 1 \else: 2 \fi:
21250   \fi:
21251   \exp_stop_f:
21252     \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 2 }
21253   \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 4 }
21254   \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 0 }
21255   \fi:
21256   \__fp_atan_normal_o:NNnwNnw #1
21257   \s__fp \__fp_chk:w #2#3#4;
21258   \s__fp \__fp_chk:w #5 #6
21259 }
21260 \cs_new:Npn \__fp_acotii_o:Nww #1#2; #3;
21261 { \__fp_atanii_o:Nww #1#3; #2; }

```

(End definition for __fp_atanii_o:Nww and __fp_acotii_o:Nww.)

__fp_atan_inf_o:NNNw This auxiliary is called whenever one number is ± 0 or $\pm \infty$ (and neither is NaN). Then the result only depends on the signs, and its value is a multiple of $\pi/4$. We use the same auxiliary as for normal numbers, __fp_atan_combine_o:NwwwwwN, with arguments the final sign #2; the octant #3; $\operatorname{atan} z/z = 1$ as a fixed point number; $z = 0$ as a fixed point number; and $z = 0$ as an extended-precision number. Given the values we provide, $\operatorname{atan} z$ is computed to be 0, and the result is $[#3/2] \cdot \pi/4$ if the sign #5 of x is positive, and $[(7 - #3)/2] \cdot \pi/4$ for negative x , where the divisions are rounded up.

```

21262 \cs_new:Npn \__fp_atan_inf_o:NNNw #1#2#3 \s__fp \__fp_chk:w #4#5#6;
21263 {
21264   \exp_after:wN \__fp_atan_combine_o:NwwwwwN
21265   \exp_after:wN #2
21266   \int_value:w \__fp_int_eval:w
21267   \if_meaning:w 2 #5 7 - \fi: #3 \exp_after:wN ;
21268   \c__fp_one_fixed_tl
21269   {0000}{0000}{0000}{0000}{0000}{0000};
21270   0,{0000}{0000}{0000}{0000}{0000}{0000}; #1
21271 }

```

(End definition for __fp_atan_inf_o:NNNw.)

__fp_atan_normal_o:NNnwNnw Here we simply reorder the floating point data into a pair of signed extended-precision numbers, that is, a sign, an exponent ending with a comma, and a six-block mantissa ending with a semi-colon. This extended precision is required by other inverse trigonometric functions, to compute things like $\operatorname{atan}(x, \sqrt{1 - x^2})$ without intermediate rounding errors.

```

21272 \cs_new_protected:Npn \__fp_atan_normal_o:NNnwNnw
21273   #1 \s__fp \__fp_chk:w 1#2#3#4; \s__fp \__fp_chk:w 1#5#6#7;
21274 {
21275   \__fp_atan_test_o:NwwNwwN
21276   #2 #3, #4{0000}{0000};
21277   #5 #6, #7{0000}{0000}; #1
21278 }

```

(End definition for _fp_atan_normal_o:NNwNnw.)

_fp_atan_test_o:NwwNwwN

This receives: the sign #1 of y , its exponent #2, its 24 digits #3 in groups of 4, and similarly for x . We prepare to call _fp_atan_combine_o:NwwwwwN which expects the sign #1, the octant, the ratio $(\operatorname{atan} z)/z = 1 - \dots$, and the value of z , both as a fixed point number and as an extended-precision floating point number with a mantissa in $[0.01, 1)$. For now, we place #1 as a first argument, and start an integer expression for the octant. The sign of x does not affect z , so we simply leave a contribution to the octant: $\langle \text{octant} \rangle \rightarrow 7 - \langle \text{octant} \rangle$ for negative x . Then we order $|y|$ and $|x|$ in a non-decreasing order: if $|y| > |x|$, insert 3- in the expression for the octant, and swap the two numbers. The finer test with 0.41421 is done by _fp_atan_div:wnwwnw after the operands have been ordered.

```

21279 \cs_new:Npn \_fp_atan_test_o:NwwNwwN #1#2,#3; #4#5,#6;
21280 {
21281   \exp_after:wN \_fp_atan_combine_o:NwwwwwN
21282   \exp_after:wN #1
21283   \int_value:w \_fp_int_eval:w
21284   \if_meaning:w 2 #4
21285     7 - \_fp_int_eval:w
21286   \fi:
21287   \if_int_compare:w
21288     \_fp_ep_compare:www #2,#3; #5,#6; > 0 \exp_stop_f:
21289     3 -
21290   \exp_after:wN \_fp_reverse_args:Nww
21291   \fi:
21292   \_fp_atan_div:wnwwnw #2,#3; #5,#6;
21293 }

```

(End definition for _fp_atan_test_o:NwwNwwN.)

_fp_atan_div:wnwwnw
_fp_atan_near:wwwN
_fp_atan_near_aux:wwN

This receives two positive numbers a and b (equal to $|x|$ and $|y|$ in some order), each as an exponent and 6 blocks of 4 digits, such that $0 < a < b$. If $0.41421b < a$, the two numbers are “near”, hence the point (y, x) that we started with is closer to the diagonals $\{|y| = |x|\}$ than to the axes $\{xy = 0\}$. In that case, the octant is 1 (possibly combined with the 7- and 3- inserted earlier) and we wish to compute $\operatorname{atan} \frac{b-a}{a+b}$. Otherwise, the octant is 0 (again, combined with earlier terms) and we wish to compute $\operatorname{atan} \frac{a}{b}$. In any case, call _fp_atan_auxi:ww followed by z , as a comma-delimited exponent and a fixed point number.

```

21294 \cs_new:Npn \_fp_atan_div:wnwwnw #1,#2#3; #4,#5#6;
21295 {
21296   \if_int_compare:w
21297     \_fp_int_eval:w 41421 * #5 < #2 000
21298     \if_case:w \_fp_int_eval:w #4 - #1 \_fp_int_eval_end:
21299     00 \or: 0 \fi:
21300   \exp_stop_f:
21301   \exp_after:wN \_fp_atan_near:wwwN
21302   \fi:
21303   0
21304   \_fp_ep_div:wwwN #1,{#2}#3; #4,{#5}#6;
21305   \_fp_atan_auxi:ww
21306 }
21307 \cs_new:Npn \_fp_atan_near:wwwN
21308   0 \_fp_ep_div:wwwN #1,#2; #3,

```

```

21309 {
21310     1
21311     \__fp_ep_to_fixed:wwn #1 - #3, #2;
21312     \__fp_atan_near_aux:wwn
21313 }
21314 \cs_new:Npn \__fp_atan_near_aux:wwn #1; #2;
21315 {
21316     \__fp_fixed_add:wwn #1; #2;
21317     { \__fp_fixed_sub:wwn #2; #1; { \__fp_ep_div:wwwn 0, } 0, }
21318 }

```

(End definition for __fp_atan_div:wwwnw, __fp_atan_near:wwwn, and __fp_atan_near_aux:wwn.)

__fp_atan_auxi:ww Convert z from a representation as an exponent and a fixed point number in $[0.01, 1)$ to a
 __fp_atan_auxii:w fixed point number only, then set up the call to __fp_atan_Taylor_loop:www, followed
 by the fixed point representation of z and the old representation.

```

21319 \cs_new:Npn \__fp_atan_auxi:ww #1,#2;
21320 { \__fp_ep_to_fixed:wwn #1,#2; \__fp_atan_auxii:w #1,#2; }
21321 \cs_new:Npn \__fp_atan_auxii:w #1;
21322 {
21323     \__fp_fixed_mul:wwn #1; #1;
21324     {
21325         \__fp_atan_Taylor_loop:www 39 ;
21326         {0000}{0000}{0000}{0000}{0000}{0000} ;
21327     }
21328     ! #1;
21329 }

```

(End definition for __fp_atan_auxi:ww and __fp_atan_auxii:w.)

__fp_atan_Taylor_loop:www We compute the series of $(\operatorname{atan} z)/z$. A typical intermediate stage has $\#1 = 2k - 1$,
 __fp_atan_Taylor_break:w $\#2 = \frac{1}{2k+1} - z^2(\frac{1}{2k+3} - z^2(\dots - z^2\frac{1}{39}))$, and $\#3 = z^2$. To go to the next step $k \rightarrow k - 1$,
 we compute $\frac{1}{2k-1}$, then subtract from it z^2 times $\#2$. The loop stops when $k = 0$: then
 $\#2$ is $(\operatorname{atan} z)/z$, and there is a need to clean up all the unnecessary data, end the integer
 expression computing the octant with a semicolon, and leave the result $\#2$ afterwards.

```

21330 \cs_new:Npn \__fp_atan_Taylor_loop:www #1; #2; #3;
21331 {
21332     \if_int_compare:w #1 = -1 \exp_stop_f:
21333     \__fp_atan_Taylor_break:w
21334     \fi:
21335     \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl #1;
21336     \__fp_rrot:www \__fp_fixed_mul_sub_back:wwwn #2; #3;
21337     {
21338         \exp_after:wN \__fp_atan_Taylor_loop:www
21339         \int_value:w \__fp_int_eval:w #1 - 2 ;
21340     }
21341     #3;
21342 }
21343 \cs_new:Npn \__fp_atan_Taylor_break:w
21344 \fi: #1 \__fp_fixed_mul_sub_back:wwwn #2; #3 !
21345 { \fi: ; #2 ; }

```

(End definition for __fp_atan_Taylor_loop:www and __fp_atan_Taylor_break:w.)

`_fp_atan_combine_o:NwwwwN` This receives a $\langle sign \rangle$, an $\langle octant \rangle$, a fixed point value of $(\text{atan } z)/z$, a fixed point number z , and another representation of z , as an $\langle exponent \rangle$ and the fixed point number $10^{-\langle exponent \rangle} z$, followed by either `\use_i:nn` (when working in radians) or `\use_ii:nn` (when working in degrees). The function computes the floating point result

$$\langle sign \rangle \left(\left\lceil \frac{\langle octant \rangle}{2} \right\rceil \frac{\pi}{4} + (-1)^{\langle octant \rangle} \frac{\text{atan } z}{z} \cdot z \right), \quad (11)$$

multiplied by $180/\pi$ if working in degrees, and using in any case the most appropriate representation of z . The floating point result is passed to `_fp_sanitize:Nw`, which checks for overflow or underflow. If the octant is 0, leave the exponent #5 for `_fp_sanitize:Nw`, and multiply #3 = $\frac{\text{atan } z}{z}$ with #6, the adjusted z . Otherwise, multiply #3 = $\frac{\text{atan } z}{z}$ with #4 = z , then compute the appropriate multiple of $\frac{\pi}{4}$ and add or subtract the product #3 · #4. In both cases, convert to a floating point with `_fp_fixed_to_float_o:wN`.

```

21346 \cs_new:Npn \_fp_atan_combine_o:NwwwwN #1 #2; #3; #4; #5,#6; #7
21347 {
21348   \exp_after:wN \_fp_sanitize:Nw
21349   \exp_after:wN #1
21350   \int_value:w \_fp_int_eval:w
21351   \if_meaning:w 0 #2
21352     \exp_after:wN \use_i:nn
21353   \else:
21354     \exp_after:wN \use_ii:nn
21355   \fi:
21356   { #5 \_fp_fixed_mul:wwn #3; #6; }
21357   {
21358     \_fp_fixed_mul:wwn #3; #4;
21359     {
21360       \exp_after:wN \_fp_atan_combine_aux:ww
21361       \int_value:w \_fp_int_eval:w #2 / 2 ; #2;
21362     }
21363   }
21364   { #7 \_fp_fixed_to_float_o:wN \_fp_fixed_to_float_rad_o:wN }
21365   #1
21366 }
21367 \cs_new:Npn \_fp_atan_combine_aux:ww #1; #2;
21368 {
21369   \_fp_fixed_mul_short:wwn
21370   {7853}{9816}{3397}{4483}{0961}{5661};
21371   {#1}{0000}{0000};
21372   {
21373     \if_int_odd:w #2 \exp_stop_f:
21374     \exp_after:wN \_fp_fixed_sub:wwn
21375   \else:
21376     \exp_after:wN \_fp_fixed_add:wwn
21377   \fi:
21378   }
21379 }

```

(End definition for `_fp_atan_combine_o:NwwwwN` and `_fp_atan_combine_aux:ww`.)

34.2.2 Arcsine and arccosine

`__fp_asin_o:w` Again, the first argument provided by `l3fp-parse` is `\use_i:nn` if we are to work in radians and `\use_ii:nn` for degrees. Then comes a floating point number. The arcsine of ± 0 or NaN is the same floating point number. The arcsine of $\pm\infty$ raises an invalid operation exception. Otherwise, call an auxiliary common with `__fp_acos_o:w`, feeding it information about what function is being performed (for “invalid operation” exceptions).

```

21380 \cs_new:Npn \__fp_asin_o:w #1 \s__fp \__fp_chk:w #2#3; @
21381 {
21382   \if_case:w #2 \exp_stop_f:
21383     \__fp_case_return_same_o:w
21384   \or:
21385     \__fp_case_use:nw
21386     { \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { asin } { asind } } }
21387   \or:
21388     \__fp_case_use:nw
21389     { \__fp_invalid_operation_o:fw { #1 { asin } { asind } } }
21390   \else:
21391     \__fp_case_return_same_o:w
21392   \fi:
21393   \s__fp \__fp_chk:w #2 #3;
21394 }

```

(End definition for `__fp_asin_o:w`.)

`__fp_acos_o:w` The arccosine of ± 0 is $\pi/2$ (in degrees, 90). The arccosine of $\pm\infty$ raises an invalid operation exception. The arccosine of NaN is itself. Otherwise, call an auxiliary common with `__fp_sin_o:w`, informing it that it was called by `acos` or `acosd`, and preparing to swap some arguments down the line.

```

21395 \cs_new:Npn \__fp_acos_o:w #1 \s__fp \__fp_chk:w #2#3; @
21396 {
21397   \if_case:w #2 \exp_stop_f:
21398     \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 4 }
21399   \or:
21400     \__fp_case_use:nw
21401     {
21402       \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { acos } { acosd } }
21403       \__fp_reverse_args:Nww
21404     }
21405   \or:
21406     \__fp_case_use:nw
21407     { \__fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
21408   \else:
21409     \__fp_case_return_same_o:w
21410   \fi:
21411   \s__fp \__fp_chk:w #2 #3;
21412 }

```

(End definition for `__fp_acos_o:w`.)

`__fp_asin_normal_o:NfwNnnnnw` If the exponent #5 is at most 0, the operand lies within $(-1, 1)$ and the operation is permitted: call `__fp_asin_auxi_o:NnNw` with the appropriate arguments. If the number is exactly ± 1 (the test works because we know that $\#5 \geq 1$, $\#6\#7 \geq 10000000$, $\#8\#9 \geq 0$,

with equality only for ± 1), we also call `__fp_asin_auxi_o:NnNww`. Otherwise, `__fp_use_i:ww` gets rid of the `asin` auxiliary, and raises instead an invalid operation, because the operand is outside the domain of arcsine or arccosine.

```

21413 \cs_new:Npn \__fp_asin_normal_o:NfwNnnnnw
21414   #1#2#3 \s__fp \__fp_chk:w 1#4#5#6#7#8#9;
21415   {
21416     \if_int_compare:w #5 < 1 \exp_stop_f:
21417       \exp_after:wN \__fp_use_none_until_s:w
21418     \fi:
21419     \if_int_compare:w \__fp_int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
21420       \exp_after:wN \__fp_use_none_until_s:w
21421     \fi:
21422     \__fp_use_i:ww
21423     \__fp_invalid_operation_o:fw {#2}
21424     \s__fp \__fp_chk:w 1#4#{#5}{#6}{#7}{#8}{#9};
21425     \__fp_asin_auxi_o:NnNww
21426     #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000};
21427   }

```

(End definition for `__fp_asin_normal_o:NfwNnnnnw`.)

`__fp_asin_auxi_o:NnNww`
`__fp_asin_isqrt:wn`

We compute $x/\sqrt{1-x^2}$. This function is used by `asin` and `acos`, but also by `acsc` and `asec` after inverting the operand, thus it must manipulate extended-precision numbers. First evaluate $1-x^2$ as $(1+x)(1-x)$: this behaves better near $x = 1$. We do the addition/subtraction with fixed point numbers (they are not implemented for extended-precision floats), but go back to extended-precision floats to multiply and compute the inverse square root $1/\sqrt{1-x^2}$. Finally, multiply by the (positive) extended-precision float $|x|$, and feed the (signed) result, and the number $+1$, as arguments to the arctangent function. When computing the arccosine, the arguments $x/\sqrt{1-x^2}$ and $+1$ are swapped by `#2` (`__fp_reverse_args:Nww` in that case) before `__fp_atan_test_o:NwwNwwN` is evaluated. Note that the arctangent function requires normalized arguments, hence the need for `ep_to_ep` and continue after `ep_mul`.

```

21428 \cs_new:Npn \__fp_asin_auxi_o:NnNww #1#2#3#4,#5;
21429   {
21430     \__fp_ep_to_fixed:wwn #4,#5;
21431     \__fp_asin_isqrt:wn
21432     \__fp_ep_mul:wwwwn #4,#5;
21433     \__fp_ep_to_ep:wwN
21434     \__fp_fixed_continue:wn
21435     { #2 \__fp_atan_test_o:NwwNwwN #3 }
21436     0 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1
21437   }
21438 \cs_new:Npn \__fp_asin_isqrt:wn #1;
21439   {
21440     \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl #1;
21441     {
21442       \__fp_fixed_add_one:wn #1;
21443       \__fp_fixed_continue:wn { \__fp_ep_mul:wwwwn 0, } 0,
21444     }
21445     \__fp_ep_isqrt:wwn
21446   }

```

(End definition for `__fp_asin_auxi_o:NnNww` and `__fp_asin_isqrt:wn`.)

34.2.3 Arccosecant and arcsecant

`__fp_acsc_o:w` Cases are mostly labelled by #2, except when #2 is 2: then we use #3#2, which is 02 = 2 when the number is $+\infty$ and 22 when the number is $-\infty$. The arccosecant of ± 0 raises an invalid operation exception. The arccosecant of $\pm\infty$ is ± 0 with the same sign. The arcosecant of NaN is itself. Otherwise, `__fp_acsc_normal_o:NfwNnw` does some more tests, keeping the function name (acsc or acscd) as an argument for invalid operation exceptions.

```

21447 \cs_new:Npn \__fp_acsc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
21448 {
21449     \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
21450         \__fp_case_use:nw
21451         { \__fp_invalid_operation_o:fw { #1 { acsc } { acscd } } }
21452     \or: \__fp_case_use:nw
21453         { \__fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acscd } } }
21454     \or: \__fp_case_return_o:Nw \c_zero_fp
21455     \or: \__fp_case_return_same_o:w
21456     \else: \__fp_case_return_o:Nw \c_minus_zero_fp
21457     \fi:
21458     \s__fp \__fp_chk:w #2 #3 #4;
21459 }

```

(End definition for `__fp_acsc_o:w`.)

`__fp_asec_o:w` The arcsecant of ± 0 raises an invalid operation exception. The arcsecant of $\pm\infty$ is $\pi/2$ (in degrees, 90). The arcosecant of NaN is itself. Otherwise, do some more tests, keeping the function name asec (or asecd) as an argument for invalid operation exceptions, and a `__fp_reverse_args:Nww` following precisely that appearing in `__fp_acos_o:w`.

```

21460 \cs_new:Npn \__fp_asec_o:w #1 \s__fp \__fp_chk:w #2#3; @
21461 {
21462     \if_case:w #2 \exp_stop_f:
21463         \__fp_case_use:nw
21464         { \__fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
21465     \or:
21466         \__fp_case_use:nw
21467         {
21468             \__fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
21469             \__fp_reverse_args:Nww
21470         }
21471     \or: \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 4 }
21472     \else: \__fp_case_return_same_o:w
21473     \fi:
21474     \s__fp \__fp_chk:w #2 #3;
21475 }

```

(End definition for `__fp_asec_o:w`.)

`__fp_acsc_normal_o:NfwNnw` If the exponent is non-positive, the operand is less than 1 in absolute value, which is always an invalid operation: complain. Otherwise, compute the inverse of the operand, and feed it to `__fp_asin_auxi_o:NnNww` (with all the appropriate arguments). This computes what we want thanks to $\text{acsc}(x) = \text{asin}(1/x)$ and $\text{asec}(x) = \text{acos}(1/x)$.

```

21476 \cs_new:Npn \__fp_acsc_normal_o:NfwNnw #1#2#3 \s__fp \__fp_chk:w 1#4#5#6;
21477 {
21478     \int_compare:nNnTF {#5} < 1

```



```

21479     {
21480       \__fp_invalid_operation_o:fw {#2}
21481       \s__fp \__fp_chk:w 1#4{#5}#6;
21482     }
21483     {
21484       \__fp_ep_div:wwwwn
21485       1,{1000}{0000}{0000}{0000}{0000}{0000};
21486       #5,#6{0000}{0000};
21487     { \__fp_asin_auxi_o:NnNww #1 {#3} #4 }
21488     }
21489   }

```

(End definition for __fp_acsc_normal_o:NfwNnw.)

```
21490 \end{package}
```

35 13fp-convert implementation

```
21491 \begin{package}
```

```
21492 \let\fp=fp
```

35.1 Dealing with tuples

The first argument is for instance __fp_to_t1_dispatch:w, which converts any floating point object to the appropriate representation. We loop through all items, putting ,~ between all of them and making sure to remove the leading ,~.

```

21493 \cs_new:Npn \__fp_tuple_convert:Nw #1 \s__fp_tuple \__fp_tuple_chk:w #2 ;
21494 {
21495   \int_case:nnF { \__fp_array_count:n {#2} }
21496   {
21497     { 0 } { ( ) }
21498     { 1 } { \__fp_tuple_convert_end:w @ { #1 #2 , } }
21499   }
21500   {
21501     \__fp_tuple_convert_loop:nNw { } #1
21502     #2 { ? \__fp_tuple_convert_end:w } ;
21503     @ { \use_none:nn }
21504   }
21505 }
21506 \cs_new:Npn \__fp_tuple_convert_loop:nNw #1#2#3#4; #5 @ #6
21507 {
21508   \use_none:n #3
21509   \exp_args:Nf \__fp_tuple_convert_loop:nNw { #2 #3#4 ; } #2 #5
21510   @ { #6 , ~ #1 }
21511 }
21512 \cs_new:Npn \__fp_tuple_convert_end:w #1 @ #2
21513 { \exp_after:wN ( \exp:w \exp_end_continue_f:w #2 ) }

```

(End definition for __fp_tuple_convert:Nw, __fp_tuple_convert_loop:nNw, and __fp_tuple_convert_end:w.)

35.2 Trimming trailing zeros

`__fp_trim_zeros:w` If #1 ends with a 0, the loop auxiliary takes that zero as an end-delimiter for its first argument, and the second argument is the same loop auxiliary. Once the last trailing zero is reached, the second argument is the dot auxiliary, which removes a trailing dot if any. We then clean-up with the end auxiliary, keeping only the number.

```
21514 \cs_new:Npn \__fp_trim_zeros:w #1 ;
21515 {
21516   \__fp_trim_zeros_loop:w #1
21517   ; \__fp_trim_zeros_loop:w 0; \__fp_trim_zeros_dot:w .; \s__fp_stop
21518 }
21519 \cs_new:Npn \__fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
21520 \cs_new:Npn \__fp_trim_zeros_dot:w #1 .; { \__fp_trim_zeros_end:w #1 ; }
21521 \cs_new:Npn \__fp_trim_zeros_end:w #1 ; #2 \s__fp_stop { #1 }
```

(End definition for `__fp_trim_zeros:w` and others.)

35.3 Scientific notation

`\fp_to_scientific:N` The three public functions evaluate their argument, then pass it to `__fp_to_scientific_dispatch:w`.

```
\fp_to_scientific:c
\fp_to_scientific:n
21522 \cs_new:Npn \fp_to_scientific:N #1
21523 { \exp_after:wN \__fp_to_scientific_dispatch:w #1 }
21524 \cs_generate_variant:Nn \fp_to_scientific:N { c }
21525 \cs_new:Npn \fp_to_scientific:n
21526 {
21527   \exp_after:wN \__fp_to_scientific_dispatch:w
21528   \exp:w \exp_end_continue_f:w \__fp_parse:n
21529 }
```

(End definition for `\fp_to_scientific:N` and `\fp_to_scientific:n`. These functions are documented on page 204.)

`_fp_to_scientific_dispatch:w` We allow tuples.

```
\_fp_to_scientific_recover:w
\__fp_tuple_to_scientific:w
21530 \cs_new:Npn \__fp_to_scientific_dispatch:w #1
21531 {
21532   \__fp_change_func_type:NNN
21533   #1 \__fp_to_scientific:w \__fp_to_scientific_recover:w
21534   #1
21535 }
21536 \cs_new:Npn \__fp_to_scientific_recover:w #1 #2 ;
21537 {
21538   \__fp_error:nffn { fp-unknown-type } { \tl_to_str:n { #2 ; } } { } { }
21539   nan
21540 }
21541 \cs_new:Npn \__fp_tuple_to_scientific:w
21542 { \__fp_tuple_convert:Nw \__fp_to_scientific_dispatch:w }
```

(End definition for `__fp_to_scientific_dispatch:w`, `__fp_to_scientific_recover:w`, and `__fp_tuple_to_scientific:w`.)

`__fp_to_scientific:w` Expressing an internal floating point number in scientific notation is quite easy: no rounding, and the format is very well defined. First cater for the sign: negative numbers (`#2 = 2`) start with `-`; we then only need to care about positive numbers and `nan`. Then

filter the special cases: ± 0 are represented as 0; infinities are converted to a number slightly larger than the largest after an “invalid_operation” exception; `nan` is represented as 0 after an “invalid_operation” exception. In the normal case, decrement the exponent and unbrace the 4 brace groups, then in a second step grab the first digit (previously hidden in braces) to order the various parts correctly.

```

21543 \cs_new:Npn \__fp_to_scientific:w \s__fp \__fp_chk:w #1#2
21544 {
21545   \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
21546   \if_case:w #1 \exp_stop_f:
21547     \__fp_case_return:nw { 0.000000000000000e0 }
21548   \or: \exp_after:wN \__fp_to_scientific_normal:wnnnnn
21549   \or:
21550     \__fp_case_use:nw
21551     {
21552       \__fp_invalid_operation:nnw
21553       { \fp_to_scientific:N \c__fp_overflowing_fp }
21554       { fp_to_scientific }
21555     }
21556   \or:
21557     \__fp_case_use:nw
21558     {
21559       \__fp_invalid_operation:nnw
21560       { \fp_to_scientific:N \c_zero_fp }
21561       { fp_to_scientific }
21562     }
21563   \fi:
21564   \s__fp \__fp_chk:w #1 #2
21565 }
21566 \cs_new:Npn \__fp_to_scientific_normal:wnnnnn
21567 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
21568 {
21569   \exp_after:wN \__fp_to_scientific_normal:wNw
21570   \exp_after:wN e
21571   \int_value:w \__fp_int_eval:w #2 - 1
21572   ; #3 #4 #5 #6 ;
21573 }
21574 \cs_new:Npn \__fp_to_scientific_normal:wNw #1 ; #2#3;
21575 { #2.#3 #1 }

```

(End definition for `__fp_to_scientific:w`, `__fp_to_scientific_normal:wnnnnn`, and `__fp_to_scientific_normal:wNw`.)

35.4 Decimal representation

`\fp_to_decimal:N` All three public variants are based on the same `__fp_to_decimal_dispatch:w` after evaluating their argument to an internal floating point.

```

\fp_to_decimal:c
\fp_to_decimal:n
21576 \cs_new:Npn \fp_to_decimal:N #1
21577 { \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
21578 \cs_generate_variant:Nn \fp_to_decimal:N { c }
21579 \cs_new:Npn \fp_to_decimal:n
21580 {
21581   \exp_after:wN \__fp_to_decimal_dispatch:w
21582   \exp:w \exp_end_continue_f:w \__fp_parse:n
21583 }

```

(End definition for `\fp_to_decimal:N` and `\fp_to_decimal:n`. These functions are documented on page 204.)

`__fp_to_decimal_dispatch:w`
`__fp_to_decimal_recover:w`
`__fp_tuple_to_decimal:w`

We allow tuples.

```

21584 \cs_new:Npn \__fp_to_decimal_dispatch:w #1
21585 {
21586   \__fp_change_func_type:NNN
21587   #1 \__fp_to_decimal:w \__fp_to_decimal_recover:w
21588   #1
21589 }
21590 \cs_new:Npn \__fp_to_decimal_recover:w #1 #2 ;
21591 {
21592   \__fp_error:nffn { fp-unknown-type } { \tl_to_str:n { #2 ; } } { } { }
21593   nan
21594 }
21595 \cs_new:Npn \__fp_tuple_to_decimal:w
21596 { \__fp_tuple_convert:Nw \__fp_to_decimal_dispatch:w }
```

(End definition for `__fp_to_decimal_dispatch:w`, `__fp_to_decimal_recover:w`, and `__fp_tuple_to_decimal:w`.)

`__fp_to_decimal:w`
`_fp_to_decimal_normal:wnnnnn`
`__fp_to_decimal_large:Nnnw`
`__fp_to_decimal_huge:wnnnn`

The structure is similar to `__fp_to_scientific:w`. Insert `-` for negative numbers. Zero gives 0, $\pm\infty$ and NaN yield an “invalid operation” exception; note that $\pm\infty$ produces a very large output, which we don’t expand now since it most likely won’t be needed. Normal numbers with an exponent in the range [1, 15] have that number of digits before the decimal separator: “decimate” them, and remove leading zeros with `\int_value:w`, then trim trailing zeros and dot. Normal numbers with an exponent 16 or larger have no decimal separator, we only need to add trailing zeros. When the exponent is non-positive, the result should be 0.<zeros><digits>, trimmed.

```

21597 \cs_new:Npn \__fp_to_decimal:w \s__fp \__fp_chk:w #1#2
21598 {
21599   \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
21600   \if_case:w #1 \exp_stop_f:
21601     \__fp_case_return:nw { 0 }
21602   \or: \exp_after:wN \__fp_to_decimal_normal:wnnnnn
21603   \or:
21604     \__fp_case_use:nw
21605     {
21606       \__fp_invalid_operation:nnw
21607       { \fp_to_decimal:N \c__fp_overflowing_fp }
21608       { fp_to_decimal }
21609     }
21610   \or:
21611     \__fp_case_use:nw
21612     {
21613       \__fp_invalid_operation:nnw
21614       { 0 }
21615       { fp_to_decimal }
21616     }
21617   \fi:
21618   \s__fp \__fp_chk:w #1 #2
21619 }
21620 \cs_new:Npn \__fp_to_decimal_normal:wnnnnn
21621 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
```

```

21622 {
21623   \int_compare:nNnTF {#2} > 0
21624   {
21625     \int_compare:nNnTF {#2} < \c__fp_prec_int
21626     {
21627       \__fp_decimate:nNnnnn { \c__fp_prec_int - #2 }
21628       \__fp_to_decimal_large:Nnnw
21629     }
21630     {
21631       \exp_after:wN \exp_after:wN
21632       \exp_after:wN \__fp_to_decimal_huge:wnnnn
21633       \prg_replicate:nn { #2 - \c__fp_prec_int } { 0 } ;
21634     }
21635     {#3} {#4} {#5} {#6}
21636   }
21637   {
21638     \exp_after:wN \__fp_trim_zeros:w
21639     \exp_after:wN 0
21640     \exp_after:wN .
21641     \exp:w \exp_end_continue_f:w \prg_replicate:nn { - #2 } { 0 }
21642     #3#4#5#6 ;
21643   }
21644 }
21645 \cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4;
21646 {
21647   \exp_after:wN \__fp_trim_zeros:w \int_value:w
21648   \if_int_compare:w #2 > 0 \exp_stop_f:
21649     #2
21650   \fi:
21651   \exp_stop_f:
21652   #3.#4 ;
21653 }
21654 \cs_new:Npn \__fp_to_decimal_huge:wnnnn #1; #2#3#4#5 { #2#3#4#5 #1 }

```

(End definition for __fp_to_decimal:w and others.)

35.5 Token list representation

\fp_to_tl:N These three public functions evaluate their argument, then pass it to __fp_to_tl_dispatch:w.
\fp_to_tl:c dispatch:w.

\fp_to_tl:n

```

21655 \cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \__fp_to_tl_dispatch:w #1 }
21656 \cs_generate_variant:Nn \fp_to_tl:N { c }
21657 \cs_new:Npn \fp_to_tl:n
21658 {
21659   \exp_after:wN \__fp_to_tl_dispatch:w
21660   \exp:w \exp_end_continue_f:w \__fp_parse:n
21661 }

```

(End definition for \fp_to_tl:N and \fp_to_tl:n. These functions are documented on page 205.)

__fp_to_tl_dispatch:w We allow tuples.

```

21662 \cs_new:Npn \__fp_to_tl_dispatch:w #1
21663 { \__fp_change_func_type:NNN #1 \__fp_to_tl:w \__fp_to_tl_recover:w #1 }
21664 \cs_new:Npn \__fp_to_tl_recover:w #1 #2 ;

```

```

21665 {
21666     \__fp_error:nffn { fp-unknown-type } { \tl_to_str:n { #2 ; } } { } { }
21667     nan
21668 }
21669 \cs_new:Npn \__fp_tuple_to_tl:w
21670 { \__fp_tuple_convert:Nw \__fp_to_tl_dispatch:w }

```

(End definition for __fp_to_tl_dispatch:w, __fp_to_tl_recover:w, and __fp_tuple_to_tl:w.)

__fp_to_tl:w A structure similar to __fp_to_scientific_dispatch:w and __fp_to_decimal_dispatch:w, but without the “invalid operation” exception. First filter special cases. We express normal numbers in decimal notation if the exponent is in the range $[-2, 16]$, and otherwise use scientific notation.

```

21671 \cs_new:Npn \__fp_to_tl:w \s__fp \__fp_chk:w #1#2
21672 {
21673     \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
21674     \if_case:w #1 \exp_stop_f:
21675         \__fp_case_return:nw { 0 }
21676     \or: \exp_after:wN \__fp_to_tl_normal:nnnnn
21677     \or: \__fp_case_return:nw { inf }
21678     \else: \__fp_case_return:nw { nan }
21679     \fi:
21680 }
21681 \cs_new:Npn \__fp_to_tl_normal:nnnnn #1
21682 {
21683     \int_compare:nTF
21684         { -2 <= #1 <= \c__fp_prec_int }
21685         { \__fp_to_decimal_normal:wnnnnnn }
21686         { \__fp_to_tl_scientific:wnnnnnn }
21687     \s__fp \__fp_chk:w 1 0 {#1}
21688 }
21689 \cs_new:Npn \__fp_to_tl_scientific:wnnnnnn
21690     \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
21691 {
21692     \exp_after:wN \__fp_to_tl_scientific:wNw
21693     \exp_after:wN e
21694     \int_value:w \__fp_int_eval:w #2 - 1
21695     ; #3 #4 #5 #6 ;
21696 }
21697 \cs_new:Npn \__fp_to_tl_scientific:wNw #1 ; #2#3;
21698 { \__fp_trim_zeros:w #2.#3 ; #1 }

```

(End definition for __fp_to_tl:w and others.)

35.6 Formatting

This is not implemented yet, as it is not yet clear what a correct interface would be, for this kind of structured conversion from a floating point (or other types of variables) to a string. Ideas welcome.

35.7 Convert to dimension or integer

`\fp_to_dim:N` All three public variants are based on the same `__fp_to_dim_dispatch:w` after evaluating their argument to an internal floating point. We only allow floating point numbers, not tuples.

```

\__fp_to_dim_dispatch:w 21699 \cs_new:Npn \fp_to_dim:N #1
\__fp_to_dim_recover:w 21700 { \exp_after:wN \__fp_to_dim_dispatch:w #1 }
\__fp_to_dim:w 21701 \cs_generate_variant:Nn \fp_to_dim:N { c }
21702 \cs_new:Npn \fp_to_dim:n
21703 {
21704     \exp_after:wN \__fp_to_dim_dispatch:w
21705     \exp:w \exp_end_continue_f:w \__fp_parse:n
21706 }
21707 \cs_new:Npn \__fp_to_dim_dispatch:w #1#2 ;
21708 {
21709     \__fp_change_func_type:NNN #1 \__fp_to_dim:w \__fp_to_dim_recover:w
21710     #1 #2 ;
21711 }
21712 \cs_new:Npn \__fp_to_dim_recover:w #1
21713 { \__fp_invalid_operation:nnw { Opt } { fp_to_dim } }
21714 \cs_new:Npn \__fp_to_dim:w #1 ; { \__fp_to_decimal:w #1 ; pt }
```

(End definition for `\fp_to_dim:N` and others. These functions are documented on page 204.)

`\fp_to_int:N` For the most part identical to `\fp_to_dim:N` but without `pt`, and where `__fp_to_int:w` does more work. To convert to an integer, first round to 0 places (to the nearest integer), then express the result as a decimal number: the definition of `__fp_to_decimal_dispatch:w` is such that there are no trailing dot nor zero.

```

\__fp_to_int_dispatch:w 21715 \cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \__fp_to_int_dispatch:w #1 }
\__fp_to_int_recover:w 21716 \cs_generate_variant:Nn \fp_to_int:N { c }
21717 \cs_new:Npn \fp_to_int:n
21718 {
21719     \exp_after:wN \__fp_to_int_dispatch:w
21720     \exp:w \exp_end_continue_f:w \__fp_parse:n
21721 }
21722 \cs_new:Npn \__fp_to_int_dispatch:w #1#2 ;
21723 {
21724     \__fp_change_func_type:NNN #1 \__fp_to_int:w \__fp_to_int_recover:w
21725     #1 #2 ;
21726 }
21727 \cs_new:Npn \__fp_to_int_recover:w #1
21728 { \__fp_invalid_operation:nnw { 0 } { fp_to_int } }
21729 \cs_new:Npn \__fp_to_int:w #1;
21730 {
21731     \exp_after:wN \__fp_to_decimal:w \exp:w \exp_end_continue_f:w
21732     \__fp_round:Nwn \__fp_round_to_nearest:NNN #1; { 0 }
21733 }
```

(End definition for `\fp_to_int:N` and others. These functions are documented on page 204.)

35.8 Convert from a dimension

`\dim_to_fp:n` The dimension expression (which can in fact be a glue expression) is evaluated, converted to a number (*i.e.*, expressed in scaled points), then multiplied by $2^{-16} =$

```

\__fp_from_dim_test:ww
\__fp_from_dim:wNw
\__fp_from_dim:wNNnnnnnn
\__fp_from_dim:wnnnnwNw
```

0.0000152587890625 to give a value expressed in points. The auxiliary `__fp_mul_npos_o:Nww` expects the desired *final sign* and two floating point operands (of the form `\s__fp ... ;`) as arguments. This set of functions is also used to convert dimension registers to floating points while parsing expressions: in this context there is an additional exponent, which is the first argument of `__fp_from_dim_test:ww`, and is combined with the exponent -4 of 2^{-16} . There is also a need to expand afterwards: this is performed by `__fp_mul_npos_o:Nww`, and cancelled by `\prg_do_nothing:` here.

```

21734 \cs_new:Npn \dim_to_fp:n #1
21735 {
21736   \exp_after:wN \__fp_from_dim_test:ww
21737   \exp_after:wN 0
21738   \exp_after:wN ,
21739   \int_value:w \tex_glueexpr:D #1 ;
21740 }
21741 \cs_new:Npn \__fp_from_dim_test:ww #1, #2
21742 {
21743   \if_meaning:w 0 #2
21744     \__fp_case_return:nw { \exp_after:wN \c_zero_fp }
21745   \else:
21746     \exp_after:wN \__fp_from_dim:wNw
21747     \int_value:w \__fp_int_eval:w #1 - 4
21748     \if_meaning:w - #2
21749       \exp_after:wN , \exp_after:wN 2 \int_value:w
21750     \else:
21751       \exp_after:wN , \exp_after:wN 0 \int_value:w #2
21752     \fi:
21753   \fi:
21754 }
21755 \cs_new:Npn \__fp_from_dim:wNw #1,#2#3;
21756 {
21757   \__fp_pack_twice_four:wNNNNNNNN \__fp_from_dim:wNNnnnnnn ;
21758   #3 000 0000 00 {10}987654321; #2 {#1}
21759 }
21760 \cs_new:Npn \__fp_from_dim:wNNnnnnnn #1; #2#3#4#5#6#7#8#9
21761 { \__fp_from_dim:wnnnnwNn #1 {#2#300} {0000} ; }
21762 \cs_new:Npn \__fp_from_dim:wnnnnwNn #1; #2#3#4#5#6; #7#8
21763 {
21764   \__fp_mul_npos_o:Nww #7
21765   \s__fp \__fp_chk:w 1 #7 {#5} #1 ;
21766   \s__fp \__fp_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} ;
21767   \prg_do_nothing:
21768 }

```

(End definition for `\dim_to_fp:n` and others. This function is documented on page 178.)

35.9 Use and eval

`\fp_use:N` Those public functions are simple copies of the decimal conversions.
`\fp_use:c` 21769 \cs_new_eq:NN \fp_use:N \fp_to_decimal:N
`\fp_eval:n` 21770 \cs_generate_variant:Nn \fp_use:N { c }
21771 \cs_new_eq:NN \fp_eval:n \fp_to_decimal:n

(End definition for `\fp_use:N` and `\fp_eval:n`. These functions are documented on page 205.)

\fp_sign:n Trivial but useful. See the implementation of `\fp_add:Nn` for an explanation of why to use `__fp_parse:n`, namely, for better error reporting.

```
21772 \cs_new:Npn \fp_sign:n #1
21773 { \fp_to_decimal:n { sign \__fp_parse:n {#1} } }
```

(End definition for `\fp_sign:n`. This function is documented on page 204.)

\fp_abs:n Trivial but useful. See the implementation of `\fp_add:Nn` for an explanation of why to use `__fp_parse:n`, namely, for better error reporting.

```
21774 \cs_new:Npn \fp_abs:n #1
21775 { \fp_to_decimal:n { abs \__fp_parse:n {#1} } }
```

(End definition for `\fp_abs:n`. This function is documented on page 219.)

\fp_max:nn Similar to `\fp_abs:n`, for consistency with `\int_max:nn`, etc.

\fp_min:nn

```
21776 \cs_new:Npn \fp_max:nn #1#2
21777 { \fp_to_decimal:n { max ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
21778 \cs_new:Npn \fp_min:nn #1#2
21779 { \fp_to_decimal:n { min ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
```

(End definition for `\fp_max:nn` and `\fp_min:nn`. These functions are documented on page 219.)

35.10 Convert an array of floating points to a comma list

`__fp_array_to_clist:n` Converts an array of floating point numbers to a comma-list. If speed here ends up irrelevant, we can simplify the code for the auxiliary to become

```
\cs_new:Npn \__fp_array_to_clist_loop:Nw #1#2;
{
  \use_none:n #1
  { , ~ } \fp_to_tl:n { #1 #2 ; }
  \__fp_array_to_clist_loop:Nw
}
```

The `\use_ii:nn` function is expanded after `__fp_expand:n` is done, and it removes `,~` from the start of the representation.

```
21780 \cs_new:Npn \__fp_array_to_clist:n #1
21781 {
21782   \tl_if_empty:nF {#1}
21783   {
21784     \exp_last_unbraced:Ne \use_ii:nn
21785     {
21786       \__fp_array_to_clist_loop:Nw #1 { ? \prg_break: } ;
21787       \prg_break_point:
21788     }
21789   }
21790 }
21791 \cs_new:Npn \__fp_array_to_clist_loop:Nw #1#2;
21792 {
21793   \use_none:n #1
21794   , ~
21795   \exp_not:f { \__fp_to_tl_dispatch:w #1 #2 ; }
21796   \__fp_array_to_clist_loop:Nw
21797 }
```

(End definition for `_fp_array_to_clist:n` and `_fp_array_to_clist_loop:Nw`.)

21798 `\</package>`

36 13fp-random Implementation

21799 `*package>`

21800 `<@@=fp>`

`_fp_parse_word_rand:N`
`_fp_parse_word_randint:N`

Those functions may receive a variable number of arguments. We won't use the argument ?.

21801 `\cs_new:Npn _fp_parse_word_rand:N`
 21802 `{ _fp_parse_function:NNN _fp_rand_o:Nw ? }`
 21803 `\cs_new:Npn _fp_parse_word_randint:N`
 21804 `{ _fp_parse_function:NNN _fp_randint_o:Nw ? }`

(End definition for `_fp_parse_word_rand:N` and `_fp_parse_word_randint:N`.)

36.1 Engine support

Most engines provide random numbers, but not all. We write the test twice simply in order to write the false branch first.

21805 `\sys_if_rand_exist:F`
 21806 `{`
 21807 `_kernel_msg_new:nnn { kernel } { fp-no-random }`
 21808 `{ Random-numbers-unavailable-for~#1 }`
 21809 `\cs_new:Npn _fp_rand_o:Nw ? #1 @`
 21810 `{`
 21811 `_kernel_msg_expandable_error:nnn { kernel } { fp-no-random }`
 21812 `{ fp-rand }`
 21813 `\exp_after:wN \c_nan_fp`
 21814 `}`
 21815 `\cs_new_eq:NN _fp_randint_o:Nw _fp_rand_o:Nw`
 21816 `\cs_new:Npn \int_rand:nn #1#2`
 21817 `{`
 21818 `_kernel_msg_expandable_error:nnn { kernel } { fp-no-random }`
 21819 `{ \int_rand:nn {#1} {#2} }`
 21820 `\int_eval:n {#1}`
 21821 `}`
 21822 `\cs_new:Npn \int_rand:n #1`
 21823 `{`
 21824 `_kernel_msg_expandable_error:nnn { kernel } { fp-no-random }`
 21825 `{ \int_rand:n {#1} }`
 21826 `1`
 21827 `}`
 21828 `}`
 21829 `\sys_if_rand_exist:T`
 21830 `{`

Obviously, every word “random” below means “pseudo-random”, as we have no access to entropy (except a very unreliable source of entropy: the time it takes to run some code).

The primitive random number generator (RNG) is provided as `\tex_uniformdeviate:D`. Under the hood, it maintains an array of 55 28-bit numbers, updated with a linear recursion relation (similar to Fibonacci numbers) modulo 2^{28} . When `\tex_uniformdeviate:D` $\langle integer \rangle$ is called (for brevity denote by N the $\langle integer \rangle$), the next 28-bit number is read from the array, scaled by $N/2^{28}$, and rounded. To prevent 0 and N from appearing half as often as other numbers, they are both mapped to the result 0.

This process means that `\tex_uniformdeviate:D` only gives a uniform distribution from 0 to $N-1$ if N is a divisor of 2^{28} , so we will mostly call the RNG with such power of 2 arguments. If N does not divide 2^{28} , then the relative non-uniformity (difference between probabilities of getting different numbers) is about $N/2^{28}$. This implies that detecting deviation from $1/N$ of the probability of a fixed value X requires about $2^{56}/N$ random trials. But collective patterns can reduce this to about $2^{56}/N^2$. For instance with $N = 3 \times 2^k$, the modulo 3 repartition of such random numbers is biased with a non-uniformity about $2^k/2^{28}$ (which is much worse than the circa $3/2^{28}$ non-uniformity from taking directly $N = 3$). This is detectable after about $2^{56}/2^{2k} = 9 \cdot 2^{56}/N^2$ random numbers. For $k = 15$, $N = 98304$, this means roughly 2^{26} calls to the RNG (experimentally this takes at the very least 16 seconds on a 2 giga-hertz processor). While this bias is not quite problematic, it is uncomfortably close to being so, and it becomes worse as N is increased. In our code, we shall thus combine several results from the RNG.

The RNG has three types of unexpected correlations. First, everything is linear modulo 2^{28} , hence the lowest k bits of the random numbers only depend on the lowest k bits of the seed (and of course the number of times the RNG was called since setting the seed). The recommended way to get a number from 0 to $N-1$ is thus to scale the raw 28-bit integer, as the engine's RNG does. We will go further and in fact typically we discard some of the lowest bits.

Second, suppose that we call the RNG with the same argument N to get a set of K integers in $[0, N-1]$ (throwing away repeats), and suppose that $N > K^3$ and $K > 55$. The recursion used to construct more 28-bit numbers from previous ones is linear: $x_n = x_{n-55} - x_{n-24}$ or $x_n = x_{n-55} - x_{n-24} + 2^{28}$. After rescaling and rounding we find that the result $N_n \in [0, N-1]$ is among $N_{n-55} - N_{n-24} + \{-1, 0, 1\}$ modulo N (a more detailed analysis shows that 0 appears with frequency close to $3/4$). The resulting set thus has more triplets (a, b, c) than expected obeying $a = b + c$ modulo N . Namely it will have of order $(K-55) \times 3/4$ such triplets, when one would expect $K^3/(6N)$. This starts to be detectable around $N = 2^{18} > 55^3$ (earlier if one keeps track of positions too, but this is more subtle than it looks because the array of 28-bit integers is read backwards by the engine). Hopefully the correlation is subtle enough to not affect realistic documents so we do not specifically mitigate against this. Since we typically use two calls to the RNG per `\int_rand:nn` we would need to investigate linear relations between the x_{2n} on the one hand and between the x_{2n+1} on the other hand. Such relations will have more complicated coefficients than ± 1 , which alleviates the issue.

Third, consider successive batches of 165 calls to the RNG (with argument 2^{28} or with argument 2 for instance), then most batches have more odd than even numbers. Note that this does not mean that there are more odd than even numbers overall. Similar issues are discussed in Knuth's TAOCP volume 2 near exercise 3.3.2-31. We do not have any mitigation strategy for this.

Ideally, our algorithm should be:

- Uniform. The result should be as uniform as possible assuming that the RNG's underlying 28-bit integers are uniform.

- Uncorrelated. The result should not have detectable correlations between different seeds, similar to the lowest-bit ones mentioned earlier.
- Quick. The algorithm should be fast in $\text{T}_{\text{E}}\text{X}$, so no “bit twiddling”, but “digit twiddling” is ok.
- Simple. The behaviour must be documentable precisely.
- Predictable. The number of calls to the RNG should be the same for any `\int_rand:nn`, because then the algorithm can be modified later without changing the result of other uses of the RNG.
- Robust. It should work even for `\int_rand:nn { - \c_max_int } { \c_max_int }` where the range is not representable as an integer. In fact, we also provide later a floating-point `randint` whose range can go all the way up to $2 \times 10^{16} - 1$ possible values.

Some of these requirements conflict. For instance, uniformity cannot be achieved with a fixed number of calls to the RNG.

Denote by $\text{random}(N)$ one call to `\tex_uniformdeviate:D` with argument N , and by $\text{ediv}(p, q)$ the $\varepsilon\text{-T}_{\text{E}}\text{X}$ rounding division giving $\lfloor p/q + 1/2 \rfloor$. Denote by $\langle \min \rangle$, $\langle \max \rangle$ and $R = \langle \max \rangle - \langle \min \rangle + 1$ the arguments of `\int_min:nn` and the number of possible outcomes. Note that $R \in [1, 2^{32} - 1]$ cannot necessarily be represented as an integer (however, $R - 2^{31}$ can). Our strategy is to get two 28-bit integers X and Y from the RNG, split each into 14-bit integers, as $X = X_1 \times 2^{14} + X_0$ and $Y = Y_1 \times 2^{14} + Y_0$ then return essentially $\langle \min \rangle + \lfloor R(X_1 \times 2^{-14} + Y_1 \times 2^{-28} + Y_0 \times 2^{-42} + X_0 \times 2^{-56}) \rfloor$. For small R the X_0 term has a tiny effect so we ignore it and we can compute $R \times Y/2^{28}$ much more directly by $\text{random}(R)$.

- If $R \leq 2^{17} - 1$ then return $\text{ediv}(R \text{random}(2^{14}) + \text{random}(R) + 2^{13}, 2^{14}) - 1 + \langle \min \rangle$. The shifts by 2^{13} and -1 convert $\varepsilon\text{-T}_{\text{E}}\text{X}$ division to truncated division. The bound on R ensures that the number obtained after the shift is less than `\c_max_int`. The non-uniformity is at most of order $2^{17}/2^{42} = 2^{-25}$.
- Split $R = R_2 \times 2^{28} + R_1 \times 2^{14} + R_0$, where $R_2 \in [0, 15]$. Compute $\langle \min \rangle + R_2 X_1 2^{14} + (R_2 Y_1 + R_1 X_1) + \text{ediv}(R_2 Y_0 + R_1 Y_1 + R_0 X_1 + \text{ediv}(R_2 X_0 + R_0 Y_1 + \text{ediv}((2^{14} R_1 + R_0)(2^{14} Y_0 + X_0), 2^{28}), 2^{14}), 2^{14})$ then map a result of $\langle \max \rangle + 1$ to $\langle \min \rangle$. Writing each ediv in terms of truncated division with a shift, and using $\lfloor (p + \lfloor r/s \rfloor)/q \rfloor = \lfloor (ps + r)/(sq) \rfloor$, what we compute is equal to $\lfloor \langle \text{exact} \rangle + 2^{-29} + 2^{-15} + 2^{-1} \rfloor$ with $\langle \text{exact} \rangle = \langle \min \rangle + R \times 0.X_1 Y_1 Y_0 X_0$. Given we map $\langle \max \rangle + 1$ to $\langle \min \rangle$, the shift has no effect on uniformity. The non-uniformity is bounded by $R/2^{56} < 2^{-24}$. It may be possible to speed up the code by dropping tiny terms such as $R_0 X_0$, but the analysis of non-uniformity proves too difficult.

To avoid the overflow when the computation yields $\langle \max \rangle + 1$ with $\langle \max \rangle = 2^{31} - 1$ (note that R is then arbitrary), we compute the result in two pieces. Compute $\langle \text{first} \rangle = \langle \min \rangle + R_2 X_1 2^{14}$ if $R_2 < 8$ or $\langle \min \rangle + 8 X_1 2^{14} + (R_2 - 8) X_1 2^{14}$ if $R_2 \geq 8$, the expressions being chosen to avoid overflow. Compute $\langle \text{second} \rangle = R_2 Y_1 + R_1 X_1 + \text{ediv}(\dots)$, at most $R_2 2^{14} + R_1 2^{14} + R_0 \leq 2^{28} + 15 \times 2^{14} - 1$, not at risk of overflowing. We have $\langle \text{first} \rangle + \langle \text{second} \rangle = \langle \max \rangle + 1 = \langle \min \rangle + R$ if and only if $\langle \text{second} \rangle = R 12^{14} + R_0 + R_2 2^{14}$ and $2^{14} R_2 X_1 = 2^{28} R_2 - 2^{14} R_2$ (namely $R_2 = 0$ or $X_1 = 2^{14} - 1$). In that case, return $\langle \min \rangle$, otherwise return $\langle \text{first} \rangle + \langle \text{second} \rangle$, which is safe because it is at most $\langle \max \rangle$. Note that the decision of what to return

does not need $\langle first \rangle$ explicitly so we don't actually compute it, just put it in an integer expression in which $\langle second \rangle$ is eventually added (or not).

- To get a floating point number in $[0, 1)$ just call the $R = 10000 \leq 2^{17} - 1$ procedure above to produce four blocks of four digits.
- To get an integer floating point number in a range (whose size can be up to $2 \times 10^{16} - 1$), work with fixed-point numbers: get six times four digits to build a fixed point number, multiply by R and add $\langle min \rangle$. This requires some care because l3fp-extended only supports non-negative numbers.

`\c__kernel_randint_max_int` Constant equal to $2^{17} - 1$, the maximal size of a range that `\int_range:nn` can do with its “simple” algorithm.

```
21831 \int_const:Nn \c__kernel_randint_max_int { 131071 }
```

(End definition for `\c__kernel_randint_max_int`.)

`__kernel_randint:n` Used in an integer expression, `__kernel_randint:n {R}` gives a random number $1 + \lfloor (R \text{random}(2^{14}) + \text{random}(R)) / 2^{14} \rfloor$ that is in $[1, R]$. Previous code was computing $\lfloor p / 2^{14} \rfloor$ as `ediv(p - 2^{13}, 2^{14})` but that wrongly gives -1 for $p = 0$.

```
21832 \cs_new:Npn \__kernel_randint:n #1
21833 {
21834   (#1 * \tex_uniformdeviate:D 16384
21835   + \tex_uniformdeviate:D #1 + 8192 ) / 16384
21836 }
```

(End definition for `__kernel_randint:n`.)

`__fp_rand_myriads:n` Used as `__fp_rand_myriads:n {XXX}` with one letter X (specifically) per block of four digit we want; it expands to `;` followed by the requested number of brace groups, each containing four (pseudo-random) digits. Digits are produced as a random number in $[10000, 19999]$ for the usual reason of preserving leading zeros.

```
21837 \cs_new:Npn \__fp_rand_myriads:n #1
21838 { \__fp_rand_myriads_loop:w #1 \prg_break: X \prg_break_point: ; }
21839 \cs_new:Npn \__fp_rand_myriads_loop:w #1 X
21840 {
21841   #1
21842   \exp_after:wN \__fp_rand_myriads_get:w
21843   \int_value:w \__fp_int_eval:w 9999 +
21844   \__kernel_randint:n { 10000 }
21845   \__fp_rand_myriads_loop:w
21846 }
21847 \cs_new:Npn \__fp_rand_myriads_get:w 1 #1 ; { ; {#1} }
```

(End definition for `__fp_rand_myriads:n`, `__fp_rand_myriads_loop:w`, and `__fp_rand_myriads_get:w`.)

36.2 Random floating point

`__fp_rand_o:Nw` First we check that `random` was called without argument. Then get four blocks of four digits and convert that fixed point number to a floating point number (this correctly sets the exponent). This has a minor bug: if all of the random numbers are zero then the result is correctly 0 but it raises the underflow flag; it should not do that.

`__fp_rand_o:w`

```

21848 \cs_new:Npn \__fp_rand_o:Nw ? #1 @
21849 {
21850   \tl_if_empty:nTF {#1}
21851   {
21852     \exp_after:wN \__fp_rand_o:w
21853     \exp:w \exp_end_continue_f:w
21854     \__fp_rand_myriads:n { XXXX } { 0000 } { 0000 } ; 0
21855   }
21856   {
21857     \__kernel_msg_expandable_error:nnnnn
21858     { kernel } { fp-num-args } { rand() } { 0 } { 0 }
21859     \exp_after:wN \c_nan_fp
21860   }
21861 }
21862 \cs_new:Npn \__fp_rand_o:w ;
21863 {
21864   \exp_after:wN \__fp_sanitize:Nw
21865   \exp_after:wN 0
21866   \int_value:w \__fp_int_eval:w \c_zero_int
21867   \__fp_fixed_to_float_o:wN
21868 }

```

(End definition for `__fp_rand_o:Nw` and `__fp_rand_o:w`.)

36.3 Random integer

`__fp_randint_o:Nw` Enforce that there is one argument (then add first argument 1) or two arguments. Call `__fp_randint_badarg:w` on each; this function inserts 1 `\exp_stop_f:` to end the `\if_case:w` statement if either the argument is not an integer or if its absolute value is $\geq 10^{16}$. Also bail out if `__fp_compare_back:ww` yields 1, meaning that the bounds are not in the right order. Otherwise an auxiliary converts each argument times 10^{-16} (hence the shift in exponent) to a 24-digit fixed point number (see `l3fp-extended`). Then compute the number of choices, $\langle max \rangle + 1 - \langle min \rangle$. Create a random 24-digit fixed-point number with `__fp_rand_myriads:n`, then use a fused multiply-add instruction to multiply the number of choices to that random number and add it to $\langle min \rangle$. Then truncate to 16 digits (namely select the integer part of 10^{16} times the result) before converting back to a floating point number (`__fp_sanitize:Nw` takes care of zero). To avoid issues with negative numbers, add 1 to all fixed point numbers (namely 10^{16} to the integers they represent), except of course when it is time to convert back to a float.

```

21869 \cs_new:Npn \__fp_randint_o:Nw ?
21870 {
21871   \__fp_parse_function_one_two:nnw
21872   { randint }
21873   { \__fp_randint_default:w \__fp_randint_o:w }
21874 }
21875 \cs_new:Npn \__fp_randint_default:w #1 { \exp_after:wN #1 \c_one_fp }
21876 \cs_new:Npn \__fp_randint_badarg:w \s__fp \__fp_chk:w #1#2#3;

```

```

21877 {
21878   \__fp_int:wTF \s__fp \__fp_chk:w #1#2#3;
21879   {
21880     \if_meaning:w 1 #1
21881     \if_int_compare:w
21882       \__fp_use_i_until_s:nw #3 ; > \c__fp_prec_int
21883       1 \exp_stop_f:
21884     \fi:
21885     \fi:
21886   }
21887   { 1 \exp_stop_f: }
21888 }
21889 \cs_new:Npn \__fp_randint_o:w #1; #2; @
21890 {
21891   \if_case:w
21892     \__fp_randint_badarg:w #1;
21893     \__fp_randint_badarg:w #2;
21894     \if:w 1 \__fp_compare_back:ww #2; #1; 1 \exp_stop_f: \fi:
21895     0 \exp_stop_f:
21896     \__fp_randint_auxi_o:ww #1; #2;
21897   \or:
21898     \__fp_invalid_operation_tl_o:ff
21899     { randint } { \__fp_array_to_clist:n { #1; #2; } }
21900   \exp:w
21901   \fi:
21902   \exp_after:wN \exp_end:
21903 }
21904 \cs_new:Npn \__fp_randint_auxi_o:ww #1 ; #2 ; #3 \exp_end:
21905 {
21906   \fi:
21907   \__fp_randint_auxii:wn #2 ;
21908   { \__fp_randint_auxii:wn #1 ; \__fp_randint_auxiii_o:ww }
21909 }
21910 \cs_new:Npn \__fp_randint_auxii:wn \s__fp \__fp_chk:w #1#2#3#4 ;
21911 {
21912   \if_meaning:w 0 #1
21913     \exp_after:wN \use_i:nn
21914   \else:
21915     \exp_after:wN \use_ii:nn
21916   \fi:
21917   { \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl }
21918   {
21919     \exp_after:wN \__fp_ep_to_fixed:wwn
21920     \int_value:w \__fp_int_eval:w
21921     #3 - \c__fp_prec_int , #4 {0000} {0000} ;
21922     {
21923       \if_meaning:w 0 #2
21924         \exp_after:wN \use_i:nnnn
21925         \exp_after:wN \__fp_fixed_add_one:wN
21926       \fi:
21927       \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl
21928     }
21929     \__fp_fixed_continue:wn
21930   }

```

```

21931     }
21932 \cs_new:Npn \__fp_randint_auxiii_o:ww #1 ; #2 ;
21933 {
21934     \__fp_fixed_add:wwn #2 ;
21935     {0000} {0000} {0000} {0001} {0000} {0000} ;
21936     \__fp_fixed_sub:wwn #1 ;
21937     {
21938         \exp_after:wN \use_i:nn
21939         \exp_after:wN \__fp_fixed_mul_add:wwwn
21940         \exp:w \exp_end_continue_f:w \__fp_rand_myriads:n { XXXXXX } ;
21941     }
21942     #1 ;
21943     \__fp_randint_auxiv_o:ww
21944     #2 ;
21945     \__fp_randint_auxv_o:w #1 ; @
21946 }
21947 \cs_new:Npn \__fp_randint_auxiv_o:ww #1#2#3#4#5 ; #6#7#8#9
21948 {
21949     \if_int_compare:w
21950         \if_int_compare:w #1#2 > #6#7 \exp_stop_f: 1 \else:
21951         \if_int_compare:w #1#2 < #6#7 \exp_stop_f: - \fi: \fi:
21952         #3#4 > #8#9 \exp_stop_f:
21953         \__fp_use_i_until_s:nw
21954         \fi:
21955         \__fp_randint_auxv_o:w {#1}{#2}{#3}{#4}#5
21956 }
21957 \cs_new:Npn \__fp_randint_auxv_o:w #1#2#3#4#5 ; #6 @
21958 {
21959     \exp_after:wN \__fp_sanitize:Nw
21960     \int_value:w
21961     \if_int_compare:w #1 < 10000 \exp_stop_f:
21962     2
21963     \else:
21964     0
21965     \exp_after:wN \exp_after:wN
21966     \exp_after:wN \__fp_reverse_args:Nww
21967     \fi:
21968     \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl
21969     {#1} {#2} {#3} {#4} {0000} {0000} ;
21970     {
21971         \exp_after:wN \exp_stop_f:
21972         \int_value:w \__fp_int_eval:w \c__fp_prec_int
21973         \__fp_fixed_to_float_o:wN
21974     }
21975     0
21976     \exp:w \exp_after:wN \exp_end:
21977 }

```

(End definition for __fp_randint_o:Nw and others.)

\int_rand:nn Evaluate the argument and filter out the case where the lower bound #1 is more than
__fp_randint:ww the upper bound #2. Then determine whether the range is narrower than **\c__kernel_-randint_max_int**; #2-#1 may overflow for very large positive #2 and negative #1. If the range is narrow, call **__kernel_randint:n {<choices>}** where <choices> is the number

of possible outcomes. If the range is wide, use somewhat slower code.

```

21978 \cs_new:Npn \int_rand:nn #1#2
21979 {
21980   \int_eval:n
21981   {
21982     \exp_after:wN \__fp_randint:ww
21983     \int_value:w \int_eval:n {#1} \exp_after:wN ;
21984     \int_value:w \int_eval:n {#2} ;
21985   }
21986 }
21987 \cs_new:Npn \__fp_randint:ww #1; #2;
21988 {
21989   \if_int_compare:w #1 > #2 \exp_stop_f:
21990   \__kernel_msg_expandable_error:nnnn
21991   { kernel } { randint-backward-range } {#1} {#2}
21992   \__fp_randint:ww #2; #1;
21993   \else:
21994     \if_int_compare:w \__fp_int_eval:w #2
21995     \if_int_compare:w #1 > \c_zero_int
21996     - #1 < \__fp_int_eval:w
21997     \else:
21998       < \__fp_int_eval:w #1 +
21999       \fi:
22000       \c_kernel_randint_max_int
22001       \__fp_int_eval_end:
22002       \__kernel_randint:n
22003       { \__fp_int_eval:w #2 - #1 + 1 \__fp_int_eval_end: }
22004       - 1 + #1
22005     \else:
22006       \__kernel_randint:nn {#1} {#2}
22007     \fi:
22008   \fi:
22009 }

```

(End definition for `\int_rand:nn` and `__fp_randint:ww`. This function is documented on page 100.)

Any $n \in [-2^{31} + 1, 2^{31} - 1]$ is uniquely written as $2^{14}n_1 + n_2$ with $n_1 \in [-2^{17}, 2^{17} - 1]$ and $n_2 \in [0, 2^{14} - 1]$. Calling `__fp_randint_split_o:Nw n` ; gives n_1 ; n_2 ; and expands the next token once. We do this for two random numbers and apply `__fp_randint_split_o:Nw` twice to fully decompose the range R . One subtlety is that we compute $R - 2^{31} = \langle \max \rangle - \langle \min \rangle - (2^{31} - 1) \in [-2^{31} + 1, 2^{31} - 1]$ rather than R to avoid overflow.

Then we have `__fp_randint_wide_aux:w` $\langle X_1 \rangle; \langle X_0 \rangle; \langle Y_1 \rangle; \langle Y_0 \rangle; \langle R_2 \rangle; \langle R_1 \rangle; \langle R_0 \rangle; .$ and we apply the algorithm described earlier.

```

22010 \cs_new:Npn \__kernel_randint:nn #1#2
22011 {
22012   #1
22013   \exp_after:wN \__fp_randint_wide_aux:w
22014   \int_value:w
22015   \exp_after:wN \__fp_randint_split_o:Nw
22016   \tex_uniformdeviate:D 268435456 ;
22017   \int_value:w
22018   \exp_after:wN \__fp_randint_split_o:Nw
22019   \tex_uniformdeviate:D 268435456 ;
22020   \int_value:w

```

```

22021         \exp_after:wN \__fp_randint_split_o:Nw
22022         \int_value:w \__fp_int_eval:w 131072 +
22023         \exp_after:wN \__fp_randint_split_o:Nw
22024         \int_value:w
22025         \__kernel_int_add:nnn {#2} { -#1 } { -\c_max_int } ;
22026     .
22027 }
22028 \cs_new:Npn \__fp_randint_split_o:Nw #1#2 ;
22029 {
22030     \if_meaning:w 0 #1
22031     0 \exp_after:wN ; \int_value:w 0
22032     \else:
22033         \exp_after:wN \__fp_randint_split_aux:w
22034         \int_value:w \__fp_int_eval:w (#1#2 - 8192) / 16384 ;
22035         + #1#2
22036     \fi:
22037     \exp_after:wN ;
22038 }
22039 \cs_new:Npn \__fp_randint_split_aux:w #1 ;
22040 {
22041     #1 \exp_after:wN ;
22042     \int_value:w \__fp_int_eval:w - #1 * 16384
22043 }
22044 \cs_new:Npn \__fp_randint_wide_aux:w #1;#2; #3;#4; #5;#6;#7; .
22045 {
22046     \exp_after:wN \__fp_randint_wide_auxii:w
22047     \int_value:w \__fp_int_eval:w #5 * #3 + #6 * #1 +
22048     (#5 * #4 + #6 * #3 + #7 * #1 +
22049     (#5 * #2 + #7 * #3 +
22050     (16384 * #6 + #7) * (16384 * #4 + #2) / 268435456) / 16384
22051     ) / 16384 \exp_after:wN ;
22052     \int_value:w \__fp_int_eval:w (#5 + #6) * 16384 + #7 ;
22053     #1 ; #5 ;
22054 }
22055 \cs_new:Npn \__fp_randint_wide_auxii:w #1; #2; #3; #4;
22056 {
22057     \if_int_odd:w 0
22058     \if_int_compare:w #1 = #2 \else: \exp_stop_f: \fi:
22059     \if_int_compare:w #4 = \c_zero_int 1 \fi:
22060     \if_int_compare:w #3 = 16383 ~ 1 \fi:
22061     \exp_stop_f:
22062     \exp_after:wN \prg_break:
22063     \fi:
22064     \if_int_compare:w #4 < 8 \exp_stop_f:
22065     + #4 * #3 * 16384
22066     \else:
22067     + 8 * #3 * 16384 + (#4 - 8) * #3 * 16384
22068     \fi:
22069     + #1
22070     \prg_break_point:
22071 }

```

(End definition for __kernel_randint:nn and others.)

\int_rand:n Similar to \int_rand:nn, but needs fewer checks.

```

\__fp_randint:n 22072 \cs_new:Npn \int_rand:n #1
22073 {
22074   \int_eval:n
22075   { \exp_args:Nf \__fp_randint:n { \int_eval:n {#1} } }
22076 }
22077 \cs_new:Npn \__fp_randint:n #1
22078 {
22079   \if_int_compare:w #1 < 1 \exp_stop_f:
22080   \__kernel_msg_expandable_error:nnnn
22081   { kernel } { randint-backward-range } { 1 } {#1}
22082   \__fp_randint:ww #1; 1;
22083   \else:
22084   \if_int_compare:w #1 > \c__kernel_randint_max_int
22085   \__kernel_randint:nn { 1 } {#1}
22086   \else:
22087   \__kernel_randint:n {#1}
22088   \fi:
22089   \fi:
22090 }

```

(End definition for \int_rand:n and __fp_randint:n. This function is documented on page 100.)

End the initial conditional that ensures these commands are only defined in engines that support random numbers.

```

22091 }
22092 \</package>

```

37 l3fparray implementation

```

22093 \*package>
22094 \@@=fp>

```

In analogy to l3intarray it would make sense to have <@@=fparray>, but we need direct access to __fp_parse:n from l3fp-parse, and a few other (less crucial) internals of the l3fp family.

37.1 Allocating arrays

There are somewhat more than $(2^{31} - 1)^2$ floating point numbers so we store each floating point number as three entries in integer arrays. To avoid having to multiply indices by three or to add 1 etc, a floating point array is just a token list consisting of three tokens: integer arrays of the same size.

\g__fp_array_int Used to generate unique names for the three integer arrays.

```

22095 \int_new:N \g__fp_array_int

```

(End definition for \g__fp_array_int.)

\l__fp_array_loop_int Used to loop in __fp_array_gzero:N.

```

22096 \int_new:N \l__fp_array_loop_int

```

(End definition for \l__fp_array_loop_int.)

\fpararray_new:Nn Build a three-token token list, then define all three tokens to be integer arrays of the same size. No need to initialize the data: the integer arrays start with zeros, and three zeros denote precisely `\c_zero_fp`, as we want.

\fpararray_new:cn

__fp_array_new:nNNN

```

22097 \cs_new_protected:Npn \fpararray_new:Nn #1#2
22098 {
22099   \tl_new:N #1
22100   \prg_replicate:nn { 3 }
22101   {
22102     \int_gincr:N \g__fp_array_int
22103     \exp_args:NNc \tl_gput_right:Nn #1
22104     { g__fp_array_ \__fp_int_to_roman:w \g__fp_array_int _intarray }
22105   }
22106   \exp_last_unbraced:Nfo \__fp_array_new:nNNNN
22107   { \int_eval:n {#2} } #1 #1
22108 }
22109 \cs_generate_variant:Nn \fpararray_new:Nn { c }
22110 \cs_new_protected:Npn \__fp_array_new:nNNNN #1#2#3#4#5
22111 {
22112   \int_compare:nNnTF {#1} < 0
22113   {
22114     \__kernel_msg_error:nnn { kernel } { negative-array-size } {#1}
22115     \cs_undefine:N #1
22116     \int_gsub:Nn \g__fp_array_int { 3 }
22117   }
22118   {
22119     \intarray_new:Nn #2 {#1}
22120     \intarray_new:Nn #3 {#1}
22121     \intarray_new:Nn #4 {#1}
22122   }
22123 }

```

(End definition for `\fpararray_new:Nn` and `__fp_array_new:nNNN`. This function is documented on page 222.)

\fpararray_count:N Size of any of the intarrays, here we pick the third.

\fpararray_count:c

```

22124 \cs_new:Npn \fpararray_count:N #1
22125 {
22126   \exp_after:wN \use_i:nnn
22127   \exp_after:wN \intarray_count:N #1
22128 }
22129 \cs_generate_variant:Nn \fpararray_count:N { c }

```

(End definition for `\fpararray_count:N`. This function is documented on page 222.)

37.2 Array items

__fp_array_bounds:NNnTF See the `l3intarray` analogue: only names change. The functions `\fpararray_gset:Nnn` and `\fpararray_item:Nn` share bounds checking. The T branch is used if #3 is within bounds of the array #2.

__fp_array_bounds_error:NNn

```

22130 \cs_new:Npn \__fp_array_bounds:NNnTF #1#2#3#4#5
22131 {
22132   \if_int_compare:w 1 > #3 \exp_stop_f:
22133   \__fp_array_bounds_error:NNn #1 #2 {#3}
22134   #5

```

```

22135     \else:
22136         \if_int_compare:w #3 > \fpararray_count:N #2 \exp_stop_f:
22137             \__fp_array_bounds_error:NNn #1 #2 {#3}
22138             #5
22139         \else:
22140             #4
22141         \fi:
22142     \fi:
22143 }
22144 \cs_new:Npn \__fp_array_bounds_error:NNn #1#2#3
22145 {
22146     #1 { kernel } { out-of-bounds }
22147     { \token_to_str:N #2 } {#3} { \fpararray_count:N #2 }
22148 }

```

(End definition for __fp_array_bounds:NNnTF and __fp_array_bounds_error:NNn.)

\fpararray_gset:Nnn

Evaluate, then store exponent in one intarray, sign and 8 digits of mantissa in the next, and 8 trailing digits in the last.

\fpararray_gset:cnn

__fp_array_gset:NNNNww
__fp_array_gset:w
__fp_array_gset_recover:Nw
__fp_array_gset_special:nnNNN
__fp_array_gset_normal:w

```

22149 \cs_new_protected:Npn \fpararray_gset:Nnn #1#2#3
22150 {
22151     \exp_after:wN \exp_after:wN
22152     \exp_after:wN \__fp_array_gset:NNNNww
22153     \exp_after:wN #1
22154     \exp_after:wN #1
22155     \int_value:w \int_eval:n {#2} \exp_after:wN ;
22156     \exp:w \exp_end_continue_f:w \__fp_parse:n {#3}
22157 }
22158 \cs_generate_variant:Nn \fpararray_gset:Nnn { c }
22159 \cs_new_protected:Npn \__fp_array_gset:NNNNww #1#2#3#4#5 ; #6 ;
22160 {
22161     \__fp_array_bounds:NNnTF \__kernel_msg_error:nnxxx #4 {#5}
22162     {
22163         \exp_after:wN \__fp_change_func_type:NNN
22164         \__fp_use_i_until_s:nw #6 ;
22165         \__fp_array_gset:w
22166         \__fp_array_gset_recover:Nw
22167         #6 ; {#5} #1 #2 #3
22168     }
22169     { }
22170 }
22171 \cs_new_protected:Npn \__fp_array_gset_recover:Nw #1#2 ;
22172 {
22173     \__fp_error:nffn { fp-unknown-type } { \tl_to_str:n { #2 ; } } { } { } { }
22174     \exp_after:wN #1 \c_nan_fp
22175 }
22176 \cs_new_protected:Npn \__fp_array_gset:w \s__fp \__fp_chk:w #1#2
22177 {
22178     \if_case:w #1 \exp_stop_f:
22179         \__fp_case_return:nw { \__fp_array_gset_special:nnNNN {#2} }
22180     \or: \exp_after:wN \__fp_array_gset_normal:w
22181     \or: \__fp_case_return:nw { \__fp_array_gset_special:nnNNN { #2 3 } }
22182     \or: \__fp_case_return:nw { \__fp_array_gset_special:nnNNN { 1 } }
22183     \fi:

```

```

22184     \s__fp \__fp_chk:w #1 #2
22185 }
22186 \cs_new_protected:Npn \__fp_array_gset_normal:w
22187 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5 ; #6#7#8#9
22188 {
22189     \__kernel_intarray_gset:Nnn #7 {#6} {#2}
22190     \__kernel_intarray_gset:Nnn #8 {#6}
22191     { \if_meaning:w 2 #1 3 \else: 1 \fi: #3#4 }
22192     \__kernel_intarray_gset:Nnn #9 {#6} { 1 \use:nn #5 }
22193 }
22194 \cs_new_protected:Npn \__fp_array_gset_special:nnNNN #1#2#3#4#5
22195 {
22196     \__kernel_intarray_gset:Nnn #3 {#2} {#1}
22197     \__kernel_intarray_gset:Nnn #4 {#2} {0}
22198     \__kernel_intarray_gset:Nnn #5 {#2} {0}
22199 }

```

(End definition for \fpararray_gset:Nnn and others. This function is documented on page 222.)

\fpararray_gzero:N

\fpararray_gzero:c

```

22200 \cs_new_protected:Npn \fpararray_gzero:N #1
22201 {
22202     \int_zero:N \l__fp_array_loop_int
22203     \prg_replicate:nn { \fpararray_count:N #1 }
22204     {
22205         \int_incr:N \l__fp_array_loop_int
22206         \exp_after:wN \__fp_array_gset_special:nnNNN
22207         \exp_after:wN 0
22208         \exp_after:wN \l__fp_array_loop_int
22209         #1
22210     }
22211 }
22212 \cs_generate_variant:Nn \fpararray_gzero:N { c }

```

(End definition for \fpararray_gzero:N. This function is documented on page 222.)

\fpararray_item:Nn

\fpararray_item:cn

\fpararray_item_to_tl:Nn

\fpararray_item_to_tl:cn

__fp_array_item:NwN

__fp_array_item:NNNnN

__fp_array_item:N

__fp_array_item:w

__fp_array_item_special:w

__fp_array_item_normal:w

```

22213 \cs_new:Npn \fpararray_item:Nn #1#2
22214 {
22215     \exp_after:wN \__fp_array_item:NwN
22216     \exp_after:wN #1
22217     \int_value:w \int_eval:n {#2} ;
22218     \__fp_to_decimal:w
22219 }
22220 \cs_generate_variant:Nn \fpararray_item:Nn { c }
22221 \cs_new:Npn \fpararray_item_to_tl:Nn #1#2
22222 {
22223     \exp_after:wN \__fp_array_item:NwN
22224     \exp_after:wN #1
22225     \int_value:w \int_eval:n {#2} ;
22226     \__fp_to_tl:w
22227 }
22228 \cs_generate_variant:Nn \fpararray_item_to_tl:Nn { c }
22229 \cs_new:Npn \__fp_array_item:NwN #1#2 ; #3
22230 {

```

```

22231 \__fp_array_bounds:NNnTF \__kernel_msg_expandable_error:nnfff #1 {#2}
22232 { \exp_after:wN \__fp_array_item:NNNnN #1 {#2} #3 }
22233 { \exp_after:wN #3 \c_nan_fp }
22234 }
22235 \cs_new:Npn \__fp_array_item:NNNnN #1#2#3#4
22236 {
22237   \exp_after:wN \__fp_array_item:N
22238   \int_value:w \__kernel_intarray_item:Nn #2 {#4} \exp_after:wN ;
22239   \int_value:w \__kernel_intarray_item:Nn #3 {#4} \exp_after:wN ;
22240   \int_value:w \__kernel_intarray_item:Nn #1 {#4} ;
22241 }
22242 \cs_new:Npn \__fp_array_item:N #1
22243 {
22244   \if_meaning:w 0 #1 \exp_after:wN \__fp_array_item_special:w \fi:
22245   \__fp_array_item:w #1
22246 }
22247 \cs_new:Npn \__fp_array_item:w #1 #2#3#4#5 #6 ; 1 #7 ;
22248 {
22249   \exp_after:wN \__fp_array_item_normal:w
22250   \int_value:w \if_meaning:w #1 1 0 \else: 2 \fi: \exp_stop_f:
22251   #7 ; {#2#3#4#5} {#6} ;
22252 }
22253 \cs_new:Npn \__fp_array_item_special:w #1 ; #2 ; #3 ; #4
22254 {
22255   \exp_after:wN #4
22256   \exp:w \exp_end_continue_f:w
22257   \if_case:w #3 \exp_stop_f:
22258     \exp_after:wN \c_zero_fp
22259   \or: \exp_after:wN \c_nan_fp
22260   \or: \exp_after:wN \c_minus_zero_fp
22261   \or: \exp_after:wN \c_inf_fp
22262   \else: \exp_after:wN \c_minus_inf_fp
22263   \fi:
22264 }
22265 \cs_new:Npn \__fp_array_item_normal:w #1 #2#3#4#5 #6 ; #7 ; #8 ; #9
22266 { #9 \s__fp \__fp_chk:w 1 #1 {#8} #7 {#2#3#4#5} {#6} ; }

```

(End definition for `\fparray_item:Nn` and others. These functions are documented on page 222.)

```

22267 \endpackage

```

38 l3cctab implementation

```

22268 \*package>
22269 \@@=cctab>

```

As LuaTeX offers engine support for category code tables, and this is entirely lacking from the other engines, we need two complementary approaches. (Some future XeTeX may add support, at which point the conditionals below would be different.)

38.1 Variables

`\g__cctab_stack_seq` List of catcode tables saved by nested `\cctab_begin:N`, to restore catcodes at the matching `\cctab_end:.` When popped from the `\g__cctab_stack_seq` the table numbers are stored in `\g__cctab_unused_seq` for later reuse.

```

22270 \seq_new:N \g__cctab_stack_seq
22271 \seq_new:N \g__cctab_unused_seq

```

(End definition for `\g__cctab_stack_seq` and `\g__cctab_unused_seq`.)

`\g__cctab_group_seq` A stack to store the group level when a catcode table started.

```

22272 \seq_new:N \g__cctab_group_seq

```

(End definition for `\g__cctab_group_seq`.)

`\g__cctab_allocate_int` Integer to keep track of what category code table to allocate. In LuaTeX it is only used in format mode to implement `\cctab_new:N`. In other engines it is used to make csnames for dynamic tables.

```

22273 \int_new:N \g__cctab_allocate_int

```

(End definition for `\g__cctab_allocate_int`.)

`\l__cctab_internal_a_tl` Scratch space. For instance, when popping `\g__cctab_stack_seq`/`\g__cctab_unused_seq`, consists of the catcodetable number (integer denotation) in LuaTeX, or of an intarray variable (as a single token) in other engines.

```

22274 \tl_new:N \l__cctab_internal_a_tl

```

```

22275 \tl_new:N \l__cctab_internal_b_tl

```

(End definition for `\l__cctab_internal_a_tl` and `\l__cctab_internal_b_tl`.)

`\g__cctab_endlinechar_prop` In LuaTeX we store the `\endlinechar` associated to each `\catcodetable` in a property list, unless it is the default value 13.

```

22276 \prop_new:N \g__cctab_endlinechar_prop

```

(End definition for `\g__cctab_endlinechar_prop`.)

38.2 Allocating category code tables

`\cctab_new:N` The `__cctab_new:N` auxiliary allocates a new catcode table but does not attempt to set its value consistently across engines. It is used both in `\cctab_new:N`, which sets catcodes to iniTeX values, and in `\cctab_begin:N`/`\cctab_end:` for dynamically allocated tables.

`\cctab_new:c`

`__cctab_new:N` First, the LuaTeX case. Creating a new category code table is done like other registers. In ConTeXt, `\newcatcodetable` does not include the initialisation, so that is added explicitly.

`__cctab_gstore:Nnn`

```

22277 \sys_if_engine luatex:TF
22278 {
22279   \cs_new_protected:Npn \cctab_new:N #1
22280   {
22281     \__kernel_chk_if_free_cs:N #1
22282     \__cctab_new:N #1
22283   }
22284   \cs_new_protected:Npn \__cctab_new:N #1
22285   {
22286     \newcatcodetable #1
22287     \tex_initcatcodetable:D #1
22288   }
22289 }

```


Now the case for other engines. Here, each table is an integer array. Following the LuaTeX pattern, a new table starts with `initEX` codes. The index base is out-by-one, so we have an internal function to handle that. The `initEX \endlinechar` is 13.

```

22290 {
22291   \cs_new_protected:Npn \__cctab_new:N #1
22292     { \intarray_new:Nn #1 { 257 } }
22293   \cs_new_protected:Npn \__cctab_gstore:Nnn #1#2#3
22294     { \intarray_gset:Nnn #1 { \int_eval:n { #2 + 1 } } {#3} }
22295   \cs_new_protected:Npn \cctab_new:N #1
22296     {
22297       \__kernel_chk_if_free_cs:N #1
22298       \__cctab_new:N #1
22299       \int_step_inline:nn { 256 }
22300         { \__kernel_intarray_gset:Nnn #1 {##1} { 12 } }
22301       \__kernel_intarray_gset:Nnn #1 { 257 } { 13 }
22302       \__cctab_gstore:Nnn #1 { 0 } { 9 }
22303       \__cctab_gstore:Nnn #1 { 13 } { 5 }
22304       \__cctab_gstore:Nnn #1 { 32 } { 10 }
22305       \__cctab_gstore:Nnn #1 { 37 } { 14 }
22306       \int_step_inline:nnn { 65 } { 90 }
22307         { \__cctab_gstore:Nnn #1 {##1} { 11 } }
22308       \__cctab_gstore:Nnn #1 { 92 } { 0 }
22309       \int_step_inline:nnn { 97 } { 122 }
22310         { \__cctab_gstore:Nnn #1 {##1} { 11 } }
22311       \__cctab_gstore:Nnn #1 { 127 } { 15 }
22312     }
22313   }
22314   \cs_generate_variant:Nn \cctab_new:N { c }

```

(End definition for `\cctab_new:N`, `__cctab_new:N`, and `__cctab_gstore:Nnn`. This function is documented on page 223.)

38.3 Saving category code tables

`__cctab_gset:n`
`__cctab_gset_aux:n`

In various functions we need to save the current catcodes (globally) in a table. In LuaTeX, saving the catcodes is a primitives, but the `\endlinechar` needs more work: to avoid filling `\g__cctab_endlinechar_prop` with many entries we special-case the default value 13. In other engines we store 256 current catcodes and the `\endlinechar` in an intarray variable.

```

22315 \sys_if_engine luatex:TF
22316 {
22317   \cs_new_protected:Npn \__cctab_gset:n #1
22318     { \exp_args:Nf \__cctab_gset_aux:n { \int_eval:n {#1} } }
22319   \cs_new_protected:Npn \__cctab_gset_aux:n #1
22320     {
22321       \tex_savecatcodetable:D #1 \scan_stop:
22322       \int_compare:nNnTF { \tex_endlinechar:D } = { 13 }
22323         { \prop_gremove:Nn \g__cctab_endlinechar_prop {#1} }
22324         {
22325           \prop_gput:NnV \g__cctab_endlinechar_prop {#1}
22326             \tex_endlinechar:D
22327         }
22328     }

```

```

22329 }
22330 {
22331   \cs_new_protected:Npn \__cctab_gset:n #1
22332   {
22333     \int_step_inline:nn { 256 }
22334     {
22335       \__kernel_intarray_gset:Nnn #1 {##1}
22336       { \char_value_catcode:n { ##1 - 1 } }
22337     }
22338     \__kernel_intarray_gset:Nnn #1 { 257 }
22339     { \tex_endlinechar:D }
22340   }
22341 }

```

(End definition for `__cctab_gset:n` and `__cctab_gset_aux:n`.)

`\cctab_gset:Nn` Category code tables are always global, so only one version of assignments is needed.
`\cctab_gset:cn` Simply run the setup in a group and save the result in a category code table #1, provided it is valid. The internal function is defined above depending on the engine.

```

22342 \cs_new_protected:Npn \cctab_gset:Nn #1#2
22343 {
22344   \__cctab_chk_if_valid:NT #1
22345   {
22346     \group_begin:
22347     \cctab_select:N \c_initex_cctab
22348     #2 \scan_stop:
22349     \__cctab_gset:n {#1}
22350     \group_end:
22351   }
22352 }
22353 \cs_generate_variant:Nn \cctab_gset:Nn { c }

```

(End definition for `\cctab_gset:Nn`. This function is documented on page 223.)

38.4 Using category code tables

`\g__cctab_internal_cctab` In LuaTeX, we must ensure that the saved tables are read-only. This is done by applying the saved table, then switching immediately to a scratch table. Any later catcode assignment will affect that scratch table rather than the saved one. If we simply switched to the saved tables, then `\char_set_catcode_other:N` in the example below would change `\c_document_cctab` and a later use of that table would give the wrong category code to `_`.

```

\use:n
{
  \cctab_begin:N \c_document_cctab
  \char_set_catcode_other:N \_
  \cctab_end:
  \cctab_begin:N \c_document_cctab
  \int_compare:nTF { \char_value_catcode:n { '_' } = 8 }
  { \TRUE } { \ERROR }
  \cctab_end:
}

```

We must also make sure that a scratch table is never reused in a nested group: in the following example, the scratch table used by the first `\cctab_begin:N` would be changed globally by the second one issuing `\savecatcodetable`, and after `\group_end:` the wrong category codes (those of `\c_str_cctab`) would be imposed. Note that the inner `\cctab_end:` restores the correct catcodes only locally, so the problem really comes up because of the different grouping level. The simplest is to use a scratch table labeled by the `\currentgrouplevel`. We initialize one of them as an example.

```
\use:n
{
  \cctab_begin:N \c_document_cctab
  \group_begin:
    \cctab_begin:N \c_str_cctab
    \cctab_end:
  \group_end:
  \cctab_end:
}

22354 \sys_if_engine luatex:T
22355 {
22356   \__cctab_new:N \g__cctab_internal_cctab
22357   \cs_new:Npn \__cctab_internal_cctab_name:
22358     {
22359       g__cctab_internal
22360       \tex_romannumeral:D \tex_currentgrouplevel:D
22361       _cctab
22362     }
22363 }
```

(End definition for `\g__cctab_internal_cctab` and `__cctab_internal_cctab_name:.`)

`\cctab_select:N` The public function simply checks the `\cctab var` exists before using the engine-dependent `__cctab_select:N`. Skipping these checks would result in low-level engine-dependent errors. First, the LuaTeX case. In other engines, selecting a catcode table is a matter of doing 256 catcode assignments and setting the `\endlinechar`.

```
22364 \cs_new_protected:Npn \cctab_select:N #1
22365   { \__cctab_chk_if_valid:NT #1 { \__cctab_select:N #1 } }
22366 \cs_generate_variant:Nn \cctab_select:N { c }
22367 \sys_if_engine luatex:TF
22368 {
22369   \cs_new_protected:Npn \__cctab_select:N #1
22370     {
22371       \tex_catcodetable:D #1
22372       \prop_get:NVNTF \g__cctab_endlinechar_prop #1 \l__cctab_internal_a_tl
22373       { \int_set:Nn \tex_endlinechar:D { \l__cctab_internal_a_tl } }
22374       { \int_set:Nn \tex_endlinechar:D { 13 } }
22375       \cs_if_exist:cF { \__cctab_internal_cctab_name: }
22376       { \exp_args:Nc \__cctab_new:N { \__cctab_internal_cctab_name: } }
22377       \exp_args:Nc \tex_savecatcodetable:D { \__cctab_internal_cctab_name: }
22378       \exp_args:Nc \tex_catcodetable:D { \__cctab_internal_cctab_name: }
22379     }
22380 }
22381 {
```

```

22382 \cs_new_protected:Npn \__cctab_select:N #1
22383 {
22384   \int_step_inline:nn { 256 }
22385   {
22386     \char_set_catcode:nn { ##1 - 1 }
22387     { \__kernel_intarray_item:Nn #1 {##1} }
22388   }
22389   \int_set:Nn \tex_endlinechar:D
22390   { \__kernel_intarray_item:Nn #1 { 257 } }
22391 }
22392 }

```

(End definition for `\cctab_select:N` and `__cctab_select:N`. This function is documented on page 223.)

`\g__cctab_next_cctab` For `\cctab_begin:N/\cctab_end:` we will need to allocate dynamic tables. This is done here by `__cctab_begin_aux:`, which puts a table number (in LuaTeX) or name (in other engines) into `\l__cctab_internal_a_tl`. In LuaTeX this simply calls `__cctab_new:N` and uses the resulting catcodetable number; in other engines we need to give a name to the intarray variable and use that. In LuaTeX, to restore catcodes at `\cctab_end:` we cannot just set `\catcodetable` to its value before `\cctab_begin:N`, because that table may have been altered by other code in the mean time. So we must make sure to save the catcodes in a table we control and restore them at `\cctab_end:`.

```

22393 \sys_if_engine luatex:TF
22394 {
22395   \cs_new_protected:Npn \__cctab_begin_aux:
22396   {
22397     \__cctab_new:N \g__cctab_next_cctab
22398     \tl_set:NV \l__cctab_internal_a_tl \g__cctab_next_cctab
22399     \cs_undefine:N \g__cctab_next_cctab
22400   }
22401 }
22402 {
22403   \cs_new_protected:Npn \__cctab_begin_aux:
22404   {
22405     \int_gincr:N \g__cctab_allocate_int
22406     \exp_args:Nc \__cctab_new:N
22407     { g__cctab_ \int_use:N \g__cctab_allocate_int _cctab }
22408     \exp_args:NNc \tl_set:Nn \l__cctab_internal_a_tl
22409     { g__cctab_ \int_use:N \g__cctab_allocate_int _cctab }
22410   }
22411 }

```

(End definition for `\g__cctab_next_cctab` and `__cctab_begin_aux:`.)

`\cctab_begin:N` Check the `\cctab var` exists, to avoid low-level errors. Get in `\l__cctab_internal_a_tl` the number/name of a dynamic table, either from `\g__cctab_unused_seq` where we save tables that are not currently in use, or from `__cctab_begin_aux:` if none are available. Then save the current catcodes into the table (pointed to by) `\l__cctab_internal_a_tl` and save that table number in a stack before selecting the desired catcodes.

```

22412 \cs_new_protected:Npn \cctab_begin:N #1
22413 {
22414   \__cctab_chk_if_valid:NT #1
22415   {

```

```

22416 \seq_gpop:N NF \g__cctab_unused_seq \l__cctab_internal_a_tl
22417 { \__cctab_begin_aux: }
22418 \exp_args:Nx \__cctab_chk_group_begin:n
22419 { \__cctab_nesting_number:N \l__cctab_internal_a_tl }
22420 \seq_gpush:NV \g__cctab_stack_seq \l__cctab_internal_a_tl
22421 \exp_args:NV \__cctab_gset:n \l__cctab_internal_a_tl
22422 \__cctab_select:N #1
22423 }
22424 }
22425 \cs_generate_variant:Nn \cctab_begin:N { c }

```

(End definition for `\cctab_begin:N`. This function is documented on page 223.)

`\cctab_end:` Make sure a `\cctab_begin:N` was used some time earlier, get in `\l__cctab_internal_a_tl` the catcode table number/name in which the prevailing catcodes were stored, then restore these catcodes. The dynamic table is now unused hence stored in `\g__cctab_unused_seq` for recycling by later `\cctab_begin:N`.

```

22426 \cs_new_protected:Npn \cctab_end:
22427 {
22428   \seq_gpop:NNTF \g__cctab_stack_seq \l__cctab_internal_a_tl
22429   {
22430     \seq_gpush:NV \g__cctab_unused_seq \l__cctab_internal_a_tl
22431     \exp_args:Nx \__cctab_chk_group_end:n
22432     { \__cctab_nesting_number:N \l__cctab_internal_a_tl }
22433     \__cctab_select:N \l__cctab_internal_a_tl
22434   }
22435   { \__kernel_msg_error:nn { kernel } { cctab-extra-end } }
22436 }

```

(End definition for `\cctab_end:`. This function is documented on page 223.)

`__cctab_chk_group_begin:n` `__cctab_chk_group_end:n` Catcode tables are not allowed to be intermixed with groups, so here we check that they are properly nested regarding \TeX groups. `__cctab_chk_group_begin:n` stores the current group level in a stack, and locally defines a dummy control sequence `__cctab_group_⟨cctab-level⟩_chk:`.

`__cctab_chk_group_end:n` pops the stack, and compares the returned value with `\tex_currentgrouplevel:D`. If they differ, `\cctab_end:` is in a different grouping level than the matching `\cctab_begin:N`. If they are the same, both happened at the same level, however a group might have ended and another started between `\cctab_begin:N` and `\cctab_end:`:

```

\group_begin:
  \cctab_begin:N \c_document_cctab
\group_end:
\group_begin:
  \cctab_end:
\group_end:

```

In this case checking `\tex_currentgrouplevel:D` is not enough, so we locally define `__cctab_group_⟨cctab-level⟩_chk:`, and then check if it exist in `\cctab_end:`. If it doesn't, we know there was a group end where it shouldn't.

The `⟨cctab-level⟩` in the sentinel macro above cannot be replaced by the more convenient `\tex_currentgrouplevel:D` because with the latter we might be tricked. Suppose:

```

\group_begin:
  \cctab_begin:N \c_code_cctab % A
\group_end:
\group_begin:
  \cctab_begin:N \c_code_cctab % B
  \cctab_end: % C
  \cctab_end: % D
\group_end:

```

The line marked with A would start a `cctab` with a sentinel token named `__cctab_group_1_chk:`, which would disappear at the `\group_end:` that follows. But B would create the same sentinel token, since both are at the same group level. Line C would end the `cctab` from line B correctly, but so would line D because line B created the same sentinel token. Using `\cctab_level` works correctly because it signals that certain `cctab` level was activated somewhere, but if it doesn't exist when the `\cctab_end:` is reached, we had a problem.

Unfortunately these tests only flag the wrong usage at the `\cctab_end:`, which might be far from the `\cctab_begin:N`. However it isn't possible to signal the wrong usage at the `\group_end:` without using `\tex_aftergroup:D`, which is unsafe in certain types of groups.

The three cases checked here just raise an error, and no recovery is attempted: usually interleaving groups and catcode tables will work predictably.

```

22437 \cs_new_protected:Npn \__cctab_chk_group_begin:n #1
22438 {
22439   \seq_gpush:Nx \g__cctab_group_seq
22440   { \int_use:N \tex_currentgrouplevel:D }
22441   \cs_set_eq:cN { __cctab_group_ #1 _chk: } \prg_do_nothing:
22442 }
22443 \cs_new_protected:Npn \__cctab_chk_group_end:n #1
22444 {
22445   \seq_gpop:NN \g__cctab_group_seq \l__cctab_internal_b_tl
22446   \bool_lazy_and:nnF
22447   {
22448     \int_compare_p:nNn
22449     { \tex_currentgrouplevel:D } = { \l__cctab_internal_b_tl }
22450   }
22451   { \cs_if_exist_p:c { __cctab_group_ #1 _chk: } }
22452   {
22453     \__kernel_msg_error:nxx { kernel } { cctab-group-mismatch }
22454     {
22455       \int_sign:n
22456       { \tex_currentgrouplevel:D - \l__cctab_internal_b_tl }
22457     }
22458   }
22459   \cs_undefine:c { __cctab_group_ #1 _chk: }
22460 }

```

(End definition for `__cctab_chk_group_begin:n` and `__cctab_chk_group_end:n`.)

`__cctab_nesting_number:N`
`__cctab_nesting_number:w`

This macro returns the numeric index of the current catcode table. In LuaTeX this is just the argument, which is a count reference to a `\catcodetable` register. In other engines, the number is extracted from the `cctab` variable.

```

22461 \sys_if_engine luatex:TF
22462 { \cs_new:Npn \__cctab_nesting_number:N #1 {#1} }
22463 {
22464   \cs_new:Npn \__cctab_nesting_number:N #1
22465   {
22466     \exp_after:wN \exp_after:wN \exp_after:wN \__cctab_nesting_number:w
22467     \exp_after:wN \token_to_str:N #1
22468   }
22469   \use:x
22470   {
22471     \cs_new:Npn \exp_not:N \__cctab_nesting_number:w
22472     ##1 \tl_to_str:n { g__cctab_ } ##2 \tl_to_str:n { _cctab } {##2}
22473   }
22474 }

```

(End definition for `__cctab_nesting_number:N` and `__cctab_nesting_number:w`.)

Finally, install some code at the end of the \TeX run to check that all `\cctab_begin:N` were ended by some `\cctab_end:`.

```

22475 \cs_if_exist:NT \hook_gput_code:nnn
22476 {
22477   \hook_gput_code:nnn { enddocument/end } { kernel }
22478   {
22479     \seq_if_empty:NF \g__cctab_stack_seq
22480     { \__kernel_msg_error:nn { kernel } { cctab-missing-end } }
22481   }
22482 }

```

38.5 Category code table conditionals

`\cctab_if_exist:N` Checks whether a $\langle cctab\ var \rangle$ is defined.

```

\cctab_if_exist:c
22483 \prg_new_eq_conditional:NNn \cctab_if_exist:N \cs_if_exist:N
22484 { TF , T , F , p }
22485 \prg_new_eq_conditional:NNn \cctab_if_exist:c \cs_if_exist:c
22486 { TF , T , F , p }

```

(End definition for `\cctab_if_exist:N`. This function is documented on page ??.)

`__cctab_chk_if_valid:NTF` Checks whether the argument is defined and whether it is a valid $\langle cctab\ var \rangle$. In Lua \TeX the validity of the $\langle cctab\ var \rangle$ is checked by the engine, which complains if the argument is not a `\chardef`'ed constant. In other engines, check if the given command is an intarray variable (the underlying definition is a copy of the `cmr10` font).

`__cctab_chk_if_valid_aux:NTF`

```

22487 \prg_new_protected_conditional:Npnn \__cctab_chk_if_valid:N #1
22488 { TF , T , F }
22489 {
22490   \cctab_if_exist:NTF #1
22491   {
22492     \__cctab_chk_if_valid_aux:NTF #1
22493     { \prg_return_true: }
22494     {
22495       \__kernel_msg_error:nnx { kernel } { invalid-cctab }
22496       { \token_to_str:N #1 }
22497       \prg_return_false:
22498     }
22499   }

```

```

22499     }
22500     {
22501         \__kernel_msg_error:nxx { kernel } { command-not-defined }
22502         { \token_to_str:N #1 }
22503         \prg_return_false:
22504     }
22505 }
22506 \sys_if_engine luatex:TF
22507 {
22508     \cs_new_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
22509     {
22510         \int_compare:nNnTF {#1-1} < { \e@alloc@ccodetable@count }
22511     }
22512     \cs_if_exist:NT \c_syst_catcodes_n
22513     {
22514         \cs_gset_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
22515         {
22516             \int_compare:nTF { #1 <= \c_syst_catcodes_n }
22517         }
22518     }
22519 }
22520 {
22521     \cs_new_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
22522     {
22523         \exp_args:Nf \str_if_in:nnTF
22524         { \cs_meaning:N #1 }
22525         { select~font~cmr10~at~ }
22526     }
22527 }

```

(End definition for __cctab_chk_if_valid:NTF and __cctab_chk_if_valid_aux:NTF.)

38.6 Constant category code tables

\cctab_const:Nn Creates a new *cctab var* then sets it with the current and user-supplied codes.

```

\cctab_const:cn
22528 \cs_new_protected:Npn \cctab_const:Nn #1#2
22529 {
22530     \cctab_new:N #1
22531     \cctab_gset:Nn #1 {#2}
22532 }
22533 \cs_generate_variant:Nn \cctab_const:Nn { c }

```

(End definition for \cctab_const:Nn. This function is documented on page 223.)

\c_initex_cctab Creating category code tables means thinking starting from iniT_EX. For all-other and
\c_other_cctab the standard “string” tables that’s easy.

```

\c_str_cctab
22534 \cctab_new:N \c_initex_cctab
22535 \cctab_const:Nn \c_other_cctab
22536 {
22537     \cctab_select:N \c_initex_cctab
22538     \int_set:Nn \tex_endlinechar:D { -1 }
22539     \int_step_inline:nnn { 0 } { 127 }
22540     { \char_set_catcode_other:n {#1} }
22541 }

```



```

22542 \cctab_const:Nn \c_str_cctab
22543 {
22544   \cctab_select:N \c_other_cctab
22545   \char_set_catcode_space:n { 32 }
22546 }

```

(End definition for `\c_initex_cctab`, `\c_other_cctab`, and `\c_str_cctab`. These variables are documented on page 224.)

`\c_document_cctab` To pick up document-level category codes, we need to delay set up to the end of the
`\c_other_cctab` format, where that’s possible. Also, as there are a *lot* of category codes to set, we avoid using the official interface and store the document codes using internal code. Depending on whether we are in the hook or not, the catcodes may be code or document, so we explicitly set up both correctly.

```

22547 \cs_if_exist:NTF \@expl@finalise@setup@@
22548 { \tl_gput_right:Nn \@expl@finalise@setup@@ }
22549 { \use:n }
22550 {
22551   \__cctab_new:N \c_code_cctab
22552   \group_begin:
22553     \int_set:Nn \tex_endlinechar:D { 32 }
22554     \char_set_catcode_invalid:n { 0 }
22555     \bool_lazy_or:nnTF
22556       { \sys_if_engine_xetex_p: } { \sys_if_engine_luatex_p: }
22557       { \int_step_function:nn { 31 } { 64 } \char_set_catcode_invalid:n }
22558       { \int_step_function:nn { 31 } { 96 } \char_set_catcode_active:n }
22559     \int_step_function:nnN { 65 } { 90 } \char_set_catcode_letter:n
22560     \int_step_function:nnN { 91 } { 96 } \char_set_catcode_other:n
22561     \int_step_function:nnN { 97 } { 122 } \char_set_catcode_letter:n
22562     \char_set_catcode_ignore:n { 9 } % tab
22563     \char_set_catcode_other:n { 10 } % lf
22564     \char_set_catcode_active:n { 12 } % ff
22565     \char_set_catcode_end_line:n { 13 } % cr
22566     \char_set_catcode_ignore:n { 32 } % space
22567     \char_set_catcode_parameter:n { 35 } % hash
22568     \char_set_catcode_math_toggle:n { 36 } % dollar
22569     \char_set_catcode_comment:n { 37 } % percent
22570     \char_set_catcode_alignment:n { 38 } % ampersand
22571     \char_set_catcode_letter:n { 58 } % colon
22572     \char_set_catcode_escape:n { 92 } % backslash
22573     \char_set_catcode_math_superscript:n { 94 } % circumflex
22574     \char_set_catcode_letter:n { 95 } % underscore
22575     \char_set_catcode_group_begin:n { 123 } % left brace
22576     \char_set_catcode_other:n { 124 } % pipe
22577     \char_set_catcode_group_end:n { 125 } % right brace
22578     \char_set_catcode_space:n { 126 } % tilde
22579     \char_set_catcode_invalid:n { 127 } % ^^?
22580     \__cctab_gset:n { \c_code_cctab }
22581   \group_end:
22582   \cctab_const:Nn \c_document_cctab
22583   {
22584     \cctab_select:N \c_code_cctab
22585     \int_set:Nn \tex_endlinechar:D { 13 }
22586

```

```

22587         \char_set_catcode_space:n          { 9 }
22588         \char_set_catcode_space:n          { 32 }
22589         \char_set_catcode_other:n           { 58 }
22590         \char_set_catcode_math_subscript:n { 95 }
22591         \char_set_catcode_active:n          { 126 }
22592     }
22593 }

```

(End definition for `\c_document_cctab` and `\c_other_cctab`. These variables are documented on page 224.)

38.7 Messages

```

22594 \__kernel_msg_new:nnnn { kernel } { cctab-stack-full }
22595 { The~category~code~table~stack~is~exhausted. }
22596 {
22597     LaTeX~has~been~asked~to~switch~to~a~new~category~code~table,~
22598     but~there~is~no~more~space~to~do~this!
22599 }
22600 \__kernel_msg_new:nnnn { kernel } { cctab-extra-end }
22601 { Extra~\iow_char:N\\cctab_end:~ignored~\msg_line_context:. }
22602 {
22603     LaTeX~came~across~a~\iow_char:N\\cctab_end:~without~a~matching~
22604     \iow_char:N\\cctab_begin:N.~This~command~will~be~ignored.
22605 }
22606 \__kernel_msg_new:nnnn { kernel } { cctab-missing-end }
22607 { Missing~\iow_char:N\\cctab_end:~before~end~of~TeX~run. }
22608 {
22609     LaTeX~came~across~more~\iow_char:N\\cctab_begin:N~than~
22610     \iow_char:N\\cctab_end:.
22611 }
22612 \__kernel_msg_new:nnnn { kernel } { invalid-cctab }
22613 { Invalid~\iow_char:N\\catcode~table. }
22614 {
22615     You~can~only~switch~to~a~\iow_char:N\\catcode~table~that~is~
22616     initialized~using~\iow_char:N\\cctab_new:N~or~
22617     \iow_char:N\\cctab_const:Nn.
22618 }
22619 \__kernel_msg_new:nnnn { kernel } { cctab-group-mismatch }
22620 {
22621     \iow_char:N\\cctab_end:~occurred~in~a~
22622     \int_case:nm {#1}
22623     {
22624         { 0 } { different~group }
22625         { 1 } { higher~group~level }
22626         { -1 } { lower~group~level }
22627     } ~than~
22628     the~matching~\iow_char:N\\cctab_begin:N.
22629 }
22630 {
22631     Catcode~tables~and~groups~must~be~properly~nested,~but~
22632     you~tried~to~interleave~them.~LaTeX~will~try~to~proceed,~
22633     but~results~may~be~unexpected.
22634 }

```

22635 `\package`

39 l3sort implementation

22636 `*package`

22637 `\@@=sort`

39.1 Variables

`\g__sort_internal_seq`
`\g__sort_internal_tl` Sorting happens in a group; the result is stored in those global variables before being copied outside the group to the proper places. For `seq` and `tl` this is more efficient than using `\use:x` (or some `\exp_args:NNNx`) to smuggle the definition outside the group since \TeX does not need to re-read tokens. For `clist` we don't gain anything since the result is converted from `seq` to `clist` anyways.

22638 `\seq_new:N \g__sort_internal_seq`

22639 `\tl_new:N \g__sort_internal_tl`

(End definition for `\g__sort_internal_seq` and `\g__sort_internal_tl`.)

`\l__sort_length_int`
`\l__sort_min_int`
`\l__sort_top_int`
`\l__sort_max_int`
`\l__sort_true_max_int` The sequence has `\l__sort_length_int` items and is stored from `\l__sort_min_int` to `\l__sort_top_int - 1`. While reading the sequence in memory, we check that `\l__sort_top_int` remains at most `\l__sort_max_int`, precomputed by `__sort-compute_range:.` That bound is such that the merge sort only uses `\toks` registers less than `\l__sort_true_max_int`, namely those that have not been allocated for use in other code: the user's comparison code could alter these.

22640 `\int_new:N \l__sort_length_int`

22641 `\int_new:N \l__sort_min_int`

22642 `\int_new:N \l__sort_top_int`

22643 `\int_new:N \l__sort_max_int`

22644 `\int_new:N \l__sort_true_max_int`

(End definition for `\l__sort_length_int` and others.)

`\l__sort_block_int` Merge sort is done in several passes. In each pass, blocks of size `\l__sort_block_int` are merged in pairs. The block size starts at 1, and, for a length in the range $[2^k + 1, 2^{k+1}]$, reaches 2^k in the last pass.

22645 `\int_new:N \l__sort_block_int`

(End definition for `\l__sort_block_int`.)

`\l__sort_begin_int`
`\l__sort_end_int` When merging two blocks, `\l__sort_begin_int` marks the lowest index in the two blocks, and `\l__sort_end_int` marks the highest index, plus 1.

22646 `\int_new:N \l__sort_begin_int`

22647 `\int_new:N \l__sort_end_int`

(End definition for `\l__sort_begin_int` and `\l__sort_end_int`.)

`\l__sort_A_int`
`\l__sort_B_int`
`\l__sort_C_int` When merging two blocks (whose end-points are `beg` and `end`), `A` starts from the high end of the low block, and decreases until reaching `beg`. The index `B` starts from the top of the range and marks the register in which a sorted item should be put. Finally, `C` points to the copy of the high block in the interval of registers starting at `\l__sort_length_int`, upwards. `C` starts from the upper limit of that range.

22648 `\int_new:N \l__sort_A_int`

22649 `\int_new:N \l__sort_B_int`

22650 `\int_new:N \l__sort_C_int`

(End definition for `\l__sort_A_int`, `\l__sort_B_int`, and `\l__sort_C_int`.)

```
\s__sort_mark Internal scan marks.
\s__sort_stop 22651 \scan_new:N \s__sort_mark
               22652 \scan_new:N \s__sort_stop
```

(End definition for `\s__sort_mark` and `\s__sort_stop`.)

39.2 Finding available `\toks` registers

`__sort_shrink_range:` After `__sort_compute_range:` (defined below) determines that `\toks` registers between `\l__sort_min_int` (included) and `\l__sort_true_max_int` (excluded) have not yet been assigned, `__sort_shrink_range:` computes `\l__sort_max_int` to reflect the need for a buffer when merging blocks in the merge sort. Given $2^n \leq A \leq 2^n + 2^{n-1}$ registers we can sort $\lfloor A/2 \rfloor + 2^{n-2}$ items while if we have $2^n + 2^{n-1} \leq A \leq 2^{n+1}$ registers we can sort $A - 2^{n-1}$ items. We first find out a power 2^n such that $2^n \leq A \leq 2^{n+1}$ by repeatedly halving `\l__sort_block_int`, starting at 2^{15} or 2^{14} namely half the total number of registers, then we use the formulas and set `\l__sort_max_int`.

```
22653 \cs_new_protected:Npn \__sort_shrink_range:
22654 {
22655   \int_set:Nn \l__sort_A_int
22656     { \l__sort_true_max_int - \l__sort_min_int + 1 }
22657   \int_set:Nn \l__sort_block_int { \c_max_register_int / 2 }
22658   \__sort_shrink_range_loop:
22659   \int_set:Nn \l__sort_max_int
22660     {
22661     \int_compare:nNnTF
22662       { \l__sort_block_int * 3 / 2 } > \l__sort_A_int
22663       {
22664         \l__sort_min_int
22665         + ( \l__sort_A_int - 1 ) / 2
22666         + \l__sort_block_int / 4
22667         - 1
22668       }
22669       { \l__sort_true_max_int - \l__sort_block_int / 2 }
22670     }
22671 }
22672 \cs_new_protected:Npn \__sort_shrink_range_loop:
22673 {
22674   \if_int_compare:w \l__sort_A_int < \l__sort_block_int
22675     \tex_divide:D \l__sort_block_int 2 \exp_stop_f:
22676     \exp_after:wN \__sort_shrink_range_loop:
22677   \fi:
22678 }
```

(End definition for `__sort_shrink_range:` and `__sort_shrink_range_loop:`.)

`__sort_compute_range:` First find out what `\toks` have not yet been assigned. There are many cases. In $\text{\LaTeX} 2_{\epsilon}$ with no package, available `\toks` range from `\count15 + 1` to `\c_max_register_int` included (this was not altered despite the 2015 changes). When `\loctoks` is defined, namely in plain (e) \TeX , or when the package `etex` is loaded in $\text{\LaTeX} 2_{\epsilon}$, redefine `__sort_compute_range:` to use the range `\count265` to `\count275 - 1`. The `elocalloc` package also defines `\loctoks` but uses yet another number for the upper bound, namely

`\e@alloc@top` (minus one). We must check for `\loctoks` every time a sorting function is called, as `etex` or `elocalloc` could be loaded.

In ConT_EXt MkIV the range is from `\c_syst_last_allocated_toks+1` to `\c_max_register_int`, and in MkII it is from `\lastallocatedtoks+1` to `\c_max_register_int`. In all these cases, call `__sort_shrink_range:`.

```

22679 \cs_new_protected:Npn \__sort_compute_range:
22680 {
22681   \int_set:Nn \l__sort_min_int { \tex_count:D 15 + 1 }
22682   \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
22683   \__sort_shrink_range:
22684   \if_meaning:w \loctoks \tex_undefined:D \else:
22685     \if_meaning:w \loctoks \scan_stop: \else:
22686       \__sort_redefine_compute_range:
22687       \__sort_compute_range:
22688     \fi:
22689   \fi:
22690 }
22691 \cs_new_protected:Npn \__sort_redefine_compute_range:
22692 {
22693   \cs_if_exist:cTF { ver@elocalloc.sty }
22694   {
22695     \cs_gset_protected:Npn \__sort_compute_range:
22696     {
22697       \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
22698       \int_set_eq:NN \l__sort_true_max_int \e@alloc@top
22699       \__sort_shrink_range:
22700     }
22701   }
22702   {
22703     \cs_gset_protected:Npn \__sort_compute_range:
22704     {
22705       \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
22706       \int_set:Nn \l__sort_true_max_int { \tex_count:D 275 }
22707       \__sort_shrink_range:
22708     }
22709   }
22710 }
22711 \cs_if_exist:NT \loctoks { \__sort_redefine_compute_range: }
22712 \tl_map_inline:nn { \lastallocatedtoks \c_syst_last_allocated_toks }
22713 {
22714   \cs_if_exist:NT #1
22715   {
22716     \cs_gset_protected:Npn \__sort_compute_range:
22717     {
22718       \int_set:Nn \l__sort_min_int { #1 + 1 }
22719       \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
22720       \__sort_shrink_range:
22721     }
22722   }
22723 }

```

(End definition for `__sort_compute_range:`, `__sort_redefine_compute_range:`, and `\c__sort_max_length_int`.)

39.3 Protected user commands

`__sort_main:NNNn` Sorting happens in three steps. First store items in `\toks` registers ranging from `\l__sort_min_int` to `\l__sort_top_int - 1`, while checking that the list is not too long. If we reach the maximum length, that's an error; exit the group. Secondly, sort the array of `\toks` registers, using the user-defined sorting function: `__sort_level:` calls `__sort_compare:nn` as needed. Finally, unpack the `\toks` registers (now sorted) into the target `tl`, or into `\g__sort_internal_seq` for `seq` and `clist`. This is done by `__sort_seq:NNNNn` and `__sort_tl:NNn`.

```

22724 \cs_new_protected:Npn \__sort_main:NNNn #1#2#3#4
22725 {
22726   \__sort_disable_toksdef:
22727   \__sort_compute_range:
22728   \int_set_eq:NN \l__sort_top_int \l__sort_min_int
22729   #1 #3
22730   {
22731     \if_int_compare:w \l__sort_top_int = \l__sort_max_int
22732       \__sort_too_long_error:NNw #2 #3
22733     \fi:
22734     \tex_toks:D \l__sort_top_int {##1}
22735     \int_incr:N \l__sort_top_int
22736   }
22737   \int_set:Nn \l__sort_length_int
22738     { \l__sort_top_int - \l__sort_min_int }
22739   \cs_set:Npn \__sort_compare:nn ##1 ##2 {#4}
22740   \int_set:Nn \l__sort_block_int { 1 }
22741   \__sort_level:
22742 }

```

(End definition for `__sort_main:NNNn`.)

`\tl_sort:Nn` Call the main sorting function then unpack `\toks` registers outside the group into the target token list. The unpacking is done by `__sort_tl_toks:w`; registers are numbered from `\l__sort_min_int` to `\l__sort_top_int - 1`. For expansion behaviour we need a couple of primitives. The `\tl_gclear:N` reduces memory usage. The `\prg_break_point:` is used by `__sort_main:NNNn` when the list is too long.

```

\__sort_tl:NNn
\__sort_tl_toks:w
22743 \cs_new_protected:Npn \tl_sort:Nn { \__sort_tl:NNn \tl_set_eq:NN }
22744 \cs_generate_variant:Nn \tl_sort:Nn { c }
22745 \cs_new_protected:Npn \tl_gsort:Nn { \__sort_tl:NNn \tl_gset_eq:NN }
22746 \cs_generate_variant:Nn \tl_gsort:Nn { c }
22747 \cs_new_protected:Npn \__sort_tl:NNn #1#2#3
22748 {
22749   \group_begin:
22750     \__sort_main:NNNn \tl_map_inline:Nn \tl_map_break:n #2 {#3}
22751     \tl_gset:Nx \g__sort_internal_tl
22752       { \__sort_tl_toks:w \l__sort_min_int ; }
22753   \group_end:
22754   #1 #2 \g__sort_internal_tl
22755   \tl_gclear:N \g__sort_internal_tl
22756   \prg_break_point:
22757 }
22758 \cs_new:Npn \__sort_tl_toks:w #1 ;
22759 {

```

```

22760 \if_int_compare:w #1 < \l__sort_top_int
22761 { \tex_the:D \tex_toks:D #1 }
22762 \exp_after:wN \__sort_tl_toks:w
22763 \int_value:w \int_eval:n { #1 + 1 } \exp_after:wN ;
22764 \fi:
22765 }

```

(End definition for `\tl_sort:Nn` and others. These functions are documented on page 53.)

```

\seq_sort:Nn Use the same general framework for seq and clist. Apply the general sorting code, then
\seq_sort:cn unpack \toks into \g__sort_internal_seq. Outside the group copy or convert (for
\seq_gsort:Nn clist) the data to the target variable. The \seq_gclear:N reduces memory usage. The
\seq_gsort:cn \prg_break_point: is used by \__sort_main:NNNn when the list is too long.
\clist_sort:Nn
22766 \cs_new_protected:Npn \seq_sort:Nn
\clist_sort:cn 22767 { \__sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_set_eq:NN }
\clist_gsort:Nn 22768 \cs_generate_variant:Nn \seq_sort:Nn { c }
\clist_gsort:cn 22769 \cs_new_protected:Npn \seq_gsort:Nn
\__sort_seq:NNNNn 22770 { \__sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_gset_eq:NN }
22771 \cs_generate_variant:Nn \seq_gsort:Nn { c }
22772 \cs_new_protected:Npn \clist_sort:Nn
22773 {
22774 \__sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n
22775 \clist_set_from_seq:NN
22776 }
22777 \cs_generate_variant:Nn \clist_sort:Nn { c }
22778 \cs_new_protected:Npn \clist_gsort:Nn
22779 {
22780 \__sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n
22781 \clist_gset_from_seq:NN
22782 }
22783 \cs_generate_variant:Nn \clist_gsort:Nn { c }
22784 \cs_new_protected:Npn \__sort_seq:NNNNn #1#2#3#4#5
22785 {
22786 \group_begin:
22787 \__sort_main:NNNn #1 #2 #4 {#5}
22788 \seq_gset_from_inline_x:Nnn \g__sort_internal_seq
22789 {
22790 \int_step_function:nnN
22791 { \l__sort_min_int } { \l__sort_top_int - 1 }
22792 }
22793 { \tex_the:D \tex_toks:D ##1 }
22794 \group_end:
22795 #3 #4 \g__sort_internal_seq
22796 \seq_gclear:N \g__sort_internal_seq
22797 \prg_break_point:
22798 }

```

(End definition for `\seq_sort:Nn` and others. These functions are documented on page 80.)

39.4 Merge sort

`__sort_level:` This function is called once blocks of size `\l__sort_block_int` (initially 1) are each sorted. If the whole list fits in one block, then we are done (this also takes care of the

case of an empty list or a list with one item). Otherwise, go through pairs of blocks starting from 0, then double the block size, and repeat.

```

22799 \cs_new_protected:Npn \__sort_level:
22800 {
22801   \if_int_compare:w \l__sort_block_int < \l__sort_length_int
22802     \l__sort_end_int \l__sort_min_int
22803     \__sort_merge_blocks:
22804     \tex_advance:D \l__sort_block_int \l__sort_block_int
22805     \exp_after:wN \__sort_level:
22806   \fi:
22807 }

```

(End definition for __sort_level:.)

__sort_merge_blocks: This function is called to merge a pair of blocks, starting at the last value of `\l__sort_end_int` (end-point of the previous pair of blocks). If shifting by one block to the right we reach the end of the list, then this pass has ended: the end of the list is sorted already. Otherwise, store the result of that shift in *A*, which indexes the first block starting from the top end. Then locate the end-point (maximum) of the second block: shift `end` upwards by one more block, but keeping it \leq `top`. Copy this upper block of `\toks` registers in registers above `length`, indexed by *C*: this is covered by `__sort_copy_block:`. Once this is done we are ready to do the actual merger using `__sort_merge_blocks_aux:`, after shifting *A*, *B* and *C* so that they point to the largest index in their respective ranges rather than pointing just beyond those ranges. Of course, once that pair of blocks is merged, move on to the next pair.

```

22808 \cs_new_protected:Npn \__sort_merge_blocks:
22809 {
22810   \l__sort_begin_int \l__sort_end_int
22811   \tex_advance:D \l__sort_end_int \l__sort_block_int
22812   \if_int_compare:w \l__sort_end_int < \l__sort_top_int
22813     \l__sort_A_int \l__sort_end_int
22814     \tex_advance:D \l__sort_end_int \l__sort_block_int
22815     \if_int_compare:w \l__sort_end_int > \l__sort_top_int
22816       \l__sort_end_int \l__sort_top_int
22817     \fi:
22818     \l__sort_B_int \l__sort_A_int
22819     \l__sort_C_int \l__sort_top_int
22820     \__sort_copy_block:
22821     \int_decr:N \l__sort_A_int
22822     \int_decr:N \l__sort_B_int
22823     \int_decr:N \l__sort_C_int
22824     \exp_after:wN \__sort_merge_blocks_aux:
22825     \exp_after:wN \__sort_merge_blocks:
22826   \fi:
22827 }

```

(End definition for __sort_merge_blocks:.)

__sort_copy_block: We wish to store a copy of the “upper” block of `\toks` registers, ranging between the initial value of `\l__sort_B_int` (included) and `\l__sort_end_int` (excluded) into a new range starting at the initial value of `\l__sort_C_int`, namely `\l__sort_top_int`.

```

22828 \cs_new_protected:Npn \__sort_copy_block:
22829 {

```



```

22830 \tex_toks:D \l__sort_C_int \tex_toks:D \l__sort_B_int
22831 \int_incr:N \l__sort_C_int
22832 \int_incr:N \l__sort_B_int
22833 \if_int_compare:w \l__sort_B_int = \l__sort_end_int
22834 \use_i:nn
22835 \fi:
22836 \__sort_copy_block:
22837 }

```

(End definition for __sort_copy_block:.)

__sort_merge_blocks_aux: At this stage, the first block starts at \l__sort_begin_int, and ends at \l__sort_A_int, and the second block starts at \l__sort_top_int and ends at \l__sort_C_int. The result of the merger is stored at positions indexed by \l__sort_B_int, which starts at \l__sort_end_int - 1 and decreases down to \l__sort_begin_int, covering the full range of the two blocks. In other words, we are building the merger starting with the largest values. The comparison function is defined to return either `swapped` or `same`. Of course, this means the arguments need to be given in the order they appear originally in the list.

```

22838 \cs_new_protected:Npn \__sort_merge_blocks_aux:
22839 {
22840 \exp_after:wN \__sort_compare:nn \exp_after:wN
22841 { \tex_the:D \tex_toks:D \exp_after:wN \l__sort_A_int \exp_after:wN }
22842 \exp_after:wN { \tex_the:D \tex_toks:D \l__sort_C_int }
22843 \prg_do_nothing:
22844 \__sort_return_mark:w
22845 \__sort_return_mark:w
22846 \s__sort_mark
22847 \__sort_return_none_error:
22848 }

```

(End definition for __sort_merge_blocks_aux:.)

`\sort_return_same:` Each comparison should call `\sort_return_same:` or `\sort_return_swapped:` exactly once. If neither is called, `__sort_return_none_error:` is called, since the `return_mark` removes tokens until `\s__sort_mark`. If one is called, the `return_mark` auxiliary removes everything except `__sort_return_same:w` (or its `swapped` analogue) followed by `__sort_return_none_error:.` Finally if two or more are called, `__sort_return_two_error:` ends up before any `__sort_return_mark:w`, so that it produces an error.

```

22849 \cs_new_protected:Npn \sort_return_same:
22850 #1 \__sort_return_mark:w #2 \s__sort_mark
22851 {
22852 #1
22853 #2
22854 \__sort_return_two_error:
22855 \__sort_return_mark:w
22856 \s__sort_mark
22857 \__sort_return_same:w
22858 }
22859 \cs_new_protected:Npn \sort_return_swapped:
22860 #1 \__sort_return_mark:w #2 \s__sort_mark
22861 {
22862 #1

```

```

22863      #2
22864      \__sort_return_two_error:
22865      \__sort_return_mark:w
22866      \s__sort_mark
22867      \__sort_return_swapped:w
22868    }
22869 \cs_new_protected:Npn \__sort_return_mark:w #1 \s__sort_mark { }
22870 \cs_new_protected:Npn \__sort_return_none_error:
22871 {
22872   \__kernel_msg_error:nnxx { kernel } { return-none }
22873   { \tex_the:D \tex_toks:D \l__sort_A_int }
22874   { \tex_the:D \tex_toks:D \l__sort_C_int }
22875   \__sort_return_same:w \__sort_return_none_error:
22876 }
22877 \cs_new_protected:Npn \__sort_return_two_error:
22878 {
22879   \__kernel_msg_error:nnxx { kernel } { return-two }
22880   { \tex_the:D \tex_toks:D \l__sort_A_int }
22881   { \tex_the:D \tex_toks:D \l__sort_C_int }
22882 }

```

(End definition for `\sort_return_same:` and others. These functions are documented on page 225.)

`__sort_return_same:w` If the comparison function returns `same`, then the second argument fed to `__sort_compare:nn` should remain to the right of the other one. Since we build the merger starting from the right, we copy that `\toks` register into the allotted range, then shift the pointers *B* and *C*, and go on to do one more step in the merger, unless the second block has been exhausted: then the remainder of the first block is already in the correct registers and we are done with merging those two blocks.

```

22883 \cs_new_protected:Npn \__sort_return_same:w #1 \__sort_return_none_error:
22884 {
22885   \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
22886   \int_decr:N \l__sort_B_int
22887   \int_decr:N \l__sort_C_int
22888   \if_int_compare:w \l__sort_C_int < \l__sort_top_int
22889     \use_i:nn
22890   \fi:
22891   \__sort_merge_blocks_aux:
22892 }

```

(End definition for `__sort_return_same:w`.)

`__sort_return_swapped:w` If the comparison function returns `swapped`, then the next item to add to the merger is the first argument, contents of the `\toks` register *A*. Then shift the pointers *A* and *B* to the left, and go for one more step for the merger, unless the left block was exhausted (*A* goes below the threshold). In that case, all remaining `\toks` registers in the second block, indexed by *C*, are copied to the merger by `__sort_merge_blocks_end:`.

```

22893 \cs_new_protected:Npn \__sort_return_swapped:w #1 \__sort_return_none_error:
22894 {
22895   \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_A_int
22896   \int_decr:N \l__sort_B_int
22897   \int_decr:N \l__sort_A_int
22898   \if_int_compare:w \l__sort_A_int < \l__sort_begin_int
22899     \__sort_merge_blocks_end: \use_i:nn

```

```

22900   \fi:
22901   \__sort_merge_blocks_aux:
22902 }

```

(End definition for __sort_return_swapped:w.)

__sort_merge_blocks_end: This function’s task is to copy the \toks registers in the block indexed by C to the merger indexed by B . The end can equally be detected by checking when B reaches the threshold `begin`, or when C reaches top.

```

22903 \cs_new_protected:Npn \__sort_merge_blocks_end:
22904 {
22905   \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
22906   \int_decr:N \l__sort_B_int
22907   \int_decr:N \l__sort_C_int
22908   \if_int_compare:w \l__sort_B_int < \l__sort_begin_int
22909     \use_i:nn
22910   \fi:
22911   \__sort_merge_blocks_end:
22912 }

```

(End definition for __sort_merge_blocks_end:.)

39.5 Expandable sorting

Sorting expandably is very different from sorting and assigning to a variable. Since tokens cannot be stored, they must remain in the input stream, and be read through at every step. It is thus necessarily much slower (at best $O(n^2 \ln n)$) than non-expandable sorting functions ($O(n \ln n)$).

A prototypical version of expandable quicksort is as follows. If the argument has no item, return nothing, otherwise partition, using the first item as a pivot (argument #4 of __sort:nnNnn). The arguments of __sort:nnNnn are 1. items less than #4, 2. items greater or equal to #4, 3. comparison, 4. pivot, 5. next item to test. If #5 is the tail of the list, call \tl_sort:nN on #1 and on #2, placing #4 in between; \use:ff expands the parts to make \tl_sort:nN f-expandable. Otherwise, compare #4 and #5 using #3. If they are ordered, place #5 amongst the “greater” items, otherwise amongst the “lesser” items, and continue partitioning.

```

\cs_new:Npn \tl_sort:nN #1#2
{
  \tl_if_blank:nF {#1}
  {
    \__sort:nnNnn { } { } #2
    #1 \q__sort_recursion_tail \q__sort_recursion_stop
  }
}
\cs_new:Npn \__sort:nnNnn #1#2#3#4#5
{
  \quark_if_recursion_tail_stop_do:nn {#5}
  { \use:ff { \tl_sort:nN {#1} #3 {#4} } { \tl_sort:nN {#2} #3 } }
  #3 {#4} {#5}
  { \__sort:nnNnn {#1} { #2 {#5} } #3 {#4} }
  { \__sort:nnNnn { #1 {#5} } {#2} #3 {#4} }
}

```

```

}
\cs_generate_variant:Nn \use:nn { ff }

```

There are quite a few optimizations available here: the code below is less legible, but more than twice as fast.

In the simple version of the code, `__sort:nnNnn` is called $O(n \ln n)$ times on average (the number of comparisons required by the quicksort algorithm). Hence most of our focus is on optimizing that function.

The first speed up is to avoid testing for the end of the list at every call to `__sort:nnNnn`. For this, the list is prepared by changing each *<item>* of the original token list into *<command>* `{<item>}`, just like sequences are stored. We arrange things such that the *<command>* is the *<conditional>* provided by the user: the loop over the *<prepared tokens>* then looks like

```

\cs_new:Npn \__sort_loop:wNn ... #6#7
{
  #6 {<pivot>} {#7} <loop big> <loop small>
  <extra arguments>
}
\__sort_loop:wNn ... <prepared tokens>
<end-loop> {} \s__sort_stop

```

In this example, which matches the structure of `__sort_quick_split_i:NnnnnNn` and a few other functions below, the `__sort_loop:wNn` auxiliary normally receives the user's *<conditional>* as #6 and an *<item>* as #7. This is compared to the *<pivot>* (the argument #5, not shown here), and the *<conditional>* leaves the *<loop big>* or *<loop small>* auxiliary, which both have the same form as `__sort_loop:wNn`, receiving the next pair *<conditional>* `{<item>}` as #6 and #7. At the end, #6 is the *<end-loop>* function, which terminates the loop.

The second speed up is to minimize the duplicated tokens between the **true** and **false** branches of the conditional. For this, we introduce two versions of `__sort:nnNnn`, which receive the new item as #1 and place it either into the list #2 of items less than the pivot #4 or into the list #3 of items greater or equal to the pivot.

```

\cs_new:Npn \__sort_i:nnnnNn #1#2#3#4#5#6
{
  #5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
  {#6} { #2 {#1} } {#3} {#4}
}
\cs_new:Npn \__sort_ii:nnnnNn #1#2#3#4#5#6
{
  #5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
  {#6} {#2} { #3 {#1} } {#4}
}

```

Note that the two functions have the form of `__sort_loop:wNn` above, receiving as #5 the conditional or a function to end the loop. In fact, the lists #2 and #3 must be made of pairs *<conditional>* `{<item>}`, so we have to replace `{#6}` above by `{ #5 {#6} }`, and `{#1}` by `#1`. The actual functions have one more argument, so all argument numbers are shifted compared to this code.

The third speed up is to avoid `\use:ff` using a continuation-passing style: `__sort_quick_split:NnNn` expects a list followed by `\s__sort_mark {⟨code⟩}`, and expands to `⟨code⟩ ⟨sorted list⟩`. Sorting the two parts of the list around the pivot is done with

```
\__sort_quick_split:NnNn #2 ... \s__sort_mark
{
\__sort_quick_split:NnNn #1 ... \s__sort_mark {⟨code⟩}
{⟨pivot⟩}
}
```

Items which are larger than the `⟨pivot⟩` are sorted, then placed after code that sorts the smaller items, and after the (braced) `⟨pivot⟩`.

The fourth speed up is avoid the recursive call to `\tl_sort:nN` with an empty first argument. For this, we introduce functions similar to the `__sort_i:nnnnNn` of the last example, but aware of whether the list of `⟨conditional⟩ {⟨item⟩}` read so far that are less than the pivot, and the list of those greater or equal, are empty or not: see `__sort_quick_split:NnNn` and functions defined below. Knowing whether the lists are empty or not is useless if we do not use distinct ending codes as appropriate. The splitting auxiliaries communicate to the `⟨end-loop⟩` function (that is initially placed after the “prepared” list) by placing a specific ending function, ignored when looping, but useful at the end. In fact, the `⟨end-loop⟩` function does nothing but place the appropriate ending function in front of all its arguments. The ending functions take care of sorting non-empty sublists, placing the pivot in between, and the continuation before.

The final change in fact slows down the code a little, but is required to avoid memory issues: schematically, when `TEX` encounters

```
\use:n { \use:n { \use:n { ... } ... } ... }
```

the argument of the first `\use:n` is not completely read by the second `\use:n`, hence must remain in memory; then the argument of the second `\use:n` is not completely read when grabbing the argument of the third `\use:n`, hence must remain in memory, and so on. The memory consumption grows quadratically with the number of nested `\use:n`. In practice, this means that we must read everything until a trailing `\s__sort_stop` once in a while, otherwise sorting lists of more than a few thousand items would exhaust a typical `TEX`’s memory.

`\tl_sort:nN`

```
\__sort_quick_prepare:Nnnn
\__sort_quick_prepare_end:NNNnw
\__sort_quick_cleanup:w
```

The code within the `\exp_not:f` sorts the list, leaving in most cases a leading `\exp_not:f`, which stops the expansion, letting the result be return within `\exp_not:n`. We filter out the case of a list with no item, which would otherwise cause problems. Then prepare the token list `#1` by inserting the conditional `#2` before each item. The `prepare` auxiliary receives the conditional as `#1`, the prepared token list so far as `#2`, the next prepared item as `#3`, and the item after that as `#4`. The loop ends when `#4` contains `\prg_break_point:`, then the `prepare_end` auxiliary finds the prepared token list as `#4`. The scene is then set up for `__sort_quick_split:NnNn`, which sorts the prepared list and perform the post action placed after `\s__sort_mark`, namely removing the trailing `\s__sort_stop` and `\s__sort_stop` and leaving `\exp_stop_f:` to stop `f`-expansion.

```
22913 \cs_new:Npn \tl_sort:nN #1#2
22914 {
22915   \exp_not:f
22916   {
22917     \tl_if_blank:nF {#1}
```

```

22918         {
22919             \__sort_quick_prepare:Nnnn #2 { } { }
22920             #1
22921             { \prg_break_point: \__sort_quick_prepare_end:NNNnw }
22922             \s__sort_stop
22923         }
22924     }
22925 }
22926 \cs_new:Npn \__sort_quick_prepare:Nnnn #1#2#3#4
22927 {
22928     \prg_break: #4 \prg_break_point:
22929     \__sort_quick_prepare:Nnnn #1 { #2 #3 } { #1 {#4} }
22930 }
22931 \cs_new:Npn \__sort_quick_prepare_end:NNNnw #1#2#3#4#5 \s__sort_stop
22932 {
22933     \__sort_quick_split:NnNn #4 \__sort_quick_end:nnTFNn { }
22934     \s__sort_mark { \__sort_quick_cleanup:w \exp_stop_f: }
22935     \s__sort_mark \s__sort_stop
22936 }
22937 \cs_new:Npn \__sort_quick_cleanup:w #1 \s__sort_mark \s__sort_stop {#1}

```

(End definition for `\tl_sort:nN` and others. This function is documented on page 53.)

```

\__sort_quick_split:NnNn
\__sort_quick_only_i:NnnnnNn
  \_sort_quick_only_ii:NnnnnNn
  \_sort_quick_split_i:NnnnnNn
  \_sort_quick_split_ii:NnnnnNn

```

The `only_i`, `only_ii`, `split_i` and `split_ii` auxiliaries receive a useless first argument, the new item #2 (that they append to either one of the next two arguments), the list #3 of items less than the pivot, bigger items #4, the pivot #5, a *function* #6, and an item #7. The *function* is the user's *conditional* except at the end of the list where it is `__sort_quick_end:nnTFNn`. The comparison is applied to the *pivot* and the *item*, and calls the `only_i` or `split_i` auxiliaries if the *item* is smaller, and the `only_ii` or `split_ii` auxiliaries otherwise. In both cases, the next auxiliary goes to work right away, with no intermediate expansion that would slow down operations. Note that the argument #2 left for the next call has the form *conditional* {*item*}, so that the lists #3 and #4 keep the right form to be fed to the next sorting function. The `split` auxiliary differs from these in that it is missing three of the arguments, which would be empty, and its first argument is always the user's *conditional* rather than an ending function.

```

22938 \cs_new:Npn \__sort_quick_split:NnNn #1#2#3#4
22939 {
22940     #3 {#2} {#4} \__sort_quick_only_ii:NnnnnNn
22941     \__sort_quick_only_i:NnnnnNn
22942     \__sort_quick_single_end:nnwnw
22943     { #3 {#4} } { } { } {#2}
22944 }
22945 \cs_new:Npn \__sort_quick_only_i:NnnnnNn #1#2#3#4#5#6#7
22946 {
22947     #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
22948     \__sort_quick_only_i:NnnnnNn
22949     \__sort_quick_only_i_end:nnwnw
22950     { #6 {#7} } { #3 #2 } { } {#5}
22951 }
22952 \cs_new:Npn \__sort_quick_only_ii:NnnnnNn #1#2#3#4#5#6#7
22953 {
22954     #6 {#5} {#7} \__sort_quick_only_ii:NnnnnNn
22955     \__sort_quick_split_i:NnnnnNn

```

```

22956     \__sort_quick_only_ii_end:nnwnw
22957     { #6 {#7} } { } { #4 #2 } {#5}
22958 }
22959 \cs_new:Npn \__sort_quick_split_i:NnnnnNn #1#2#3#4#5#6#7
22960 {
22961     #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
22962     \__sort_quick_split_i:NnnnnNn
22963     \__sort_quick_split_end:nnwnw
22964     { #6 {#7} } { #3 #2 } {#4} {#5}
22965 }
22966 \cs_new:Npn \__sort_quick_split_ii:NnnnnNn #1#2#3#4#5#6#7
22967 {
22968     #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
22969     \__sort_quick_split_i:NnnnnNn
22970     \__sort_quick_split_end:nnwnw
22971     { #6 {#7} } {#3} { #4 #2 } {#5}
22972 }

```

(End definition for __sort_quick_split:NnNn and others.)

```

\__sort_quick_end:nnTFNn
  \__sort_quick_single_end:nnwnw
  \__sort_quick_only_i_end:nnwnw
  \__sort_quick_only_ii_end:nnwnw
  \__sort_quick_split_end:nnwnw

```

The __sort_quick_end:nnTFNn appears instead of the user's conditional, and receives as its arguments the pivot #1, a fake item #2, a true and a false branches #3 and #4, followed by an ending function #5 (one of the four auxiliaries here) and another copy #6 of the fake item. All those are discarded except the function #5. This function receives lists #1 and #2 of items less than or greater than the pivot #3, then a continuation code #5 just after \s__sort_mark. To avoid a memory problem described earlier, all of the ending functions read #6 until \s__sort_stop and place #6 back into the input stream. When the lists #1 and #2 are empty, the single auxiliary simply places the continuation #5 before the pivot {#3}. When #2 is empty, #1 is sorted and placed before the pivot {#3}, taking care to feed the continuation #5 as a continuation for the function sorting #1. When #1 is empty, #2 is sorted, and the continuation argument is used to place the continuation #5 and the pivot {#3} before the sorted result. Finally, when both lists are non-empty, items larger than the pivot are sorted, then items less than the pivot, and the continuations are done in such a way to place the pivot in between.

```

22973 \cs_new:Npn \__sort_quick_end:nnTFNn #1#2#3#4#5#6 {#5}
22974 \cs_new:Npn \__sort_quick_single_end:nnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
22975 { #5 {#3} #6 \s__sort_stop }
22976 \cs_new:Npn \__sort_quick_only_i_end:nnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
22977 {
22978     \__sort_quick_split:NnNn #1
22979     \__sort_quick_end:nnTFNn { } \s__sort_mark {#5}
22980     {#3}
22981     #6 \s__sort_stop
22982 }
22983 \cs_new:Npn \__sort_quick_only_ii_end:nnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
22984 {
22985     \__sort_quick_split:NnNn #2
22986     \__sort_quick_end:nnTFNn { } \s__sort_mark { #5 {#3} }
22987     #6 \s__sort_stop
22988 }
22989 \cs_new:Npn \__sort_quick_split_end:nnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
22990 {
22991     \__sort_quick_split:NnNn #2 \__sort_quick_end:nnTFNn { } \s__sort_mark

```

```

22992     {
22993         \__sort_quick_split:NnNn #1
22994         \__sort_quick_end:nnTFNn { } \s__sort_mark {#5}
22995         {#3}
22996     }
22997     #6 \s__sort_stop
22998 }

```

(End definition for __sort_quick_end:nnTFNn and others.)

39.6 Messages

__sort_error: Bailing out of the sorting code is a bit tricky. It may not be safe to use a delimited argument, so instead we redefine many l3sort commands to be trivial, with __sort_level: jumping to the break point. This error recovery won't work in a group.

```

22999 \cs_new_protected:Npn \__sort_error:
23000 {
23001     \cs_set_eq:NN \__sort_merge_blocks_aux: \prg_do_nothing:
23002     \cs_set_eq:NN \__sort_merge_blocks: \prg_do_nothing:
23003     \cs_set_protected:Npn \__sort_level: { \group_end: \prg_break: }
23004 }

```

(End definition for __sort_error:.)

__sort_disable_toksdef: While sorting, \toksdef is locally disabled to prevent users from using \newtoks or similar commands in their comparison code: the \toks registers that would be assigned are in use by l3sort. In format mode, none of this is needed since there is no \toks allocator.

```

23005 \cs_new_protected:Npn \__sort_disable_toksdef:
23006 { \cs_set_eq:NN \toksdef \__sort_disabled_toksdef:n }
23007 \cs_new_protected:Npn \__sort_disabled_toksdef:n #1
23008 {
23009     \__kernel_msg_error:nnx { kernel } { toksdef }
23010     { \token_to_str:N #1 }
23011     \__sort_error:
23012     \tex_toksdef:D #1
23013 }
23014 \__kernel_msg_new:nnnn { kernel } { toksdef }
23015 { Allocation~of~\iow_char:N\ toks-registers-impossible~while~sorting. }
23016 {
23017     The~comparison-code-used~for~sorting~a~list~has~attempted~to~
23018     define~#1~as~a~new~\iow_char:N\ toks-register-using~
23019     \iow_char:N\ newtoks~
23020     or~a~similar~command.~The~list~will~not~be~sorted.
23021 }

```

(End definition for __sort_disable_toksdef: and __sort_disabled_toksdef:n.)

__sort_too_long_error:NNw When there are too many items in a sequence, this is an error, and we clean up properly the mapping over items in the list: break using the type-specific breaking function #1.

```

23022 \cs_new_protected:Npn \__sort_too_long_error:NNw #1#2 \fi:
23023 {
23024     \fi:
23025     \__kernel_msg_error:nnxxx { kernel } { too-large }

```



```

23026 { \token_to_str:N #2 }
23027 { \int_eval:n { \l__sort_true_max_int - \l__sort_min_int } }
23028 { \int_eval:n { \l__sort_top_int - \l__sort_min_int } }
23029 #1 \__sort_error:
23030 }
23031 \__kernel_msg_new:nnnn { kernel } { too-large }
23032 { The~list~#1~is~too~long~to~be~sorted~by~TeX. }
23033 {
23034 TeX~has~#2~toks~registers~still~available:~
23035 this~only~allows~to~sort~with~up~to~#3~
23036 items.~The~list~will~not~be~sorted.
23037 }
(End definition for \__sort_too_long_error:NNw.)
23038 \__kernel_msg_new:nnnn { kernel } { return-none }
23039 { The~comparison~code~did~not~return. }
23040 {
23041 When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~
23042 did~not~call~
23043 \iow_char:N\sort_return_same: ~nor~
23044 \iow_char:N\sort_return_swapped: .~
23045 Exactly~one~of~these~should~be~called.
23046 }
23047 \__kernel_msg_new:nnnn { kernel } { return-two }
23048 { The~comparison~code~returned~multiple~times. }
23049 {
23050 When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~called~
23051 \iow_char:N\sort_return_same: ~or~
23052 \iow_char:N\sort_return_swapped: ~multiple~times.~
23053 Exactly~one~of~these~should~be~called.
23054 }
23055 </package>

```

40 l3tl-analysis implementation

```
23056 <@@=tl>
```

40.1 Internal functions

`\s__tl` The format used to store token lists internally uses the scan mark `\s__tl` as a delimiter.

(End definition for `\s__tl`.)

40.2 Internal format

The task of the `l3tl-analysis` module is to convert token lists to an internal format which allows us to extract all the relevant information about individual tokens (category code, character code), as well as reconstruct the token list quickly. This internal format is used in `l3regex` where we need to support arbitrary tokens, and it is used in conversion functions in `l3str-convert`, where we wish to support clusters of characters instead of single tokens.

We thus need a way to encode any *<token>* (even begin-group and end-group character tokens) in a way amenable to manipulating tokens individually. The best we can do is

to find $\langle tokens \rangle$ which both *o*-expand and *x*-expand to the given $\langle token \rangle$. Collecting more information about the category code and character code is also useful for regular expressions, since most regexes are catcode-agnostic. The internal format thus takes the form of a succession of items of the form

$\langle tokens \rangle \backslash s_tl \langle catcode \rangle \langle char\ code \rangle \backslash s_tl$

The $\langle tokens \rangle$ *o*- and *x*-expand to the original token in the token list or to the cluster of tokens corresponding to one Unicode character in the given encoding (for *l3str-convert*). The $\langle catcode \rangle$ is given as a single hexadecimal digit, 0 for control sequences. The $\langle char\ code \rangle$ is given as a decimal number, -1 for control sequences.

Using delimited arguments lets us build the $\langle tokens \rangle$ progressively when doing an encoding conversion in *l3str-convert*. On the other hand, the delimiter $\backslash s_tl$ may not appear unbraced in $\langle tokens \rangle$. This is not a problem because we are careful to wrap control sequences in braces (as an argument to $\backslash exp_not:n$) when converting from a general token list to the internal format.

The current rule for converting a $\langle token \rangle$ to a balanced set of $\langle tokens \rangle$ which both *o*-expands and *x*-expands to it is the following.

- A control sequence $\backslash cs$ becomes $\backslash exp_not:n \{ \backslash cs \} \backslash s_tl 0 -1 \backslash s_tl$.
- A begin-group character $\{$ becomes $\backslash exp_after:wN \{ \backslash if_false: \} \backslash fi: \backslash s_tl 1 \langle char\ code \rangle \backslash s_tl$.
- An end-group character $\}$ becomes $\backslash if_false: \{ \backslash fi: \} \backslash s_tl 2 \langle char\ code \rangle \backslash s_tl$.
- A character with any other category code becomes $\backslash exp_not:n \{ \langle character \rangle \} \backslash s_tl \langle hex\ catcode \rangle \langle char\ code \rangle \backslash s_tl$.

23057 $\langle *package \rangle$

40.3 Variables and helper functions

$\backslash s_tl$ The scan mark $\backslash s_tl$ is used as a delimiter in the internal format. This is more practical than using a quark, because we would then need to control expansion much more carefully: compare $\backslash int_value:w \text{'\#1 \backslash s_tl}$ with $\backslash int_value:w \text{'\#1 \backslash exp_stop_f: \backslash exp_not:N \backslash q_mark}$ to extract a character code followed by the delimiter in an *x*-expansion.

23058 $\backslash scan_new:N \backslash s_tl$

(End definition for $\backslash s_tl$.)

$\backslash l_tl_analysis_token$ The tokens in the token list are probed with the T_EX primitive $\backslash futurelet$. We use $\backslash l_tl_analysis_token$ in that construction. In some cases, we convert the following token to a string before probing it: then the token variable used is $\backslash l_tl_analysis_char_token$.

23059 $\backslash cs_new_eq:NN \backslash l_tl_analysis_token ?$

23060 $\backslash cs_new_eq:NN \backslash l_tl_analysis_char_token ?$

(End definition for $\backslash l_tl_analysis_token$ and $\backslash l_tl_analysis_char_token$.)

$\backslash l_tl_analysis_normal_int$ The number of normal (N-type argument) tokens since the last special token.

23061 $\backslash int_new:N \backslash l_tl_analysis_normal_int$

(End definition for \l__tl_analysis_normal_int.)

\l__tl_analysis_index_int During the first pass, this is the index in the array being built. During the second pass, it is equal to the maximum index in the array from the first pass.

23062 \int_new:N \l__tl_analysis_index_int

(End definition for \l__tl_analysis_index_int.)

\l__tl_analysis_nesting_int Nesting depth of explicit begin-group and end-group characters during the first pass. This lets us detect the end of the token list without a reserved end-marker.

23063 \int_new:N \l__tl_analysis_nesting_int

(End definition for \l__tl_analysis_nesting_int.)

\l__tl_analysis_type_int When encountering special characters, we record their “type” in this integer.

23064 \int_new:N \l__tl_analysis_type_int

(End definition for \l__tl_analysis_type_int.)

\g__tl_analysis_result_tl The result of the conversion is stored in this token list, with a succession of items of the form

$\langle tokens \rangle \backslash s_tl \langle catcode \rangle \langle char\ code \rangle \backslash s_tl$

23065 \tl_new:N \g__tl_analysis_result_tl

(End definition for \g__tl_analysis_result_tl.)

_tl_analysis_extract_charcode: Extracting the character code from the meaning of \l__tl_analysis_token. This has no error checking, and should only be assumed to work for begin-group and end-group character tokens. It produces a number in the form ‘ $\langle char \rangle$ ’.

_tl_analysis_extract_charcode_aux:w

23066 \cs_new:Npn _tl_analysis_extract_charcode:

23067 {

23068 \exp_after:wN _tl_analysis_extract_charcode_aux:w

23069 \token_to_meaning:N \l__tl_analysis_token

23070 }

23071 \cs_new:Npn _tl_analysis_extract_charcode_aux:w #1 ~ #2 ~ { ‘ ’ }

(End definition for _tl_analysis_extract_charcode: and _tl_analysis_extract_charcode_aux:w.)

_tl_analysis_cs_space_count:NN Counts the number of spaces in the string representation of its second argument, as well as the number of characters following the last space in that representation, and feeds the two numbers as semicolon-delimited arguments to the first argument. When this function is used, the escape character is printable and non-space.

_tl_analysis_cs_space_count:w

_tl_analysis_cs_space_count_end:w

23072 \cs_new:Npn _tl_analysis_cs_space_count:NN #1 #2

23073 {

23074 \exp_after:wN #1

23075 \int_value:w \int_eval:w 0

23076 \exp_after:wN _tl_analysis_cs_space_count:w

23077 \token_to_str:N #2

23078 \fi: _tl_analysis_cs_space_count_end:w ; ~ !

23079 }

23080 \cs_new:Npn _tl_analysis_cs_space_count:w #1 ~

23081 {

```

23082     \if_false: #1 #1 \fi:
23083     + 1
23084     \__tl_analysis_cs_space_count:w
23085   }
23086 \cs_new:Npn \__tl_analysis_cs_space_count_end:w ; #1 \fi: #2 !
23087   { \exp_after:wN ; \int_value:w \str_count_ignore_spaces:n {#1} ; }

```

(End definition for `__tl_analysis_cs_space_count:NN`, `__tl_analysis_cs_space_count:w`, and `__tl_analysis_cs_space_count_end:w`.)

40.4 Plan of attack

Our goal is to produce a token list of the form roughly

```

⟨token 1⟩ \s@__ ⟨catcode 1⟩ ⟨char code 1⟩ \s@__
⟨token 2⟩ \s__tl ⟨catcode 2⟩ ⟨char code 2⟩ \s__tl
... ⟨token N⟩ \s__tl ⟨catcode N⟩ ⟨char code N⟩ \s__tl

```

Most but not all tokens can be grabbed as an undelimited (N-type) argument by `TEX`. The plan is to have a two pass system. In the first pass, locate special tokens, and store them in various `\toks` registers. In the second pass, which is done within an `x`-expanding assignment, normal tokens are taken in as N-type arguments, and special tokens are retrieved from the `\toks` registers, and removed from the input stream by some means. The whole process takes linear time, because we avoid building the result one item at a time.

We make the escape character printable (backslash, but this later oscillates between slash and backslash): this allows us to distinguish characters from control sequences.

A token has two characteristics: its `\meaning`, and what it looks like for `TEX` when it is in scanning mode (*e.g.*, when capturing parameters for a macro). For our purposes, we distinguish the following meanings:

- begin-group token (category code 1), either space (character code 32), or non-space;
- end-group token (category code 2), either space (character code 32), or non-space;
- space token (category code 10, character code 32);
- anything else (then the token is always an N-type argument).

The token itself can “look like” one of the following

- a non-active character, in which case its meaning is automatically that associated to its character code and category code, we call it “true” character;
- an active character;
- a control sequence.

The only tokens which are not valid N-type arguments are true begin-group characters, true end-group characters, and true spaces. We detect those characters by scanning ahead with `\futurelet`, then distinguishing true characters from control sequences set equal to them using the `\string` representation.

The second pass is a simple exercise in expandable loops.

`__tl_analysis:n` Everything is done within a group, and all definitions are local. We use `\group_align_safe_begin/end:` to avoid problems in case `__tl_analysis:n` is used within an alignment and its argument contains alignment tab tokens.

```

23088 \cs_new_protected:Npn \__tl_analysis:n #1
23089 {
23090   \group_begin:
23091   \group_align_safe_begin:
23092     \__tl_analysis_a:n {#1}
23093     \__tl_analysis_b:n {#1}
23094   \group_align_safe_end:
23095   \group_end:
23096 }

```

(End definition for `__tl_analysis:n`.)

40.5 Disabling active characters

`__tl_analysis_disable:n` Active characters can cause problems later on in the processing, so we provide a way to disable them, by setting them to undefined. Since Unicode contains too many characters to loop over all of them, we instead do this whenever we encounter a character. For pTeX and upTeX we skip characters beyond [0, 255] because `\lccode` only allows those values.

```

23097 \group_begin:
23098   \char_set_catcode_active:N \^^@
23099   \cs_new_protected:Npn \__tl_analysis_disable:n #1
23100   {
23101     \tex_lccode:D 0 = #1 \exp_stop_f:
23102     \tex_lowercase:D { \tex_let:D \^^@ } \tex_undefined:D
23103   }
23104   \bool_lazy_or:nnT
23105     { \sys_if_engine_ptex_p: }
23106     { \sys_if_engine_uptex_p: }
23107   {
23108     \cs_gset_protected:Npn \__tl_analysis_disable:n #1
23109     {
23110       \if_int_compare:w 256 > #1 \exp_stop_f:
23111       \tex_lccode:D 0 = #1 \exp_stop_f:
23112       \tex_lowercase:D { \tex_let:D \^^@ } \tex_undefined:D
23113     }
23114   }
23115 }
23116 \group_end:

```

(End definition for `__tl_analysis_disable:n`.)

40.6 First pass

The goal of this pass is to detect special (non-N-type) tokens, and count how many N-type tokens lie between special tokens. Also, we wish to store some representation of each special token in a `\toks` register.

We have 11 types of tokens:

1. a true non-space begin-group character;
2. a true space begin-group character;

3. a true non-space end-group character;
4. a true space end-group character;
5. a true space blank space character;
6. an active character;
7. any other true character;
8. a control sequence equal to a begin-group token (category code 1);
9. a control sequence equal to an end-group token (category code 2);
10. a control sequence equal to a space token (character code 32, category code 10);
11. any other control sequence.

Our first tool is `\futurelet`. This cannot distinguish case 8 from 1 or 2, nor case 9 from 3 or 4, nor case 10 from case 5. Those cases are later distinguished by applying the `\string` primitive to the following token, after possibly changing the escape character to ensure that a control sequence’s string representation cannot be mistaken for the true character.

In cases 6, 7, and 11, the following token is a valid N-type argument, so we grab it and distinguish the case of a character from a control sequence: in the latter case, `\str_tail:n {\token}` is non-empty, because the escape character is printable.

`__tl_analysis_a:n` We read tokens one by one using `\futurelet`. While performing the loop, we keep track of the number of true begin-group characters minus the number of true end-group characters in `\l__tl_analysis_nesting_int`. This reaches `-1` when we read the closing brace.

```

23117 \cs_new_protected:Npn \__tl_analysis_a:n #1
23118 {
23119   \__tl_analysis_disable:n { 32 }
23120   \int_set:Nn \tex_escapechar:D { 92 }
23121   \int_zero:N \l__tl_analysis_normal_int
23122   \int_zero:N \l__tl_analysis_index_int
23123   \int_zero:N \l__tl_analysis_nesting_int
23124   \if_false: { \fi: \__tl_analysis_a_loop:w #1 }
23125   \int_decr:N \l__tl_analysis_index_int
23126 }

```

(End definition for `__tl_analysis_a:n`.)

`__tl_analysis_a_loop:w` Read one character and check its type.

```

23127 \cs_new_protected:Npn \__tl_analysis_a_loop:w
23128 { \tex_futurelet:D \l__tl_analysis_token \__tl_analysis_a_type:w }

```

(End definition for `__tl_analysis_a_loop:w`.)

`__tl_analysis_a_type:w` At this point, `\l__tl_analysis_token` holds the meaning of the following token. We store in `\l__tl_analysis_type_int` information about the meaning of the token ahead:

- 0 space token;
- 1 begin-group token;

- -1 end-group token;
- 2 other.

The values 0, 1, -1 correspond to how much a true such character changes the nesting level (2 is used only here, and is irrelevant later). Then call the auxiliary for each case. Note that nesting conditionals here is safe because we only skip over `\l__tl_analysis_token` if it matches with one of the character tokens (hence is not a primitive conditional).

```

23129 \cs_new_protected:Npn \__tl_analysis_a_type:w
23130 {
23131   \l__tl_analysis_type_int =
23132   \if_meaning:w \l__tl_analysis_token \c_space_token
23133     0
23134   \else:
23135     \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_begin_token
23136       1
23137     \else:
23138       \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_end_token
23139         - 1
23140       \else:
23141         2
23142     \fi:
23143   \fi:
23144   \fi:
23145   \exp_stop_f:
23146   \if_case:w \l__tl_analysis_type_int
23147     \exp_after:wN \__tl_analysis_a_space:w
23148   \or: \exp_after:wN \__tl_analysis_a_bgroup:w
23149   \or: \exp_after:wN \__tl_analysis_a_safe:N
23150   \else: \exp_after:wN \__tl_analysis_a_egroup:w
23151   \fi:
23152 }

```

(End definition for `__tl_analysis_a_type:w`.)

`__tl_analysis_a_space:w`
`__tl_analysis_a_space_test:w`

In this branch, the following token's meaning is a blank space. Apply `\string` to that token: a true blank space gives a space, a control sequence gives a result starting with the escape character, an active character gives something else than a space since we disabled the space. We grab as `\l__tl_analysis_char_token` the first character of the string representation then test it in `__tl_analysis_a_space_test:w`. Also, since `__tl_analysis_a_store:` expects the special token to be stored in the relevant `\toks` register, we do that. The extra `\exp_not:n` is unnecessary of course, but it makes the treatment of all tokens more homogeneous. If we discover that the next token was actually a control sequence or an active character instead of a true space, then we step the counter of normal tokens. We now have in front of us the whole string representation of the control sequence, including potential spaces; those will appear to be true spaces later in this pass. Hence, all other branches of the code in this first pass need to consider the string representation, so that the second pass does not need to test the meaning of tokens, only strings.

```

23153 \cs_new_protected:Npn \__tl_analysis_a_space:w
23154 {
23155   \tex_afterassignment:D \__tl_analysis_a_space_test:w
23156   \exp_after:wN \cs_set_eq:NN

```

```

23157 \exp_after:wN \l__tl_analysis_char_token
23158 \token_to_str:N
23159 }
23160 \cs_new_protected:Npn \__tl_analysis_a_space_test:w
23161 {
23162   \if_meaning:w \l__tl_analysis_char_token \c_space_token
23163     \tex_toks:D \l__tl_analysis_index_int { \exp_not:n { ~ } }
23164     \__tl_analysis_a_store:
23165   \else:
23166     \int_incr:N \l__tl_analysis_normal_int
23167   \fi:
23168   \__tl_analysis_a_loop:w
23169 }

```

(End definition for __tl_analysis_a_space:w and __tl_analysis_a_space_test:w.)

```

\__tl_analysis_a_bgroup:w
\__tl_analysis_a_egroup:w
\__tl_analysis_a_group:nw
\__tl_analysis_a_group_aux:w
  \__tl_analysis_a_group_auxii:w
  \__tl_analysis_a_group_test:w

```

The token is most likely a true character token with catcode 1 or 2, but it might be a control sequence, or an active character. Optimizing for the first case, we store in a toks register some code that expands to that token. Since we will turn what follows into a string, we make sure the escape character is different from the current character code (by switching between solidus and backslash). To detect the special case of an active character let to the catcode 1 or 2 character with the same character code, we disable the active character with that character code and re-test: if the following token has become undefined we can in fact safely grab it. We are finally ready to turn what follows to a string and test it. This is one place where we need \l__tl_analysis_char_token to be a separate control sequence from \l__tl_analysis_token, to compare them.

```

23170 \group_begin:
23171   \char_set_catcode_group_begin:N \^^@ % {
23172   \cs_new_protected:Npn \__tl_analysis_a_bgroup:w
23173     { \__tl_analysis_a_group:nw { \exp_after:wN \^^@ \if_false: } \fi: } }
23174   \char_set_catcode_group_end:N \^^@
23175   \cs_new_protected:Npn \__tl_analysis_a_egroup:w
23176     { \__tl_analysis_a_group:nw { \if_false: { \fi: \^^@ } } % }
23177 \group_end:
23178 \cs_new_protected:Npn \__tl_analysis_a_group:nw #1
23179 {
23180   \tex_lccode:D 0 = \__tl_analysis_extract_charcode: \scan_stop:
23181   \tex_lowercase:D { \tex_toks:D \l__tl_analysis_index_int {#1} }
23182   \if_int_compare:w \tex_lccode:D 0 = \tex_escapechar:D
23183     \int_set:Nn \tex_escapechar:D { 139 - \tex_escapechar:D }
23184   \fi:
23185   \__tl_analysis_disable:n { \tex_lccode:D 0 }
23186   \tex_futurelet:D \l__tl_analysis_token \__tl_analysis_a_group_aux:w
23187 }
23188 \cs_new_protected:Npn \__tl_analysis_a_group_aux:w
23189 {
23190   \if_meaning:w \l__tl_analysis_token \tex_undefined:D
23191     \exp_after:wN \__tl_analysis_a_safe:N
23192   \else:
23193     \exp_after:wN \__tl_analysis_a_group_auxii:w
23194   \fi:
23195 }
23196 \cs_new_protected:Npn \__tl_analysis_a_group_auxii:w
23197 {

```



```

23198 \tex_afterassignment:D \__tl_analysis_a_group_test:w
23199 \exp_after:wN \cs_set_eq:NN
23200 \exp_after:wN \l__tl_analysis_char_token
23201 \token_to_str:N
23202 }
23203 \cs_new_protected:Npn \__tl_analysis_a_group_test:w
23204 {
23205   \if_charcode:w \l__tl_analysis_token \l__tl_analysis_char_token
23206     \__tl_analysis_a_store:
23207   \else:
23208     \int_incr:N \l__tl_analysis_normal_int
23209   \fi:
23210   \__tl_analysis_a_loop:w
23211 }

```

(End definition for `__tl_analysis_a_bgroup:w` and others.)

`__tl_analysis_a_store:` This function is called each time we meet a special token; at this point, the `\toks` register `\l__tl_analysis_index_int` holds a token list which expands to the given special token. Also, the value of `\l__tl_analysis_type_int` indicates which case we are in:

- -1 end-group character;
- 0 space character;
- 1 begin-group character.

We need to distinguish further the case of a space character (code 32) from other character codes, because those behave differently in the second pass. Namely, after testing the `\lccode` of 0 (which holds the present character code) we change the cases above to

- -2 space end-group character;
- -1 non-space end-group character;
- 0 space blank space character;
- 1 non-space begin-group character;
- 2 space begin-group character.

This has the property that non-space characters correspond to odd values of `\l__tl_analysis_type_int`. The number of normal tokens until here and the type of special token are packed into a `\skip` register. Finally, we check whether we reached the last closing brace, in which case we stop by disabling the looping function (locally).

```

23212 \cs_new_protected:Npn \__tl_analysis_a_store:
23213 {
23214   \tex_advance:D \l__tl_analysis_nesting_int \l__tl_analysis_type_int
23215   \if_int_compare:w \tex_lccode:D 0 = '\ \exp_stop_f:
23216     \tex_advance:D \l__tl_analysis_type_int \l__tl_analysis_type_int
23217   \fi:
23218   \tex_skip:D \l__tl_analysis_index_int
23219     = \l__tl_analysis_normal_int sp
23220     plus \l__tl_analysis_type_int sp \scan_stop:
23221   \int_incr:N \l__tl_analysis_index_int
23222   \int_zero:N \l__tl_analysis_normal_int

```

```

23223 \if_int_compare:w \l__tl_analysis_nesting_int = -1 \exp_stop_f:
23224 \cs_set_eq:NN \__tl_analysis_a_loop:w \scan_stop:
23225 \fi:
23226 }

```

(End definition for __tl_analysis_a_store:.)

```

\__tl_analysis_a_safe:N
\__tl_analysis_a_cs:ww

```

This should be the simplest case: since the upcoming token is safe, we can simply grab it in a second pass. If the token is a single character (including space), the `\if_charcode:w` test yields true; we disable a potentially active character (that could otherwise masquerade as the true character in the next pass) and we count one “normal” token. On the other hand, if the token is a control sequence, we should replace it by its string representation for compatibility with other code branches. Instead of slowly looping through the characters with the main code, we use the knowledge of how the second pass works: if the control sequence name contains no space, count that token as a number of normal tokens equal to its string length. If the control sequence contains spaces, they should be registered as special characters by increasing `\l__tl_analysis_index_int` (no need to carefully count character between each space), and all characters after the last space should be counted in the following sequence of “normal” tokens.

```

23227 \cs_new_protected:Npn \__tl_analysis_a_safe:N #1
23228 {
23229   \if_charcode:w
23230     \scan_stop:
23231     \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
23232     \scan_stop:
23233     \exp_after:wN \use_i:nn
23234   \else:
23235     \exp_after:wN \use_ii:nn
23236   \fi:
23237   {
23238     \__tl_analysis_disable:n { '#1 }
23239     \int_incr:N \l__tl_analysis_normal_int
23240   }
23241   { \__tl_analysis_cs_space_count:NN \__tl_analysis_a_cs:ww #1 }
23242   \__tl_analysis_a_loop:w
23243 }
23244 \cs_new_protected:Npn \__tl_analysis_a_cs:ww #1; #2;
23245 {
23246   \if_int_compare:w #1 > 0 \exp_stop_f:
23247   \tex_skip:D \l__tl_analysis_index_int
23248   = \int_eval:n { \l__tl_analysis_normal_int + 1 } sp \exp_stop_f:
23249   \tex_advance:D \l__tl_analysis_index_int #1 \exp_stop_f:
23250   \else:
23251     \tex_advance:D
23252   \fi:
23253   \l__tl_analysis_normal_int #2 \exp_stop_f:
23254 }

```

(End definition for __tl_analysis_a_safe:N and __tl_analysis_a_cs:ww.)

40.7 Second pass

The second pass is an exercise in expandable loops. All the necessary information is stored in `\skip` and `\toks` registers.

`__tl_analysis_b:n` Start the loop with the index 0. No need for an end-marker: the loop stops by itself when the last index is read. We repeatedly oscillate between reading long stretches of normal tokens, and reading special tokens.

```

23255 \cs_new_protected:Npn \__tl_analysis_b:n #1
23256 {
23257   \tl_gset:Nx \g__tl_analysis_result_tl
23258   {
23259     \__tl_analysis_b_loop:w 0; #1
23260     \prg_break_point:
23261   }
23262 }
23263 \cs_new:Npn \__tl_analysis_b_loop:w #1;
23264 {
23265   \exp_after:wN \__tl_analysis_b_normals:ww
23266   \int_value:w \tex_skip:D #1 ; #1 ;
23267 }

```

(End definition for `__tl_analysis_b:n` and `__tl_analysis_b_loop:w`.)

`__tl_analysis_b_normals:ww` The first argument is the number of normal tokens which remain to be read, and the
`__tl_analysis_b_normal:wwN` second argument is the index in the array produced in the first step. A character's string representation is always one character long, while a control sequence is always longer (we have set the escape character to a printable value). In both cases, we leave `\exp_not:n` $\langle token \rangle$ `\s__tl` in the input stream (after x-expansion). Here, `\exp_not:n` is used rather than `\exp_not:N` because #3 could be a macro parameter character or could be `\s__tl` (which must be hidden behind braces in the result).

```

23268 \cs_new:Npn \__tl_analysis_b_normals:ww #1;
23269 {
23270   \if_int_compare:w #1 = 0 \exp_stop_f:
23271   \__tl_analysis_b_special:w
23272   \fi:
23273   \__tl_analysis_b_normal:wwN #1;
23274 }
23275 \cs_new:Npn \__tl_analysis_b_normal:wwN #1; #2; #3
23276 {
23277   \exp_not:n { \exp_not:n { #3 } } \s__tl
23278   \if_charcode:w
23279     \scan_stop:
23280     \exp_after:wN \use_none:n \token_to_str:N #3 \prg_do_nothing:
23281     \scan_stop:
23282     \exp_after:wN \__tl_analysis_b_char:Nww
23283   \else:
23284     \exp_after:wN \__tl_analysis_b_cs:Nww
23285   \fi:
23286   #3 #1; #2;
23287 }

```

(End definition for `__tl_analysis_b_normals:ww` and `__tl_analysis_b_normal:wwN`.)

`__tl_analysis_b_char:Nww` If the normal token we grab is a character, leave $\langle catcode \rangle$ $\langle charcode \rangle$ followed by `\s__tl` in the input stream, and call `__tl_analysis_b_normals:ww` with its first argument decremented.

```

23288 \cs_new:Npx \__tl_analysis_b_char:Nww #1

```

```

23289 {
23290   \exp_not:N \if_meaning:w #1 \exp_not:N \tex_undefined:D
23291   \token_to_str:N D \exp_not:N \else:
23292   \exp_not:N \if_catcode:w #1 \c_catcode_other_token
23293   \token_to_str:N C \exp_not:N \else:
23294   \exp_not:N \if_catcode:w #1 \c_catcode_letter_token
23295   \token_to_str:N B \exp_not:N \else:
23296   \exp_not:N \if_catcode:w #1 \c_math_toggle_token      3
23297   \exp_not:N \else:
23298   \exp_not:N \if_catcode:w #1 \c_alignment_token        4
23299   \exp_not:N \else:
23300   \exp_not:N \if_catcode:w #1 \c_math_superscript_token 7
23301   \exp_not:N \else:
23302   \exp_not:N \if_catcode:w #1 \c_math_subscript_token   8
23303   \exp_not:N \else:
23304   \exp_not:N \if_catcode:w #1 \c_space_token
23305   \token_to_str:N A \exp_not:N \else:
23306   6
23307   \exp_not:n { \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi: }
23308   \exp_not:N \int_value:w '#1 \s__tl
23309   \exp_not:N \exp_after:wN \exp_not:N \__tl_analysis_b_normals:ww
23310   \exp_not:N \int_value:w \exp_not:N \int_eval:w - 1 +
23311 }

```

(End definition for __tl_analysis_b_char:Nww.)

__tl_analysis_b_cs:Nww If the token we grab is a control sequence, leave 0 -1 (as category code and character code) in the input stream, followed by \s__tl, and call __tl_analysis_b_normals:ww with updated arguments.

```

23312 \cs_new:Npn \__tl_analysis_b_cs:Nww #1
23313 {
23314   0 -1 \s__tl
23315   \__tl_analysis_cs_space_count:NN \__tl_analysis_b_cs_test:ww #1
23316 }
23317 \cs_new:Npn \__tl_analysis_b_cs_test:ww #1 ; #2 ; #3 ; #4 ;
23318 {
23319   \exp_after:wN \__tl_analysis_b_normals:ww
23320   \int_value:w \int_eval:w
23321   \if_int_compare:w #1 = 0 \exp_stop_f:
23322   #3
23323   \else:
23324     \tex_skip:D \int_eval:n { #4 + #1 } \exp_stop_f:
23325   \fi:
23326   - #2
23327   \exp_after:wN ;
23328   \int_value:w \int_eval:n { #4 + #1 } ;
23329 }

```

(End definition for __tl_analysis_b_cs:Nww and __tl_analysis_b_cs_test:ww.)

__tl_analysis_b_special:w Here, #1 is the current index in the array built in the first pass. Check now whether we reached the end (we shouldn't keep the trailing end-group character that marked the end of the token list in the first pass). Unpack the \toks register: when x-expanding again,

we will get the special token. Then leave the category code in the input stream, followed by the character code, and call `__tl_analysis_b_loop:w` with the next index.

```

23330 \group_begin:
23331   \char_set_catcode_other:N A
23332   \cs_new:Npn \__tl_analysis_b_special:w
23333     \fi: \__tl_analysis_b_normal:wwN 0 ; #1 ;
23334   {
23335     \fi:
23336     \if_int_compare:w #1 = \l__tl_analysis_index_int
23337       \exp_after:wN \prg_break:
23338     \fi:
23339     \tex_the:D \tex_toks:D #1 \s__tl
23340     \if_case:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
23341       \token_to_str:N A
23342     \or: 1
23343     \or: 1
23344     \else: 2
23345     \fi:
23346     \if_int_odd:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
23347       \exp_after:wN \__tl_analysis_b_special_char:wN \int_value:w
23348     \else:
23349       \exp_after:wN \__tl_analysis_b_special_space:w \int_value:w
23350     \fi:
23351     \int_eval:n { 1 + #1 } \exp_after:wN ;
23352     \token_to_str:N
23353   }
23354 \group_end:
23355 \cs_new:Npn \__tl_analysis_b_special_char:wN #1 ; #2
23356   {
23357     \int_value:w ‘#2 \s__tl
23358     \__tl_analysis_b_loop:w #1 ;
23359   }
23360 \cs_new:Npn \__tl_analysis_b_special_space:w #1 ; ~
23361   {
23362     32 \s__tl
23363     \__tl_analysis_b_loop:w #1 ;
23364   }

```

(End definition for `__tl_analysis_b_special:w`, `__tl_analysis_b_special_char:wN`, and `__tl_analysis_b_special_space:w`.)

40.8 Mapping through the analysis

`\tl_analysis_map_inline:nn` First obtain the analysis of the token list into `\g__tl_analysis_result_tl`. To allow nested mappings, increase the nesting depth `\g__kernel_prng_map_int` (shared between all modules), then define the looping macro, which has a name specific to that nesting depth. That looping grabs the $\langle tokens \rangle$, $\langle catcode \rangle$ and $\langle char code \rangle$; it checks for the end of the loop with `\use_none:n ##2`, normally empty, but which becomes `\tl_map_break:` at the end; it then performs the user’s code #2, and loops by calling itself. When the loop ends, remember to decrease the nesting depth.

```

23365 \cs_new_protected:Npn \tl_analysis_map_inline:nn #1
23366   {
23367     \__tl_analysis:n {#1}

```

```

23368 \int_gincr:N \g__kernel_prg_map_int
23369 \exp_args:Nc \__tl_analysis_map_inline_aux:Nn
23370 { \__tl_analysis_map_inline_ \int_use:N \g__kernel_prg_map_int :wNw }
23371 }
23372 \cs_new_protected:Npn \tl_analysis_map_inline:Nn #1
23373 { \exp_args:No \tl_analysis_map_inline:nn #1 }
23374 \cs_new_protected:Npn \__tl_analysis_map_inline_aux:Nn #1#2
23375 {
23376   \cs_gset_protected:Npn #1 ##1 \s__tl ##2 ##3 \s__tl
23377   {
23378     \use_none:n ##2
23379     \__tl_analysis_map_inline_aux:nnn {##1} {##3} {##2}
23380   }
23381   \cs_gset_protected:Npn \__tl_analysis_map_inline_aux:nnn ##1##2##3
23382   {
23383     #2
23384     #1
23385   }
23386   \exp_after:wN #1
23387   \g__tl_analysis_result_tl
23388   \s__tl { ? \tl_map_break: } \s__tl
23389   \prg_break_point:Nn \tl_map_break:
23390   { \int_gdecr:N \g__kernel_prg_map_int }
23391 }

```

(End definition for `\tl_analysis_map_inline:nn` and others. These functions are documented on page 226.)

40.9 Showing the results

`\tl_analysis_show:N` Add to `__tl_analysis:n` a third pass to display tokens to the terminal. If the token list variable is not defined, throw the same error as `\tl_show:N` by simply calling that function.

```

23392 \cs_new_protected:Npn \tl_analysis_show:N #1
23393 {
23394   \tl_if_exist:NTF #1
23395   {
23396     \exp_args:No \__tl_analysis:n {#1}
23397     \msg_show:nnxxxx { LaTeX / kernel } { show-tl-analysis }
23398     { \token_to_str:N #1 } { \__tl_analysis_show: } { } { }
23399   }
23400   { \tl_show:N #1 }
23401 }
23402 \cs_new_protected:Npn \tl_analysis_show:n #1
23403 {
23404   \__tl_analysis:n {#1}
23405   \msg_show:nnxxxx { LaTeX / kernel } { show-tl-analysis }
23406   { } { \__tl_analysis_show: } { } { }
23407 }

```

(End definition for `\tl_analysis_show:N` and `\tl_analysis_show:n`. These functions are documented on page 226.)

`__tl_analysis_show:` Here, #1 o- and x-expands to the token; #2 is the category code (one uppercase hexadecimal digit), 0 for control sequences; #3 is the character code, which we ignore. In the

cases of control sequences and active characters, the meaning may overflow one line, and we want to truncate it. Those cases are thus separated out.

```

23408 \cs_new:Npn \__tl_analysis_show:
23409 {
23410   \exp_after:wN \__tl_analysis_show_loop:wNw \g__tl_analysis_result_tl
23411   \s__tl { ? \prg_break: } \s__tl
23412   \prg_break_point:
23413 }
23414 \cs_new:Npn \__tl_analysis_show_loop:wNw #1 \s__tl #2 #3 \s__tl
23415 {
23416   \use_none:n #2
23417   \iow_newline: > \use:nn { ~ } { ~ }
23418   \if_int_compare:w "#2 = 0 \exp_stop_f:
23419     \exp_after:wN \__tl_analysis_show_cs:n
23420   \else:
23421     \if_int_compare:w "#2 = 13 \exp_stop_f:
23422     \exp_after:wN \exp_after:wN
23423     \exp_after:wN \__tl_analysis_show_active:n
23424   \else:
23425     \exp_after:wN \exp_after:wN
23426     \exp_after:wN \__tl_analysis_show_normal:n
23427   \fi:
23428   \fi:
23429   {#1}
23430   \__tl_analysis_show_loop:wNw
23431 }

```

(End definition for `__tl_analysis_show:` and `__tl_analysis_show_loop:wNw`.)

`__tl_analysis_show_normal:n` Non-active characters are a simple matter of printing the character, and its meaning. Our test suite checks that begin-group and end-group characters do not mess up \TeX 's alignment status.

```

23432 \cs_new:Npn \__tl_analysis_show_normal:n #1
23433 {
23434   \exp_after:wN \token_to_str:N #1 ~
23435   ( \exp_after:wN \token_to_meaning:N #1 )
23436 }

```

(End definition for `__tl_analysis_show_normal:n`.)

`__tl_analysis_show_value:N` This expands to the value of #1 if it has any.

```

23437 \cs_new:Npn \__tl_analysis_show_value:N #1
23438 {
23439   \token_if_expandable:NF #1
23440   {
23441     \token_if_chardef:NTF #1 \prg_break: { }
23442     \token_if_mathchardef:NTF #1 \prg_break: { }
23443     \token_if_dim_register:NTF #1 \prg_break: { }
23444     \token_if_int_register:NTF #1 \prg_break: { }
23445     \token_if_skip_register:NTF #1 \prg_break: { }
23446     \token_if_toks_register:NTF #1 \prg_break: { }
23447     \use_none:nnn
23448     \prg_break_point:
23449     \use:n { \exp_after:wN = \tex_the:D #1 }

```

```

23450     }
23451 }

```

(End definition for `_tl_analysis_show_value:N`.)

`_tl_analysis_show_cs:n` Control sequences and active characters are printed in the same way, making sure not to go beyond the `\l_iow_line_count_int`. In case of an overflow, we replace the last characters by `\c_tl_analysis_show_etc_str`.

```

\_tl\_analysis\_show\_active:n
\_tl\_analysis\_show\_long:nn
\_tl\_analysis\_show\_long\_aux:nnnn
23452 \cs\_new:Npn \_tl\_analysis\_show\_cs:n #1
23453 { \exp\_args:No \_tl\_analysis\_show\_long:nn {#1} { control~sequence= } }
23454 \cs\_new:Npn \_tl\_analysis\_show\_active:n #1
23455 { \exp\_args:No \_tl\_analysis\_show\_long:nn {#1} { active~character= } }
23456 \cs\_new:Npn \_tl\_analysis\_show\_long:nn #1
23457 {
23458   \_tl\_analysis\_show\_long\_aux:oofn
23459   { \token\_to\_str:N #1 }
23460   { \token\_to\_meaning:N #1 }
23461   { \_tl\_analysis\_show\_value:N #1 }
23462 }
23463 \cs\_new:Npn \_tl\_analysis\_show\_long\_aux:nnnn #1#2#3#4
23464 {
23465   \int\_compare:nNnTF
23466   { \str\_count:n { #1 ~ ( #4 #2 #3 ) } }
23467   > { \l\_iow\_line\_count\_int - 3 }
23468   {
23469     \str\_range:nnn { #1 ~ ( #4 #2 #3 ) } { 1 }
23470     {
23471       \l\_iow\_line\_count\_int - 3
23472       - \str\_count:N \c\_tl\_analysis\_show\_etc\_str
23473     }
23474     \c\_tl\_analysis\_show\_etc\_str
23475   }
23476   { #1 ~ ( #4 #2 #3 ) }
23477 }
23478 \cs\_generate\_variant:Nn \_tl\_analysis\_show\_long\_aux:nnnn { oof }

```

(End definition for `_tl_analysis_show_cs:n` and others.)

40.10 Messages

`\c_tl_analysis_show_etc_str` When a control sequence (or active character) and its meaning are too long to fit in one line of the terminal, the end is replaced by this token list.

```

23479 \tl\_const:Nx \c\_tl\_analysis\_show\_etc\_str % (
23480 { \token\_to\_str:N \ETC.) }

```

(End definition for `\c_tl_analysis_show_etc_str`.)

```

23481 \__kernel\_msg\_new:nnn { kernel } { show~tl~analysis }
23482 {
23483   The~token~list~ \tl\_if\_empty:nF {#1} { #1 ~ }
23484   \tl\_if\_empty:nTF {#2}
23485   { is~empty }
23486   { contains~the~tokens: #2 }
23487 }
23488 </package>

```


41 l3regex implementation

23489 `*package`

23490 `\@@=regex`

41.1 Plan of attack

Most regex engines use backtracking. This allows to provide very powerful features (back-references come to mind first), but it is costly, and raises the problem of catastrophic backtracking. Since T_EX is not first and foremost a programming language, complicated code tends to run slowly, and we must use faster, albeit slightly more restrictive, techniques, coming from automata theory.

Given a regular expression of n characters, we do the following:

- (Compiling.) Analyse the regex, finding invalid input, and convert it to an internal representation.
- (Building.) Convert the compiled regex to a non-deterministic finite automaton (NFA) with $O(n)$ states which accepts precisely token lists matching that regex.
- (Matching.) Loop through the query token list one token (one “position”) at a time, exploring in parallel every possible path (“active thread”) through the NFA, considering active threads in an order determined by the quantifiers’ greediness.

We use the following vocabulary in the code comments (and in variable names).

- *Group*: index of the capturing group, -1 for non-capturing groups.
- *Position*: each token in the query is labelled by an integer $\langle position \rangle$, with $\text{min_pos} - 1 \leq \langle position \rangle \leq \text{max_pos}$. The lowest and highest positions correspond to imaginary begin and end markers (with inaccessible category code and character code).
- *Query*: the token list to which we apply the regular expression.
- *State*: each state of the NFA is labelled by an integer $\langle state \rangle$ with $\text{min_state} \leq \langle state \rangle < \text{max_state}$.
- *Active thread*: state of the NFA that is reached when reading the query token list for the matching. Those threads are ordered according to the greediness of quantifiers.
- *Step*: used when matching, starts at 0, incremented every time a character is read, and is not reset when searching for repeated matches. The integer `\l__regex_step_int` is a unique id for all the steps of the matching algorithm.

We use `l3intarray` to manipulate arrays of integers (stored into some dimension registers in scaled points). We also abuse T_EX’s `\toks` registers, by accessing them directly by number rather than tying them to control sequence using the `\newtoks` allocation functions. Specifically, these arrays and `\toks` are used as follows. When building, `\toks\langle state \rangle` holds the tests and actions to perform in the $\langle state \rangle$ of the NFA. When matching,

- `\g__regex_state_active_intarray` holds the last $\langle step \rangle$ in which each $\langle state \rangle$ was active.

- `\g__regex_thread_state_intarray` maps each $\langle thread \rangle$ (with `min_active` $\leq \langle thread \rangle < \text{max_active}$) to the $\langle state \rangle$ in which the $\langle thread \rangle$ currently is. The $\langle threads \rangle$ are ordered starting from the best to the least preferred.
- `\toks\langle thread \rangle` holds the submatch information for the $\langle thread \rangle$, as the contents of a property list.
- `\g__regex_charcode_intarray` and `\g__regex_catcode_intarray` hold the character codes and category codes of tokens at each $\langle position \rangle$ in the query.
- `\g__regex_balance_intarray` holds the balance of begin-group and end-group character tokens which appear before that point in the token list.
- `\toks\langle position \rangle` holds $\langle tokens \rangle$ which o- and x-expand to the $\langle position \rangle$ -th token in the query.
- `\g__regex_submatch_prev_intarray`, `\g__regex_submatch_begin_intarray` and `\g__regex_submatch_end_intarray` hold, for each submatch (as would be extracted by `\regex_extract_all:nnN`), the place where the submatch started to be looked for and its two end-points. For historical reasons, the minimum index is twice `max_state`, and the used registers go up to `\l__regex_submatch_int`. They are organized in blocks of `\l__regex_capturing_group_int` entries, each block corresponding to one match with all its submatches stored in consecutive entries.

The code is structured as follows. Variables are introduced in the relevant section. First we present some generic helper functions. Then comes the code for compiling a regular expression, and for showing the result of the compilation. The building phase converts a compiled regex to NFA states, and the automaton is run by the code in the following section. The only remaining brick is parsing the replacement text and performing the replacement. We are then ready for all the user functions. Finally, messages, and a little bit of tracing code.

41.2 Helpers

`__regex_int_eval:w` Access the primitive: performance is key here, so we do not use the slower route *via* `\int_eval:n`.

```
23491 \cs_new_eq:NN \__regex_int_eval:w \tex_numexpr:D
```

(End definition for `__regex_int_eval:w`.)

`__regex_standard_escapechar:` Make the `\escapechar` into the standard backslash.

```
23492 \cs_new_protected:Npn \__regex_standard_escapechar:
23493 { \int_set:Nn \tex_escapechar:D { '\ } }
```

(End definition for `__regex_standard_escapechar:.`)

`__regex_toks_use:w` Unpack a `\toks` given its number.

```
23494 \cs_new:Npn \__regex_toks_use:w { \tex_the:D \tex_toks:D }
```

(End definition for `__regex_toks_use:w`.)

`__regex_toks_clear:N` Empty a `\toks` or set it to a value, given its number.
`__regex_toks_set:Nn` 23495 `\cs_new_protected:Npn __regex_toks_clear:N #1`
`__regex_toks_set:No` 23496 `{ __regex_toks_set:Nn #1 { } }`
23497 `\cs_new_eq:NN __regex_toks_set:Nn \tex_toks:D`
23498 `\cs_new_protected:Npn __regex_toks_set:No #1`
23499 `{ __regex_toks_set:Nn #1 \exp_after:wN }`

(End definition for `__regex_toks_clear:N` and `__regex_toks_set:Nn`.)

`__regex_toks_memcpy:NNn` Copy #3 `\toks` registers from #2 onwards to #1 onwards, like C's `memcpy`.

23500 `\cs_new_protected:Npn __regex_toks_memcpy:NNn #1#2#3`
23501 `{`
23502 `\prg_replicate:nn {#3}`
23503 `{`
23504 `\tex_toks:D #1 = \tex_toks:D #2`
23505 `\int_incr:N #1`
23506 `\int_incr:N #2`
23507 `}`
23508 `}`

(End definition for `__regex_toks_memcpy:NNn`.)

`__regex_toks_put_left:Nx` During the building phase we wish to add x-expanded material to `\toks`, either to the left
`__regex_toks_put_right:Nx` or to the right. The expansion is done “by hand” for optimization (these operations are
`__regex_toks_put_right:Nn` used quite a lot). The `Nn` version of `__regex_toks_put_right:Nx` is provided because
it is more efficient than x-expanding with `\exp_not:n`.

23509 `\cs_new_protected:Npn __regex_toks_put_left:Nx #1#2`
23510 `{`
23511 `\cs_set:Npx __regex_tmp:w { #2 }`
23512 `\tex_toks:D #1 \exp_after:wN \exp_after:wN \exp_after:wN`
23513 `{ \exp_after:wN __regex_tmp:w \tex_the:D \tex_toks:D #1 }`
23514 `}`
23515 `\cs_new_protected:Npn __regex_toks_put_right:Nx #1#2`
23516 `{`
23517 `\cs_set:Npx __regex_tmp:w {#2}`
23518 `\tex_toks:D #1 \exp_after:wN`
23519 `{ \tex_the:D \tex_toks:D \exp_after:wN #1 __regex_tmp:w }`
23520 `}`
23521 `\cs_new_protected:Npn __regex_toks_put_right:Nn #1#2`
23522 `{ \tex_toks:D #1 \exp_after:wN { \tex_the:D \tex_toks:D #1 #2 } }`

(End definition for `__regex_toks_put_left:Nx` and `__regex_toks_put_right:Nx`.)

`__regex_curr_cs_to_str:` Expands to the string representation of the token (known to be a control sequence) at
the current position `\l__regex_curr_pos_int`. It should only be used in x-expansion to
avoid losing a leading space.

23523 `\cs_new:Npn __regex_curr_cs_to_str:`
23524 `{`
23525 `\exp_after:wN \exp_after:wN \exp_after:wN \cs_to_str:N`
23526 `\tex_the:D \tex_toks:D \l__regex_curr_pos_int`
23527 `}`

(End definition for `__regex_curr_cs_to_str:.`)

41.2.1 Constants and variables

`__regex_tmp:w` Temporary function used for various short-term purposes.

```
23528 \cs_new:Npn \__regex_tmp:w { }
```

(End definition for __regex_tmp:w.)

`\l__regex_internal_a_tl` Temporary variables used for various purposes.

```
\l__regex_internal_b_tl 23529 \tl_new:N \l__regex_internal_a_tl
\l__regex_internal_a_int 23530 \tl_new:N \l__regex_internal_b_tl
\l__regex_internal_b_int 23531 \int_new:N \l__regex_internal_a_int
\l__regex_internal_c_int 23532 \int_new:N \l__regex_internal_b_int
\l__regex_internal_bool 23533 \int_new:N \l__regex_internal_c_int
\l__regex_internal_seq 23534 \bool_new:N \l__regex_internal_bool
\g__regex_internal_tl 23535 \seq_new:N \l__regex_internal_seq
23536 \tl_new:N \g__regex_internal_tl
```

(End definition for \l__regex_internal_a_tl and others.)

`\l__regex_build_tl` This temporary variable is specifically for use with the `tl_build` machinery.

```
23537 \tl_new:N \l__regex_build_tl
```

(End definition for \l__regex_build_tl.)

`\c__regex_no_match_regex` This regular expression matches nothing, but is still a valid regular expression. We could use a failing assertion, but I went for an empty class. It is used as the initial value for regular expressions declared using `\regex_new:N`.

```
23538 \tl_const:Nn \c__regex_no_match_regex
23539 {
23540     \__regex_branch:n
23541     { \__regex_class:NnnnN \c_true_bool { } { 1 } { 0 } \c_true_bool }
23542 }
```

(End definition for \c__regex_no_match_regex.)

`\g__regex_charcode_intarray` The first thing we do when matching is to go once through the query token list and store the information for each token into `\g__regex_charcode_intarray`, `\g__regex_catcode_intarray` and `\toks` registers. We also store the balance of begin-group/end-group characters into `\g__regex_balance_intarray`.

```
23543 \intarray_new:Nn \g__regex_charcode_intarray { 65536 }
23544 \intarray_new:Nn \g__regex_catcode_intarray { 65536 }
23545 \intarray_new:Nn \g__regex_balance_intarray { 65536 }
```

(End definition for \g__regex_charcode_intarray, \g__regex_catcode_intarray, and \g__regex_balance_intarray.)

`\l__regex_balance_int` During this phase, `\l__regex_balance_int` counts the balance of begin-group and end-group character tokens which appear before a given point in the token list. This variable is also used to keep track of the balance in the replacement text.

```
23546 \int_new:N \l__regex_balance_int
```

(End definition for \l__regex_balance_int.)

`\l__regex_cs_name_tl` This variable is used in `__regex_item_cs:n` to store the csname of the currently-tested token when the regex contains a sub-regex for testing csnames.

```
23547 \tl_new:N \l__regex_cs_name_tl
```

(End definition for \l__regex_cs_name_tl.)

41.2.2 Testing characters

```
\c__regex_ascii_min_int
    \c__regex_ascii_max_control_int 23548 \int_const:Nn \c__regex_ascii_min_int { 0 }
\c__regex_ascii_max_int 23549 \int_const:Nn \c__regex_ascii_max_control_int { 31 }
23550 \int_const:Nn \c__regex_ascii_max_int { 127 }

(End definition for \c__regex_ascii_min_int, \c__regex_ascii_max_control_int, and \c__regex_
ascii_max_int.)
```

```
\c__regex_ascii_lower_int
23551 \int_const:Nn \c__regex_ascii_lower_int { ‘a - ‘A }

(End definition for \c__regex_ascii_lower_int.)
```

41.2.3 Internal auxiliaries

```
\q__regex_recursion_stop Internal recursion quarks.
23552 \quark_new:N \q__regex_recursion_stop

(End definition for \q__regex_recursion_stop.)
```

```
regex_use_none_delimit_by_q_recursion_stop:w Functions to gobble up to a quark.
__regex_use_i_delimit_by_q_recursion_stop:nw
23553 \cs_new:Npn \__regex_use_none_delimit_by_q_recursion_stop:w
23554 #1 \q__regex_recursion_stop { }
23555 \cs_new:Npn \__regex_use_i_delimit_by_q_recursion_stop:nw
23556 #1 #2 \q__regex_recursion_stop {#1}

(End definition for \__regex_use_none_delimit_by_q_recursion_stop:w and \__regex_use_i_delimit_
by_q_recursion_stop:nw.)
```

```
\q__regex_nil Internal quarks.
23557 \quark_new:N \q__regex_nil

(End definition for \q__regex_nil.)
```

```
\__regex_quark_if_nil_p:n Branching quark conditional.
\__regex_quark_if_nil:nTF 23558 \__kernel_quark_new_conditional:Nn \__regex_quark_if_nil:N { F }

(End definition for \__regex_quark_if_nil:nTF.)
```

```
\__regex_break_point:TF When testing whether a character of the query token list matches a given character class
\__regex_break_true:w in the regular expression, we often have to test it against several ranges of characters,
checking if any one of those matches. This is done with a structure like
```

```
⟨test1⟩ ... ⟨testn⟩
\__regex_break_point:TF {⟨true code⟩} {⟨false code⟩}
```

If any of the tests succeeds, it calls `__regex_break_true:w`, which cleans up and leaves `⟨true code⟩` in the input stream. Otherwise, `__regex_break_point:TF` leaves the `⟨false code⟩` in the input stream.

```
23559 \cs_new_protected:Npn \__regex_break_true:w
23560 #1 \__regex_break_point:TF #2 #3 {#2}
23561 \cs_new_protected:Npn \__regex_break_point:TF #1 #2 { #2 }

(End definition for \__regex_break_point:TF and \__regex_break_true:w.)
```

`__regex_item_reverse:n` This function makes showing regular expressions easier, and lets us define `\D` in terms of `\d` for instance. There is a subtlety: the end of the query is marked by `-2`, and thus matches `\D` and other negated properties; this case is caught by another part of the code.

```

23562 \cs_new_protected:Npn \__regex_item_reverse:n #1
23563 {
23564     #1
23565     \__regex_break_point:TF { } \__regex_break_true:w
23566 }

```

(End definition for `__regex_item_reverse:n`.)

`__regex_item_caseful_equal:n` Simple comparisons triggering `__regex_break_true:w` when true.

```

\__regex_item_caseful_range:nn
23567 \cs_new_protected:Npn \__regex_item_caseful_equal:n #1
23568 {
23569     \if_int_compare:w #1 = \l__regex_curr_char_int
23570     \exp_after:wN \__regex_break_true:w
23571     \fi:
23572 }
23573 \cs_new_protected:Npn \__regex_item_caseful_range:nn #1 #2
23574 {
23575     \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
23576     \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
23577     \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
23578     \fi:
23579     \fi:
23580 }

```

(End definition for `__regex_item_caseful_equal:n` and `__regex_item_caseful_range:nn`.)

`__regex_item_caseless_equal:n` For caseless matching, we perform the test both on the `current_char` and on the `case_changed_char`. Before doing the second set of tests, we make sure that `case_changed_char` has been computed.

```

\__regex_item_caseless_range:nn
23581 \cs_new_protected:Npn \__regex_item_caseless_equal:n #1
23582 {
23583     \if_int_compare:w #1 = \l__regex_curr_char_int
23584     \exp_after:wN \__regex_break_true:w
23585     \fi:
23586     \if_int_compare:w \l__regex_case_changed_char_int = \c_max_int
23587     \__regex_compute_case_changed_char:
23588     \fi:
23589     \if_int_compare:w #1 = \l__regex_case_changed_char_int
23590     \exp_after:wN \__regex_break_true:w
23591     \fi:
23592 }
23593 \cs_new_protected:Npn \__regex_item_caseless_range:nn #1 #2
23594 {
23595     \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
23596     \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
23597     \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
23598     \fi:
23599     \fi:
23600     \if_int_compare:w \l__regex_case_changed_char_int = \c_max_int
23601     \__regex_compute_case_changed_char:
23602     \fi:

```

```

23603     \reverse_if:N \if_int_compare:w #1 > \l__regex_case_changed_char_int
23604     \reverse_if:N \if_int_compare:w #2 < \l__regex_case_changed_char_int
23605     \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
23606     \fi:
23607   \fi:
23608 }

```

(End definition for __regex_item_caseless_equal:n and __regex_item_caseless_range:nn.)

__regex_compute_case_changed_char: This function is called when \l__regex_case_changed_char_int has not yet been computed (or rather, when it is set to the marker value \c_max_int). If the current character code is in the range [65,90] (upper-case), then add 32, making it lowercase. If it is in the lower-case letter range [97,122], subtract 32.

```

23609 \cs_new_protected:Npn \__regex_compute_case_changed_char:
23610 {
23611   \int_set_eq:NN \l__regex_case_changed_char_int \l__regex_curr_char_int
23612   \if_int_compare:w \l__regex_curr_char_int > 'Z \exp_stop_f:
23613     \if_int_compare:w \l__regex_curr_char_int > 'z \exp_stop_f: \else:
23614       \if_int_compare:w \l__regex_curr_char_int < 'a \exp_stop_f: \else:
23615         \int_sub:Nn \l__regex_case_changed_char_int
23616         { \c__regex_ascii_lower_int }
23617       \fi:
23618     \fi:
23619   \else:
23620     \if_int_compare:w \l__regex_curr_char_int < 'A \exp_stop_f: \else:
23621       \int_add:Nn \l__regex_case_changed_char_int
23622       { \c__regex_ascii_lower_int }
23623     \fi:
23624   \fi:
23625 }

```

(End definition for __regex_compute_case_changed_char:.)

__regex_item_equal:n Those must always be defined to expand to a **caseful** (default) or **caseless** version, and not be protected: they must expand when compiling, to hard-code which tests are caseless or caseful.

```

23626 \cs_new_eq:NN \__regex_item_equal:n ?
23627 \cs_new_eq:NN \__regex_item_range:nn ?

```

(End definition for __regex_item_equal:n and __regex_item_range:nn.)

__regex_item_catcode:nT The argument is a sum of powers of 4 with exponents given by the allowed category codes (between 0 and 13). Dividing by a given power of 4 gives an odd result if and only if that category code is allowed. If the catcode does not match, then skip the character code tests which follow.

```

23628 \cs_new_protected:Npn \__regex_item_catcode:
23629 {
23630   "
23631   \if_case:w \l__regex_curr_catcode_int
23632     1      \or: 4      \or: 10      \or: 40
23633   \or: 100  \or:      \or: 1000    \or: 4000
23634   \or: 10000 \or:      \or: 100000 \or: 400000
23635   \or: 1000000 \or: 4000000 \else: 1*0
23636   \fi:

```

```

23637 }
23638 \cs_new_protected:Npn \__regex_item_catcode:nT #1
23639 {
23640   \if_int_odd:w \int_eval:n { #1 / \__regex_item_catcode: } \exp_stop_f:
23641   \exp_after:wN \use:n
23642   \else:
23643     \exp_after:wN \use_none:n
23644   \fi:
23645 }
23646 \cs_new_protected:Npn \__regex_item_catcode_reverse:nT #1#2
23647 { \__regex_item_catcode:nT {#1} { \__regex_item_reverse:n {#2} } }

(End definition for \__regex_item_catcode:nT, \__regex_item_catcode_reverse:nT, and \__regex_
item_catcode:.)

```

`__regex_item_exact:nn` This matches an exact $\langle category \rangle$ - $\langle character code \rangle$ pair, or an exact control sequence, more precisely one of several possible control sequences.

```

23648 \cs_new_protected:Npn \__regex_item_exact:nn #1#2
23649 {
23650   \if_int_compare:w #1 = \l__regex_curr_catcode_int
23651   \if_int_compare:w #2 = \l__regex_curr_char_int
23652   \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
23653   \fi:
23654   \fi:
23655 }
23656 \cs_new_protected:Npn \__regex_item_exact_cs:n #1
23657 {
23658   \int_compare:nNnTF \l__regex_curr_catcode_int = 0
23659   {
23660     \tl_set:Nx \l__regex_internal_a_tl
23661     { \scan_stop: \__regex_curr_cs_to_str: \scan_stop: }
23662     \tl_if_in:noTF { \scan_stop: #1 \scan_stop: }
23663     \l__regex_internal_a_tl
23664     { \__regex_break_true:w } { }
23665   }
23666   { }
23667 }

```

(End definition for `__regex_item_exact:nn` and `__regex_item_exact_cs:n`.)

`__regex_item_cs:n` Match a control sequence (the argument is a compiled regex). First test the catcode of the current token to be zero. Then perform the matching test, and break if the csname indeed matches. The three `\exp_after:wN` expand the contents of the `\toks` $\langle current position \rangle$ (of the form `\exp_not:n { \langle control sequence \rangle }`) to $\langle control sequence \rangle$. We store the cs name before building states for the cs, as those states may overlap with `\toks` registers storing the user's input.

```

23668 \cs_new_protected:Npn \__regex_item_cs:n #1
23669 {
23670   \int_compare:nNnTF \l__regex_curr_catcode_int = 0
23671   {
23672     \group_begin:
23673     \tl_set:Nx \l__regex_cs_name_tl { \__regex_curr_cs_to_str: }
23674     \__regex_single_match:
23675     \__regex_disable_submatches:

```



```

23676         \__regex_build_for_cs:n {#1}
23677         \bool_set_eq:NN \l__regex_saved_success_bool
23678         \g__regex_success_bool
23679         \exp_args:NV \__regex_match_cs:n \l__regex_cs_name_tl
23680         \if_meaning:w \c_true_bool \g__regex_success_bool
23681         \group_insert_after:N \__regex_break_true:w
23682         \fi:
23683         \bool_gset_eq:NN \g__regex_success_bool
23684         \l__regex_saved_success_bool
23685     \group_end:
23686 }
23687 }

```

(End definition for __regex_item_cs:n.)

41.2.4 Character property tests

__regex_prop_d: Character property tests for \d, \W, etc. These character properties are not affected by the (?i) option. The characters recognized by each one are as follows: \d=[0-9], __regex_prop_h: \w=[0-9A-Z_a-z], \s=[_\^\^I\^\^J\^\^L\^\^M], \h=[_\^\^I], \v=[\^\^J-\^\^M], and the upper case counterparts match anything that the lower case does not match. The order in which the various tests appear is optimized for usual mostly lower case letter text.

```

\__regex_prop_v:
\__regex_prop_w:
\__regex_prop_N:
23688 \cs_new_protected:Npn \__regex_prop_d:
23689 { \__regex_item_caseful_range:nn { '0 } { '9 } }
23690 \cs_new_protected:Npn \__regex_prop_h:
23691 {
23692     \__regex_item_caseful_equal:n { '\ }
23693     \__regex_item_caseful_equal:n { '\^\^I }
23694 }
23695 \cs_new_protected:Npn \__regex_prop_s:
23696 {
23697     \__regex_item_caseful_equal:n { '\ }
23698     \__regex_item_caseful_equal:n { '\^\^I }
23699     \__regex_item_caseful_equal:n { '\^\^J }
23700     \__regex_item_caseful_equal:n { '\^\^L }
23701     \__regex_item_caseful_equal:n { '\^\^M }
23702 }
23703 \cs_new_protected:Npn \__regex_prop_v:
23704 { \__regex_item_caseful_range:nn { '\^\^J } { '\^\^M } } % lf, vtab, ff, cr
23705 \cs_new_protected:Npn \__regex_prop_w:
23706 {
23707     \__regex_item_caseful_range:nn { 'a } { 'z }
23708     \__regex_item_caseful_range:nn { 'A } { 'Z }
23709     \__regex_item_caseful_range:nn { '0 } { '9 }
23710     \__regex_item_caseful_equal:n { '_' }
23711 }
23712 \cs_new_protected:Npn \__regex_prop_N:
23713 {
23714     \__regex_item_reverse:n
23715     { \__regex_item_caseful_equal:n { '\^\^J } }
23716 }

```

(End definition for __regex_prop_d: and others.)

```

__regex_posix_alnum: POSIX properties. No surprise.
__regex_posix_alpha: 23717 \cs_new_protected:Npn __regex_posix_alnum:
__regex_posix_ascii: 23718 { __regex_posix_alpha: __regex_posix_digit: }
__regex_posix_blank: 23719 \cs_new_protected:Npn __regex_posix_alpha:
__regex_posix_cntrl: 23720 { __regex_posix_lower: __regex_posix_upper: }
__regex_posix_digit: 23721 \cs_new_protected:Npn __regex_posix_ascii:
__regex_posix_graph: 23722 {
__regex_posix_lower: 23723   __regex_item_caseful_range:nn
__regex_posix_print: 23724   __regex_ascii_min_int
__regex_posix_punct: 23725   __regex_ascii_max_int
__regex_posix_space: 23726 }
__regex_posix_upper: 23727 \cs_new_eq:NN __regex_posix_blank: __regex_prop_h:
__regex_posix_word: 23728 \cs_new_protected:Npn __regex_posix_cntrl:
__regex_posix_xdigit: 23729 {
23730   __regex_item_caseful_range:nn
23731   __regex_ascii_min_int
23732   __regex_ascii_max_control_int
23733   __regex_item_caseful_equal:n __regex_ascii_max_int
23734 }
23735 \cs_new_eq:NN __regex_posix_digit: __regex_prop_d:
23736 \cs_new_protected:Npn __regex_posix_graph:
23737 { __regex_item_caseful_range:nn { '!' } { '~ } }
23738 \cs_new_protected:Npn __regex_posix_lower:
23739 { __regex_item_caseful_range:nn { 'a' } { 'z' } }
23740 \cs_new_protected:Npn __regex_posix_print:
23741 { __regex_item_caseful_range:nn { '\ ' } { '~ } }
23742 \cs_new_protected:Npn __regex_posix_punct:
23743 {
23744   __regex_item_caseful_range:nn { '!' } { '/' }
23745   __regex_item_caseful_range:nn { ':' } { '@' }
23746   __regex_item_caseful_range:nn { '[' } { '"' }
23747   __regex_item_caseful_range:nn { '{' } { '~' }
23748 }
23749 \cs_new_protected:Npn __regex_posix_space:
23750 {
23751   __regex_item_caseful_equal:n { '\ ' }
23752   __regex_item_caseful_range:nn { '^I' } { '^M' }
23753 }
23754 \cs_new_protected:Npn __regex_posix_upper:
23755 { __regex_item_caseful_range:nn { 'A' } { 'Z' } }
23756 \cs_new_eq:NN __regex_posix_word: __regex_prop_w:
23757 \cs_new_protected:Npn __regex_posix_xdigit:
23758 {
23759   __regex_posix_digit:
23760   __regex_item_caseful_range:nn { 'A' } { 'F' }
23761   __regex_item_caseful_range:nn { 'a' } { 'f' }
23762 }

```

(End definition for `__regex_posix_alnum:` and others.)

41.2.5 Simple character escape

Before actually parsing the regular expression or the replacement text, we go through them once, converting `\n` to the character 10, *etc.* In this pass, we also convert any special

character (*, ?, {, etc.) or escaped alphanumeric character into a marker indicating that this was a special sequence, and replace escaped special characters and non-escaped alphanumeric characters by markers indicating that those were “raw” characters. The rest of the code can then avoid caring about escaping issues (those can become quite complex to handle in combination with ranges in character classes).

Usage: `__regex_escape_use:nnnn` *<inline 1>* *<inline 2>* *<inline 3>* *{<token list>}*
The *<token list>* is converted to a string, then read from left to right, interpreting backslashes as escaping the next character. Unescaped characters are fed to the function *<inline 1>*, and escaped characters are fed to the function *<inline 2>* within an x-expansion context (typically those functions perform some tests on their argument to decide how to output them). The escape sequences `\a`, `\e`, `\f`, `\n`, `\r`, `\t` and `\x` are recognized, and those are replaced by the corresponding character, then fed to *<inline 3>*. The result is then left in the input stream. Spaces are ignored unless escaped.

The conversion is done within an x-expanding assignment.

`__regex_escape_use:nnnn` The result is built in `\l__regex_internal_a_tl`, which is then left in the input stream. Tracing code is added as appropriate inside this token list. Go through #4 once, applying #1, #2, or #3 as relevant to each character (after de-escaping it).

```

23763 \cs_new_protected:Npn \__regex_escape_use:nnnn #1#2#3#4
23764 {
23765   \group_begin:
23766     \tl_clear:N \l__regex_internal_a_tl
23767     \cs_set:Npn \__regex_escape_unescaped:N ##1 { #1 }
23768     \cs_set:Npn \__regex_escape_escaped:N ##1 { #2 }
23769     \cs_set:Npn \__regex_escape_raw:N ##1 { #3 }
23770     \__regex_standard_escapechar:
23771     \tl_gset:Nx \g__regex_internal_tl
23772       { \__kernel_str_to_other_fast:n {#4} }
23773     \tl_put_right:Nx \l__regex_internal_a_tl
23774       {
23775         \exp_after:wN \__regex_escape_loop:N \g__regex_internal_tl
23776         { break } \prg_break_point:
23777       }
23778     \exp_after:wN
23779     \group_end:
23780     \l__regex_internal_a_tl
23781   }

```

(End definition for `__regex_escape_use:nnnn`.)

`__regex_escape_loop:N` `__regex_escape_loop:N` reads one character: if it is special (space, backslash, or end-marker), perform the associated action, otherwise it is simply an unescaped character. After a backslash, the same is done, but unknown characters are “escaped”.

```

23782 \cs_new:Npn \__regex_escape_loop:N #1
23783 {
23784   \cs_if_exist_use:cF { __regex_escape_token_to_str:N #1:w }
23785     { \__regex_escape_unescaped:N #1 }
23786   \__regex_escape_loop:N
23787 }
23788 \cs_new:cpn { __regex_escape_ c_backslash_str :w }
23789   \__regex_escape_loop:N #1
23790 {
23791   \cs_if_exist_use:cF { __regex_escape_/token_to_str:N #1:w }

```

```

23792     { \__regex_escape_escaped:N #1 }
23793     \__regex_escape_loop:N
23794 }

```

(End definition for __regex_escape_loop:N and __regex_escape_w.)

__regex_escape_unescaped:N Those functions are never called before being given a new meaning, so their definitions here don't matter.

```

23795 \cs_new_eq:NN \__regex_escape_unescaped:N ?
23796 \cs_new_eq:NN \__regex_escape_escaped:N ?
23797 \cs_new_eq:NN \__regex_escape_raw:N ?

```

(End definition for __regex_escape_unescaped:N, __regex_escape_escaped:N, and __regex_escape_raw:N.)

__regex_escape_break:w The loop is ended upon seeing the end-marker “break”, with an error if the string ended in a backslash. Spaces are ignored, and \a, \e, \f, \n, \r, \t take their meaning here.

```

\__regex_escape_/break:w
\__regex_escape_/a:w
\__regex_escape_/e:w
\__regex_escape_/f:w
\__regex_escape_/n:w
\__regex_escape_/r:w
\__regex_escape_/t:w
\__regex_escape_\w:w
23798 \cs_new_eq:NN \__regex_escape_break:w \prg_break:
23799 \cs_new:cpn { \__regex_escape_/break:w }
23800 {
23801     \__kernel_msg_expandable_error:nn { kernel } { trailing-backslash }
23802     \prg_break:
23803 }
23804 \cs_new:cpn { \__regex_escape_~:w } { }
23805 \cs_new:cpx { \__regex_escape_/a:w }
23806 { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^G }
23807 \cs_new:cpx { \__regex_escape_/t:w }
23808 { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^I }
23809 \cs_new:cpx { \__regex_escape_/n:w }
23810 { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^J }
23811 \cs_new:cpx { \__regex_escape_/f:w }
23812 { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^L }
23813 \cs_new:cpx { \__regex_escape_/r:w }
23814 { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^M }
23815 \cs_new:cpx { \__regex_escape_/e:w }
23816 { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^[ }

```

(End definition for __regex_escape_break:w and others.)

__regex_escape_/x:w When \x is encountered, __regex_escape_x_test:N is responsible for grabbing some hexadecimal digits, and feeding the result to __regex_escape_x_end:w. If the number is too big interrupt the assignment and produce an error, otherwise call __regex_escape_raw:N on the corresponding character token.

```

23817 \cs_new:cpn { \__regex_escape_/x:w } \__regex_escape_loop:N
23818 {
23819     \exp_after:wN \__regex_escape_x_end:w
23820     \int_value:w "0 \__regex_escape_x_test:N
23821 }
23822 \cs_new:Npn \__regex_escape_x_end:w #1 ;
23823 {
23824     \int_compare:nNnTF {#1} > \c_max_char_int
23825     {
23826         \__kernel_msg_expandable_error:nnff { kernel } { x-overflow }
23827         {#1} { \int_to_Hex:n {#1} }

```

```

23828     }
23829     {
23830         \exp_last_unbraced:Nf \__regex_escape_raw:N
23831         { \char_generate:nn {#1} { 12 } }
23832     }
23833 }

```

(End definition for __regex_escape_/x:w, __regex_escape_x_end:w, and __regex_escape_x_large:n.)

__regex_escape_x_test:N Find out whether the first character is a left brace (allowing any number of hexadecimal digits), or not (allowing up to two hexadecimal digits). We need to check for the end-of-string marker. Eventually, call either __regex_escape_x_loop:N or __regex_escape_x:N.

```

23834 \cs_new:Npn \__regex_escape_x_test:N #1
23835 {
23836     \str_if_eq:nnTF {#1} { break } { ; }
23837     {
23838         \if_charcode:w \c_space_token #1
23839         \exp_after:wN \__regex_escape_x_test:N
23840         \else:
23841         \exp_after:wN \__regex_escape_x_testii:N
23842         \exp_after:wN #1
23843         \fi:
23844     }
23845 }
23846 \cs_new:Npn \__regex_escape_x_testii:N #1
23847 {
23848     \if_charcode:w \c_left_brace_str #1
23849     \exp_after:wN \__regex_escape_x_loop:N
23850     \else:
23851     \__regex_hexadecimal_use:NTF #1
23852     { \exp_after:wN \__regex_escape_x:N }
23853     { ; \exp_after:wN \__regex_escape_loop:N \exp_after:wN #1 }
23854     \fi:
23855 }

```

(End definition for __regex_escape_x_test:N and __regex_escape_x_testii:N.)

__regex_escape_x:N This looks for the second digit in the unbraced case.

```

23856 \cs_new:Npn \__regex_escape_x:N #1
23857 {
23858     \str_if_eq:nnTF {#1} { break } { ; }
23859     {
23860         \__regex_hexadecimal_use:NTF #1
23861         { ; \__regex_escape_loop:N }
23862         { ; \__regex_escape_loop:N #1 }
23863     }
23864 }

```

(End definition for __regex_escape_x:N.)

__regex_escape_x_loop:N Grab hexadecimal digits, skip spaces, and at the end, check that there is a right brace, otherwise raise an error outside the assignment.

```

23865 \cs_new:Npn \__regex_escape_x_loop:N #1

```

```

23866 {
23867   \str_if_eq:nnTF {#1} { break }
23868   { ; \__regex_escape_x_loop_error:n { } {#1} }
23869   {
23870     \__regex_hexadecimal_use:NNTF #1
23871     { \__regex_escape_x_loop:N }
23872     {
23873       \token_if_eq_charcode:NNTF \c_space_token #1
23874       { \__regex_escape_x_loop:N }
23875       {
23876         ;
23877         \exp_after:wN
23878         \token_if_eq_charcode:NNTF \c_right_brace_str #1
23879         { \__regex_escape_loop:N }
23880         { \__regex_escape_x_loop_error:n {#1} }
23881       }
23882     }
23883   }
23884 }
23885 \cs_new:Npn \__regex_escape_x_loop_error:n #1
23886 {
23887   \__kernel_msg_expandable_error:nnn { kernel } { x-missing-rbrace } {#1}
23888   \__regex_escape_loop:N #1
23889 }

```

(End definition for __regex_escape_x_loop:N and __regex_escape_x_loop_error:.)

__regex_hexadecimal_use:NNTF TEX detects uppercase hexadecimal digits for us but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```

23890 \prg_new_conditional:Npnn \__regex_hexadecimal_use:N #1 { TF }
23891 {
23892   \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
23893   #1 \prg_return_true:
23894   \else:
23895     \if_case:w
23896     \int_eval:n { \exp_after:wN ‘ \token_to_str:N #1 - ‘a }
23897     A
23898     \or: B
23899     \or: C
23900     \or: D
23901     \or: E
23902     \or: F
23903     \else:
23904       \prg_return_false:
23905       \exp_after:wN \use_none:n
23906     \fi:
23907     \prg_return_true:
23908   \fi:
23909 }

```

(End definition for __regex_hexadecimal_use:NNTF.)

__regex_char_if_alphanumeric:NNTF These two tests are used in the first pass when parsing a regular expression. That pass is responsible for finding escaped and non-escaped characters, and recognizing which ones have special meanings and which should be interpreted as “raw” characters. Namely,

- alphanumerics are “raw” if they are not escaped, and may have a special meaning when escaped;
- non-alphanumeric printable ascii characters are “raw” if they are escaped, and may have a special meaning when not escaped;
- characters other than printable ascii are always “raw”.

The code is ugly, and highly based on magic numbers and the ascii codes of characters. This is mostly unavoidable for performance reasons. Maybe the tests can be optimized a little bit more. Here, “alphanumeric” means 0–9, A–Z, a–z; “special” character means non-alphanumeric but printable ascii, from space (hex 20) to del (hex 7E).

```

23910 \prg_new_conditional:Npnn \__regex_char_if_special:N #1 { TF }
23911 {
23912   \if_int_compare:w '#1 > 'Z \exp_stop_f:
23913   \if_int_compare:w '#1 > 'z \exp_stop_f:
23914     \if_int_compare:w '#1 < \c__regex_ascii_max_int
23915       \prg_return_true: \else: \prg_return_false: \fi:
23916   \else:
23917     \if_int_compare:w '#1 < 'a \exp_stop_f:
23918       \prg_return_true: \else: \prg_return_false: \fi:
23919   \fi:
23920 \else:
23921   \if_int_compare:w '#1 > '9 \exp_stop_f:
23922   \if_int_compare:w '#1 < 'A \exp_stop_f:
23923     \prg_return_true: \else: \prg_return_false: \fi:
23924   \else:
23925     \if_int_compare:w '#1 < '0 \exp_stop_f:
23926     \if_int_compare:w '#1 < '\' \exp_stop_f:
23927       \prg_return_false: \else: \prg_return_true: \fi:
23928   \else: \prg_return_false: \fi:
23929   \fi:
23930 \fi:
23931 }
23932 \prg_new_conditional:Npnn \__regex_char_if_alphanumeric:N #1 { TF }
23933 {
23934   \if_int_compare:w '#1 > 'Z \exp_stop_f:
23935   \if_int_compare:w '#1 > 'z \exp_stop_f:
23936     \prg_return_false:
23937   \else:
23938     \if_int_compare:w '#1 < 'a \exp_stop_f:
23939       \prg_return_false: \else: \prg_return_true: \fi:
23940   \fi:
23941 \else:
23942   \if_int_compare:w '#1 > '9 \exp_stop_f:
23943   \if_int_compare:w '#1 < 'A \exp_stop_f:
23944     \prg_return_false: \else: \prg_return_true: \fi:
23945   \else:
23946     \if_int_compare:w '#1 < '0 \exp_stop_f:
23947       \prg_return_false: \else: \prg_return_true: \fi:
23948   \fi:
23949 \fi:
23950 }
```

(End definition for __regex_char_if_alphanumeric:N and __regex_char_if_special:N.)

41.3 Compiling

A regular expression starts its life as a string of characters. In this section, we convert it to internal instructions, resulting in a “compiled” regular expression. This compiled expression is then turned into states of an automaton in the building phase. Compiled regular expressions consist of the following:

- `__regex_class:NnnnN` $\langle \text{boolean} \rangle$ $\{\langle \text{tests} \rangle\}$ $\{\langle \text{min} \rangle\}$ $\{\langle \text{more} \rangle\}$ $\langle \text{lazyness} \rangle$
- `__regex_group:nnnN` $\{\langle \text{branches} \rangle\}$ $\{\langle \text{min} \rangle\}$ $\{\langle \text{more} \rangle\}$ $\langle \text{lazyness} \rangle$, also `__regex_group_no_capture:nnnN` and `__regex_group_resetting:nnnN` with the same syntax.
- `__regex_branch:n` $\{\langle \text{contents} \rangle\}$
- `__regex_command_K:`
- `__regex_assertion:Nn` $\langle \text{boolean} \rangle$ $\{\langle \text{assertion test} \rangle\}$, where the $\langle \text{assertion test} \rangle$ is `__regex_b_test:` or `__regex_anchor:N` $\langle \text{integer} \rangle$

Tests can be the following:

- `__regex_item_caseful_equal:n` $\{\langle \text{char code} \rangle\}$
- `__regex_item_caseless_equal:n` $\{\langle \text{char code} \rangle\}$
- `__regex_item_caseful_range:nn` $\{\langle \text{min} \rangle\}$ $\{\langle \text{max} \rangle\}$
- `__regex_item_caseless_range:nn` $\{\langle \text{min} \rangle\}$ $\{\langle \text{max} \rangle\}$
- `__regex_item_catcode:nT` $\{\langle \text{catcode bitmap} \rangle\}$ $\{\langle \text{tests} \rangle\}$
- `__regex_item_catcode_reverse:nT` $\{\langle \text{catcode bitmap} \rangle\}$ $\{\langle \text{tests} \rangle\}$
- `__regex_item_reverse:n` $\{\langle \text{tests} \rangle\}$
- `__regex_item_exact:nn` $\{\langle \text{catcode} \rangle\}$ $\{\langle \text{char code} \rangle\}$
- `__regex_item_exact_cs:n` $\{\langle \text{csnames} \rangle\}$, more precisely given as $\langle \text{cname} \rangle$ `\scan_stop:` $\langle \text{cname} \rangle$ `\scan_stop:` $\langle \text{cname} \rangle$ and so on in a brace group.
- `__regex_item_cs:n` $\{\langle \text{compiled regex} \rangle\}$

41.3.1 Variables used when compiling

`\l__regex_group_level_int`

We make sure to open the same number of groups as we close.

23951 `\int_new:N \l__regex_group_level_int`

(End definition for `\l__regex_group_level_int`.)

`\l__regex_mode_int`

While compiling, ten modes are recognized, labelled -63 , -23 , -6 , -2 , 0 , 2 , 3 , 6 , 23 , 63 .

`\c__regex_cs_in_class_mode_int`

See section 41.3.3. We only define some of these as constants.

`\c__regex_cs_mode_int`

23952 `\int_new:N \l__regex_mode_int`

`\c__regex_outer_mode_int`

23953 `\int_const:Nn \c__regex_cs_in_class_mode_int { -6 }`

`\c__regex_catcode_mode_int`

23954 `\int_const:Nn \c__regex_cs_mode_int { -2 }`

`\c__regex_class_mode_int`

23955 `\int_const:Nn \c__regex_outer_mode_int { 0 }`

`\c__regex_catcode_in_class_mode_int`

23956 `\int_const:Nn \c__regex_catcode_mode_int { 2 }`

23957 `\int_const:Nn \c__regex_class_mode_int { 3 }`

23958 `\int_const:Nn \c__regex_catcode_in_class_mode_int { 6 }`

(End definition for `\l__regex_mode_int` and others.)

`\l__regex_catcodes_int` We wish to allow constructions such as `\c[^BE](. . \cL[a-z] . .)`, where the outer catcode test applies to the whole group, but is superseded by the inner catcode test. For this to work, we need to keep track of lists of allowed category codes: `\l__regex_catcodes_int` and `\l__regex_default_catcodes_int` are bitmaps, sums of 4^c , for all allowed catcodes c . The latter is local to each capturing group, and we reset `\l__regex_catcodes_int` to that value after each character or class, changing it only when encountering a `\c` escape. The boolean records whether the list of categories of a catcode test has to be inverted: compare `\c[^BE]` and `\c[BE]`.

```
23959 \int_new:N \l__regex_catcodes_int
23960 \int_new:N \l__regex_default_catcodes_int
23961 \bool_new:N \l__regex_catcodes_bool
```

(End definition for `\l__regex_catcodes_int`, `\l__regex_default_catcodes_int`, and `\l__regex_catcodes_bool`.)

`\c__regex_catcode_C_int` Constants: 4^c for each category, and the sum of all powers of 4.

```
23962 \int_const:Nn \c__regex_catcode_C_int { "1 }
23963 \int_const:Nn \c__regex_catcode_B_int { "4 }
23964 \int_const:Nn \c__regex_catcode_E_int { "10 }
23965 \int_const:Nn \c__regex_catcode_M_int { "40 }
23966 \int_const:Nn \c__regex_catcode_T_int { "100 }
23967 \int_const:Nn \c__regex_catcode_P_int { "1000 }
23968 \int_const:Nn \c__regex_catcode_U_int { "4000 }
23969 \int_const:Nn \c__regex_catcode_D_int { "10000 }
23970 \int_const:Nn \c__regex_catcode_S_int { "100000 }
23971 \int_const:Nn \c__regex_catcode_L_int { "400000 }
23972 \int_const:Nn \c__regex_catcode_O_int { "1000000 }
23973 \int_const:Nn \c__regex_catcode_A_int { "4000000 }
23974 \int_const:Nn \c__regex_all_catcodes_int { "5515155 }
```

(End definition for `\c__regex_catcode_C_int` and others.)

`\l__regex_internal_regex` The compilation step stores its result in this variable.

```
23975 \cs_new_eq:NN \l__regex_internal_regex \c__regex_no_match_regex
```

(End definition for `\l__regex_internal_regex`.)

`\l__regex_show_prefix_seq` This sequence holds the prefix that makes up the line displayed to the user. The various items must be removed from the right, which is tricky with a token list, hence we use a sequence.

```
23976 \seq_new:N \l__regex_show_prefix_seq
```

(End definition for `\l__regex_show_prefix_seq`.)

`\l__regex_show_lines_int` A hack. To know whether a given class has a single item in it or not, we count the number of lines when showing the class.

```
23977 \int_new:N \l__regex_show_lines_int
```

(End definition for `\l__regex_show_lines_int`.)

41.3.2 Generic helpers used when compiling

`_regex_two_if_eq:NNNTF` Used to compare pairs of things like `_regex_compile_special:N ?` together. It's often inconvenient to get the catcodes of the character to match so we just compare the character code. Besides, the expanding behaviour of `\\if:w` is very useful as that means we can use `\\c_left_brace_str` and the like.

```

23978 \\prg_new_conditional:Npnn \\_regex_two_if_eq:NNNN #1#2#3#4 { TF }
23979 {
23980   \\if_meaning:w #1 #3
23981   \\if:w #2 #4
23982   \\prg_return_true:
23983   \\else:
23984   \\prg_return_false:
23985   \\fi:
23986   \\else:
23987   \\prg_return_false:
23988   \\fi:
23989 }
```

(End definition for _regex_two_if_eq:NNNTF.)

`_regex_get_digits:NTFw` If followed by some raw digits, collect them one by one in the integer variable `#1`, and
`_regex_get_digits_loop:w` take the true branch. Otherwise, take the false branch.

```

23990 \\cs_new_protected:Npn \\_regex_get_digits:NTFw #1#2#3#4#5
23991 {
23992   \\_regex_if_raw_digit:NNTF #4 #5
23993   { #1 = #5 \\_regex_get_digits_loop:nw {#2} }
23994   { #3 #4 #5 }
23995 }
23996 \\cs_new:Npn \\_regex_get_digits_loop:nw #1#2#3
23997 {
23998   \\_regex_if_raw_digit:NNTF #2 #3
23999   { #3 \\_regex_get_digits_loop:nw {#1} }
24000   { \\scan_stop: #1 #2 #3 }
24001 }
```

(End definition for _regex_get_digits:NTFw and _regex_get_digits_loop:w.)

`_regex_if_raw_digit:NNTF` Test used when grabbing digits for the `{m,n}` quantifier. It only accepts non-escaped digits.

```

24002 \\prg_new_conditional:Npnn \\_regex_if_raw_digit:NN #1#2 { TF }
24003 {
24004   \\if_meaning:w \\_regex_compile_raw:N #1
24005   \\if_int_compare:w 1 < 1 #2 \\exp_stop_f:
24006   \\prg_return_true:
24007   \\else:
24008   \\prg_return_false:
24009   \\fi:
24010   \\else:
24011   \\prg_return_false:
24012   \\fi:
24013 }
```

(End definition for _regex_if_raw_digit:NNTF.)

41.3.3 Mode

When compiling the NFA corresponding to a given regex string, we can be in ten distinct modes, which we label by some magic numbers:

- 6 `[\c{...}]` control sequence in a class,
- 2 `\c{...}` control sequence,
- 0 ... outer,
- 2 `\c...` catcode test,
- 6 `[\c...]` catcode test in a class,
- 63 `[\c{[...]}]` class inside mode -6,
- 23 `\c{[...]}` class inside mode -2,
- 3 `[...]` class inside mode 0,
- 23 `\c[...]` class inside mode 2,
- 63 `[\c[...]]` class inside mode 6.

This list is exhaustive, because `\c` escape sequences cannot be nested, and character classes cannot be nested directly. The choice of numbers is such as to optimize the most useful tests, and make transitions from one mode to another as simple as possible.

- Even modes mean that we are not directly in a character class. In this case, a left bracket appends 3 to the mode. In a character class, a right bracket changes the mode as $m \rightarrow (m - 15)/13$, truncated.
- Grouping, assertion, and anchors are allowed in non-positive even modes (0, -2, -6), and do not change the mode. Otherwise, they trigger an error.
- A left bracket is special in even modes, appending 3 to the mode; in those modes, quantifiers and the dot are recognized, and the right bracket is normal. In odd modes (within classes), the left bracket is normal, but the right bracket ends the class, changing the mode from m to $(m - 15)/13$, truncated; also, ranges are recognized.
- In non-negative modes, left and right braces are normal. In negative modes, however, left braces trigger a warning; right braces end the control sequence, going from -2 to 0 or -6 to 3, with error recovery for odd modes.
- Properties (such as the `\d` character class) can appear in any mode.

`__regex_if_in_class:TF` Test whether we are directly in a character class (at the innermost level of nesting). There, many escape sequences are not recognized, and special characters are normal. Also, for every raw character, we must look ahead for a possible raw dash.

```
24014 \cs_new:Npn \__regex_if_in_class:TF
24015 {
24016   \if_int_odd:w \l__regex_mode_int
24017     \exp_after:wN \use_i:nn
24018   \else:
24019     \exp_after:wN \use_ii:nn
24020   \fi:
24021 }
```

(End definition for `_regex_if_in_class:TF`.)

`_regex_if_in_cs:TF` Right braces are special only directly inside control sequences (at the inner-most level of nesting, not counting groups).

```

24022 \cs_new:Npn \_regex_if_in_cs:TF
24023   {
24024     \if_int_odd:w \l__regex_mode_int
24025       \exp_after:wN \use_ii:nn
24026     \else:
24027       \if_int_compare:w \l__regex_mode_int < \c__regex_outer_mode_int
24028         \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
24029       \else:
24030         \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
24031       \fi:
24032     \fi:
24033   }

```

(End definition for `_regex_if_in_cs:TF`.)

`_regex_if_in_class_or_catcode:TF` Assertions are only allowed in modes 0, -2, and -6, *i.e.*, even, non-positive modes.

```

24034 \cs_new:Npn \_regex_if_in_class_or_catcode:TF
24035   {
24036     \if_int_odd:w \l__regex_mode_int
24037       \exp_after:wN \use_i:nn
24038     \else:
24039       \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
24040         \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
24041       \else:
24042         \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
24043       \fi:
24044     \fi:
24045   }

```

(End definition for `_regex_if_in_class_or_catcode:TF`.)

`_regex_if_within_catcode:TF` This test takes the true branch if we are in a catcode test, either immediately following it (modes 2 and 6) or in a class on which it applies (modes 23 and 63). This is used to tweak how left brackets behave in modes 2 and 6.

```

24046 \cs_new:Npn \_regex_if_within_catcode:TF
24047   {
24048     \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
24049       \exp_after:wN \use_i:nn
24050     \else:
24051       \exp_after:wN \use_ii:nn
24052     \fi:
24053   }

```

(End definition for `_regex_if_within_catcode:TF`.)

`_regex_chk_c_allowed:T` The `\c` escape sequence is only allowed in modes 0 and 3, *i.e.*, not within any other `\c` escape sequence.

```

24054 \cs_new_protected:Npn \_regex_chk_c_allowed:T
24055   {
24056     \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int

```

```

24057     \exp_after:wN \use:n
24058 \else:
24059     \if_int_compare:w \l__regex_mode_int = \c__regex_class_mode_int
24060     \exp_after:wN \exp_after:wN \exp_after:wN \use:n
24061 \else:
24062     \__kernel_msg_error:nn { kernel } { c-bad-mode }
24063     \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
24064 \fi:
24065 \fi:
24066 }

```

(End definition for __regex_chk_c_allowed:T.)

__regex_mode_quit_c: This function changes the mode as it is needed just after a catcode test.

```

24067 \cs_new_protected:Npn \__regex_mode_quit_c:
24068 {
24069     \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
24070     \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
24071 \else:
24072     \if_int_compare:w \l__regex_mode_int =
24073     \c__regex_catcode_in_class_mode_int
24074     \int_set_eq:NN \l__regex_mode_int \c__regex_class_mode_int
24075 \fi:
24076 \fi:
24077 }

```

(End definition for __regex_mode_quit_c:.)

41.3.4 Framework

__regex_compile:w Used when compiling a user regex or a regex for the \c{...} escape sequence within another regex. Start building a token list within a group (with x-expansion at the outset), and set a few variables (group level, catcodes), then start the first branch. At the end, make sure there are no dangling classes nor groups, close the last branch: we are done building \l__regex_internal_regex.

```

24078 \cs_new_protected:Npn \__regex_compile:w
24079 {
24080     \group_begin:
24081     \tl_build_begin:N \l__regex_build_tl
24082     \int_zero:N \l__regex_group_level_int
24083     \int_set_eq:NN \l__regex_default_catcodes_int
24084     \c__regex_all_catcodes_int
24085     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
24086     \cs_set:Npn \__regex_item_equal:n { \__regex_item_caseful_equal:n }
24087     \cs_set:Npn \__regex_item_range:nn { \__regex_item_caseful_range:nn }
24088     \tl_build_put_right:Nn \l__regex_build_tl
24089     { \__regex_branch:n { \if_false: } \fi: }
24090 }
24091 \cs_new_protected:Npn \__regex_compile_end:
24092 {
24093     \__regex_if_in_class:TF
24094     {
24095         \__kernel_msg_error:nn { kernel } { missing-rbrack }
24096         \use:c { __regex_compile_]: }

```

```

24097         \prg_do_nothing: \prg_do_nothing:
24098     }
24099     { }
24100     \if_int_compare:w \l__regex_group_level_int > 0 \exp_stop_f:
24101         \__kernel_msg_error:nnx { kernel } { missing-rparen }
24102         { \int_use:N \l__regex_group_level_int }
24103     \prg_replicate:nn
24104         { \l__regex_group_level_int }
24105     {
24106         \tl_build_put_right:Nn \l__regex_build_tl
24107         {
24108             \if_false: { \fi: }
24109             \if_false: { \fi: } { 1 } { 0 } \c_true_bool
24110         }
24111         \tl_build_end:N \l__regex_build_tl
24112         \exp_args:NNNo
24113         \group_end:
24114         \tl_build_put_right:Nn \l__regex_build_tl
24115         { \l__regex_build_tl }
24116     }
24117     \fi:
24118     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
24119     \tl_build_end:N \l__regex_build_tl
24120     \exp_args:NNNx
24121     \group_end:
24122     \tl_set:Nn \l__regex_internal_regex { \l__regex_build_tl }
24123 }

```

(End definition for __regex_compile:w and __regex_compile_end:.)

__regex_compile:n The compilation is done between __regex_compile:w and __regex_compile_end:, starting in mode 0. Then __regex_escape_use:nnnn distinguishes special characters, escaped alphanumerics, and raw characters, interpreting \a, \x and other sequences. The 4 trailing \prg_do_nothing: are needed because some functions defined later look up to 4 tokens ahead. Before ending, make sure that any \c{...} is properly closed. No need to check that brackets are closed properly since __regex_compile_end: does that. However, catch the case of a trailing \cL construction.

```

24124 \cs_new_protected:Npn \__regex_compile:n #1
24125 {
24126     \__regex_compile:w
24127     \__regex_standard_escapechar:
24128     \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
24129     \__regex_escape_use:nnnn
24130     {
24131         \__regex_char_if_special:NTF ##1
24132         \__regex_compile_special:N \__regex_compile_raw:N ##1
24133     }
24134     {
24135         \__regex_char_if_alphanumeric:NTF ##1
24136         \__regex_compile_escaped:N \__regex_compile_raw:N ##1
24137     }
24138     { \__regex_compile_raw:N ##1 }
24139     { #1 }
24140     \prg_do_nothing: \prg_do_nothing:

```

```

24141 \prg_do_nothing: \prg_do_nothing:
24142 \int_compare:nNnT \l__regex_mode_int = \c__regex_catcode_mode_int
24143 { \__kernel_msg_error:nn { kernel } { c-trailing } }
24144 \int_compare:nNnT \l__regex_mode_int < \c__regex_outer_mode_int
24145 {
24146   \__kernel_msg_error:nn { kernel } { c-missing-rbrace }
24147   \__regex_compile_end_cs:
24148   \prg_do_nothing: \prg_do_nothing:
24149   \prg_do_nothing: \prg_do_nothing:
24150 }
24151 \__regex_compile_end:
24152 }

```

(End definition for __regex_compile:n.)

__regex_compile_escaped:N If the special character or escaped alphanumeric has a particular meaning in regexes, the corresponding function is used. Otherwise, it is interpreted as a raw character. We distinguish special characters from escaped alphanumeric characters because they behave differently when appearing as an end-point of a range.

```

24153 \cs_new_protected:Npn \__regex_compile_special:N #1
24154 {
24155   \cs_if_exist_use:cF { __regex_compile_#1: }
24156   { \__regex_compile_raw:N #1 }
24157 }
24158 \cs_new_protected:Npn \__regex_compile_escaped:N #1
24159 {
24160   \cs_if_exist_use:cF { __regex_compile_/#1: }
24161   { \__regex_compile_raw:N #1 }
24162 }

```

(End definition for __regex_compile_escaped:N and __regex_compile_special:N.)

__regex_compile_one:n This is used after finding one “test”, such as \d, or a raw character. If that followed a catcode test (e.g., \cL), then restore the mode. If we are not in a class, then the test is “standalone”, and we need to add __regex_class:NnnnN and search for quantifiers. In any case, insert the test, possibly together with a catcode test if appropriate.

```

24163 \cs_new_protected:Npn \__regex_compile_one:n #1
24164 {
24165   \__regex_mode_quit_c:
24166   \__regex_if_in_class:TF { }
24167   {
24168     \tl_build_put_right:Nn \l__regex_build_tl
24169     { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
24170   }
24171   \tl_build_put_right:Nx \l__regex_build_tl
24172   {
24173     \if_int_compare:w \l__regex_catcodes_int <
24174     \c__regex_all_catcodes_int
24175     \__regex_item_catcode:nT { \int_use:N \l__regex_catcodes_int }
24176     { \exp_not:N \exp_not:n {#1} }
24177     \else:
24178     \exp_not:N \exp_not:n {#1}
24179     \fi:
24180   }

```

```

24181     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
24182     \__regex_if_in_class:TF { } { \__regex_compile_quantifier:w }
24183   }

```

(End definition for __regex_compile_one:n.)

__regex_compile_abort_tokens:n This function places the collected tokens back in the input stream, each as a raw character.
 __regex_compile_abort_tokens:x Spaces are not preserved.

```

24184 \cs_new_protected:Npn \__regex_compile_abort_tokens:n #1
24185 {
24186   \use:x
24187   {
24188     \exp_args:No \tl_map_function:nN { \tl_to_str:n {#1} }
24189     \__regex_compile_raw:N
24190   }
24191 }
24192 \cs_generate_variant:Nn \__regex_compile_abort_tokens:n { x }

```

(End definition for __regex_compile_abort_tokens:n.)

41.3.5 Quantifiers

__regex_compile_quantifier:w This looks ahead and finds any quantifier (special character equal to either of ?+*{).

```

24193 \cs_new_protected:Npn \__regex_compile_quantifier:w #1#2
24194 {
24195   \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
24196   {
24197     \cs_if_exist_use:cF { \__regex_compile_quantifier_#2:w }
24198     { \__regex_compile_quantifier_none: #1 #2 }
24199   }
24200   { \__regex_compile_quantifier_none: #1 #2 }
24201 }

```

(End definition for __regex_compile_quantifier:w.)

__regex_compile_quantifier_none: Those functions are called whenever there is no quantifier, or a braced construction is
 __regex_compile_quantifier_abort:xNN invalid (equivalent to no quantifier, and whatever characters were grabbed are left raw).

```

24202 \cs_new_protected:Npn \__regex_compile_quantifier_none:
24203 {
24204   \tl_build_put_right:Nn \l__regex_build_tl
24205   { \if_false: { \fi: } { 1 } { 0 } \c_false_bool }
24206 }
24207 \cs_new_protected:Npn \__regex_compile_quantifier_abort:xNN #1#2#3
24208 {
24209   \__regex_compile_quantifier_none:
24210   \__kernel_msg_warning:nxx { kernel } { invalid-quantifier } {#1} {#3}
24211   \__regex_compile_abort_tokens:x {#1}
24212   #2 #3
24213 }

```

(End definition for __regex_compile_quantifier_none: and __regex_compile_quantifier_abort:xNN.)

`__regex_compile_quantifier_lazyness:nnNN`

Once the “main” quantifier (`?`, `*`, `+` or a braced construction) is found, we check whether it is lazy (followed by a question mark). We then add to the compiled regex a closing brace (ending `__regex_class:NnnnN` and friends), the start-point of the range, its end-point, and a boolean, `true` for lazy and `false` for greedy operators.

```
24214 \cs_new_protected:Npn \__regex_compile_quantifier_lazyness:nnNN #1#2#3#4
24215 {
24216   \__regex_two_if_eq:NNNTF #3 #4 \__regex_compile_special:N ?
24217   {
24218     \tl_build_put_right:Nn \l__regex_build_tl
24219     { \if_false: { \fi: } { #1 } { #2 } \c_true_bool }
24220   }
24221   {
24222     \tl_build_put_right:Nn \l__regex_build_tl
24223     { \if_false: { \fi: } { #1 } { #2 } \c_false_bool }
24224     #3 #4
24225   }
24226 }
```

(End definition for `__regex_compile_quantifier_lazyness:nnNN`.)

`__regex_compile_quantifier?:w`
`__regex_compile_quantifier*:w`
`__regex_compile_quantifier+:w`

For each “basic” quantifier, `?`, `*`, `+`, feed the correct arguments to `__regex_compile_quantifier_lazyness:nnNN`, `-1` means that there is no upper bound on the number of repetitions.

```
24227 \cs_new_protected:cpn { __regex_compile_quantifier?:w }
24228 { \__regex_compile_quantifier_lazyness:nnNN { 0 } { 1 } }
24229 \cs_new_protected:cpn { __regex_compile_quantifier*:w }
24230 { \__regex_compile_quantifier_lazyness:nnNN { 0 } { -1 } }
24231 \cs_new_protected:cpn { __regex_compile_quantifier+:w }
24232 { \__regex_compile_quantifier_lazyness:nnNN { 1 } { -1 } }
```

(End definition for `__regex_compile_quantifier?:w`, `__regex_compile_quantifier*:w`, and `__regex_compile_quantifier+:w`.)

`__regex_compile_quantifier{:w`
`__regex_compile_quantifier_braced_auxi:w`
`__regex_compile_quantifier_braced_auxii:w`
`__regex_compile_quantifier_braced_auxiii:w`

Three possible syntaxes: `{<int>}`, `{<int>},` or `{<int>,<int>}`. Any other syntax causes us to abort and put whatever we collected back in the input stream, as `raw` characters, including the opening brace. Grab a number into `\l__regex_internal_a_int`. If the number is followed by a right brace, the range is `[a, a]`. If followed by a comma, grab one more number, and call the `_ii` or `_iii` auxiliary. Those auxiliaries check for a closing brace, leading to the range `[a, ∞]` or `[a, b]`, encoded as `{a}{-1}` and `{a}{b-a}`.

```
24233 \cs_new_protected:cpn { __regex_compile_quantifier_ \c_left_brace_str :w }
24234 {
24235   \__regex_get_digits:NTFw \l__regex_internal_a_int
24236   { \__regex_compile_quantifier_braced_auxi:w }
24237   { \__regex_compile_quantifier_abort:xNN { \c_left_brace_str } }
24238 }
24239 \cs_new_protected:Npn \__regex_compile_quantifier_braced_auxi:w #1#2
24240 {
24241   \str_case_e:nnF { #1 #2 }
24242   {
24243     { \__regex_compile_special:N \c_right_brace_str }
24244     {
24245       \exp_args:No \__regex_compile_quantifier_lazyness:nnNN
24246       { \int_use:N \l__regex_internal_a_int } { 0 }
24247     }
24248   }
```

```

24248     { \_regex_compile_special:N , }
24249     {
24250         \_regex_get_digits:NTFw \l__regex_internal_b_int
24251         { \_regex_compile_quantifier_braced_auxiii:w }
24252         { \_regex_compile_quantifier_braced_auxii:w }
24253     }
24254 }
24255 {
24256     \_regex_compile_quantifier_abort:xNN
24257     { \c_left_brace_str \int_use:N \l__regex_internal_a_int }
24258     #1 #2
24259 }
24260 }
24261 \cs_new_protected:Npn \_regex_compile_quantifier_braced_auxii:w #1#2
24262 {
24263     \_regex_two_if_eq:NNNTF #1 #2 \_regex_compile_special:N \c_right_brace_str
24264     {
24265         \exp_args:No \_regex_compile_quantifier_lazyness:nnNN
24266         { \int_use:N \l__regex_internal_a_int } { -1 }
24267     }
24268     {
24269         \_regex_compile_quantifier_abort:xNN
24270         { \c_left_brace_str \int_use:N \l__regex_internal_a_int , }
24271         #1 #2
24272     }
24273 }
24274 \cs_new_protected:Npn \_regex_compile_quantifier_braced_auxiii:w #1#2
24275 {
24276     \_regex_two_if_eq:NNNTF #1 #2 \_regex_compile_special:N \c_right_brace_str
24277     {
24278         \if_int_compare:w \l__regex_internal_a_int >
24279         \l__regex_internal_b_int
24280         \_kernel_msg_error:nxxx { kernel } { backwards-quantifier }
24281         { \int_use:N \l__regex_internal_a_int }
24282         { \int_use:N \l__regex_internal_b_int }
24283         \int_zero:N \l__regex_internal_b_int
24284     \else:
24285         \int_sub:Nn \l__regex_internal_b_int \l__regex_internal_a_int
24286     \fi:
24287     \exp_args:Noo \_regex_compile_quantifier_lazyness:nnNN
24288     { \int_use:N \l__regex_internal_a_int }
24289     { \int_use:N \l__regex_internal_b_int }
24290 }
24291 {
24292     \_regex_compile_quantifier_abort:xNN
24293     {
24294         \c_left_brace_str
24295         \int_use:N \l__regex_internal_a_int ,
24296         \int_use:N \l__regex_internal_b_int
24297     }
24298     #1 #2
24299 }
24300 }

```

(End definition for _regex_compile_quantifier_{:w and others.)

41.3.6 Raw characters

`_regex_compile_raw_error:N` Within character classes, and following catcode tests, some escaped alphanumeric sequences such as `\b` do not have any meaning. They are replaced by a raw character, after spitting out an error.

```

24301 \cs_new_protected:Npn \_regex_compile_raw_error:N #1
24302 {
24303     \__kernel_msg_error:nnx { kernel } { bad-escape } {#1}
24304     \_regex_compile_raw:N #1
24305 }
```

(End definition for `_regex_compile_raw_error:N`.)

`_regex_compile_raw:N` If we are in a character class and the next character is an unescaped dash, this denotes a range. Otherwise, the current character `#1` matches itself.

```

24306 \cs_new_protected:Npn \_regex_compile_raw:N #1#2#3
24307 {
24308     \_regex_if_in_class:TF
24309     {
24310         \_regex_two_if_eq:NNNTF #2 #3 \_regex_compile_special:N -
24311         { \_regex_compile_range:Nw #1 }
24312         {
24313             \_regex_compile_one:n
24314             { \_regex_item_equal:n { \int_value:w '#1 } }
24315             #2 #3
24316         }
24317     }
24318     {
24319         \_regex_compile_one:n
24320         { \_regex_item_equal:n { \int_value:w '#1 } }
24321         #2 #3
24322     }
24323 }
```

(End definition for `_regex_compile_raw:N`.)

`_regex_compile_range:Nw` We have just read a raw character followed by a dash; this should be followed by an end-point for the range. Valid end-points are: any raw character; any special character, except a right bracket. In particular, escaped characters are forbidden.

`_regex_if_end_range:NNTF`

```

24324 \prg_new_protected_conditional:Npnn \_regex_if_end_range:NN #1#2 { TF }
24325 {
24326     \if_meaning:w \_regex_compile_raw:N #1
24327     \prg_return_true:
24328     \else:
24329         \if_meaning:w \_regex_compile_special:N #1
24330         \if_charcode:w ] #2
24331         \prg_return_false:
24332         \else:
24333             \prg_return_true:
24334         \fi:
24335     \else:
24336         \prg_return_false:
24337     \fi:
24338 }
```

```

24339     }
24340     \cs_new_protected:Npn \__regex_compile_range:Nw #1#2#3
24341     {
24342         \__regex_if_end_range:NNTF #2 #3
24343         {
24344             \if_int_compare:w '#1 > '#3 \exp_stop_f:
24345             \__kernel_msg_error:nxxx { kernel } { range-backwards } {#1} {#3}
24346             \else:
24347                 \tl_build_put_right:Nx \l__regex_build_tl
24348                 {
24349                     \if_int_compare:w '#1 = '#3 \exp_stop_f:
24350                     \__regex_item_equal:n
24351                     \else:
24352                         \__regex_item_range:nn { \int_value:w '#1 }
24353                     \fi:
24354                     { \int_value:w '#3 }
24355                 }
24356             \fi:
24357         }
24358         {
24359             \__kernel_msg_warning:nxxx { kernel } { range-missing-end }
24360             {#1} { \c_backslash_str #3 }
24361             \tl_build_put_right:Nx \l__regex_build_tl
24362             {
24363                 \__regex_item_equal:n { \int_value:w '#1 \exp_stop_f: }
24364                 \__regex_item_equal:n { \int_value:w '- \exp_stop_f: }
24365             }
24366             #2#3
24367         }
24368     }

```

(End definition for __regex_compile_range:Nw and __regex_if_end_range:NNTF.)

41.3.7 Character properties

__regex_compile_.: In a class, the dot has no special meaning. Outside, insert __regex_prop_., which matches any character or control sequence, and refuses -2 (end-marker).

```

24369 \cs_new_protected:cpx { __regex_compile_.: }
24370 {
24371     \exp_not:N \__regex_if_in_class:TF
24372     { \__regex_compile_raw:N . }
24373     { \__regex_compile_one:n \exp_not:c { __regex_prop_.: } }
24374 }
24375 \cs_new_protected:cpn { __regex_prop_.: }
24376 {
24377     \if_int_compare:w \l__regex_curr_char_int > - 2 \exp_stop_f:
24378     \exp_after:wN \__regex_break_true:w
24379     \fi:
24380 }

```

(End definition for __regex_compile_.: and __regex_prop_.:)

__regex_compile_/d: The constants __regex_prop_d:, etc. hold a list of tests which match the corresponding character class, and jump to the __regex_break_point:TF marker. As for a normal character, we check for quantifiers.

```

\__regex_compile_/D:
\__regex_compile_/h:
\__regex_compile_/H:
\__regex_compile_/s:
\__regex_compile_/S:
\__regex_compile_/v:
\__regex_compile_/V:
\__regex_compile_/w:
\__regex_compile_/W:
\__regex_compile_/N:

```

```

24381 \cs_set_protected:Npn \__regex_tmp:w #1#2
24382 {
24383   \cs_new_protected:cpx { __regex_compile_/#1: }
24384   { \__regex_compile_one:n \exp_not:c { __regex_prop_#1: } }
24385   \cs_new_protected:cpx { __regex_compile_/#2: }
24386   {
24387     \__regex_compile_one:n
24388     { \__regex_item_reverse:n \exp_not:c { __regex_prop_#1: } }
24389   }
24390 }
24391 \__regex_tmp:w d D
24392 \__regex_tmp:w h H
24393 \__regex_tmp:w s S
24394 \__regex_tmp:w v V
24395 \__regex_tmp:w w W
24396 \cs_new_protected:cpn { __regex_compile_/N: }
24397 { \__regex_compile_one:n \__regex_prop_N: }

```

(End definition for __regex_compile_/d: and others.)

41.3.8 Anchoring and simple assertions

__regex_compile_anchor:NF In modes where assertions are allowed, anchor to the start of the query, the start of the match, or the end of the query, depending on the integer #1. In other modes, #2 treats the character as raw, with an error for escaped letters (\$ is valid in a class, but \A is definitely a mistake on the user's part).

```

\__regex_compile_anchor:NF
  \__regex_compile_#:
  \__regex_compile_/A:
  \__regex_compile_/G:
  \__regex_compile_/$:
  \__regex_compile_/Z:
  \__regex_compile_/z:
24398 \cs_new_protected:Npn \__regex_compile_anchor:NF #1#2
24399 {
24400   \__regex_if_in_class_or_catcode:TF {#2}
24401   {
24402     \tl_build_put_right:Nn \l__regex_build_tl
24403     { \__regex_assertion:Nn \c_true_bool { \__regex_anchor:N #1 } }
24404   }
24405 }
24406 \cs_set_protected:Npn \__regex_tmp:w #1#2
24407 {
24408   \cs_new_protected:cpn { __regex_compile_/#1: }
24409   { \__regex_compile_anchor:NF #2 { \__regex_compile_raw_error:N #1 } }
24410 }
24411 \__regex_tmp:w A \l__regex_min_pos_int
24412 \__regex_tmp:w G \l__regex_start_pos_int
24413 \__regex_tmp:w Z \l__regex_max_pos_int
24414 \__regex_tmp:w z \l__regex_max_pos_int
24415 \cs_set_protected:Npn \__regex_tmp:w #1#2
24416 {
24417   \cs_new_protected:cpn { __regex_compile_#1: }
24418   { \__regex_compile_anchor:NF #2 { \__regex_compile_raw:N #1 } }
24419 }
24420 \exp_args:Nx \__regex_tmp:w { \iow_char:N \^ } \l__regex_min_pos_int
24421 \exp_args:Nx \__regex_tmp:w { \iow_char:N \$ } \l__regex_max_pos_int

```

(End definition for __regex_compile_anchor:NF and others.)

`_regex_compile/b:` Contrarily to `^` and `$`, which could be implemented without really knowing what precedes in the token list, this requires more information, namely, the knowledge of the last character code.

`_regex_compile/B:`

```

24422 \cs_new_protected:cpn { \_regex_compile/b: }
24423 {
24424   \_regex_if_in_class_or_catcode:TF
24425   { \_regex_compile_raw_error:N b }
24426   {
24427     \tl_build_put_right:Nn \l__regex_build_tl
24428     { \_regex_assertion:Nn \c_true_bool { \_regex_b_test: } }
24429   }
24430 }
24431 \cs_new_protected:cpn { \_regex_compile/B: }
24432 {
24433   \_regex_if_in_class_or_catcode:TF
24434   { \_regex_compile_raw_error:N B }
24435   {
24436     \tl_build_put_right:Nn \l__regex_build_tl
24437     { \_regex_assertion:Nn \c_false_bool { \_regex_b_test: } }
24438   }
24439 }

```

(End definition for `_regex_compile/b:` and `_regex_compile/B:`)

41.3.9 Character classes

`_regex_compile_]:` Outside a class, right brackets have no meaning. In a class, change the mode ($m \rightarrow (m - 15)/13$, truncated) to reflect the fact that we are leaving the class. Look for quantifiers, unless we are still in a class after leaving one (the case of `[... \cL[...]`). quantifiers.

```

24440 \cs_new_protected:cpn { \_regex_compile_]: }
24441 {
24442   \_regex_if_in_class:TF
24443   {
24444     \if_int_compare:w \l__regex_mode_int >
24445     \c__regex_catcode_in_class_mode_int
24446     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
24447     \fi:
24448     \tex_advance:D \l__regex_mode_int - 15 \exp_stop_f:
24449     \tex_divide:D \l__regex_mode_int 13 \exp_stop_f:
24450     \if_int_odd:w \l__regex_mode_int \else:
24451     \exp_after:wN \_regex_compile_quantifier:w
24452     \fi:
24453   }
24454   { \_regex_compile_raw:N ] }
24455 }

```

(End definition for `_regex_compile_]:`)

`_regex_compile[:` In a class, left brackets might introduce a POSIX character class, or mean nothing. Immediately following `\c<category>`, we must insert the appropriate catcode test, then parse the class; we pre-expand the catcode as an optimization. Otherwise (modes 0, -2 and -6) just parse the class. The mode is updated later.

```

24456 \cs_new_protected:cpn { \_regex_compile[: }

```

```

24457 {
24458     \__regex_if_in_class:TF
24459     { \__regex_compile_class_posix_test:w }
24460     {
24461         \__regex_if_within_catcode:TF
24462         {
24463             \exp_after:wN \__regex_compile_class_catcode:w
24464             \int_use:N \l__regex_catcodes_int ;
24465         }
24466         { \__regex_compile_class_normal:w }
24467     }
24468 }

```

(End definition for __regex_compile_[:.])

__regex_compile_class_normal:w In the “normal” case, we insert __regex_class:NnnnN *(boolean)* in the compiled code. The *(boolean)* is true for positive classes, and false for negative classes, characterized by a leading `^`. The auxiliary __regex_compile_class:TFNN also checks for a leading `]` which has a special meaning.

```

24469 \cs_new_protected:Npn \__regex_compile_class_normal:w
24470 {
24471     \__regex_compile_class:TFNN
24472     { \__regex_class:NnnnN \c_true_bool }
24473     { \__regex_class:NnnnN \c_false_bool }
24474 }

```

(End definition for __regex_compile_class_normal:w.)

__regex_compile_class_catcode:w This function is called for a left bracket in modes 2 or 6 (catcode test, and catcode test within a class). In mode 2 the whole construction needs to be put in a class (like single character). Then determine if the class is positive or negative, inserting __regex_item_catcode:nT or the reverse variant as appropriate, each with the current catcodes bitmap #1 as an argument, and reset the catcodes.

```

24475 \cs_new_protected:Npn \__regex_compile_class_catcode:w #1;
24476 {
24477     \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
24478     \tl_build_put_right:Nn \l__regex_build_tl
24479     { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
24480     \fi:
24481     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
24482     \__regex_compile_class:TFNN
24483     { \__regex_item_catcode:nT {#1} }
24484     { \__regex_item_catcode_reverse:nT {#1} }
24485 }

```

(End definition for __regex_compile_class_catcode:w.)

__regex_compile_class:TFNN If the first character is `^`, then the class is negative (use #2), otherwise it is positive (use #1). If the next character is a right bracket, then it should be changed to a raw one.

```

24486 \cs_new_protected:Npn \__regex_compile_class:TFNN #1#2#3#4
24487 {
24488     \l__regex_mode_int = \int_value:w \l__regex_mode_int 3 \exp_stop_f:
24489     \__regex_two_if_eq:NNNTF #3 #4 \__regex_compile_special:N ^
24490     {

```

```

24491         \tl_build_put_right:Nn \l__regex_build_tl { #2 { \if_false: } \fi: }
24492         \__regex_compile_class:NN
24493     }
24494     {
24495         \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
24496         \__regex_compile_class:NN #3 #4
24497     }
24498 }
24499 \cs_new_protected:Npn \__regex_compile_class:NN #1#2
24500 {
24501     \token_if_eq_charcode:NNTF #2 ]
24502     { \__regex_compile_raw:N #2 }
24503     { #1 #2 }
24504 }

```

(End definition for __regex_compile_class:TFNN and __regex_compile_class:NN.)

__regex_compile_class_posix_test:w Here we check for a syntax such as [:alpha:]. We also detect [= and [. which have a meaning in POSIX regular expressions, but are not implemented in l3regex. In case we see [:, grab raw characters until hopefully reaching :]. If that's missing, or the POSIX class is unknown, abort. If all is right, add the test to the current class, with an extra __regex_item_reverse:n for negative classes.

```

24505 \cs_new_protected:Npn \__regex_compile_class_posix_test:w #1#2
24506 {
24507     \token_if_eq_meaning:NNT \__regex_compile_special:N #1
24508     {
24509         \str_case:nn { #2 }
24510         {
24511             : { \__regex_compile_class_posix:NNNNw }
24512             = {
24513                 \__kernel_msg_warning:nxx { kernel }
24514                 { posix-unsupported } { = }
24515             }
24516             . {
24517                 \__kernel_msg_warning:nxx { kernel }
24518                 { posix-unsupported } { . }
24519             }
24520         }
24521     }
24522     \__regex_compile_raw:N [ #1 #2
24523 }
24524 \cs_new_protected:Npn \__regex_compile_class_posix:NNNNw #1#2#3#4#5#6
24525 {
24526     \__regex_two_if_eq:NNNTF #5 #6 \__regex_compile_special:N ^
24527     {
24528         \bool_set_false:N \l__regex_internal_bool
24529         \tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
24530         \__regex_compile_class_posix_loop:w
24531     }
24532     {
24533         \bool_set_true:N \l__regex_internal_bool
24534         \tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
24535         \__regex_compile_class_posix_loop:w #5 #6
24536     }

```



```

24537 }
24538 \cs_new:Npn \__regex_compile_class_posix_loop:w #1#2
24539 {
24540   \token_if_eq_meaning:NNTF \__regex_compile_raw:N #1
24541   { #2 \__regex_compile_class_posix_loop:w }
24542   { \if_false: { \fi: } \__regex_compile_class_posix_end:w #1 #2 }
24543 }
24544 \cs_new_protected:Npn \__regex_compile_class_posix_end:w #1#2#3#4
24545 {
24546   \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N :
24547   { \__regex_two_if_eq:NNNTF #3 #4 \__regex_compile_special:N ] }
24548   { \use_ii:nn }
24549   {
24550     \cs_if_exist:cTF { __regex_posix_ \l__regex_internal_a_tl : }
24551     {
24552       \__regex_compile_one:n
24553       {
24554         \bool_if:NF \l__regex_internal_bool \__regex_item_reverse:n
24555         \exp_not:c { __regex_posix_ \l__regex_internal_a_tl : }
24556       }
24557     }
24558     {
24559       \__kernel_msg_warning:nxx { kernel } { posix-unknown }
24560       { \l__regex_internal_a_tl }
24561       \__regex_compile_abort_tokens:x
24562       {
24563         [: \bool_if:NF \l__regex_internal_bool { ^ }
24564         \l__regex_internal_a_tl :]
24565       }
24566     }
24567   }
24568   {
24569     \__kernel_msg_error:nxxx { kernel } { posix-missing-close }
24570     { [: \l__regex_internal_a_tl ] { #2 #4 }
24571     \__regex_compile_abort_tokens:x { [: \l__regex_internal_a_tl ]
24572     #1 #2 #3 #4
24573   }
24574 }

```

(End definition for __regex_compile_class_posix_test:w and others.)

41.3.10 Groups and alternations

__regex_compile_group_begin:N The contents of a regex group are turned into compiled code in \l__regex_build_t1, which ends up with items of the form __regex_branch:n {⟨concatenation⟩}. This construction is done using \t1_build... functions within a T_EX group, which automatically makes sure that options (case-sensitivity and default catcode) are reset at the end of the group. The argument #1 is __regex_group:nnnN or a variant thereof. A small subtlety to support \cL(abc) as a shorthand for (\cLa\cLb\cLc): exit any pending catcode test, save the category code at the start of the group as the default catcode for that group, and make sure that the catcode is restored to the default outside the group.

```

24575 \cs_new_protected:Npn \__regex_compile_group_begin:N #1

```

```

24576 {
24577   \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
24578   \__regex_mode_quit_c:
24579   \group_begin:
24580     \tl_build_begin:N \l__regex_build_tl
24581     \int_set_eq:NN \l__regex_default_catcodes_int \l__regex_catcodes_int
24582     \int_incr:N \l__regex_group_level_int
24583     \tl_build_put_right:Nn \l__regex_build_tl
24584       { \__regex_branch:n { \if_false: } \fi: }
24585   }
24586 \cs_new_protected:Npn \__regex_compile_group_end:
24587 {
24588   \if_int_compare:w \l__regex_group_level_int > 0 \exp_stop_f:
24589     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
24590     \tl_build_end:N \l__regex_build_tl
24591     \exp_args:NNNx
24592     \group_end:
24593     \tl_build_put_right:Nn \l__regex_build_tl { \l__regex_build_tl }
24594     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
24595     \exp_after:wN \__regex_compile_quantifier:w
24596   \else:
24597     \__kernel_msg_warning:nn { kernel } { extra-rparen }
24598     \exp_after:wN \__regex_compile_raw:N \exp_after:wN )
24599   \fi:
24600 }

```

(End definition for __regex_compile_group_begin:N and __regex_compile_group_end:.)

__regex_compile(: In a class, parentheses are not special. In a catcode test inside a class, a left parenthesis gives an error, to catch [a\cL(bcd)e]. Otherwise check for a ?, denoting special groups, and run the code for the corresponding special group.

```

24601 \cs_new_protected:cpn { __regex_compile(: }
24602 {
24603   \__regex_if_in_class:TF { \__regex_compile_raw:N ( }
24604   {
24605     \if_int_compare:w \l__regex_mode_int =
24606       \c__regex_catcode_in_class_mode_int
24607       \__kernel_msg_error:nn { kernel } { c-lparen-in-class }
24608       \exp_after:wN \__regex_compile_raw:N \exp_after:wN (
24609     \else:
24610       \exp_after:wN \__regex_compile_lparen:w
24611     \fi:
24612   }
24613 }
24614 \cs_new_protected:Npn \__regex_compile_lparen:w #1#2#3#4
24615 {
24616   \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N ?
24617   {
24618     \cs_if_exist_use:cF
24619     { __regex_compile_special_group\token_to_str:N #4 :w }
24620     {
24621       \__kernel_msg_warning:nnx { kernel } { special-group-unknown }
24622       { (? #4 }
24623       \__regex_compile_group_begin:N \__regex_group:nnnN

```

```

24624         \_regex_compile_raw:N ? #3 #4
24625     }
24626 }
24627 {
24628     \_regex_compile_group_begin:N \_regex_group:nnnN
24629     #1 #2 #3 #4
24630 }
24631 }

```

(End definition for _regex_compile(:))

_regex_compile_|: In a class, the pipe is not special. Otherwise, end the current branch and open another one.

```

24632 \cs_new_protected:cpn { \_regex_compile_|: }
24633 {
24634     \_regex_if_in_class:TF { \_regex_compile_raw:N | }
24635     {
24636         \tl_build_put_right:Nn \l__regex_build_tl
24637         { \if_false: { \fi: } \_regex_branch:n { \if_false: } \fi: }
24638     }
24639 }

```

(End definition for _regex_compile/|:)

_regex_compile_): Within a class, parentheses are not special. Outside, close a group.

```

24640 \cs_new_protected:cpn { \_regex_compile_): }
24641 {
24642     \_regex_if_in_class:TF { \_regex_compile_raw:N ) }
24643     { \_regex_compile_group_end: }
24644 }

```

(End definition for _regex_compile_):)

_regex_compile_special_group::w Non-capturing, and resetting groups are easy to take care of during compilation; for those groups, the harder parts come when building.

```

24645 \cs_new_protected:cpn { \_regex_compile_special_group::w }
24646 { \_regex_compile_group_begin:N \_regex_group_no_capture:nnnN }
24647 \cs_new_protected:cpn { \_regex_compile_special_group_|:w }
24648 { \_regex_compile_group_begin:N \_regex_group_resetting:nnnN }

```

(End definition for _regex_compile_special_group::w and _regex_compile_special_group_|:w.)

_regex_compile_special_group_i:w The match can be made case-insensitive by setting the option with (?i); the original behaviour is restored by (?-i). This is the only supported option.

```

24649 \cs_new_protected:Npn \_regex_compile_special_group_i:w #1#2
24650 {
24651     \_regex_two_if_eq:NNNTF #1 #2 \_regex_compile_special:N )
24652     {
24653         \cs_set:Npn \_regex_item_equal:n
24654         { \_regex_item_caseless_equal:n }
24655         \cs_set:Npn \_regex_item_range:nn
24656         { \_regex_item_caseless_range:nn }
24657     }
24658     {
24659         \_kernel_msg_warning:nnx { kernel } { unknown-option } { (?i #2 }

```

```

24660     \_regex_compile_raw:N (
24661     \_regex_compile_raw:N ?
24662     \_regex_compile_raw:N i
24663     #1 #2
24664   }
24665 }
24666 \cs_new_protected:cpn { __regex_compile_special_group_-:w } #1#2#3#4
24667 {
24668   \_regex_two_if_eq:NNNTF #1 #2 \_regex_compile_raw:N i
24669   { \_regex_two_if_eq:NNNTF #3 #4 \_regex_compile_special:N ) }
24670   { \use_ii:nn }
24671   {
24672     \cs_set:Npn \_regex_item_equal:n
24673     { \_regex_item_caseful_equal:n }
24674     \cs_set:Npn \_regex_item_range:nn
24675     { \_regex_item_caseful_range:nn }
24676   }
24677   {
24678     \_kernel_msg_warning:nnx { kernel } { unknown-option } { (?-#2#4 }
24679     \_regex_compile_raw:N (
24680     \_regex_compile_raw:N ?
24681     \_regex_compile_raw:N -
24682     #1 #2 #3 #4
24683   }
24684 }

```

(End definition for _regex_compile_special_group_i:w and _regex_compile_special_group_-:w.)

41.3.11 Catcodes and csnames

_regex_compile_/c: The \c escape sequence can be followed by a capital letter representing a character category, by a left bracket which starts a list of categories, or by a brace group holding a regular expression for a control sequence name. Otherwise, raise an error.

```

24685 \cs_new_protected:cpn { __regex_compile_/c: }
24686 { \_regex_chk_c_allowed:T { \_regex_compile_c_test:NN } }
24687 \cs_new_protected:Npn \_regex_compile_c_test:NN #1#2
24688 {
24689   \token_if_eq_meaning:NNTF #1 \_regex_compile_raw:N
24690   {
24691     \int_if_exist:cTF { c__regex_catcode_#2_int }
24692     {
24693       \int_set_eq:Nc \l__regex_catcodes_int
24694       { c__regex_catcode_#2_int }
24695       \l__regex_mode_int
24696       = \if_case:w \l__regex_mode_int
24697       \c__regex_catcode_mode_int
24698       \else:
24699       \c__regex_catcode_in_class_mode_int
24700       \fi:
24701       \token_if_eq_charcode:NNT C #2 { \_regex_compile_c_C:NN }
24702     }
24703   }
24704   { \cs_if_exist_use:cF { __regex_compile_c_#2:w } }
24705   {

```

```

24706         \__kernel_msg_error:nxx { kernel } { c-missing-category } {#2}
24707         #1 #2
24708     }
24709 }

```

(End definition for __regex_compile_/c: and __regex_compile_c_test:NN.)

__regex_compile_c_C:NN If \cC is not followed by . or (...) then complain because that construction cannot match anything, except in cases like \cC[\c{...}], where it has no effect.

```

24710 \cs_new_protected:Npn \__regex_compile_c_C:NN #1#2
24711 {
24712     \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
24713     {
24714         \token_if_eq_charcode:NNTF #2 .
24715         { \use_none:n }
24716         { \token_if_eq_charcode:NNTF #2 ( } % )
24717     }
24718     { \use:n }
24719     { \__kernel_msg_error:nnn { kernel } { c-C-invalid } {#2} }
24720     #1 #2
24721 }

```

(End definition for __regex_compile_c_C:NN.)

__regex_compile_c_[:w When encountering \c[, the task is to collect uppercase letters representing character categories. First check for ^ which negates the list of category codes.

```

\__regex_compile_c_lbrack_loop:NN
\__regex_compile_c_lbrack_add:N
\__regex_compile_c_lbrack_end:
24722 \cs_new_protected:cpn { __regex_compile_c_[:w } #1#2
24723 {
24724     \l__regex_mode_int
24725     = \if_case:w \l__regex_mode_int
24726         \c__regex_catcode_mode_int
24727     \else:
24728         \c__regex_catcode_in_class_mode_int
24729     \fi:
24730     \int_zero:N \l__regex_catcodes_int
24731     \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N ^
24732     {
24733         \bool_set_false:N \l__regex_catcodes_bool
24734         \__regex_compile_c_lbrack_loop:NN
24735     }
24736     {
24737         \bool_set_true:N \l__regex_catcodes_bool
24738         \__regex_compile_c_lbrack_loop:NN
24739         #1 #2
24740     }
24741 }
24742 \cs_new_protected:Npn \__regex_compile_c_lbrack_loop:NN #1#2
24743 {
24744     \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
24745     {
24746         \int_if_exist:cTF { c__regex_catcode_#2_int }
24747         {
24748             \exp_args:Nc \__regex_compile_c_lbrack_add:N
24749             { c__regex_catcode_#2_int }

```

```

24750         \_regex_compile_c_lbrack_loop:NN
24751     }
24752 }
24753 {
24754     \token_if_eq_charcode:NNTF #2 ]
24755     { \_regex_compile_c_lbrack_end: }
24756 }
24757 {
24758     \_kernel_msg_error:nxx { kernel } { c-missing-rbrack } {#2}
24759     \_regex_compile_c_lbrack_end:
24760     #1 #2
24761 }
24762 }
24763 \cs_new_protected:Npn \_regex_compile_c_lbrack_add:N #1
24764 {
24765     \if_int_odd:w \int_eval:n { \l__regex_catcodes_int / #1 } \exp_stop_f:
24766     \else:
24767         \int_add:Nn \l__regex_catcodes_int {#1}
24768     \fi:
24769 }
24770 \cs_new_protected:Npn \_regex_compile_c_lbrack_end:
24771 {
24772     \if_meaning:w \c_false_bool \l__regex_catcodes_bool
24773     \int_set:Nn \l__regex_catcodes_int
24774     { \c__regex_all_catcodes_int - \l__regex_catcodes_int }
24775     \fi:
24776 }

```

(End definition for _regex_compile_c[:w and others.)

_regex_compile_c_{: The case of a left brace is easy, based on what we have done so far: in a group, compile the regular expression, after changing the mode to forbid nesting \c. Additionally, disable submatch tracking since groups don't escape the scope of \c{...}.

```

24777 \cs_new_protected:cpn { __regex_compile_c_ \c_left_brace_str :w }
24778 {
24779     \_regex_compile:w
24780     \_regex_disable_submatches:
24781     \l__regex_mode_int
24782     = \if_case:w \l__regex_mode_int
24783         \c__regex_cs_mode_int
24784     \else:
24785         \c__regex_cs_in_class_mode_int
24786     \fi:
24787 }

```

(End definition for _regex_compile_c_{:.)

_regex_compile_}: Non-escaped right braces are only special if they appear when compiling the regular expression for a csname, but not within a class: \c{[{}]} matches the control sequences \{ and \}. So, end compiling the inner regex (this closes any dangling class or group). Then insert the corresponding test in the outer regex. As an optimization, if the control sequence test simply consists of several explicit possibilities (branches) then use _regex_item_exact_cs:n with an argument consisting of all possibilities separated by \scan_stop:.

```

24788 \flag_new:n { __regex_cs }
24789 \cs_new_protected:cpn { __regex_compile_ \c_right_brace_str : }
24790 {
24791   \__regex_if_in_cs:TF
24792   { \__regex_compile_end_cs: }
24793   { \exp_after:wN \__regex_compile_raw:N \c_right_brace_str }
24794 }
24795 \cs_new_protected:Npn \__regex_compile_end_cs:
24796 {
24797   \__regex_compile_end:
24798   \flag_clear:n { __regex_cs }
24799   \tl_set:Nx \l__regex_internal_a_tl
24800   {
24801     \exp_after:wN \__regex_compile_cs_aux:Nn \l__regex_internal_regex
24802     \q__regex_nil \q__regex_nil \q__regex_recursion_stop
24803   }
24804   \exp_args:Nx \__regex_compile_one:n
24805   {
24806     \flag_if_raised:nTF { __regex_cs }
24807     { \__regex_item_cs:n { \exp_not:o \l__regex_internal_regex } }
24808     {
24809       \__regex_item_exact_cs:n
24810       { \tl_tail:N \l__regex_internal_a_tl }
24811     }
24812   }
24813 }
24814 \cs_new:Npn \__regex_compile_cs_aux:Nn #1#2
24815 {
24816   \cs_if_eq:NNTF #1 \__regex_branch:n
24817   {
24818     \scan_stop:
24819     \__regex_compile_cs_aux:NNnnN #2
24820     \q__regex_nil \q__regex_nil \q__regex_nil
24821     \q__regex_nil \q__regex_nil \q__regex_nil \q__regex_recursion_stop
24822     \__regex_compile_cs_aux:Nn
24823   }
24824   {
24825     \__regex_quark_if_nil:NF #1 { \flag_raise_if_clear:n { __regex_cs } }
24826     \__regex_use_none_delimit_by_q_recursion_stop:w
24827   }
24828 }
24829 \cs_new:Npn \__regex_compile_cs_aux:NNnnN #1#2#3#4#5#6
24830 {
24831   \bool_lazy_all:nTF
24832   {
24833     { \cs_if_eq_p:NN #1 \__regex_class:NnnN }
24834     {#2}
24835     { \tl_if_head_eq_meaning_p:nN {#3} \__regex_item_caseful_equal:n }
24836     { \int_compare_p:nNn { \tl_count:n {#3} } = { 2 } }
24837     { \int_compare_p:nNn {#5} = { 0 } }
24838   }
24839   {
24840     \prg_replicate:nn {#4}
24841     { \char_generate:nn { \use_ii:nn #3 } {12} }

```

```

24842     \_regex_compile_cs_aux:NNnnnN
24843   }
24844   {
24845     \_regex_quark_if_nil:NF #1
24846     {
24847       \flag_raise_if_clear:n { \_regex_cs }
24848       \_regex_use_i_delimit_by_q_recursion_stop:nw
24849     }
24850     \_regex_use_none_delimit_by_q_recursion_stop:w
24851   }
24852 }

```

(End definition for _regex_compile_}: and others.)

41.3.12 Raw token lists with \u

_regex_compile_/u: The \u escape is invalid in classes and directly following a catcode test. Otherwise, it must be followed by a left brace. We then collect the characters for the argument of \u within an x-expanding assignment. In principle we could just wait to encounter a right brace, but this is unsafe: if the right brace was missing, then we would reach the end-markers of the regex, and continue, leading to obscure fatal errors. Instead, we only allow raw and special characters, and stop when encountering a special right brace, any escaped character, or the end-marker.

```

24853 \cs_new_protected:cpn { \_regex_compile_/u: } #1#2
24854   {
24855     \_regex_if_in_class_or_catcode:TF
24856     { \_regex_compile_raw_error:N u #1 #2 }
24857     {
24858       \_regex_two_if_eq:NNNTF #1 #2 \_regex_compile_special:N \c_left_brace_str
24859       {
24860         \tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
24861         \_regex_compile_u_loop:NN
24862       }
24863       {
24864         \__kernel_msg_error:nn { kernel } { u-missing-lbrace }
24865         \_regex_compile_raw:N u #1 #2
24866       }
24867     }
24868   }
24869 \cs_new:Npn \_regex_compile_u_loop:NN #1#2
24870   {
24871     \token_if_eq_meaning:NNTF #1 \_regex_compile_raw:N
24872     { #2 \_regex_compile_u_loop:NN }
24873     {
24874       \token_if_eq_meaning:NNTF #1 \_regex_compile_special:N
24875       {
24876         \exp_after:wN \token_if_eq_charcode:NNTF \c_right_brace_str #2
24877         { \if_false: { \fi: } \_regex_compile_u_end: }
24878         { #2 \_regex_compile_u_loop:NN }
24879       }
24880       {
24881         \if_false: { \fi: }
24882         \__kernel_msg_error:nxx { kernel } { u-missing-rbrace } {#2}
24883         \_regex_compile_u_end:

```



```

24884         #1 #2
24885     }
24886 }
24887 }

```

(End definition for `_regex_compile/u:` and `_regex_compile_u_loop:NN`.)

`_regex_compile_u_end:` Once we have extracted the variable's name, we store the contents of that variable in `\l_regex_internal_a_tl`. The behaviour of `\u` then depends on whether we are within a `\c{...}` escape (in this case, the variable is turned to a string), or not.

```

24888 \cs_new_protected:Npn \_regex_compile_u_end:
24889 {
24890     \tl_set:Nv \l\_regex_internal_a_tl { \l\_regex_internal_a_tl }
24891     \if_int_compare:w \l\_regex_mode_int = \c\_regex_outer_mode_int
24892         \_regex_compile_u_not_cs:
24893     \else:
24894         \_regex_compile_u_in_cs:
24895     \fi:
24896 }

```

(End definition for `_regex_compile_u_end:.`)

`_regex_compile_u_in_cs:` When `\u` appears within a control sequence, we convert the variable to a string with escaped spaces. Then for each character insert a class matching exactly that character, once.

```

24897 \cs_new_protected:Npn \_regex_compile_u_in_cs:
24898 {
24899     \tl_gset:Nx \g\_regex_internal_tl
24900     {
24901         \exp_args:No \__kernel_str_to_other_fast:n
24902         { \l\_regex_internal_a_tl }
24903     }
24904     \tl_build_put_right:Nx \l\_regex_build_tl
24905     {
24906         \tl_map_function:NN \g\_regex_internal_tl
24907         \_regex_compile_u_in_cs_aux:n
24908     }
24909 }
24910 \cs_new:Npn \_regex_compile_u_in_cs_aux:n #1
24911 {
24912     \_regex_class:NnnN \c_true_bool
24913     { \_regex_item_caseful_equal:n { \int_value:w '#1 } }
24914     { 1 } { 0 } \c_false_bool
24915 }

```

(End definition for `_regex_compile_u_in_cs:.`)

`_regex_compile_u_not_cs:` In mode 0, the `\u` escape adds one state to the NFA for each token in `\l_regex_internal_a_tl`. If a given *<token>* is a control sequence, then insert a string comparison test, otherwise, `_regex_item_exact:nn` which compares catcode and character code.

```

24916 \cs_new_protected:Npn \_regex_compile_u_not_cs:
24917 {
24918     \tl_analysis_map_inline:Nn \l\_regex_internal_a_tl
24919     {

```

```

24920 \tl_build_put_right:Nx \l__regex_build_tl
24921 {
24922   \__regex_class:NnnnN \c_true_bool
24923   {
24924     \if_int_compare:w "##3 = 0 \exp_stop_f:
24925       \__regex_item_exact_cs:n
24926       { \exp_after:wN \cs_to_str:N ##1 }
24927     \else:
24928       \__regex_item_exact:nn { \int_value:w "##3 } { ##2 }
24929     \fi:
24930   }
24931   { 1 } { 0 } \c_false_bool
24932 }
24933 }
24934 }

```

(End definition for __regex_compile_u_not_cs:.)

41.3.13 Other

__regex_compile_/K: The \K control sequence is currently the only “command”, which performs some action, rather than matching something. It is allowed in the same contexts as \b. At the compilation stage, we leave it as a single control sequence, defined later.

```

24935 \cs_new_protected:cpn { __regex_compile_/K: }
24936 {
24937   \int_compare:nNnTF \l__regex_mode_int = \c__regex_outer_mode_int
24938     { \tl_build_put_right:Nn \l__regex_build_tl { \__regex_command_K: } }
24939     { \__regex_compile_raw_error:N K }
24940 }

```

(End definition for __regex_compile_/K:.)

41.3.14 Showing regexes

__regex_show:N Within a group and within \tl_build_begin:N ... \tl_build_end:N we redefine all the function that can appear in a compiled regex, then run the regex. The result stored in \l__regex_internal_a_tl is then meant to be shown.

```

24941 \cs_new_protected:Npn \__regex_show:N #1
24942 {
24943   \group_begin:
24944     \tl_build_begin:N \l__regex_build_tl
24945     \cs_set_protected:Npn \__regex_branch:n
24946     {
24947       \seq_pop_right:NN \l__regex_show_prefix_seq
24948       \l__regex_internal_a_tl
24949       \__regex_show_one:n { +-branch }
24950       \seq_put_right:No \l__regex_show_prefix_seq
24951       \l__regex_internal_a_tl
24952     }
24953     \use:n
24954     \cs_set_protected:Npn \__regex_group:nnnN
24955     { \__regex_show_group_aux:nnnnN { } }
24956     \cs_set_protected:Npn \__regex_group_no_capture:nnnN
24957     { \__regex_show_group_aux:nnnnN { ~(no~capture) } }

```

```

24958 \cs_set_protected:Npn \__regex_group_resetting:nnnN
24959 { \__regex_show_group_aux:nnnnN { ~(resetting) } }
24960 \cs_set_eq:NN \__regex_class:NnnnN \__regex_show_class:NnnnN
24961 \cs_set_protected:Npn \__regex_command_K:
24962 { \__regex_show_one:n { reset~match~start~(\iow_char:N\K) } }
24963 \cs_set_protected:Npn \__regex_assertion:Nn ##1##2
24964 {
24965   \__regex_show_one:n
24966   { \bool_if:NF ##1 { negative~ } assertion:~##2 }
24967 }
24968 \cs_set:Npn \__regex_b_test: { word~boundary }
24969 \cs_set_eq:NN \__regex_anchor:N \__regex_show_anchor_to_str:N
24970 \cs_set_protected:Npn \__regex_item_caseful_equal:n ##1
24971 { \__regex_show_one:n { char~code~\int_eval:n{##1} } }
24972 \cs_set_protected:Npn \__regex_item_caseful_range:nn ##1##2
24973 {
24974   \__regex_show_one:n
24975   { range~[\int_eval:n{##1}, \int_eval:n{##2}] }
24976 }
24977 \cs_set_protected:Npn \__regex_item_caseless_equal:n ##1
24978 { \__regex_show_one:n { char~code~\int_eval:n{##1}~(caseless) } }
24979 \cs_set_protected:Npn \__regex_item_caseless_range:nn ##1##2
24980 {
24981   \__regex_show_one:n
24982   { Range~[\int_eval:n{##1}, \int_eval:n{##2}]~(caseless) }
24983 }
24984 \cs_set_protected:Npn \__regex_item_catcode:nT
24985 { \__regex_show_item_catcode:NnT \c_true_bool }
24986 \cs_set_protected:Npn \__regex_item_catcode_reverse:nT
24987 { \__regex_show_item_catcode:NnT \c_false_bool }
24988 \cs_set_protected:Npn \__regex_item_reverse:n
24989 { \__regex_show_scope:nn { Reversed~match } }
24990 \cs_set_protected:Npn \__regex_item_exact:nn ##1##2
24991 { \__regex_show_one:n { char~##2,~catcode~##1 } }
24992 \cs_set_eq:NN \__regex_item_exact_cs:n \__regex_show_item_exact_cs:n
24993 \cs_set_protected:Npn \__regex_item_cs:n
24994 { \__regex_show_scope:nn { control~sequence } }
24995 \cs_set:cpn { __regex_prop_.. } { \__regex_show_one:n { any~token } }
24996 \seq_clear:N \l__regex_show_prefix_seq
24997 \__regex_show_push:n { ~ }
24998 \cs_if_exist_use:N #1
24999 \tl_build_end:N \l__regex_build_tl
25000 \exp_args:NNNo
25001 \group_end:
25002 \tl_set:Nn \l__regex_internal_a_tl { \l__regex_build_tl }
25003 }

```

(End definition for __regex_show:N.)

__regex_show_one:n Every part of the final message go through this function, which adds one line to the output, with the appropriate prefix.

```

25004 \cs_new_protected:Npn \__regex_show_one:n #1
25005 {
25006   \int_incr:N \l__regex_show_lines_int

```

```

25007 \tl_build_put_right:Nx \l__regex_build_tl
25008 {
25009   \exp_not:N \iow_newline:
25010   \seq_map_function:NN \l__regex_show_prefix_seq \use:n
25011   #1
25012 }
25013 }

```

(End definition for __regex_show_one:n.)

__regex_show_push:n Enter and exit levels of nesting. The scope function prints its first argument as an “introduction”, then performs its second argument in a deeper level of nesting.
 __regex_show_pop:
 __regex_show_scope:nn

```

25014 \cs_new_protected:Npn \__regex_show_push:n #1
25015 { \seq_put_right:Nx \l__regex_show_prefix_seq { #1 ~ } }
25016 \cs_new_protected:Npn \__regex_show_pop:
25017 { \seq_pop_right:NN \l__regex_show_prefix_seq \l__regex_internal_a_tl }
25018 \cs_new_protected:Npn \__regex_show_scope:nn #1#2
25019 {
25020   \__regex_show_one:n {#1}
25021   \__regex_show_push:n { ~ }
25022   #2
25023   \__regex_show_pop:
25024 }

```

(End definition for __regex_show_push:n, __regex_show_pop:, and __regex_show_scope:nn.)

__regex_show_group_aux:nnnnN We display all groups in the same way, simply adding a message, (no capture) or (resetting), to special groups. The odd \use_ii:nn avoids printing a spurious +-branch for the first branch.

```

25025 \cs_new_protected:Npn \__regex_show_group_aux:nnnnN #1#2#3#4#5
25026 {
25027   \__regex_show_one:n { , -group~begin #1 }
25028   \__regex_show_push:n { | }
25029   \use_ii:nn #2
25030   \__regex_show_pop:
25031   \__regex_show_one:n
25032   { ‘-group~end \__regex_msg_repeated:nnN {#3} {#4} #5 }
25033 }

```

(End definition for __regex_show_group_aux:nnnnN.)

__regex_show_class:NnnnN I’m entirely unhappy about this function: I couldn’t find a way to test if a class is a single test. Instead, collect the representation of the tests in the class. If that had more than one line, write Match or Don’t match on its own line, with the repeating information if any. Then the various tests on lines of their own, and finally a line. Otherwise, we need to evaluate the representation of the tests again (since the prefix is incorrect). That’s clunky, but not too expensive, since it’s only one test.

```

25034 \cs_set:Npn \__regex_show_class:NnnnN #1#2#3#4#5
25035 {
25036   \group_begin:
25037   \tl_build_begin:N \l__regex_build_tl
25038   \int_zero:N \l__regex_show_lines_int
25039   \__regex_show_push:n {~}
25040   #2

```

```

25041 \int_compare:nTF { \l__regex_show_lines_int = 0 }
25042 {
25043   \group_end:
25044   \__regex_show_one:n { \bool_if:NTF #1 { Fail } { Pass } }
25045 }
25046 {
25047   \bool_if:nTF
25048   { #1 && \int_compare_p:n { \l__regex_show_lines_int = 1 } }
25049   {
25050     \group_end:
25051     #2
25052     \tl_build_put_right:Nn \l__regex_build_tl
25053     { \__regex_msg_repeated:nnN {#3} {#4} #5 }
25054   }
25055   {
25056     \tl_build_end:N \l__regex_build_tl
25057     \exp_args:NNNo
25058     \group_end:
25059     \tl_set:Nn \l__regex_internal_a_tl \l__regex_build_tl
25060     \__regex_show_one:n
25061     {
25062       \bool_if:NTF #1 { Match } { Don't-match }
25063       \__regex_msg_repeated:nnN {#3} {#4} #5
25064     }
25065     \tl_build_put_right:Nx \l__regex_build_tl
25066     { \exp_not:o \l__regex_internal_a_tl }
25067   }
25068 }
25069 }

```

(End definition for __regex_show_class:NnnnN.)

__regex_show_anchor_to_str:N The argument is an integer telling us where the anchor is. We convert that to the relevant info.

```

25070 \cs_new:Npn \__regex_show_anchor_to_str:N #1
25071 {
25072   anchor~at~
25073   \str_case:nnF { #1 }
25074   {
25075     { \l__regex_min_pos_int } { start~(\iow_char:N\\A) }
25076     { \l__regex_start_pos_int } { start~of~match~(\iow_char:N\\G) }
25077     { \l__regex_max_pos_int } { end~(\iow_char:N\\Z) }
25078   }
25079   { <error::~#1'~not~recognized> }
25080 }

```

(End definition for __regex_show_anchor_to_str:N.)

__regex_show_item_catcode:NnT Produce a sequence of categories which the catcode bitmap #2 contains, and show it, indenting the tests on which this catcode constraint applies.

```

25081 \cs_new_protected:Npn \__regex_show_item_catcode:NnT #1#2
25082 {
25083   \seq_set_split:Nnn \l__regex_internal_seq { } { CBEMTPUDSLOA }
25084   \seq_set_filter:NNn \l__regex_internal_seq \l__regex_internal_seq

```

```

25085     { \int_if_odd_p:n { #2 / \int_use:c { c__regex_catcode_##1_int } } }
25086 \__regex_show_scope:nn
25087 {
25088     categories~
25089     \seq_map_function:NN \l__regex_internal_seq \use:n
25090     , ~
25091     \bool_if:NF #1 { negative~ } class
25092 }
25093 }

```

(End definition for __regex_show_item_catcode:NnT.)

__regex_show_item_exact_cs:n

```

25094 \cs_new_protected:Npn \__regex_show_item_exact_cs:n #1
25095 {
25096     \seq_set_split:Nnn \l__regex_internal_seq { \scan_stop: } {#1}
25097     \seq_set_map_x:Nnn \l__regex_internal_seq
25098         \l__regex_internal_seq { \iow_char:N\##1 }
25099     \__regex_show_one:n
25100     { control~sequence~ \seq_use:Nn \l__regex_internal_seq { ~or~ } }
25101 }

```

(End definition for __regex_show_item_exact_cs:n.)

41.4 Building

41.4.1 Variables used while building

\l__regex_min_state_int
\l__regex_max_state_int

The last state that was allocated is `\l__regex_max_state_int - 1`, so that `\l__regex_max_state_int` always points to a free state. The `min_state` variable is 1 to begin with, but gets shifted in nested calls to the matching code, namely in `\c{...}` constructions.

```

25102 \int_new:N \l__regex_min_state_int
25103 \int_set:Nn \l__regex_min_state_int { 1 }
25104 \int_new:N \l__regex_max_state_int

```

(End definition for \l__regex_min_state_int and \l__regex_max_state_int.)

\l__regex_left_state_int
\l__regex_right_state_int
\l__regex_left_state_seq
\l__regex_right_state_seq

Alternatives are implemented by branching from a `left` state into the various choices, then merging those into a `right` state. We store information about those states in two sequences. Those states are also used to implement group quantifiers. Most often, the left and right pointers only differ by 1.

```

25105 \int_new:N \l__regex_left_state_int
25106 \int_new:N \l__regex_right_state_int
25107 \seq_new:N \l__regex_left_state_seq
25108 \seq_new:N \l__regex_right_state_seq

```

(End definition for \l__regex_left_state_int and others.)

\l__regex_capturing_group_int

`\l__regex_capturing_group_int` is the next ID number to be assigned to a capturing group. This starts at 0 for the group enclosing the full regular expression, and groups are counted in the order of their left parenthesis, except when encountering `resetting` groups.

```

25109 \int_new:N \l__regex_capturing_group_int

```

(End definition for \l__regex_capturing_group_int.)

41.4.2 Framework

This phase is about going from a compiled regex to an NFA. Each state of the NFA is stored in a `\toks`. The operations which can appear in the `\toks` are

- `__regex_action_start_wildcard`: inserted at the start of the regular expression to make it unanchored.
- `__regex_action_success`: marks the exit state of the NFA.
- `__regex_action_cost:n {\langle shift \rangle}` is a transition from the current $\langle state \rangle$ to $\langle state \rangle + \langle shift \rangle$, which consumes the current character: the target state is saved and will be considered again when matching at the next position.
- `__regex_action_free:n {\langle shift \rangle}`, and `__regex_action_free_group:n {\langle shift \rangle}` are free transitions, which immediately perform the actions for the state $\langle state \rangle + \langle shift \rangle$ of the NFA. They differ in how they detect and avoid infinite loops. For now, we just need to know that the `group` variant must be used for transitions back to the start of a group.
- `__regex_action_submatch:n {\langle key \rangle}` where the $\langle key \rangle$ is a group number followed by `<` or `>` for the beginning or end of group. This causes the current position in the query to be stored as the $\langle key \rangle$ submatch boundary.

We strive to preserve the following properties while building.

- The current capturing group is `capturing_group - 1`, and if a group opened now it would be labelled `capturing_group`.
- The last allocated state is `max_state - 1`, so `max_state` is a free state.
- The `left_state` points to a state to the left of the current group or of the last class.
- The `right_state` points to a newly created, empty state, with some transitions leading to it.
- The `left/right` sequences hold a list of the corresponding end-points of nested groups.

`__regex_build:n` The `n`-type function first compiles its argument. Reset some variables. Allocate two states, and put a wildcard in state 0 (transitions to state 1 and 0 state). Then build the regex within a (capturing) group numbered 0 (current value of `capturing_group`). Finally, if the match reaches the last state, it is successful.

```
\__regex_build:N
25110 \cs_new_protected:Npn \__regex_build:n #1
25111 {
25112   \__regex_compile:n {#1}
25113   \__regex_build:N \l__regex_internal_regex
25114 }
25115 \cs_new_protected:Npn \__regex_build:N #1
25116 {
25117   \__regex_standard_escapechar:
25118   \int_zero:N \l__regex_capturing_group_int
25119   \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
25120   \__regex_build_new_state:
```

```

25121     \__regex_build_new_state:
25122     \__regex_toks_put_right:Nn \l__regex_left_state_int
25123         { \__regex_action_start_wildcard: }
25124     \__regex_group:nnnN {#1} { 1 } { 0 } \c_false_bool
25125     \__regex_toks_put_right:Nn \l__regex_right_state_int
25126         { \__regex_action_success: }
25127 }

```

(End definition for __regex_build:n and __regex_build:N.)

__regex_build_for_cs:n The matching code relies on some global intarray variables, but only uses a range of their entries. Specifically,

- \g__regex_state_active_intarray from \l__regex_min_state_int to \l__regex_max_state_int;
- \g__regex_thread_state_intarray from \l__regex_min_active_int to \l__regex_max_active_int.

In fact, some data is stored in \toks registers (local) in the same ranges so these ranges mustn't overlap. This is done by setting \l__regex_min_active_int to \l__regex_max_state_int after building the NFA. Here, in this nested call to the matching code, we need the new versions of these ranges to involve completely new entries of the intarray variables, so we begin by setting (the new) \l__regex_min_state_int to (the old) \l__regex_max_active_int to use higher entries.

When using a regex to match a cs, we don't insert a wildcard, we anchor at the end, and since we ignore submatches, there is no need to surround the expression with a group. However, for branches to work properly at the outer level, we need to put the appropriate left and right states in their sequence.

```

25128 \cs_new_protected:Npn \__regex_build_for_cs:n #1
25129 {
25130     \int_set_eq:NN \l__regex_min_state_int \l__regex_max_active_int
25131     \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
25132     \__regex_build_new_state:
25133     \__regex_build_new_state:
25134     \__regex_push_lr_states:
25135     #1
25136     \__regex_pop_lr_states:
25137     \__regex_toks_put_right:Nn \l__regex_right_state_int
25138     {
25139         \if_int_compare:w \l__regex_curr_pos_int = \l__regex_max_pos_int
25140             \exp_after:wN \__regex_action_success:
25141         \fi:
25142     }
25143 }

```

(End definition for __regex_build_for_cs:n.)

41.4.3 Helpers for building an nfa

__regex_push_lr_states: When building the regular expression, we keep track of pointers to the left-end and right-end of each group without help from T_EX's grouping.

```

25144 \cs_new_protected:Npn \__regex_push_lr_states:
25145 {

```



```

25146     \seq_push:No \l__regex_left_state_seq
25147     { \int_use:N \l__regex_left_state_int }
25148     \seq_push:No \l__regex_right_state_seq
25149     { \int_use:N \l__regex_right_state_int }
25150   }
25151   \cs_new_protected:Npn \__regex_pop_lr_states:
25152   {
25153     \seq_pop:N \l__regex_left_state_seq \l__regex_internal_a_tl
25154     \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
25155     \seq_pop:N \l__regex_right_state_seq \l__regex_internal_a_tl
25156     \int_set:Nn \l__regex_right_state_int \l__regex_internal_a_tl
25157   }

```

(End definition for __regex_push_lr_states: and __regex_pop_lr_states:.)

__regex_build_transition_left:NNN Add a transition from #2 to #3 using the function #1. The left function is used for higher priority transitions, and the right function for lower priority transitions (which should be performed later). The signatures differ to reflect the differing usage later on. Both functions could be optimized.

```

25158   \cs_new_protected:Npn \__regex_build_transition_left:NNN #1#2#3
25159   { \__regex_toks_put_left:Nx #2 { #1 { \int_eval:n { #3 - #2 } } } }
25160   \cs_new_protected:Npn \__regex_build_transition_right:nNn #1#2#3
25161   { \__regex_toks_put_right:Nx #2 { #1 { \int_eval:n { #3 - #2 } } } }

```

(End definition for __regex_build_transition_left:NNN and __regex_build_transition_right:nNn.)

__regex_build_new_state: Add a new empty state to the NFA. Then update the left, right, and max states, so that the right state is the new empty state, and the left state points to the previously “current” state.

```

25162   \cs_new_protected:Npn \__regex_build_new_state:
25163   {
25164     \__regex_toks_clear:N \l__regex_max_state_int
25165     \int_set_eq:NN \l__regex_left_state_int \l__regex_right_state_int
25166     \int_set_eq:NN \l__regex_right_state_int \l__regex_max_state_int
25167     \int_incr:N \l__regex_max_state_int
25168   }

```

(End definition for __regex_build_new_state:.)

__regex_build_transitions_lazyness:NNNNN This function creates a new state, and puts two transitions starting from the old current state. The order of the transitions is controlled by #1, true for lazy quantifiers, and false for greedy quantifiers.

```

25169   \cs_new_protected:Npn \__regex_build_transitions_lazyness:NNNNN #1#2#3#4#5
25170   {
25171     \__regex_build_new_state:
25172     \__regex_toks_put_right:Nx \l__regex_left_state_int
25173     {
25174       \if_meaning:w \c_true_bool #1
25175       #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
25176       #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
25177     }
25178     \else:
25179     #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
25179     #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
25180   }

```

```

25181     }
25182 }

```

(End definition for `_regex_build_transitions_lazyness:NNNNN`.)

41.4.4 Building classes

`_regex_class:NnnnN` The arguments are: $\langle \text{boolean} \rangle$ $\{\langle \text{tests} \rangle\}$ $\{\langle \text{min} \rangle\}$ $\{\langle \text{more} \rangle\}$ $\langle \text{lazyness} \rangle$. First store the tests with a trailing `_regex_action_cost:n`, in the true branch of `_regex_break_point:TF` for positive classes, or the false branch for negative classes. The integer $\langle \text{more} \rangle$ is 0 for fixed repetitions, -1 for unbounded repetitions, and $\langle \text{max} \rangle - \langle \text{min} \rangle$ for a range of repetitions.

```

25183 \cs_new_protected:Npn \_regex_class:NnnnN #1#2#3#4#5
25184 {
25185   \cs_set:Npx \_regex_tests_action_cost:n ##1
25186   {
25187     \exp_not:n { \exp_not:n {#2} }
25188     \bool_if:NTF #1
25189       { \_regex_break_point:TF { \_regex_action_cost:n {##1} } { } }
25190       { \_regex_break_point:TF { } { \_regex_action_cost:n {##1} } }
25191   }
25192   \if_case:w - #4 \exp_stop_f:
25193     \_regex_class_repeat:n {#3}
25194   \or: \_regex_class_repeat:nN {#3} #5
25195   \else: \_regex_class_repeat:nnN {#3} {#4} #5
25196   \fi:
25197 }
25198 \cs_new:Npn \_regex_tests_action_cost:n { \_regex_action_cost:n }

```

(End definition for `_regex_class:NnnnN` and `_regex_tests_action_cost:n`.)

`_regex_class_repeat:n` This is used for a fixed number of repetitions. Build one state for each repetition, with a transition controlled by the tests that we have collected. That works just fine for $\#1 = 0$ repetitions: nothing is built.

```

25199 \cs_new_protected:Npn \_regex_class_repeat:n #1
25200 {
25201   \prg_replicate:nn {#1}
25202   {
25203     \_regex_build_new_state:
25204     \_regex_build_transition_right:nNn \_regex_tests_action_cost:n
25205     \l__regex_left_state_int \l__regex_right_state_int
25206   }
25207 }

```

(End definition for `_regex_class_repeat:n`.)

`_regex_class_repeat:nN` This implements unbounded repetitions of a single class (e.g. the $*$ and $+$ quantifiers). If the minimum number $\#1$ of repetitions is 0, then build a transition from the current state to itself governed by the tests, and a free transition to a new state (hence skipping the tests). Otherwise, call `_regex_class_repeat:n` for the code to match $\#1$ repetitions, and add free transitions from the last state to the previous one, and to a new one. In both cases, the order of transitions is controlled by the lazyness boolean $\#2$.

```

25208 \cs_new_protected:Npn \_regex_class_repeat:nN #1#2
25209 {

```

```

25210 \if_int_compare:w #1 = 0 \exp_stop_f:
25211 \__regex_build_transitions_lazyness:NNNN #2
25212 \__regex_action_free:n \l__regex_right_state_int
25213 \__regex_tests_action_cost:n \l__regex_left_state_int
25214 \else:
25215 \__regex_class_repeat:n {#1}
25216 \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
25217 \__regex_build_transitions_lazyness:NNNN #2
25218 \__regex_action_free:n \l__regex_right_state_int
25219 \__regex_action_free:n \l__regex_internal_a_int
25220 \fi:
25221 }

```

(End definition for __regex_class_repeat:nN.)

__regex_class_repeat:nnN We want to build the code to match from #1 to #1 + #2 repetitions. Match #1 repetitions (can be 0). Compute the final state of the next construction as a. Build #2 > 0 states, each with a transition to the next state governed by the tests, and a transition to the final state a. The computation of a is safe because states are allocated in order, starting from max_state.

```

25222 \cs_new_protected:Npn \__regex_class_repeat:nnN #1#2#3
25223 {
25224 \__regex_class_repeat:n {#1}
25225 \int_set:Nn \l__regex_internal_a_int
25226 { \l__regex_max_state_int + #2 - 1 }
25227 \prg_replicate:nn { #2 }
25228 {
25229 \__regex_build_transitions_lazyness:NNNN #3
25230 \__regex_action_free:n \l__regex_internal_a_int
25231 \__regex_tests_action_cost:n \l__regex_right_state_int
25232 }
25233 }

```

(End definition for __regex_class_repeat:nnN.)

41.4.5 Building groups

__regex_group_aux:nnnnN Arguments: {<label>} {<contents>} {<min>} {<more>} <lazyness>. If <min> is 0, we need to add a state before building the group, so that the thread which skips the group does not also set the start-point of the submatch. After adding one more state, the left_state is the left end of the group, from which all branches stem, and the right_state is the right end of the group, and all branches end their course in that state. We store those two integers to be queried for each branch, we build the NFA states for the contents #2 of the group, and we forget about the two integers. Once this is done, perform the repetition: either exactly #3 times, or #3 or more times, or between #3 and #3 + #4 times, with lazyness #5. The <label> #1 is used for submatch tracking. Each of the three auxiliaries expects left_state and right_state to be set properly.

```

25234 \cs_new_protected:Npn \__regex_group_aux:nnnnN #1#2#3#4#5
25235 {
25236 \if_int_compare:w #3 = 0 \exp_stop_f:
25237 \__regex_build_new_state:
25238 (assert)\assert_int:n { \l__regex_max_state_int = \l__regex_right_state_int + 1 }
25239 \__regex_build_transition_right:nNn \__regex_action_free_group:n

```

```

25240         \l__regex_left_state_int \l__regex_right_state_int
25241     \fi:
25242     \__regex_build_new_state:
25243     \__regex_push_lr_states:
25244     #2
25245     \__regex_pop_lr_states:
25246     \if_case:w - #4 \exp_stop_f:
25247         \__regex_group_repeat:nn {#1} {#3}
25248     \or: \__regex_group_repeat:nnN {#1} {#3} #5
25249     \else: \__regex_group_repeat:nnnN {#1} {#3} {#4} #5
25250     \fi:
25251 }

```

(End definition for __regex_group_aux:nnnnN.)

__regex_group:nnnN Hand to __regex_group_aux:nnnnN the label of that group (expanded), and the group itself, with some extra commands to perform.

__regex_group_no_capture:nnnN

```

25252 \cs_new_protected:Npn \__regex_group:nnnN #1
25253 {
25254     \exp_args:Nn \__regex_group_aux:nnnnN
25255     { \int_use:N \l__regex_capturing_group_int }
25256     {
25257         \int_incr:N \l__regex_capturing_group_int
25258         #1
25259     }
25260 }
25261 \cs_new_protected:Npn \__regex_group_no_capture:nnnN
25262 { \__regex_group_aux:nnnnN { -1 } }

```

(End definition for __regex_group:nnnN and __regex_group_no_capture:nnnN.)

__regex_group_resetting:nnnN
__regex_group_resetting_loop:nnNn

Again, hand the label `-1` to __regex_group_aux:nnnnN, but this time we work a little bit harder to keep track of the maximum group label at the end of any branch, and to reset the group number at each branch. This relies on the fact that a compiled regex always is a sequence of items of the form __regex_branch:n {<branch>}.

```

25263 \cs_new_protected:Npn \__regex_group_resetting:nnnN #1
25264 {
25265     \__regex_group_aux:nnnnN { -1 }
25266     {
25267         \exp_args:Noo \__regex_group_resetting_loop:nnNn
25268         { \int_use:N \l__regex_capturing_group_int }
25269         { \int_use:N \l__regex_capturing_group_int }
25270         #1
25271         { ?? \prg_break:n } { }
25272         \prg_break_point:
25273     }
25274 }
25275 \cs_new_protected:Npn \__regex_group_resetting_loop:nnNn #1#2#3#4
25276 {
25277     \use_none:nn #3 { \int_set:Nn \l__regex_capturing_group_int {#1} }
25278     \int_set:Nn \l__regex_capturing_group_int {#2}
25279     #3 {#4}
25280     \exp_args:Nf \__regex_group_resetting_loop:nnNn
25281     { \int_max:nn {#1} { \l__regex_capturing_group_int } }

```

```

25282     {#2}
25283 }

```

(End definition for `__regex_group_resetting:nnnN` and `__regex_group_resetting_loop:nnNn`.)

`__regex_branch:n` Add a free transition from the left state of the current group to a brand new state, starting point of this branch. Once the branch is built, add a transition from its last state to the right state of the group. The left and right states of the group are extracted from the relevant sequences.

```

25284 \cs_new_protected:Npn \__regex_branch:n #1
25285 {
25286   \__regex_build_new_state:
25287   \seq_get:NN \l__regex_left_state_seq \l__regex_internal_a_tl
25288   \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
25289   \__regex_build_transition_right:nNn \__regex_action_free:n
25290     \l__regex_left_state_int \l__regex_right_state_int
25291   #1
25292   \seq_get:NN \l__regex_right_state_seq \l__regex_internal_a_tl
25293   \__regex_build_transition_right:nNn \__regex_action_free:n
25294     \l__regex_right_state_int \l__regex_internal_a_tl
25295 }

```

(End definition for `__regex_branch:n`.)

`__regex_group_repeat:nn` This function is called to repeat a group a fixed number of times `#2`; if this is 0 we remove the group altogether (but don't reset the `capturing_group` label). Otherwise, the auxiliary `__regex_group_repeat_aux:n` copies `#2` times the `\toks` for the group, and leaves `internal_a` pointing to the left end of the last repetition. We only record the submatch information at the last repetition. Finally, add a state at the end (the transition to it has been taken care of by the replicating auxiliary).

```

25296 \cs_new_protected:Npn \__regex_group_repeat:nn #1#2
25297 {
25298   \if_int_compare:w #2 = 0 \exp_stop_f:
25299     \int_set:Nn \l__regex_max_state_int
25300       { \l__regex_left_state_int - 1 }
25301     \__regex_build_new_state:
25302   \else:
25303     \__regex_group_repeat_aux:n {#2}
25304     \__regex_group_submatches:nnn {#1}
25305     \l__regex_internal_a_int \l__regex_right_state_int
25306     \__regex_build_new_state:
25307   \fi:
25308 }

```

(End definition for `__regex_group_repeat:nn`.)

`__regex_group_submatches:nnn` This inserts in states `#2` and `#3` the code for tracking submatches of the group `#1`, unless inhibited by a label of `-1`.

```

25309 \cs_new_protected:Npn \__regex_group_submatches:nnn #1#2#3
25310 {
25311   \if_int_compare:w #1 > - 1 \exp_stop_f:
25312     \__regex_toks_put_left:Nx #2 { \__regex_action_submatch:n { #1 < } }
25313     \__regex_toks_put_left:Nx #3 { \__regex_action_submatch:n { #1 > } }
25314   \fi:
25315 }

```

(End definition for `_regex_group_submatches:nnN`.)

`_regex_group_repeat_aux:n` Here we repeat `\toks` ranging from `left_state` to `max_state`, `#1 > 0` times. First add a transition so that the copies “chain” properly. Compute the shift `c` between the original copy and the last copy we want. Shift the `right_state` and `max_state` to their final values. We then want to perform `c` copy operations. At the end, `b` is equal to the `max_state`, and `a` points to the left of the last copy of the group.

```

25316 \cs_new_protected:Npn \_regex_group_repeat_aux:n #1
25317 {
25318   \_regex_build_transition_right:nNn \_regex_action_free:n
25319   \l__regex_right_state_int \l__regex_max_state_int
25320   \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
25321   \int_set_eq:NN \l__regex_internal_b_int \l__regex_max_state_int
25322   \if_int_compare:w \int_eval:n {#1} > 1 \exp_stop_f:
25323     \int_set:Nn \l__regex_internal_c_int
25324     {
25325       ( #1 - 1 )
25326       * ( \l__regex_internal_b_int - \l__regex_internal_a_int )
25327     }
25328     \int_add:Nn \l__regex_right_state_int { \l__regex_internal_c_int }
25329     \int_add:Nn \l__regex_max_state_int { \l__regex_internal_c_int }
25330     \_regex_toks_memcpy:NNn
25331     \l__regex_internal_b_int
25332     \l__regex_internal_a_int
25333     \l__regex_internal_c_int
25334   \fi:
25335 }

```

(End definition for `_regex_group_repeat_aux:n`.)

`_regex_group_repeat:nnN` This function is called to repeat a group at least n times; the case $n = 0$ is very different from $n > 0$. Assume first that $n = 0$. Insert submatch tracking information at the start and end of the group, add a free transition from the right end to the “true” left state `a` (remember: in this case we had added an extra state before the left state). This forms the loop, which we break away from by adding a free transition from `a` to a new state.

Now consider the case $n > 0$. Repeat the group n times, chaining various copies with a free transition. Add submatch tracking only to the last copy, then add a free transition from the right end back to the left end of the last copy, either before or after the transition to move on towards the rest of the NFA. This transition can end up before submatch tracking, but that is irrelevant since it only does so when going again through the group, recording new matches. Finally, add a state; we already have a transition pointing to it from `_regex_group_repeat_aux:n`.

```

25336 \cs_new_protected:Npn \_regex_group_repeat:nnN #1#2#3
25337 {
25338   \if_int_compare:w #2 = 0 \exp_stop_f:
25339     \_regex_group_submatches:nnN {#1}
25340     \l__regex_left_state_int \l__regex_right_state_int
25341     \int_set:Nn \l__regex_internal_a_int
25342     { \l__regex_left_state_int - 1 }
25343     \_regex_build_transition_right:nNn \_regex_action_free:n
25344     \l__regex_right_state_int \l__regex_internal_a_int
25345     \_regex_build_new_state:
25346     \if_meaning:w \c_true_bool #3

```

```

25347     \__regex_build_transition_left:NNN \__regex_action_free:n
25348     \l__regex_internal_a_int \l__regex_right_state_int
25349 \else:
25350     \__regex_build_transition_right:nNn \__regex_action_free:n
25351     \l__regex_internal_a_int \l__regex_right_state_int
25352 \fi:
25353 \else:
25354     \__regex_group_repeat_aux:n {#2}
25355     \__regex_group_submatches:nNN {#1}
25356     \l__regex_internal_a_int \l__regex_right_state_int
25357 \if_meaning:w \c_true_bool #3
25358     \__regex_build_transition_right:nNn \__regex_action_free_group:n
25359     \l__regex_right_state_int \l__regex_internal_a_int
25360 \else:
25361     \__regex_build_transition_left:NNN \__regex_action_free_group:n
25362     \l__regex_right_state_int \l__regex_internal_a_int
25363 \fi:
25364 \__regex_build_new_state:
25365 \fi:
25366 }

```

(End definition for __regex_group_repeat:nnN.)

__regex_group_repeat:nnnN

We wish to repeat the group between #2 and #2 + #3 times, with a lazyness controlled by #4. We insert submatch tracking up front: in principle, we could avoid recording submatches for the first #2 copies of the group, but that forces us to treat specially the case #2 = 0. Repeat that group with submatch tracking #2 + #3 times (the maximum number of repetitions). Then our goal is to add #3 transitions from the end of the #2-th group, and each subsequent groups, to the end. For a lazy quantifier, we add those transitions to the left states, before submatch tracking. For the greedy case, we add the transitions to the right states, after submatch tracking and the transitions which go on with more repetitions. In the greedy case with #2 = 0, the transition which skips over all copies of the group must be added separately, because its starting state does not follow the normal pattern: we had to add it “by hand” earlier.

```

25367 \cs_new_protected:Npn \__regex_group_repeat:nnnN #1#2#3#4
25368 {
25369     \__regex_group_submatches:nNN {#1}
25370     \l__regex_left_state_int \l__regex_right_state_int
25371     \__regex_group_repeat_aux:n { #2 + #3 }
25372 \if_meaning:w \c_true_bool #4
25373     \int_set_eq:NN \l__regex_left_state_int \l__regex_max_state_int
25374     \prg_replicate:nn { #3 }
25375     {
25376         \int_sub:Nn \l__regex_left_state_int
25377         { \l__regex_internal_b_int - \l__regex_internal_a_int }
25378         \__regex_build_transition_left:NNN \__regex_action_free:n
25379         \l__regex_left_state_int \l__regex_max_state_int
25380     }
25381 \else:
25382     \prg_replicate:nn { #3 - 1 }
25383     {
25384         \int_sub:Nn \l__regex_right_state_int
25385         { \l__regex_internal_b_int - \l__regex_internal_a_int }

```

```

25386         \__regex_build_transition_right:nNn \__regex_action_free:n
25387         \l__regex_right_state_int \l__regex_max_state_int
25388     }
25389     \if_int_compare:w #2 = 0 \exp_stop_f:
25390         \int_set:Nn \l__regex_right_state_int
25391         { \l__regex_left_state_int - 1 }
25392     \else:
25393         \int_sub:Nn \l__regex_right_state_int
25394         { \l__regex_internal_b_int - \l__regex_internal_a_int }
25395     \fi:
25396     \__regex_build_transition_right:nNn \__regex_action_free:n
25397     \l__regex_right_state_int \l__regex_max_state_int
25398 \fi:
25399 \__regex_build_new_state:
25400 }

```

(End definition for __regex_group_repeat:nnnN.)

41.4.6 Others

__regex_assertion:Nn Usage: __regex_assertion:Nn *<boolean>* {*<test>*}, where the *<test>* is either of the two other functions. Add a free transition to a new state, conditionally to the assertion test.

__regex_b_test: The __regex_b_test: test is used by the \b and \B escape: check if the last character was a word character or not, and do the same to the current character. The boundary-markers of the string are non-word characters for this purpose. Anchors at the start or end of match use __regex_anchor:N, with a position controlled by the integer #1.

```

25401 \cs_new_protected:Npn \__regex_assertion:Nn #1#2
25402 {
25403     \__regex_build_new_state:
25404     \__regex_toks_put_right:Nx \l__regex_left_state_int
25405     {
25406         \exp_not:n {#2}
25407         \__regex_break_point:TF
25408         \bool_if:NF #1 { { } }
25409         {
25410             \__regex_action_free:n
25411             {
25412                 \int_eval:n
25413                 { \l__regex_right_state_int - \l__regex_left_state_int }
25414             }
25415         }
25416         \bool_if:NT #1 { { } }
25417     }
25418 }
25419 \cs_new_protected:Npn \__regex_anchor:N #1
25420 {
25421     \if_int_compare:w #1 = \l__regex_curr_pos_int
25422         \exp_after:wN \__regex_break_true:w
25423     \fi:
25424 }
25425 \cs_new_protected:Npn \__regex_b_test:
25426 {
25427     \group_begin:
25428     \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_int

```



```

25429     \__regex_prop_w:
25430     \__regex_break_point:TF
25431     { \group_end: \__regex_item_reverse:n \__regex_prop_w: }
25432     { \group_end: \__regex_prop_w: }
25433 }

```

(End definition for __regex_assertion:Nn, __regex_b_test:, and __regex_anchor:N.)

__regex_command_K: Change the starting point of the 0-th submatch (full match), and transition to a new state, pretending that this is a fresh thread.

```

25434 \cs_new_protected:Npn \__regex_command_K:
25435 {
25436     \__regex_build_new_state:
25437     \__regex_toks_put_right:Nx \l__regex_left_state_int
25438     {
25439         \__regex_action_submatch:n { 0< }
25440         \bool_set_true:N \l__regex_fresh_thread_bool
25441         \__regex_action_free:n
25442         {
25443             \int_eval:n
25444             { \l__regex_right_state_int - \l__regex_left_state_int }
25445         }
25446         \bool_set_false:N \l__regex_fresh_thread_bool
25447     }
25448 }

```

(End definition for __regex_command_K:.)

41.5 Matching

We search for matches by running all the execution threads through the NFA in parallel, reading one token of the query at each step. The NFA contains “free” transitions to other states, and transitions which “consume” the current token. For free transitions, the instruction at the new state of the NFA is performed immediately. When a transition consumes a character, the new state is appended to a list of “active states”, stored in `\g__regex_thread_state_intarray`: this thread is made active again when the next token is read from the query. At every step (for each token in the query), we unpack that list of active states and the corresponding submatch props, and empty those.

If two paths through the NFA “collide” in the sense that they reach the same state after reading a given token, then they only differ in how they previously matched, and any future execution would be identical for both. (Note that this would be wrong in the presence of back-references.) Hence, we only need to keep one of the two threads: the thread with the highest priority. Our NFA is built in such a way that higher priority actions always come before lower priority actions, which makes things work.

The explanation in the previous paragraph may make us think that we simply need to keep track of which states were visited at a given step: after all, the loop generated when matching `(a?)*` against `a` is broken, isn’t it? No. The group first matches `a`, as it should, then repeats; it attempts to match `a` again but fails; it skips `a`, and finds out that this state has already been seen at this position in the query: the match stops. The capturing group is (wrongly) `a`. What went wrong is that a thread collided with itself, and the later version, which has gone through the group one more times with an empty match, should have a higher priority than not going through the group.

We solve this by distinguishing “normal” free transitions `__regex_action_free:n` from transitions `__regex_action_free_group:n` which go back to the start of the group. The former keeps threads unless they have been visited by a “completed” thread, while the latter kind of transition also prevents going back to a state visited by the current thread.

41.5.1 Variables used when matching

<code>\l__regex_min_pos_int</code> <code>\l__regex_max_pos_int</code> <code>\l__regex_curr_pos_int</code> <code>\l__regex_start_pos_int</code> <code>\l__regex_success_pos_int</code>	<p>The tokens in the query are indexed from <code>min_pos</code> for the first to <code>max_pos - 1</code> for the last, and their information is stored in several arrays and <code>\toks</code> registers with those numbers. We don’t start from 0 because the <code>\toks</code> registers with low numbers are used to hold the states of the NFA. We match without backtracking, keeping all threads in lockstep at the <code>current_pos</code> in the query. The starting point of the current match attempt is <code>start_pos</code>, and <code>success_pos</code>, updated whenever a thread succeeds, is used as the next starting position.</p>
---	--

```

25449 \int_new:N \l__regex_min_pos_int
25450 \int_new:N \l__regex_max_pos_int
25451 \int_new:N \l__regex_curr_pos_int
25452 \int_new:N \l__regex_start_pos_int
25453 \int_new:N \l__regex_success_pos_int

```

(End definition for `\l__regex_min_pos_int` and others.)

<code>\l__regex_curr_char_int</code> <code>\l__regex_curr_catcode_int</code> <code>\l__regex_last_char_int</code> <code>\l__regex_case_changed_char_int</code>	<p>The character and category codes of the token at the current position; the character code of the token at the previous position; and the character code of the result of changing the case of the current token (<code>A-Z↔a-z</code>). This last integer is only computed when necessary, and is otherwise <code>\c_max_int</code>. The <code>current_char</code> variable is also used in various other phases to hold a character code.</p>
---	---

```

25454 \int_new:N \l__regex_curr_char_int
25455 \int_new:N \l__regex_curr_catcode_int
25456 \int_new:N \l__regex_last_char_int
25457 \int_new:N \l__regex_case_changed_char_int

```

(End definition for `\l__regex_curr_char_int` and others.)

<code>\l__regex_curr_state_int</code>	<p>For every character in the token list, each of the active states is considered in turn. The variable <code>\l__regex_curr_state_int</code> holds the state of the NFA which is currently considered: transitions are then given as shifts relative to the current state.</p>
---------------------------------------	---

```

25458 \int_new:N \l__regex_curr_state_int

```

(End definition for `\l__regex_curr_state_int`.)

<code>\l__regex_curr_submatches_prop</code> <code>\l__regex_success_submatches_prop</code>	<p>The submatches for the thread which is currently active are stored in the <code>current_submatches</code> property list variable. This property list is stored by <code>__regex_action_cost:n</code> into the <code>\toks</code> register for the target state of the transition, to be retrieved when matching at the next position. When a thread succeeds, this property list is copied to <code>\l__regex_success_submatches_prop</code>: only the last successful thread remains there.</p>
---	--

```

25459 \prop_new:N \l__regex_curr_submatches_prop
25460 \prop_new:N \l__regex_success_submatches_prop

```

(End definition for `\l__regex_curr_submatches_prop` and `\l__regex_success_submatches_prop`.)

`\l__regex_step_int` This integer, always even, is increased every time a character in the query is read, and not reset when doing multiple matches. We store in `\g__regex_state_active_intarray` the last step in which each $\langle state \rangle$ in the NFA was encountered. This lets us break infinite loops by not visiting the same state twice in the same step. In fact, the step we store is equal to `step` when we have started performing the operations of `\toks $\langle state \rangle$` , but not finished yet. However, once we finish, we store `step + 1` in `\g__regex_state_active_intarray`. This is needed to track submatches properly (see building phase). The `step` is also used to attach each set of submatch information to a given iteration (and automatically discard it when it corresponds to a past step).

```
25461 \int_new:N \l__regex_step_int
```

(End definition for `\l__regex_step_int`.)

`\l__regex_min_active_int` All the currently active threads are kept in order of precedence in `\g__regex_thread_state_intarray`, and the corresponding submatches in the `\toks`. For our purposes, those serve as an array, indexed from `min_active` (inclusive) to `max_active` (excluded).
`\l__regex_max_active_int` At the start of every step, the whole array is unpacked, so that the space can immediately be reused, and `max_active` is reset to `min_active`, effectively clearing the array.

```
25462 \int_new:N \l__regex_min_active_int
```

```
25463 \int_new:N \l__regex_max_active_int
```

(End definition for `\l__regex_min_active_int` and `\l__regex_max_active_int`.)

`\g__regex_state_active_intarray` `\g__regex_state_active_intarray` stores the last $\langle step \rangle$ in which each $\langle state \rangle$ was active.
`\g__regex_thread_state_intarray` `\g__regex_thread_state_intarray` stores threads to be considered in the next step, more precisely the states in which these threads are.

```
25464 \intarray_new:Nn \g__regex_state_active_intarray { 65536 }
```

```
25465 \intarray_new:Nn \g__regex_thread_state_intarray { 65536 }
```

(End definition for `\g__regex_state_active_intarray` and `\g__regex_thread_state_intarray`.)

`\l__regex_every_match_tl` Every time a match is found, this token list is used. For single matching, the token list is empty. For multiple matching, the token list is set to repeat the matching, after performing some operation which depends on the user function. See `__regex_single_match:` and `__regex_multi_match:n`.

```
25466 \tl_new:N \l__regex_every_match_tl
```

(End definition for `\l__regex_every_match_tl`.)

`\l__regex_fresh_thread_bool` When doing multiple matches, we need to avoid infinite loops where each iteration matches the same empty token list. When an empty token list is matched, the next successful match of the same empty token list is suppressed. We detect empty matches by setting `\l__regex_fresh_thread_bool` to `true` for threads which directly come from the start of the regex or from the `\K` command, and testing that boolean whenever a thread succeeds. The function `__regex_if_two_empty_matches:F` is redefined at every match attempt, depending on whether the previous match was empty or not: if it was, then the function must cancel a purported success if it is empty and at the same spot as the previous match; otherwise, we definitely don't have two identical empty matches, so the function is `\use:n`.

```
25467 \bool_new:N \l__regex_fresh_thread_bool
```

```
25468 \bool_new:N \l__regex_empty_success_bool
```

```
25469 \cs_new_eq:NN \__regex_if_two_empty_matches:F \use:n
```

(End definition for `\l__regex_fresh_thread_bool`, `\l__regex_empty_success_bool`, and `_regex_if_two_empty_matches:F`.)

`\g__regex_success_bool` The boolean `\l__regex_match_success_bool` is true if the current match attempt was successful, and `\g__regex_success_bool` is true if there was at least one successful match. This is the only global variable in this whole module, but we would need it to be local when matching a control sequence with `\c{...}`. This is done by saving the global variable into `\l__regex_saved_success_bool`, which is local, hence not affected by the changes due to inner regex functions.

```
25470 \bool_new:N \g__regex_success_bool
25471 \bool_new:N \l__regex_saved_success_bool
25472 \bool_new:N \l__regex_match_success_bool
```

(End definition for `\g__regex_success_bool`, `\l__regex_saved_success_bool`, and `\l__regex_match_success_bool`.)

41.5.2 Matching: framework

`__regex_match:n` First store the query into `\toks` registers and arrays (see `__regex_query_set:nnn`).
`__regex_match_cs:n` Then initialize the variables that should be set once for each user function (even for
`__regex_match_init:` multiple matches). Namely, the overall matching is not yet successful; none of the states should be marked as visited (`\g__regex_state_active_intarray`), and we start at step 0; we pretend that there was a previous match ending at the start of the query, which was not empty (to avoid smothering an empty match at the start). Once all this is set up, we are ready for the ride. Find the first match.

```
25473 \cs_new_protected:Npn \__regex_match:n #1
25474 {
25475   \int_zero:N \l__regex_balance_int
25476   \int_set:Nn \l__regex_curr_pos_int { 2 * \l__regex_max_state_int }
25477   \__regex_query_set:nnn { } { -1 } { -2 }
25478   \int_set_eq:NN \l__regex_min_pos_int \l__regex_curr_pos_int
25479   \tl_analysis_map_inline:nn {#1}
25480     { \__regex_query_set:nnn {##1} {"##3"} {##2} }
25481   \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
25482   \__regex_query_set:nnn { } { -1 } { -2 }
25483   \__regex_match_init:
25484   \__regex_match_once:
25485 }
25486 \cs_new_protected:Npn \__regex_match_cs:n #1
25487 {
25488   \int_zero:N \l__regex_balance_int
25489   \int_set:Nn \l__regex_curr_pos_int
25490     {
25491       \int_max:nn { 2 * \l__regex_max_state_int - \l__regex_min_state_int }
25492       { \l__regex_max_pos_int }
25493       + 1
25494     }
25495   \__regex_query_set:nnn { } { -1 } { -2 }
25496   \int_set_eq:NN \l__regex_min_pos_int \l__regex_curr_pos_int
25497   \str_map_inline:nn {#1}
25498     {
25499       \__regex_query_set:nnn { \exp_not:n {##1} }
25500       { \tl_if_blank:nTF {##1} { 10 } { 12 } }
```

```

25501         { '##1 }
25502     }
25503     \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
25504     \__regex_query_set:nnn { } { -1 } { -2 }
25505     \__regex_match_init:
25506     \__regex_match_once:
25507 }
25508 \cs_new_protected:Npn \__regex_match_init:
25509 {
25510     \bool_gset_false:N \g__regex_success_bool
25511     \int_step_inline:nnn
25512         \l__regex_min_state_int { \l__regex_max_state_int - 1 }
25513     {
25514         \__kernel_intarray_gset:Nnn
25515         \g__regex_state_active_intarray {##1} { 1 }
25516     }
25517     \int_set_eq:NN \l__regex_min_active_int \l__regex_max_state_int
25518     \int_zero:N \l__regex_step_int
25519     \int_set_eq:NN \l__regex_success_pos_int \l__regex_min_pos_int
25520     \int_set:Nn \l__regex_min_submatch_int
25521         { 2 * \l__regex_max_state_int }
25522     \int_set_eq:NN \l__regex_submatch_int \l__regex_min_submatch_int
25523     \bool_set_false:N \l__regex_empty_success_bool
25524 }

```

(End definition for `__regex_match:n`, `__regex_match_cs:n`, and `__regex_match_init:.`)

`__regex_match_once:` This function finds one match, then does some action defined by the `every_match` token list, which may recursively call `__regex_match_once:.` First initialize some variables: set the conditional which detects identical empty matches; this match attempt starts at the previous `success_pos`, is not yet successful, and has no submatches yet; clear the array of active threads, and put the starting state 0 in it. We are then almost ready to read our first token in the query, but we actually start one position earlier than the start, and `get` that token, to set `last_char` properly for word boundaries. Then call `__regex_match_loop:`, which runs through the query until the end or until a successful match breaks early.

```

25525 \cs_new_protected:Npn \__regex_match_once:
25526 {
25527     \if_meaning:w \c_true_bool \l__regex_empty_success_bool
25528     \cs_set:Npn \__regex_if_two_empty_matches:F
25529     {
25530         \int_compare:nNnF
25531             \l__regex_start_pos_int = \l__regex_curr_pos_int
25532     }
25533     \else:
25534         \cs_set_eq:NN \__regex_if_two_empty_matches:F \use:n
25535     \fi:
25536     \int_set_eq:NN \l__regex_start_pos_int \l__regex_success_pos_int
25537     \bool_set_false:N \l__regex_match_success_bool
25538     \prop_clear:N \l__regex_curr_submatches_prop
25539     \int_set_eq:NN \l__regex_max_active_int \l__regex_min_active_int
25540     \__regex_store_state:n { \l__regex_min_state_int }
25541     \int_set:Nn \l__regex_curr_pos_int
25542         { \l__regex_start_pos_int - 1 }

```

```

25543     \__regex_query_get:
25544     \__regex_match_loop:
25545     \l__regex_every_match_tl
25546 }

```

(End definition for __regex_match_once:.)

__regex_single_match: For a single match, the overall success is determined by whether the only match attempt is a success. When doing multiple matches, the overall matching is successful as soon as any match succeeds. Perform the action #1, then find the next match.

```

25547 \cs_new_protected:Npn \__regex_single_match:
25548 {
25549     \tl_set:Nn \l__regex_every_match_tl
25550     {
25551         \bool_gset_eq:NN
25552         \g__regex_success_bool
25553         \l__regex_match_success_bool
25554     }
25555 }
25556 \cs_new_protected:Npn \__regex_multi_match:n #1
25557 {
25558     \tl_set:Nn \l__regex_every_match_tl
25559     {
25560         \if_meaning:w \c_true_bool \l__regex_match_success_bool
25561         \bool_gset_true:N \g__regex_success_bool
25562         #1
25563         \exp_after:wN \__regex_match_once:
25564     \fi:
25565 }
25566 }

```

(End definition for __regex_single_match: and __regex_multi_match:n.)

__regex_match_loop: At each new position, set some variables and get the new character and category from the query. Then unpack the array of active threads, and clear it by resetting its length (max_active). This results in a sequence of __regex_use_state_and_submatches:nn {<state>} {<prop>}, and we consider those states one by one in order. As soon as a thread succeeds, exit the step, and, if there are threads to consider at the next position, and we have not reached the end of the string, repeat the loop. Otherwise, the last thread that succeeded is what __regex_match_once: matches. We explain the fresh_thread business when describing __regex_action_wildcard:.

```

25567 \cs_new_protected:Npn \__regex_match_loop:
25568 {
25569     \int_add:Nn \l__regex_step_int { 2 }
25570     \int_incr:N \l__regex_curr_pos_int
25571     \int_set_eq:NN \l__regex_last_char_int \l__regex_curr_char_int
25572     \int_set_eq:NN \l__regex_case_changed_char_int \c_max_int
25573     \__regex_query_get:
25574     \use:x
25575     {
25576         \int_set_eq:NN \l__regex_max_active_int \l__regex_min_active_int
25577         \int_step_function:nnN
25578         { \l__regex_min_active_int }
25579         { \l__regex_max_active_int - 1 }

```

```

25580         \_regex_match_one_active:n
25581     }
25582     \prg_break_point:
25583     \bool_set_false:N \l__regex_fresh_thread_bool
25584     \if_int_compare:w \l__regex_max_active_int > \l__regex_min_active_int
25585         \if_int_compare:w \l__regex_curr_pos_int < \l__regex_max_pos_int
25586             \exp_after:wN \exp_after:wN \exp_after:wN \_regex_match_loop:
25587         \fi:
25588     \fi:
25589 }
25590 \cs_new:Npn \_regex_match_one_active:n #1
25591 {
25592     \_regex_use_state_and_submatches:nn
25593     { \_kernel_intarray_item:Nn \g__regex_thread_state_intarray {#1} }
25594     { \_regex_toks_use:w #1 }
25595 }

```

(End definition for _regex_match_loop: and _regex_match_one_active:n.)

_regex_query_set:nnn The arguments are: tokens that o and x expand to one token of the query, the catcode, and the character code. Store those, and the current brace balance (used later to check for overall brace balance) in a \toks register and some arrays, then update the balance.

```

25596 \cs_new_protected:Npn \_regex_query_set:nnn #1#2#3
25597 {
25598     \_kernel_intarray_gset:Nnn \g__regex_charcode_intarray
25599     { \l__regex_curr_pos_int } {#3}
26000     \_kernel_intarray_gset:Nnn \g__regex_catcode_intarray
26001     { \l__regex_curr_pos_int } {#2}
26002     \_kernel_intarray_gset:Nnn \g__regex_balance_intarray
26003     { \l__regex_curr_pos_int } { \l__regex_balance_int }
26004     \_regex_toks_set:Nn \l__regex_curr_pos_int {#1}
26005     \int_incr:N \l__regex_curr_pos_int
26006     \if_case:w #2 \exp_stop_f:
26007     \or: \int_incr:N \l__regex_balance_int
26008     \or: \int_decr:N \l__regex_balance_int
26009     \fi:
26010 }

```

(End definition for _regex_query_set:nnn.)

_regex_query_get: Extract the current character and category codes at the current position from the appropriate arrays.

```

25611 \cs_new_protected:Npn \_regex_query_get:
25612 {
25613     \l__regex_curr_char_int
25614     = \_kernel_intarray_item:Nn \g__regex_charcode_intarray
25615     { \l__regex_curr_pos_int } \scan_stop:
25616     \l__regex_curr_catcode_int
25617     = \_kernel_intarray_item:Nn \g__regex_catcode_intarray
25618     { \l__regex_curr_pos_int } \scan_stop:
25619 }

```

(End definition for _regex_query_get:.)

41.5.3 Using states of the nfa

`__regex_use_state:` Use the current NFA instruction. The state is initially marked as belonging to the current **step**: this allows normal free transition to repeat, but group-repeating transitions won't. Once we are done exploring all the branches it spawned, the state is marked as **step + 1**: any thread hitting it at that point will be terminated.

```
25620 \cs_new_protected:Npn \__regex_use_state:
25621 {
25622   \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
25623     { \l__regex_curr_state_int } { \l__regex_step_int }
25624   \__regex_toks_use:w \l__regex_curr_state_int
25625   \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
25626     { \l__regex_curr_state_int }
25627     { \int_eval:n { \l__regex_step_int + 1 } }
25628 }
```

(End definition for __regex_use_state:.)

`__regex_use_state_and_submatches:nn` This function is called as one item in the array of active threads after that array has been unpacked for a new step. Update the `current_state` and `current_submatches` and use the state if it has not yet been encountered at this step.

```
25629 \cs_new_protected:Npn \__regex_use_state_and_submatches:nn #1 #2
25630 {
25631   \int_set:Nn \l__regex_curr_state_int {#1}
25632   \if_int_compare:w
25633     \__kernel_intarray_item:Nn \g__regex_state_active_intarray
25634       { \l__regex_curr_state_int }
25635       < \l__regex_step_int
25636     \tl_set:Nn \l__regex_curr_submatches_prop {#2}
25637     \exp_after:wN \__regex_use_state:
25638   \fi:
25639   \scan_stop:
25640 }
```

(End definition for __regex_use_state_and_submatches:nn.)

41.5.4 Actions when matching

`__regex_action_start_wildcard:` For an unanchored match, state 0 has a free transition to the next and a costly one to itself, to repeat at the next position. To catch repeated identical empty matches, we need to know if a successful thread corresponds to an empty match. The instruction resetting `\l__regex_fresh_thread_bool` may be skipped by a successful thread, hence we had to add it to `__regex_match_loop:` too.

```
25641 \cs_new_protected:Npn \__regex_action_start_wildcard:
25642 {
25643   \bool_set_true:N \l__regex_fresh_thread_bool
25644   \__regex_action_free:n {1}
25645   \bool_set_false:N \l__regex_fresh_thread_bool
25646   \__regex_action_cost:n {0}
25647 }
```

(End definition for __regex_action_start_wildcard:.)

`__regex_action_free:n` These functions copy a thread after checking that the NFA state has not already been used
`__regex_action_free_group:n` at this position. If not, store submatches in the new state, and insert the instructions for
`__regex_action_free_aux:nn` that state in the input stream. Then restore the old value of `\l__regex_curr_state_int`
and of the current submatches. The two types of free transitions differ by how they
test that the state has not been encountered yet: the `group` version is stricter, and will
not use a state if it was used earlier in the current thread, hence forcefully breaking the
loop, while the “normal” version will revisit a state even within the thread itself.

```

25648 \cs_new_protected:Npn \__regex_action_free:n
25649 { \__regex_action_free_aux:nn { > \l__regex_step_int \else: } }
25650 \cs_new_protected:Npn \__regex_action_free_group:n
25651 { \__regex_action_free_aux:nn { < \l__regex_step_int } }
25652 \cs_new_protected:Npn \__regex_action_free_aux:nn #1#2
25653 {
25654   \use:x
25655   {
25656     \int_add:Nn \l__regex_curr_state_int {#2}
25657     \exp_not:n
25658     {
25659       \if_int_compare:w
25660         \__kernel_intarray_item:Nn \g__regex_state_active_intarray
25661         { \l__regex_curr_state_int }
25662         #1
25663         \exp_after:wN \__regex_use_state:
25664       \fi:
25665     }
25666     \int_set:Nn \l__regex_curr_state_int
25667     { \int_use:N \l__regex_curr_state_int }
25668     \tl_set:Nn \exp_not:N \l__regex_curr_submatches_prop
25669     { \exp_not:o \l__regex_curr_submatches_prop }
25670   }
25671 }

```

(End definition for `__regex_action_free:n`, `__regex_action_free_group:n`, and `__regex_action_free_aux:nn`.)

`__regex_action_cost:n` A transition which consumes the current character and shifts the state by #1. The
resulting state is stored in the appropriate array for use at the next position, and we also
store the current submatches.

```

25672 \cs_new_protected:Npn \__regex_action_cost:n #1
25673 {
25674   \exp_args:Nx \__regex_store_state:n
25675   { \int_eval:n { \l__regex_curr_state_int + #1 } }
25676 }

```

(End definition for `__regex_action_cost:n`.)

`__regex_store_state:n` Put the given state in `\g__regex_thread_state_intarray`, and increment the length of
`__regex_store_submatches:` the array. Also store the current submatch in the appropriate `\toks`.

```

25677 \cs_new_protected:Npn \__regex_store_state:n #1
25678 {
25679   \__regex_store_submatches:
25680   \__kernel_intarray_gset:Nnn \g__regex_thread_state_intarray
25681   { \l__regex_max_active_int } {#1}
25682   \int_incr:N \l__regex_max_active_int

```

```

25683     }
25684 \cs_new_protected:Npn \__regex_store_submatches:
25685 {
25686     \__regex_toks_set:No \l__regex_max_active_int
25687     { \l__regex_curr_submatches_prop }
25688 }

```

(End definition for __regex_store_state:n and __regex_store_submatches:.)

__regex_disable_submatches: Some user functions don't require tracking submatches. We get a performance improvement by simply defining the relevant functions to remove their argument and do nothing with it.

```

25689 \cs_new_protected:Npn \__regex_disable_submatches:
25690 {
25691     \cs_set_protected:Npn \__regex_store_submatches: { }
25692     \cs_set_protected:Npn \__regex_action_submatch:n ##1 { }
25693 }

```

(End definition for __regex_disable_submatches:.)

__regex_action_submatch:n Update the current submatches with the information from the current position. Maybe a bottleneck.

```

25694 \cs_new_protected:Npn \__regex_action_submatch:n #1
25695 {
25696     \prop_put:Nno \l__regex_curr_submatches_prop {#1}
25697     { \int_use:N \l__regex_curr_pos_int }
25698 }

```

(End definition for __regex_action_submatch:n.)

__regex_action_success: There is a successful match when an execution path reaches the last state in the NFA, unless this marks a second identical empty match. Then mark that there was a successful match; it is empty if it is "fresh"; and we store the current position and submatches. The current step is then interrupted with \prg_break:, and only paths with higher precedence are pursued further. The values stored here may be overwritten by a later success of a path with higher precedence.

```

25699 \cs_new_protected:Npn \__regex_action_success:
25700 {
25701     \__regex_if_two_empty_matches:F
25702     {
25703         \bool_set_true:N \l__regex_match_success_bool
25704         \bool_set_eq:NN \l__regex_empty_success_bool
25705         \l__regex_fresh_thread_bool
25706         \int_set_eq:NN \l__regex_success_pos_int \l__regex_curr_pos_int
25707         \prop_set_eq:NN \l__regex_success_submatches_prop
25708         \l__regex_curr_submatches_prop
25709         \prg_break:
25710     }
25711 }

```

(End definition for __regex_action_success:.)

41.6 Replacement

41.6.1 Variables and helpers used in replacement

`\l_regex_replacement_csnames_int` The behaviour of closing braces inside a replacement text depends on whether a sequences `\c{` or `\u{` has been encountered. The number of “open” such sequences that should be closed by `}` is stored in `\l__regex_replacement_csnames_int`, and decreased by 1 by each `}`.

```
25712 \int_new:N \l__regex_replacement_csnames_int
```

(End definition for \l__regex_replacement_csnames_int.)

`\l_regex_replacement_category_tl` This sequence of letters is used to correctly restore categories in nested constructions such as `\cL(abc\cD(_)d)`.

```
25713 \tl_new:N \l__regex_replacement_category_tl
```

```
25714 \seq_new:N \l__regex_replacement_category_seq
```

(End definition for \l__regex_replacement_category_tl and \l__regex_replacement_category_seq.)

`\l__regex_balance_tl` This token list holds the replacement text for `__regex_replacement_balance_one_match:n` while it is being built incrementally.

```
25715 \tl_new:N \l__regex_balance_tl
```

(End definition for \l__regex_balance_tl.)

`__regex_replacement_balance_one_match:n` This expects as an argument the first index of a set of entries in `\g__regex_submatch_begin_intarray` (and related arrays) which hold the submatch information for a given match. It can be used within an integer expression to obtain the brace balance incurred by performing the replacement on that match. This combines the braces lost by removing the match, braces added by all the submatches appearing in the replacement, and braces appearing explicitly in the replacement. Even though it is always redefined before use, we initialize it as for an empty replacement. An important property is that concatenating several calls to that function must result in a valid integer expression (hence a leading + in the actual definition).

```
25716 \cs_new:Npn \__regex_replacement_balance_one_match:n #1
```

```
25717 { - \__regex_submatch_balance:n {#1} }
```

(End definition for __regex_replacement_balance_one_match:n.)

`__regex_replacement_do_one_match:n` The input is the same as `__regex_replacement_balance_one_match:n`. This function is redefined to expand to the part of the token list from the end of the previous match to a given match, followed by the replacement text. Hence concatenating the result of this function with all possible arguments (one call for each match), as well as the range from the end of the last match to the end of the string, produces the fully replaced token list. The initialization does not matter, but (as an example) we set it as for an empty replacement.

```
25718 \cs_new:Npn \__regex_replacement_do_one_match:n #1
```

```
25719 {
```

```
25720   \__regex_query_range:nn
```

```
25721   { \__kernel_intarray_item:Nn \g__regex_submatch_prev_intarray {#1} }
```

```
25722   { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
```

```
25723 }
```

(End definition for __regex_replacement_do_one_match:n.)

`_regex_replacement_exp_not:N` This function lets us navigate around the fact that the primitive `\exp_not:n` requires a braced argument. As far as I can tell, it is only needed if the user tries to include in the replacement text a control sequence set equal to a macro parameter character, such as `\c_parameter_token`. Indeed, within an x-expanding assignment, `\exp_not:N #` behaves as a single `#`, whereas `\exp_not:n {#}` behaves as a doubled `##`.

```
25724 \cs_new:Npn \_regex_replacement_exp_not:N #1 { \exp_not:n {#1} }
```

(End definition for `_regex_replacement_exp_not:N`.)

41.6.2 Query and brace balance

`_regex_query_range:nn` When it is time to extract submatches from the token list, the various tokens are stored in `\toks` registers numbered from `\l__regex_min_pos_int` inclusive to `\l__regex_max_pos_int` exclusive. The function `_regex_query_range:nn {<min>} {<max>}` unpacks registers from the position `<min>` to the position `<max> - 1` included. Once this is expanded, a second x-expansion results in the actual tokens from the query. That second expansion is only done by user functions at the very end of their operation, after checking (and correcting) the brace balance first.

```
25725 \cs_new:Npn \_regex_query_range:nn #1#2
25726 {
25727   \exp_after:wN \_regex_query_range_loop:ww
25728   \int_value:w \_regex_int_eval:w #1 \exp_after:wN ;
25729   \int_value:w \_regex_int_eval:w #2 ;
25730   \prg_break_point:
25731 }
25732 \cs_new:Npn \_regex_query_range_loop:ww #1 ; #2 ;
25733 {
25734   \if_int_compare:w #1 < #2 \exp_stop_f:
25735   \else:
25736     \exp_after:wN \prg_break:
25737   \fi:
25738   \_regex_toks_use:w #1 \exp_stop_f:
25739   \exp_after:wN \_regex_query_range_loop:ww
25740   \int_value:w \_regex_int_eval:w #1 + 1 ; #2 ;
25741 }
```

(End definition for `_regex_query_range:nn` and `_regex_query_range_loop:ww`.)

`_regex_query_submatch:n` Find the start and end positions for a given submatch (of a given match).

```
25742 \cs_new:Npn \_regex_query_submatch:n #1
25743 {
25744   \_regex_query_range:nn
25745   { \_kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
25746   { \_kernel_intarray_item:Nn \g__regex_submatch_end_intarray {#1} }
25747 }
```

(End definition for `_regex_query_submatch:n`.)

`_regex_submatch_balance:n` Every user function must result in a balanced token list (unbalanced token lists cannot be stored by TeX). When we unpacked the query, we kept track of the brace balance, hence the contribution from a given range is the difference between the brace balances at the `<max pos>` and `<min pos>`. These two positions are found in the corresponding “submatch” arrays.

```

25748 \cs_new_protected:Npn \__regex_submatch_balance:n #1
25749 {
25750   \int_eval:n
25751   {
25752     \int_compare:nNnTF
25753     {
25754       \__kernel_intarray_item:Nn
25755       \g__regex_submatch_end_intarray {#1}
25756     }
25757     = 0
25758     { 0 }
25759     {
25760       \__kernel_intarray_item:Nn \g__regex_balance_intarray
25761       {
25762         \__kernel_intarray_item:Nn
25763         \g__regex_submatch_end_intarray {#1}
25764       }
25765     }
25766   -
25767   \int_compare:nNnTF
25768   {
25769     \__kernel_intarray_item:Nn
25770     \g__regex_submatch_begin_intarray {#1}
25771   }
25772   = 0
25773   { 0 }
25774   {
25775     \__kernel_intarray_item:Nn \g__regex_balance_intarray
25776     {
25777       \__kernel_intarray_item:Nn
25778       \g__regex_submatch_begin_intarray {#1}
25779     }
25780   }
25781 }
25782 }

```

(End definition for __regex_submatch_balance:n.)

41.6.3 Framework

```

\__regex_replacement:n
\__regex_replacement_aux:n

```

The replacement text is built incrementally. We keep track in \l__regex_balance_int of the balance of explicit begin- and end-group tokens and we store in \l__regex_balance_tl some code to compute the brace balance from submatches (see its description). Detect unescaped right braces, and escaped characters, with trailing \prg_do_nothing: because some of the later function look-ahead. Once the whole replacement text has been parsed, make sure that there is no open csname. Finally, define the balance_one_match and do_one_match functions.

```

25783 \cs_new_protected:Npn \__regex_replacement:n #1
25784 {
25785   \group_begin:
25786   \tl_build_begin:N \l__regex_build_tl
25787   \int_zero:N \l__regex_balance_int
25788   \tl_clear:N \l__regex_balance_tl
25789   \__regex_escape_use:nnnn

```

```

25790     {
25791         \if_charcode:w \c_right_brace_str ##1
25792         \__regex_replacement_rbrace:N
25793     \else:
25794         \__regex_replacement_normal:n
25795     \fi:
25796     ##1
25797 }
25798 { \__regex_replacement_escaped:N ##1 }
25799 { \__regex_replacement_normal:n ##1 }
25800 {#1}
25801 \prg_do_nothing: \prg_do_nothing:
25802 \if_int_compare:w \l__regex_replacement_csnames_int > 0 \exp_stop_f:
25803     \__kernel_msg_error:nxx { kernel } { replacement-missing-rbrace }
25804     { \int_use:N \l__regex_replacement_csnames_int }
25805     \tl_build_put_right:Nx \l__regex_build_tl
25806     { \prg_replicate:nn \l__regex_replacement_csnames_int \cs_end: }
25807 \fi:
25808 \seq_if_empty:NF \l__regex_replacement_category_seq
25809 {
25810     \__kernel_msg_error:nxx { kernel } { replacement-missing-rparen }
25811     { \seq_count:N \l__regex_replacement_category_seq }
25812     \seq_clear:N \l__regex_replacement_category_seq
25813 }
25814 \cs_gset:Npx \__regex_replacement_balance_one_match:n ##1
25815 {
25816     + \int_use:N \l__regex_balance_int
25817     \l__regex_balance_tl
25818     - \__regex_submatch_balance:n {##1}
25819 }
25820 \tl_build_end:N \l__regex_build_tl
25821 \exp_args:NNo
25822 \group_end:
25823 \__regex_replacement_aux:n \l__regex_build_tl
25824 }
25825 \cs_new_protected:Npn \__regex_replacement_aux:n #1
25826 {
25827     \cs_set:Npn \__regex_replacement_do_one_match:n ##1
25828     {
25829         \__regex_query_range:nn
25830         {
25831             \__kernel_intarray_item:Nn
25832             \g__regex_submatch_prev_intarray {##1}
25833         }
25834         {
25835             \__kernel_intarray_item:Nn
25836             \g__regex_submatch_begin_intarray {##1}
25837         }
25838     }
25839     #1
25840 }

```

(End definition for __regex_replacement:n and __regex_replacement_aux:n.)

__regex_replacement_normal:n Most characters are simply sent to the output by \tl_build_put_right:Nn, unless a

particular category code has been requested: then `__regex_replacement_c_A:w` or a similar auxiliary is called. One exception is right parentheses, which restore the category code in place before the group started. Note that the sequence is non-empty there: it contains an empty entry corresponding to the initial value of `\l__regex_replacement_category_tl`.

```

25841 \cs_new_protected:Npn \__regex_replacement_normal:n #1
25842 {
25843   \tl_if_empty:NTF \l__regex_replacement_category_tl
25844   { \tl_build_put_right:Nn \l__regex_build_tl {#1} }
25845   { % (
25846     \token_if_eq_charcode:NNTF #1 )
25847     {
25848       \seq_pop:NN \l__regex_replacement_category_seq
25849       \l__regex_replacement_category_tl
25850     }
25851     {
25852       \use:c
25853       {
25854         __regex_replacement_c_
25855         \l__regex_replacement_category_tl :w
25856       }
25857       \__regex_replacement_normal:n {#1}
25858     }
25859   }
25860 }

```

(End definition for `__regex_replacement_normal:n`.)

`__regex_replacement_escaped:N` As in parsing a regular expression, we use an auxiliary built from `#1` if defined. Otherwise, check for escaped digits (standing from submatches from 0 to 9): anything else is a raw character. We use `\token_to_str:N` to give spaces the right category code.

```

25861 \cs_new_protected:Npn \__regex_replacement_escaped:N #1
25862 {
25863   \cs_if_exist_use:cF { __regex_replacement_#1:w }
25864   {
25865     \if_int_compare:w 1 < 1#1 \exp_stop_f:
25866     \__regex_replacement_put_submatch:n {#1}
25867   \else:
25868     \exp_args:No \__regex_replacement_normal:n
25869     { \token_to_str:N #1 }
25870   \fi:
25871   }
25872 }

```

(End definition for `__regex_replacement_escaped:N`.)

41.6.4 Submatches

`__regex_replacement_put_submatch:n` Insert a submatch in the replacement text. This is dropped if the submatch number is larger than the number of capturing groups. Unless the submatch appears inside a `\c{...}` or `\u{...}` construction, it must be taken into account in the brace balance. Later on, `##1` will be replaced by a pointer to the 0-th submatch for a given match.

There is an `\exp_not:N` here as at the point-of-use of `\l__regex_balance_tl` there is an `x`-type expansion which is needed to get `##1` in correctly.

```

25873 \cs_new_protected:Npn \__regex_replacement_put_submatch:n #1
25874 {
25875   \if_int_compare:w #1 < \l__regex_capturing_group_int
25876     \tl_build_put_right:Nn \l__regex_build_tl
25877       { \__regex_query_submatch:n { \int_eval:n { #1 + ##1 } } }
25878     \if_int_compare:w \l__regex_replacement_csnames_int = 0 \exp_stop_f:
25879     \tl_put_right:Nn \l__regex_balance_tl
25880       {
25881         + \__regex_submatch_balance:n
25882         { \exp_not:N \int_eval:n { #1 + ##1 } }
25883       }
25884   \fi:
25885 \fi:
25886 }

```

(End definition for `__regex_replacement_put_submatch:n`.)

`__regex_replacement_g:w` Grab digits for the `\g` escape sequence in a primitive assignment to the integer `\l__regex_internal_a_int`. At the end of the run of digits, check that it ends with a right brace.

```

25887 \cs_new_protected:Npn \__regex_replacement_g:w #1#2
25888 {
25889   \__regex_two_if_eq:NNNTF
25890     #1 #2 \__regex_replacement_normal:n \c_left_brace_str
25891     { \l__regex_internal_a_int = \__regex_replacement_g_digits:NN }
25892     { \__regex_replacement_error:NNN g #1 #2 }
25893 }
25894 \cs_new:Npn \__regex_replacement_g_digits:NN #1#2
25895 {
25896   \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
25897   {
25898     \if_int_compare:w 1 < 1#2 \exp_stop_f:
25899     #2
25900     \exp_after:wN \use_i:nnn
25901     \exp_after:wN \__regex_replacement_g_digits:NN
25902   } \else:
25903     \exp_stop_f:
25904     \exp_after:wN \__regex_replacement_error:NNN
25905     \exp_after:wN g
25906   \fi:
25907 }
25908 {
25909   \exp_stop_f:
25910   \if_meaning:w \__regex_replacement_rbrace:N #1
25911     \exp_args:No \__regex_replacement_put_submatch:n
25912       { \int_use:N \l__regex_internal_a_int }
25913     \exp_after:wN \use_none:nn
25914   } \else:
25915     \exp_after:wN \__regex_replacement_error:NNN
25916     \exp_after:wN g
25917   \fi:
25918 }

```



```

25919     #1 #2
25920   }

```

(End definition for `_regex_replacement_g:w` and `_regex_replacement_g_digits:NN`.)

41.6.5 Csnames in replacement

`_regex_replacement_c:w` `\c` may only be followed by an unescaped character. If followed by a left brace, start a control sequence by calling an auxiliary common with `\u`. Otherwise test whether the category is known; if it is not, complain.

```

25921 \cs_new_protected:Npn \_regex_replacement_c:w #1#2
25922 {
25923   \token_if_eq_meaning:NNTF #1 \_regex_replacement_normal:n
25924   {
25925     \exp_after:wN \token_if_eq_charcode:NNTF \c_left_brace_str #2
25926     { \_regex_replacement_cu_aux:Nw \_regex_replacement_exp_not:N }
25927     {
25928       \cs_if_exist:cTF { \_regex_replacement_c_#2:w }
25929       { \_regex_replacement_cat:NNN #2 }
25930       { \_regex_replacement_error:NNN c #1#2 }
25931     }
25932   }
25933   { \_regex_replacement_error:NNN c #1#2 }
25934 }

```

(End definition for `_regex_replacement_c:w`.)

`_regex_replacement_cu_aux:Nw` Start a control sequence with `\cs:w`, protected from expansion by `#1` (either `_regex_replacement_exp_not:N` or `\exp_not:V`), or turned to a string by `\tl_to_str:V` if inside another csname construction `\c` or `\u`. We use `\tl_to_str:V` rather than `\tl_to_str:N` to deal with integers and other registers.

```

25935 \cs_new_protected:Npn \_regex_replacement_cu_aux:Nw #1
25936 {
25937   \if_case:w \l__regex_replacement_csnames_int
25938   \tl_build_put_right:Nn \l__regex_build_tl
25939   { \exp_not:n { \exp_after:wN #1 \cs:w } }
25940   \else:
25941   \tl_build_put_right:Nn \l__regex_build_tl
25942   { \exp_not:n { \exp_after:wN \tl_to_str:V \cs:w } }
25943   \fi:
25944   \int_incr:N \l__regex_replacement_csnames_int
25945 }

```

(End definition for `_regex_replacement_cu_aux:Nw`.)

`_regex_replacement_u:w` Check that `\u` is followed by a left brace. If so, start a control sequence with `\cs:w`, which is then unpacked either with `\exp_not:V` or `\tl_to_str:V` depending on the current context.

```

25946 \cs_new_protected:Npn \_regex_replacement_u:w #1#2
25947 {
25948   \_regex_two_if_eq:NNNTF
25949   #1 #2 \_regex_replacement_normal:n \c_left_brace_str
25950   { \_regex_replacement_cu_aux:Nw \exp_not:V }
25951   { \_regex_replacement_error:NNN u #1#2 }
25952 }

```

(End definition for _regex_replacement_u:w.)

_regex_replacement_rbrace:N Within a \c{...} or \u{...} construction, end the control sequence, and decrease the brace count. Otherwise, this is a raw right brace.

```

25953 \cs_new_protected:Npn \_regex_replacement_rbrace:N #1
25954 {
25955   \if_int_compare:w \l__regex_replacement_csnames_int > 0 \exp_stop_f:
25956     \tl_build_put_right:Nn \l__regex_build_tl { \cs_end: }
25957     \int_decr:N \l__regex_replacement_csnames_int
25958   \else:
25959     \_regex_replacement_normal:n {#1}
25960   \fi:
25961 }

```

(End definition for _regex_replacement_rbrace:N.)

41.6.6 Characters in replacement

_regex_replacement_cat:NNN Here, #1 is a letter among BEMTPUDSLOA and #2#3 denote the next character. Complain if we reach the end of the replacement or if the construction appears inside \c{...} or \u{...}, and detect the case of a parenthesis. In that case, store the current category in a sequence and switch to a new one.

```

25962 \cs_new_protected:Npn \_regex_replacement_cat:NNN #1#2#3
25963 {
25964   \token_if_eq_meaning:NNTF \prg_do_nothing: #3
25965   { \_kernel_msg_error:nn { kernel } { replacement-catcode-end } }
25966   {
25967     \int_compare:nNnTF { \l__regex_replacement_csnames_int } > 0
25968     {
25969       \_kernel_msg_error:nnnn
25970       { kernel } { replacement-catcode-in-cs } {#1} {#3}
25971       #2 #3
25972     }
25973     {
25974       \_regex_two_if_eq:NNNTF #2 #3 \_regex_replacement_normal:n (
25975       {
25976         \seq_push:NV \l__regex_replacement_category_seq
25977         \l__regex_replacement_category_tl
25978         \tl_set:Nn \l__regex_replacement_category_tl {#1}
25979       }
25980       {
25981         \token_if_eq_meaning:NNT #2 \_regex_replacement_escaped:N
25982         {
25983           \_regex_char_if_alphanumeric:NTF #3
25984           {
25985             \_kernel_msg_error:nnnn
25986             { kernel } { replacement-catcode-escaped }
25987             {#1} {#3}
25988           }
25989           { }
25990         }
25991         \use:c { \_regex_replacement_c_#1:w } #2 #3
25992       }
25993     }

```

```

25994     }
25995 }

```

(End definition for `_regex_replacement_cat:NNN`.)

We now need to change the category code of the null character many times, hence work in a group. The catcode-specific macros below are defined in alphabetical order; if you are trying to understand the code, start from the end of the alphabet as those categories are simpler than active or begin-group.

```

25996 \group_begin:

```

`_regex_replacement_char:nnN` The only way to produce an arbitrary character-catcode pair is to use the `\lowercase` or `\uppercase` primitives. This is a wrapper for our purposes. The first argument is the null character with various catcodes. The second and third arguments are grabbed from the input stream: `#3` is the character whose character code to reproduce. We could use `\char_generate:nn` but only for some catcodes (active characters and spaces are not supported).

```

25997 \cs_new_protected:Npn \_regex_replacement_char:nnN #1#2#3
25998 {
25999   \tex_lccode:D 0 = '#3 \scan_stop:
26000   \tex_lowercase:D { \tl_build_put_right:Nn \l__regex_build_tl {#1} }
26001 }

```

(End definition for `_regex_replacement_char:nnN`.)

`_regex_replacement_c_A:w` For an active character, expansion must be avoided, twice because we later do two `x`-expansions, to unpack `\toks` for the query, and to expand their contents to tokens of the query.

```

26002 \char_set_catcode_active:N \^^@
26003 \cs_new_protected:Npn \_regex_replacement_c_A:w
26004 { \_regex_replacement_char:nnN { \exp_not:n { \exp_not:N \^^@ } } }

```

(End definition for `_regex_replacement_c_A:w`.)

`_regex_replacement_c_B:w` An explicit begin-group token increases the balance, unless within a `\c{...}` or `\u{...}` construction. Add the desired begin-group character, using the standard `\if_false:` trick. We eventually `x`-expand twice. The first time must yield a balanced token list, and the second one gives the bare begin-group token. The `\exp_after:wN` is not strictly needed, but is more consistent with `l3tl`-analysis.

```

26005 \char_set_catcode_group_begin:N \^^@
26006 \cs_new_protected:Npn \_regex_replacement_c_B:w
26007 {
26008   \if_int_compare:w \l__regex_replacement_csnames_int = 0 \exp_stop_f:
26009   \int_incr:N \l__regex_balance_int
26010   \fi:
26011   \_regex_replacement_char:nnN
26012   { \exp_not:n { \exp_after:wN \^^@ \if_false: } \fi: } }
26013 }

```

(End definition for `_regex_replacement_c_B:w`.)

`_regex_replacement_c_C:w` This is not quite catcode-related: when the user requests a character with category “control sequence”, the one-character control symbol is returned. As for the active character, we prepare for two x-expansions.

```
26014 \cs_new_protected:Npn \_regex_replacement_c_C:w #1#2
26015 {
26016   \tl_build_put_right:Nn \l__regex_build_tl
26017   { \exp_not:N \exp_not:N \exp_not:c {#2} }
26018 }
```

(End definition for _regex_replacement_c_C:w.)

`_regex_replacement_c_D:w` Subscripts fit the mould: `\lowercase` the null byte with the correct category.

```
26019 \char_set_catcode_math_subscript:N \^^@
26020 \cs_new_protected:Npn \_regex_replacement_c_D:w
26021 { \_regex_replacement_char:nNN { ^^@ } }
```

(End definition for _regex_replacement_c_D:w.)

`_regex_replacement_c_E:w` Similar to the begin-group case, the second x-expansion produces the bare end-group token.

```
26022 \char_set_catcode_group_end:N \^^@
26023 \cs_new_protected:Npn \_regex_replacement_c_E:w
26024 {
26025   \if_int_compare:w \l__regex_replacement_csnames_int = 0 \exp_stop_f:
26026   \int_decr:N \l__regex_balance_int
26027   \fi:
26028   \_regex_replacement_char:nNN
26029   { \exp_not:n { \if_false: { \fi: ^^@ } } }
26030 }
```

(End definition for _regex_replacement_c_E:w.)

`_regex_replacement_c_L:w` Simply `\lowercase` a letter null byte to produce an arbitrary letter.

```
26031 \char_set_catcode_letter:N \^^@
26032 \cs_new_protected:Npn \_regex_replacement_c_L:w
26033 { \_regex_replacement_char:nNN { ^^@ } }
```

(End definition for _regex_replacement_c_L:w.)

`_regex_replacement_c_M:w` No surprise here, we lowercase the null math toggle.

```
26034 \char_set_catcode_math_toggle:N \^^@
26035 \cs_new_protected:Npn \_regex_replacement_c_M:w
26036 { \_regex_replacement_char:nNN { ^^@ } }
```

(End definition for _regex_replacement_c_M:w.)

`_regex_replacement_c_O:w` Lowercase an other null byte.

```
26037 \char_set_catcode_other:N \^^@
26038 \cs_new_protected:Npn \_regex_replacement_c_O:w
26039 { \_regex_replacement_char:nNN { ^^@ } }
```

(End definition for _regex_replacement_c_O:w.)

`_regex_replacement_c_P:w` For macro parameters, expansion is a tricky issue. We need to prepare for two x-expansions and passing through various macro definitions. Note that we cannot replace one `\exp_not:n` by doubling the macro parameter characters because this would misbehave if a mischievous user asks for `\c{\cP\#}`, since that macro parameter character would be doubled.

```

26040 \char_set_catcode_parameter:N \^^@
26041 \cs_new_protected:Npn \_regex_replacement_c_P:w
26042 {
26043   \_regex_replacement_char:nNN
26044   { \exp_not:n { \exp_not:n { ^^@^^@^^@^^@ } } }
26045 }

```

(End definition for _regex_replacement_c_P:w.)

`_regex_replacement_c_S:w` Spaces are normalized on input by T_EX to have character code 32. It is in fact impossible to get a token with character code 0 and category code 10. Hence we use 32 instead of 0 as our base character.

```

26046 \cs_new_protected:Npn \_regex_replacement_c_S:w #1#2
26047 {
26048   \if_int_compare:w '#2 = 0 \exp_stop_f:
26049   \_kernel_msg_error:nn { kernel } { replacement-null-space }
26050   \fi:
26051   \tex_lccode:D '\ = '#2 \scan_stop:
26052   \tex_lowercase:D { \tl_build_put_right:Nn \l__regex_build_tl {~} }
26053 }

```

(End definition for _regex_replacement_c_S:w.)

`_regex_replacement_c_T:w` No surprise for alignment tabs here. Those are surrounded by the appropriate braces whenever necessary, hence they don't cause trouble in alignment settings.

```

26054 \char_set_catcode_alignment:N \^^@
26055 \cs_new_protected:Npn \_regex_replacement_c_T:w
26056 { \_regex_replacement_char:nNN { ^^@ } }

```

(End definition for _regex_replacement_c_T:w.)

`_regex_replacement_c_U:w` Simple call to `_regex_replacement_char:nNN` which lowercases the math superscript `^^@`.

```

26057 \char_set_catcode_math_superscript:N \^^@
26058 \cs_new_protected:Npn \_regex_replacement_c_U:w
26059 { \_regex_replacement_char:nNN { ^^@ } }

```

(End definition for _regex_replacement_c_U:w.)

Restore the catcode of the null byte.

```

26060 \group_end:

```

41.6.7 An error

`_regex_replacement_error:NNN` Simple error reporting by calling one of the messages `replacement-c`, `replacement-g`, or `replacement-u`.

```
26061 \cs_new_protected:Npn \_regex_replacement_error:NNN #1#2#3
26062 {
26063     \__kernel_msg_error:nnx { kernel } { replacement-#1 } {#3}
26064     #2 #3
26065 }
```

(End definition for `_regex_replacement_error:NNN`.)

41.7 User functions

`\regex_new:N` Before being assigned a sensible value, a regex variable matches nothing.

```
26066 \cs_new_protected:Npn \regex_new:N #1
26067 { \cs_new_eq:NN #1 \c__regex_no_match_regex }
```

(End definition for `\regex_new:N`. This function is documented on page 234.)

`\l_tmpa_regex` The usual scratch space.

```
\l_tmpb_regex 26068 \regex_new:N \l_tmpa_regex
\g_tmpa_regex 26069 \regex_new:N \l_tmpb_regex
\g_tmpb_regex 26070 \regex_new:N \g_tmpa_regex
26071 \regex_new:N \g_tmpb_regex
```

(End definition for `\l_tmpa_regex` and others. These variables are documented on page 236.)

`\regex_set:Nn` Compile, then store the result in the user variable with the appropriate assignment function.

`\regex_gset:Nn`

`\regex_const:Nn`

```
26072 \cs_new_protected:Npn \regex_set:Nn #1#2
26073 {
26074     \__regex_compile:n {#2}
26075     \tl_set_eq:NN #1 \l__regex_internal_regex
26076 }
26077 \cs_new_protected:Npn \regex_gset:Nn #1#2
26078 {
26079     \__regex_compile:n {#2}
26080     \tl_gset_eq:NN #1 \l__regex_internal_regex
26081 }
26082 \cs_new_protected:Npn \regex_const:Nn #1#2
26083 {
26084     \__regex_compile:n {#2}
26085     \tl_const:Nx #1 { \exp_not:o \l__regex_internal_regex }
26086 }
```

(End definition for `\regex_set:Nn`, `\regex_gset:Nn`, and `\regex_const:Nn`. These functions are documented on page 234.)

`\regex_show:N` User functions: the `n` variant requires compilation first. Then show the variable with some appropriate text. The auxiliary is defined in a different section.

`\regex_show:n`

```
26087 \cs_new_protected:Npn \regex_show:n #1
26088 {
26089     \__regex_compile:n {#1}
26090     \__regex_show:N \l__regex_internal_regex
```

```

26091 \msg_show:nnxxxx { LaTeX / kernel } { show-regex }
26092 { \tl_to_str:n {#1} } { }
26093 { \l__regex_internal_a_tl } { }
26094 }
26095 \cs_new_protected:Npn \regex_show:N #1
26096 {
26097   \__kernel_chk_defined:NT #1
26098   {
26099     \__regex_show:N #1
26100     \msg_show:nnxxxx { LaTeX / kernel } { show-regex }
26101     { } { \token_to_str:N #1 }
26102     { \l__regex_internal_a_tl } { }
26103   }
26104 }

```

(End definition for `\regex_show:N` and `\regex_show:n`. These functions are documented on page 234.)

`\regex_match:nnTF` Those conditionals are based on a common auxiliary defined later. Its first argument
`\regex_match:NnTF` builds the NFA corresponding to the regex, and the second argument is the query token
list. Once we have performed the match, convert the resulting boolean to `\prg_return_`
`true:` or `false`.

```

26105 \prg_new_protected_conditional:Npnn \regex_match:nn #1#2 { T , F , TF }
26106 {
26107   \__regex_if_match:nn { \__regex_build:n {#1} } {#2}
26108   \__regex_return:
26109 }
26110 \prg_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
26111 {
26112   \__regex_if_match:nn { \__regex_build:N #1 } {#2}
26113   \__regex_return:
26114 }

```

(End definition for `\regex_match:nnTF` and `\regex_match:NnTF`. These functions are documented on page 234.)

`\regex_count:nnN` Again, use an auxiliary whose first argument builds the NFA.
`\regex_count:NnN`

```

26115 \cs_new_protected:Npn \regex_count:nnN #1
26116 { \__regex_count:nnN { \__regex_build:n {#1} } }
26117 \cs_new_protected:Npn \regex_count:NnN #1
26118 { \__regex_count:nnN { \__regex_build:N #1 } }

```

(End definition for `\regex_count:nnN` and `\regex_count:NnN`. These functions are documented on page 235.)

`\regex_extract_once:nnN` We define here 40 user functions, following a common pattern in terms of `:nnN` auxiliaries,
`\regex_extract_once:nnTF` defined in the coming subsections. The auxiliary is handed `__regex_build:n` or `__`
`\regex_extract_once:NnN` `regex_build:N` with the appropriate regex argument, then all other necessary arguments
`\regex_extract_once:NnTF` (replacement text, token list, *etc.* The conditionals call `__regex_return:` to return
`\regex_extract_all:nnN` either `true` or `false` once matching has been performed.

```

26119 \cs_set_protected:Npn \__regex_tmp:w #1#2#3
26120 {
26121   \cs_new_protected:Npn #2 ##1 { #1 { \__regex_build:n {##1} } }
26122   \cs_new_protected:Npn #3 ##1 { #1 { \__regex_build:N ##1 } }
26123   \prg_new_protected_conditional:Npnn #2 ##1##2##3 { T , F , TF }

```

```

\regex_replace_once:nnN
\regex_replace_once:NnN
\regex_replace_all:nnN
\regex_replace_all:NnN
\regex_replace_all:NnTF
\regex_split:nnN
\regex_split:nnTF
\regex_split:NnN
\regex_split:NnTF

```

```

26124     { #1 { \__regex_build:n {##1} } {##2} ##3 \__regex_return: }
26125     \prg_new_protected_conditional:Npnn #3 ##1##2##3 { T , F , TF }
26126     { #1 { \__regex_build:N ##1 } {##2} ##3 \__regex_return: }
26127   }
26128   \__regex_tmp:w \__regex_extract_once:nnN
26129   \regex_extract_once:nnN \regex_extract_once:NnN
26130   \__regex_tmp:w \__regex_extract_all:nnN
26131   \regex_extract_all:nnN \regex_extract_all:NnN
26132   \__regex_tmp:w \__regex_replace_once:nnN
26133   \regex_replace_once:nnN \regex_replace_once:NnN
26134   \__regex_tmp:w \__regex_replace_all:nnN
26135   \regex_replace_all:nnN \regex_replace_all:NnN
26136   \__regex_tmp:w \__regex_split:nnN \regex_split:nnN \regex_split:NnN

```

(End definition for `\regex_extract_once:nnNTF` and others. These functions are documented on page [235](#).)

41.7.1 Variables and helpers for user functions

`\l__regex_match_count_int` The number of matches found so far is stored in `\l__regex_match_count_int`. This is only used in the `\regex_count:nnN` functions.

```
26137 \int_new:N \l__regex_match_count_int
```

(End definition for `\l__regex_match_count_int`.)

`__regex_begin` `__regex_end` Those flags are raised to indicate extra begin-group or end-group tokens when extracting submatches.

```
26138 \flag_new:n { __regex_begin }
26139 \flag_new:n { __regex_end }
```

(End definition for `__regex_begin` and `__regex_end`.)

`\l__regex_min_submatch_int` `\l__regex_submatch_int` `\l__regex_zeroth_submatch_int` The end-points of each submatch are stored in two arrays whose index *submatch* ranges from `\l__regex_min_submatch_int` (inclusive) to `\l__regex_submatch_int` (exclusive). Each successful match comes with a 0-th submatch (the full match), and one match for each capturing group: submatches corresponding to the last successful match are labelled starting at `zeroth_submatch`. The entry `\l__regex_zeroth_submatch_int` in `\g__regex_submatch_prev_intarray` holds the position at which that match attempt started: this is used for splitting and replacements.

```
26140 \int_new:N \l__regex_min_submatch_int
26141 \int_new:N \l__regex_submatch_int
26142 \int_new:N \l__regex_zeroth_submatch_int
```

(End definition for `\l__regex_min_submatch_int`, `\l__regex_submatch_int`, and `\l__regex_zeroth_submatch_int`.)

`\g__regex_submatch_prev_intarray` Hold the place where the match attempt begun and the end-points of each submatch.

```

\g__regex_submatch_begin_intarray 26143 \intarray_new:Nn \g__regex_submatch_prev_intarray { 65536 }
\g__regex_submatch_end_intarray    26144 \intarray_new:Nn \g__regex_submatch_begin_intarray { 65536 }
26145 \intarray_new:Nn \g__regex_submatch_end_intarray { 65536 }
```

(End definition for `\g__regex_submatch_prev_intarray`, `\g__regex_submatch_begin_intarray`, and `\g__regex_submatch_end_intarray`.)

`__regex_return:` This function triggers either `\prg_return_false:` or `\prg_return_true:` as appropriate to whether a match was found or not. It is used by all user conditionals.

```

26146 \cs_new_protected:Npn \__regex_return:
26147 {
26148   \if_meaning:w \c_true_bool \g__regex_success_bool
26149   \prg_return_true:
26150   \else:
26151     \prg_return_false:
26152   \fi:
26153 }

```

(End definition for __regex_return:.)

41.7.2 Matching

`__regex_if_match:nn` We don't track submatches, and stop after a single match. Build the NFA with #1, and perform the match on the query #2.

```

26154 \cs_new_protected:Npn \__regex_if_match:nn #1#2
26155 {
26156   \group_begin:
26157   \__regex_disable_submatches:
26158   \__regex_single_match:
26159   #1
26160   \__regex_match:n {#2}
26161   \group_end:
26162 }

```

(End definition for __regex_if_match:nn.)

`__regex_count:nnN` Again, we don't care about submatches. Instead of aborting after the first “longest match” is found, we search for multiple matches, incrementing `\l__regex_match_count_int` every time to record the number of matches. Build the NFA and match. At the end, store the result in the user's variable.

```

26163 \cs_new_protected:Npn \__regex_count:nnN #1#2#3
26164 {
26165   \group_begin:
26166   \__regex_disable_submatches:
26167   \int_zero:N \l__regex_match_count_int
26168   \__regex_multi_match:n { \int_incr:N \l__regex_match_count_int }
26169   #1
26170   \__regex_match:n {#2}
26171   \exp_args:NNNo
26172   \group_end:
26173   \int_set:Nn #3 { \int_use:N \l__regex_match_count_int }
26174 }

```

(End definition for __regex_count:nnN.)

41.7.3 Extracting submatches

`__regex_extract_once:nnN` Match once or multiple times. After each match (or after the only match), extract the submatches using `__regex_extract:.` At the end, store the sequence containing all the submatches into the user variable #3 after closing the group.

```

26175 \cs_new_protected:Npn \__regex_extract_once:nnN #1#2#3
26176 {
26177   \group_begin:
26178   \__regex_single_match:
26179   #1
26180   \__regex_match:n {#2}
26181   \__regex_extract:
26182   \__regex_group_end_extract_seq:N #3
26183 }
26184 \cs_new_protected:Npn \__regex_extract_all:nnN #1#2#3
26185 {
26186   \group_begin:
26187   \__regex_multi_match:n { \__regex_extract: }
26188   #1
26189   \__regex_match:n {#2}
26190   \__regex_group_end_extract_seq:N #3
26191 }

```

(End definition for __regex_extract_once:nnN and __regex_extract_all:nnN.)

__regex_split:nnN Splitting at submatches is a bit more tricky. For each match, extract all submatches, and replace the zeroth submatch by the part of the query between the start of the match attempt and the start of the zeroth submatch. This is inhibited if the delimiter matched an empty token list at the start of this match attempt. After the last match, store the last part of the token list, which ranges from the start of the match attempt to the end of the query. This step is inhibited if the last match was empty and at the very end: decrement \l__regex_submatch_int, which controls which matches will be used.

```

26192 \cs_new_protected:Npn \__regex_split:nnN #1#2#3
26193 {
26194   \group_begin:
26195   \__regex_multi_match:n
26196   {
26197     \if_int_compare:w
26198       \l__regex_start_pos_int < \l__regex_success_pos_int
26199       \__regex_extract:
26200       \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
26201       { \l__regex_zeroth_submatch_int } { 0 }
26202       \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
26203       { \l__regex_zeroth_submatch_int }
26204       {
26205         \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray
26206         { \l__regex_zeroth_submatch_int }
26207       }
26208       \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
26209       { \l__regex_zeroth_submatch_int }
26210       { \l__regex_start_pos_int }
26211     \fi:
26212   }
26213   #1
26214   \__regex_match:n {#2}
26215   (assert)\assert_int:n { \l__regex_curr_pos_int = \l__regex_max_pos_int }
26216   \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
26217   { \l__regex_submatch_int } { 0 }
26218   \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray

```

```

26219     { \l__regex_submatch_int }
26220     { \l__regex_max_pos_int }
26221     \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
26222     { \l__regex_submatch_int }
26223     { \l__regex_start_pos_int }
26224     \int_incr:N \l__regex_submatch_int
26225     \if_meaning:w \c_true_bool \l__regex_empty_success_bool
26226     \if_int_compare:w \l__regex_start_pos_int = \l__regex_max_pos_int
26227     \int_decr:N \l__regex_submatch_int
26228     \fi:
26229     \fi:
26230     \__regex_group_end_extract_seq:N #3
26231   }

```

(End definition for __regex_split:nnN.)

__regex_group_end_extract_seq:N The end-points of submatches are stored as entries of two arrays from \l__regex_min_submatch_int to \l__regex_submatch_int (exclusive). Extract the relevant ranges into \l__regex_internal_a_tl. We detect unbalanced results using the two flags __regex_begin and __regex_end, raised whenever we see too many begin-group or end-group tokens in a submatch.

```

26232 \cs_new_protected:Npn \__regex_group_end_extract_seq:N #1
26233 {
26234   \flag_clear:n { __regex_begin }
26235   \flag_clear:n { __regex_end }
26236   \seq_set_from_function:NnN \l__regex_internal_seq
26237   {
26238     \int_step_function:nnN { \l__regex_min_submatch_int }
26239     { \l__regex_submatch_int - 1 }
26240   }
26241   \__regex_extract_seq_aux:n
26242   \int_compare:nNnF
26243   {
26244     \flag_height:n { __regex_begin } +
26245     \flag_height:n { __regex_end }
26246   }
26247   = 0
26248   {
26249     \__kernel_msg_error:nnxxx { kernel } { result-unbalanced }
26250     { splitting~or~extracting~submatches }
26251     { \flag_height:n { __regex_end } }
26252     { \flag_height:n { __regex_begin } }
26253   }
26254   \seq_set_map_x:NNn \l__regex_internal_seq \l__regex_internal_seq {##1}
26255   \exp_args:NNNo
26256   \group_end:
26257   \tl_set:Nn #1 { \l__regex_internal_seq }
26258 }

```

(End definition for __regex_group_end_extract_seq:N.)

__regex_extract_seq_aux:n The :n auxiliary builds one item of the sequence of submatches. First compute the brace balance of the submatch, then extract the submatch from the query, adding the appropriate braces and raising a flag if the submatch is not balanced.

```

26259 \cs_new:Npn \__regex_extract_seq_aux:n #1
26260 {
26261   \exp_after:wN \__regex_extract_seq_aux:ww
26262   \int_value:w \__regex_submatch_balance:n {#1} ; #1;
26263 }
26264 \cs_new:Npn \__regex_extract_seq_aux:ww #1; #2;
26265 {
26266   \if_int_compare:w #1 < 0 \exp_stop_f:
26267     \flag_raise:n { __regex_end }
26268     \prg_replicate:nn {-#1} { \exp_not:n { { \if_false: } \fi: } }
26269   \fi:
26270   \__regex_query_submatch:n {#2}
26271   \if_int_compare:w #1 > 0 \exp_stop_f:
26272     \flag_raise:n { __regex_begin }
26273     \prg_replicate:nn {#1} { \exp_not:n { \if_false: { \fi: } } }
26274   \fi:
26275 }

```

(End definition for __regex_extract_seq_aux:n and __regex_extract_seq_aux:ww.)

__regex_extract: Our task here is to extract from the property list \l__regex_success_submatches_prop the list of end-points of submatches, and store them in appropriate array entries, from \l__regex_extract_b:wn the list of end-points of submatches, and store them in appropriate array entries, from \l__regex_extract_e:wn \l__regex_zeroth_submatch_int upwards. We begin by emptying those entries. Then for each $\langle key \rangle$ - $\langle value \rangle$ pair in the property list update the appropriate entry. This is somewhat a hack: the $\langle key \rangle$ is a non-negative integer followed by < or >, which we use in a comparison to -1. At the end, store the information about the position at which the match attempt started, in \g__regex_submatch_prev_intarray.

```

26276 \cs_new_protected:Npn \__regex_extract:
26277 {
26278   \if_meaning:w \c_true_bool \g__regex_success_bool
26279     \int_set_eq:NN \l__regex_zeroth_submatch_int \l__regex_submatch_int
26280     \prg_replicate:nn \l__regex_capturing_group_int
26281     {
26282       \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
26283       { \l__regex_submatch_int } { 0 }
26284       \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
26285       { \l__regex_submatch_int } { 0 }
26286       \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
26287       { \l__regex_submatch_int } { 0 }
26288       \int_incr:N \l__regex_submatch_int
26289     }
26290   \prop_map_inline:Nn \l__regex_success_submatches_prop
26291   {
26292     \if_int_compare:w ##1 - 1 \exp_stop_f:
26293       \exp_after:wN \__regex_extract_e:wn \int_value:w
26294     \else:
26295       \exp_after:wN \__regex_extract_b:wn \int_value:w
26296     \fi:
26297     \__regex_int_eval:w \l__regex_zeroth_submatch_int + ##1 {##2}
26298   }
26299   \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
26300   { \l__regex_zeroth_submatch_int } { \l__regex_start_pos_int }
26301   \fi:
26302 }

```

```

26303 \cs_new_protected:Npn \__regex_extract_b:wn #1 < #2
26304 {
26305   \__kernel_intarray_gset:Nnn
26306   \g__regex_submatch_begin_intarray {#1} {#2}
26307 }
26308 \cs_new_protected:Npn \__regex_extract_e:wn #1 > #2
26309 { \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray {#1} {#2} }

```

(End definition for __regex_extract:, __regex_extract_b:wn, and __regex_extract_e:wn.)

41.7.4 Replacement

__regex_replace_once:nnN Build the NFA and the replacement functions, then find a single match. If the match failed, simply exit the group. Otherwise, we do the replacement. Extract submatches. Compute the brace balance corresponding to replacing this match by the replacement (this depends on submatches). Prepare the replaced token list: the replacement function produces the tokens from the start of the query to the start of the match and the replacement text for this match; we need to add the tokens from the end of the match to the end of the query. Finally, store the result in the user's variable after closing the group: this step involves an additional x-expansion, and checks that braces are balanced in the final result.

```

26310 \cs_new_protected:Npn \__regex_replace_once:nnN #1#2#3
26311 {
26312   \group_begin:
26313   \__regex_single_match:
26314   #1
26315   \__regex_replacement:n {#2}
26316   \exp_args:No \__regex_match:n { #3 }
26317   \if_meaning:w \c_false_bool \g__regex_success_bool
26318   \group_end:
26319   \else:
26320     \__regex_extract:
26321     \int_set:Nn \l__regex_balance_int
26322     {
26323       \__regex_replacement_balance_one_match:n
26324       { \l__regex_zeroth_submatch_int }
26325     }
26326     \tl_set:Nx \l__regex_internal_a_tl
26327     {
26328       \__regex_replacement_do_one_match:n
26329       { \l__regex_zeroth_submatch_int }
26330       \__regex_query_range:nn
26331       {
26332         \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
26333         { \l__regex_zeroth_submatch_int }
26334       }
26335       { \l__regex_max_pos_int }
26336     }
26337     \__regex_group_end_replace:N #3
26338   \fi:
26339 }

```

(End definition for __regex_replace_once:nnN.)

`__regex_replace_all:nnN` Match multiple times, and for every match, extract submatches and additionally store the position at which the match attempt started. The entries from `\l__regex_min_submatch_int` to `\l__regex_submatch_int` hold information about submatches of every match in order; each match corresponds to `\l__regex_capturing_group_int` consecutive entries. Compute the brace balance corresponding to doing all the replacements: this is the sum of brace balances for replacing each match. Join together the replacement texts for each match (including the part of the query before the match), and the end of the query.

```

26340 \cs_new_protected:Npn \__regex_replace_all:nnN #1#2#3
26341 {
26342   \group_begin:
26343   \__regex_multi_match:n { \__regex_extract: }
26344   #1
26345   \__regex_replacement:n {#2}
26346   \exp_args:No \__regex_match:n {#3}
26347   \int_set:Nn \l__regex_balance_int
26348   {
26349     0
26350     \int_step_function:nnnN
26351     { \l__regex_min_submatch_int }
26352     \l__regex_capturing_group_int
26353     { \l__regex_submatch_int - 1 }
26354     \__regex_replacement_balance_one_match:n
26355   }
26356   \tl_set:Nx \l__regex_internal_a_tl
26357   {
26358     \int_step_function:nnnN
26359     { \l__regex_min_submatch_int }
26360     \l__regex_capturing_group_int
26361     { \l__regex_submatch_int - 1 }
26362     \__regex_replacement_do_one_match:n
26363     \__regex_query_range:nn
26364     \l__regex_start_pos_int \l__regex_max_pos_int
26365   }
26366   \__regex_group_end_replace:N #3
26367 }

```

(End definition for `__regex_replace_all:nnN`.)

`__regex_group_end_replace:N` If the brace balance is not 0, raise an error. Then set the user's variable `#1` to the x-expansion of `\l__regex_internal_a_tl`, adding the appropriate braces to produce a balanced result. And end the group.

```

26368 \cs_new_protected:Npn \__regex_group_end_replace:N #1
26369 {
26370   \if_int_compare:w \l__regex_balance_int = 0 \exp_stop_f:
26371   \else:
26372     \__kernel_msg_error:nnxxx { kernel } { result-unbalanced }
26373     { replacing }
26374     { \int_max:nn { - \l__regex_balance_int } { 0 } }
26375     { \int_max:nn { \l__regex_balance_int } { 0 } }
26376   \fi:
26377   \use:x
26378   {

```



```

26421 }
26422 \__kernel_msg_new:nnnn { kernel } { missing-rparen }
26423 {
26424     Missing-right~
26425     \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } ~
26426     inserted-in-regular-expression.
26427 }
26428 {
26429     LaTeX-was-given-a-regular-expression-with-\int_eval:n {#1} ~
26430     more-left-parentheses-than-right-parentheses.
26431 }
26432 \__kernel_msg_new:nnnn { kernel } { extra-rparen }
26433 { Extra-right-parenthesis-ignored-in-regular-expression. }
26434 {
26435     LaTeX-came-across-a-closing-parenthesis-when-no-submatch-group-
26436     was-open.~The-parenthesis-will-be-ignored.
26437 }

```

Some escaped alphanumerics are not allowed everywhere.

```

26438 \__kernel_msg_new:nnnn { kernel } { bad-escape }
26439 {
26440     Invalid-escape~'\iow_char:N\|#1'~
26441     \__regex_if_in_cs:TF { within-a-control-sequence. }
26442     {
26443         \__regex_if_in_class:TF
26444         { in-a-character-class. }
26445         { following-a-category-test. }
26446     }
26447 }
26448 {
26449     The-escape-sequence~'\iow_char:N\|#1'~may-not-appear~
26450     \__regex_if_in_cs:TF
26451     {
26452         within-a-control-sequence-test-introduced-by~
26453         '\iow_char:N\|c\iow_char:N\{' .
26454     }
26455     {
26456         \__regex_if_in_class:TF
26457         { within-a-character-class~ }
26458         { following-a-category-test-such-as~'\iow_char:N\|cL'~ }
26459         because-it-does-not-match-exactly-one-character.
26460     }
26461 }

```

Range errors.

```

26462 \__kernel_msg_new:nnnn { kernel } { range-missing-end }
26463 { Invalid-end-point-for-range~'#1-#2'~in-character-class. }
26464 {
26465     The-end-point~'#2'~of-the-range~'#1-#2'~may-not-serve-as-an-
26466     end-point-for-a-range:~alphanumeric-characters-should-not-be-
26467     escaped,~and-non-alphanumeric-characters-should-be-escaped.
26468 }
26469 \__kernel_msg_new:nnnn { kernel } { range-backwards }
26470 { Range~'#1-#2'~out-of-order-in-character-class. }
26471 {

```



```

26472 In-ranges-of-characters~'[x-y]~appearing-in-character-classes,~
26473 the-first-character-code-must-not-be-larger-than-the-second.~
26474 Here,~'#1'~has-character-code~\int_eval:n {'#1},~while~
26475 '#2'~has-character-code~\int_eval:n {'#2}.
26476 }

```

Errors related to \c and \u.

```

26477 \__kernel_msg_new:nnnn { kernel } { c-bad-mode }
26478 { Invalid-nested~'\iow_char:N\\c'~escape-in-regular-expression. }
26479 {
26480 The~'\iow_char:N\\c'~escape-cannot-be-used-within~
26481 a-control-sequence-test~'\iow_char:N\\c{...}'~
26482 nor-another-category-test.~
26483 To-combine-several-category-tests,~use~'\iow_char:N\\c[...]'.
26484 }
26485 \__kernel_msg_new:nnnn { kernel } { c-C-invalid }
26486 { '\iow_char:N\\cC'~should-be-followed-by~'.~'~or~'(',~not~'#1'. }
26487 {
26488 The~'\iow_char:N\\cC'~construction-restricts-the-next-item-to-be-a~
26489 control-sequence-or-the-next-group-to-be-made-of-control-sequences.~
26490 It-only-makes-sense-to-follow-it-by~'.~'~or-by-a-group.
26491 }
26492 \__kernel_msg_new:nnnn { kernel } { c-lparen-in-class }
26493 { Catcode-test-cannot-apply-to-group-in-character-class }
26494 {
26495 Construction-such-as~'\iow_char:N\\cL(abc)'~are-not-allowed-inside-a~
26496 class~'[...]~because-classes-do-not-match-multiple-characters-at-once.
26497 }
26498 \__kernel_msg_new:nnnn { kernel } { c-missing-rbrace }
26499 { Missing-right-brace-inserted-for~'\iow_char:N\\c'~escape. }
26500 {
26501 LaTeX-was-given-a-regular-expression-where-a~
26502 '\iow_char:N\\c\iow_char:N{...~construction-was-not-ended~
26503 with-a-closing-brace~'\iow_char:N}''.
26504 }
26505 \__kernel_msg_new:nnnn { kernel } { c-missing-rbrack }
26506 { Missing-right-bracket-inserted-for~'\iow_char:N\\c'~escape. }
26507 {
26508 A-construction~'\iow_char:N\\c[...~appears-in-a~
26509 regular-expression,~but-the-closing~']'~is-not-present.
26510 }
26511 \__kernel_msg_new:nnnn { kernel } { c-missing-category }
26512 { Invalid-character~'#1'~following~'\iow_char:N\\c'~escape. }
26513 {
26514 In-regular-expressions,~the~'\iow_char:N\\c'~escape-sequence~
26515 may-only-be-followed-by~a-left-brace,~a-left-bracket,~or~a~
26516 capital-letter-representing-a-character-category,~namely~
26517 one-of~'ABCDELMOPSTU'.
26518 }
26519 \__kernel_msg_new:nnnn { kernel } { c-trailing }
26520 { Trailing-category-code-escape~'\iow_char:N\\c'... }
26521 {
26522 A-regular-expression-ends-with~'\iow_char:N\\c'~followed~
26523 by-a-letter.~It-will-be-ignored.
26524 }

```

```

26525 \__kernel_msg_new:nnnn { kernel } { u-missing-lbrace }
26526 { Missing~left~brace~following~'\iow_char:N\\u'~escape. }
26527 {
26528   The~'\iow_char:N\\u'~escape~sequence~must~be~followed~by~
26529   a~brace~group~with~the~name~of~the~variable~to~use.
26530 }
26531 \__kernel_msg_new:nnnn { kernel } { u-missing-rbrace }
26532 { Missing~right~brace~inserted~for~'\iow_char:N\\u'~escape. }
26533 {
26534   LaTeX~
26535   \str_if_eq:eeTF { } {#2}
26536   { reached~the~end~of~the~string~ }
26537   { encountered~an~escaped~alphanumeric~character '\iow_char:N\\#2'~ }
26538   when~parsing~the~argument~of~an~
26539   '\iow_char:N\\u\iow_char:N{...}'~escape.
26540 }

```

Errors when encountering the POSIX syntax [:...:].

```

26541 \__kernel_msg_new:nnnn { kernel } { posix-unsupported }
26542 { POSIX~collating~element~'#1 ~ #1'~not~supported. }
26543 {
26544   The~' [.foo.] '~and~' [=bar=] '~syntaxes~have~a~special~meaning~
26545   in~POSIX~regular~expressions.~This~is~not~supported~by~LaTeX.~
26546   Maybe~you~forgot~to~escape~a~left~bracket~in~a~character~class?
26547 }
26548 \__kernel_msg_new:nnnn { kernel } { posix-unknown }
26549 { POSIX~class~'[:#1:]'~unknown. }
26550 {
26551   '[:#1:]'~is~not~among~the~known~POSIX~classes~
26552   '[:alnum:]',~'[:alpha:]',~'[:ascii:]',~'[:blank:]',~
26553   '[:cntrl:]',~'[:digit:]',~'[:graph:]',~'[:lower:]',~
26554   '[:print:]',~'[:punct:]',~'[:space:]',~'[:upper:]',~
26555   '[:word:]',~and~'[:xdigit:]'.
26556 }
26557 \__kernel_msg_new:nnnn { kernel } { posix-missing-close }
26558 { Missing~closing~':'~for~POSIX~class. }
26559 { The~POSIX~syntax~'#1'~must~be~followed~by~':'',~not~'#2'. }

```

In various cases, the result of a `l3regex` operation can leave us with an unbalanced token list, which we must re-balance by adding begin-group or end-group character tokens.

```

26560 \__kernel_msg_new:nnnn { kernel } { result-unbalanced }
26561 { Missing~brace~inserted~when~#1. }
26562 {
26563   LaTeX~was~asked~to~do~some~regular~expression~operation,~
26564   and~the~resulting~token~list~would~not~have~the~same~number~
26565   of~begin~group~and~end~group~tokens.~Braces~were~inserted:~
26566   #2~left,~#3~right.
26567 }

```

Error message for unknown options.

```

26568 \__kernel_msg_new:nnnn { kernel } { unknown-option }
26569 { Unknown~option~'#1'~for~regular~expressions. }
26570 {
26571   The~only~available~option~is~'case-insensitive',~toggled~by~

```

```

26572     '(?i)'~and~'(?-i)'.
26573 }
26574 \_kernel_msg_new:nnnn { kernel } { special-group-unknown }
26575 { Unknown~special~group~'#1~...'~in~a~regular~expression. }
26576 {
26577     The~only~valid~constructions~starting~with~'('?~are~
26578     '(:~...~)',~'(?|~...~)',~'(?i)',~and~'(?-i)'.
26579 }

```

Errors in the replacement text.

```

26580 \_kernel_msg_new:nnnn { kernel } { replacement-c }
26581 { Misused~'\iow_char:N\\c'~command~in~a~replacement~text. }
26582 {
26583     In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
26584     can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~
26585     or~a~brace~group,~not~by~'#1'.
26586 }
26587 \_kernel_msg_new:nnnn { kernel } { replacement-u }
26588 { Misused~'\iow_char:N\\u'~command~in~a~replacement~text. }
26589 {
26590     In~a~replacement~text,~the~'\iow_char:N\\u'~escape~sequence~
26591     must~be~followed~by~a~brace~group~holding~the~name~of~the~
26592     variable~to~use.
26593 }
26594 \_kernel_msg_new:nnnn { kernel } { replacement-g }
26595 {
26596     Missing~brace~for~the~'\iow_char:N\\g'~construction~
26597     in~a~replacement~text.
26598 }
26599 {
26600     In~the~replacement~text~for~a~regular~expression~search,~
26601     submatches~are~represented~either~as~'\iow_char:N \\g{dd..d}',~
26602     or~'\d',~where~'d'~are~single~digits.~Here,~a~brace~is~missing.
26603 }
26604 \_kernel_msg_new:nnnn { kernel } { replacement-catcode-end }
26605 {
26606     Missing~character~for~the~'\iow_char:N\\c<category><character>'~
26607     construction~in~a~replacement~text.
26608 }
26609 {
26610     In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
26611     can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
26612     the~character~category.~Then,~a~character~must~follow.~LaTeX~
26613     reached~the~end~of~the~replacement~when~looking~for~that.
26614 }
26615 \_kernel_msg_new:nnnn { kernel } { replacement-catcode-escaped }
26616 {
26617     Escaped~letter~or~digit~after~category~code~in~replacement~text.
26618 }
26619 {
26620     In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
26621     can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
26622     the~character~category.~Then,~a~character~must~follow,~not~
26623     '\iow_char:N\\#2'.
26624 }

```

```

26625 \__kernel_msg_new:nnnn { kernel } { replacement-catcode-in-cs }
26626 {
26627   Category-code~'\iow_char:N\c#1#3'~ignored-inside~
26628   '\iow_char:N\c\{...\}'~in~a~replacement~text.
26629 }
26630 {
26631   In~a~replacement~text,~the~category~codes~of~the~argument~of~
26632   '\iow_char:N\c\{...\}'~are~ignored~when~building~the~control~
26633   sequence~name.
26634 }
26635 \__kernel_msg_new:nnnn { kernel } { replacement-null-space }
26636 { TeX~cannot~build~a~space~token~with~character~code~0. }
26637 {
26638   You~asked~for~a~character~token~with~category~space,~
26639   and~character~code~0,~for~instance~through~
26640   '\iow_char:N\cS\iow_char:N\c00'.~
26641   This~specific~case~is~impossible~and~will~be~replaced~
26642   by~a~normal~space.
26643 }
26644 \__kernel_msg_new:nnnn { kernel } { replacement-missing-rbrace }
26645 { Missing~right~brace~inserted~in~replacement~text. }
26646 {
26647   There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
26648   missing~right~\int_compare:nTF { #1 = 1 } { brace } { braces } .
26649 }
26650 \__kernel_msg_new:nnnn { kernel } { replacement-missing-rparen }
26651 { Missing~right~parenthesis~inserted~in~replacement~text. }
26652 {
26653   There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
26654   missing~right~
26655   \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } .
26656 }

```

Used when showing a regex.

```

26657 \__kernel_msg_new:nnn { kernel } { show-regex }
26658 {
26659   >~Compiled~regex~
26660   \tl_if_empty:nTF {#1} { variable~ #2 } { {#1} } :
26661   #3
26662 }

```

`__regex_msg_repeated:nnN` This is not technically a message, but seems related enough to go there. The arguments are: #1 is the minimum number of repetitions; #2 is the number of allowed extra repetitions (−1 for infinite number), and #3 tells us about laziness.

```

26663 \cs_new:Npn \__regex_msg_repeated:nnN #1#2#3
26664 {
26665   \str_if_eq:eeF { #1 #2 } { 1 0 }
26666   {
26667     , ~ repeated ~
26668     \int_case:nnF {#2}
26669     {
26670       { -1 } { #1~or~more~times,~\bool_if:NTF #3 { lazy } { greedy } }
26671       { 0 } { #1~times }
26672     }

```

```

26673         {
26674             between~#1~and~\int_eval:n {#1+#2}~times,~
26675             \bool_if:NTF #3 { lazy } { greedy }
26676         }
26677     }
26678 }

```

(End definition for `__regex_msg_repeated:nnN`.)

41.9 Code for tracing

There is a more extensive implementation of tracing in the `l3trial` package `l3trace`. Function names are a bit different but could be merged.

`__regex_trace_push:nnN` Here `#1` is the module name (`regex`) and `#2` is typically 1. If the module's current tracing level is less than `#2` show nothing, otherwise write `#3` to the terminal.

```

\__regex_trace_pop:nnN
\__regex_trace:nnx
26679 \cs_new_protected:Npn \__regex_trace_push:nnN #1#2#3
26680 { \__regex_trace:nnx {#1} {#2} { entering~ \token_to_str:N #3 } }
26681 \cs_new_protected:Npn \__regex_trace_pop:nnN #1#2#3
26682 { \__regex_trace:nnx {#1} {#2} { leaving~ \token_to_str:N #3 } }
26683 \cs_new_protected:Npn \__regex_trace:nnx #1#2#3
26684 {
26685     \int_compare:nNnF
26686     { \int_use:c { g__regex_trace_#1_int } } < {#2}
26687     { \iow_term:x { Trace:~#3 } }
26688 }

```

(End definition for `__regex_trace_push:nnN`, `__regex_trace_pop:nnN`, and `__regex_trace:nnx`.)

`\g__regex_trace_regex_int` No tracing when that is zero.

```

26689 \int_new:N \g__regex_trace_regex_int

```

(End definition for `\g__regex_trace_regex_int`.)

`__regex_trace_states:n` This function lists the contents of all states of the NFA, stored in `\toks` from 0 to `\l__-regex_max_state_int` (excluded).

```

26690 \cs_new_protected:Npn \__regex_trace_states:n #1
26691 {
26692     \int_step_inline:nnn
26693     \l__regex_min_state_int
26694     { \l__regex_max_state_int - 1 }
26695     {
26696         \__regex_trace:nnx { regex } {#1}
26697         { \iow_char:N \toks ##1 = { \__regex_toks_use:w ##1 } }
26698     }
26699 }

```

(End definition for `__regex_trace_states:n`.)

```

26700 \endpackage

```

42 l3box implementation

```
26701 \*package\
```

```
26702 \@@=box\
```

42.1 Support code

`__box_dim_eval:w` Evaluating a dimension expression expandably. The only difference with `\dim_eval:n` is the lack of `\dim_use:N`, to produce an internal dimension rather than expand it into characters.

```
26703 \cs_new_eq:NN \__box_dim_eval:w \tex_dimexpr:D
```

```
26704 \cs_new:Npn \__box_dim_eval:n #1
```

```
26705 { \__box_dim_eval:w #1 \scan_stop: }
```

(End definition for `__box_dim_eval:w` and `__box_dim_eval:n`.)

42.2 Creating and initialising boxes

The following test files are used for this code: `m3box001.lvt`.

`\box_new:N` Defining a new `\box` register: remember that box 255 is not generally available.

```
\box_new:c 26706 \cs_new_protected:Npn \box_new:N #1
26707 {
26708     \__kernel_chk_if_free_cs:N #1
26709     \cs:w newbox \cs_end: #1
26710 }
26711 \cs_generate_variant:Nn \box_new:N { c }
```

Clear a `\box` register.

```
26712 \cs_new_protected:Npn \box_clear:N #1
\box_clear:N 26713 { \box_set_eq:NN #1 \c_empty_box }
26714 \cs_new_protected:Npn \box_gclear:N #1
\box_gclear:N 26715 { \box_gset_eq:NN #1 \c_empty_box }
26716 \cs_generate_variant:Nn \box_clear:N { c }
26717 \cs_generate_variant:Nn \box_gclear:N { c }
```

Clear or new.

```
26718 \cs_new_protected:Npn \box_clear_new:N #1
\box_clear_new:N 26719 { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
26720 \cs_new_protected:Npn \box_gclear_new:N #1
\box_gclear_new:N 26721 { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
26722 \cs_generate_variant:Nn \box_clear_new:N { c }
26723 \cs_generate_variant:Nn \box_gclear_new:N { c }
```

Assigning the contents of a box to be another box.

```
26724 \cs_new_protected:Npn \box_set_eq:NN #1#2
\box_set_eq:NN 26725 { \tex_setbox:D #1 \tex_copy:D #2 }
26726 \cs_new_protected:Npn \box_gset_eq:NN #1#2
\box_set_eq:Nc 26727 { \tex_global:D \tex_setbox:D #1 \tex_copy:D #2 }
26728 \cs_generate_variant:Nn \box_set_eq:NN { c , Nc , cc }
\box_set_eq:cc 26729 \cs_generate_variant:Nn \box_gset_eq:NN { c , Nc , cc }
\box_gset_eq:NN
\box_gset_eq:Nc
\box_gset_eq:cc
```

Assigning the contents of a box to be another box, then drops the original box.

```

\box_set_eq_drop:NN 26730 \cs_new_protected:Npn \box_set_eq_drop:NN #1#2
{ \tex_setbox:D #1 \tex_box:D #2 }
\box_set_eq_drop:cN 26731 { \tex_setbox:D #1 \tex_box:D #2 }
\box_set_eq_drop:Nc 26732 \cs_new_protected:Npn \box_gset_eq_drop:NN #1#2
{ \tex_global:D \tex_setbox:D #1 \tex_box:D #2 }
\box_set_eq_drop:cc 26733 \cs_generate_variant:Nn \box_set_eq_drop:NN { c , Nc , cc }
\box_gset_eq_drop:NN 26734 \cs_generate_variant:Nn \box_gset_eq_drop:NN { c , Nc , cc }
\box_gset_eq_drop:cN 26735
\box_gset_eq_drop:Nc
\box_gset_eq_drop:cc
\box_if_exist:Np 26736 \prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N
{ TF , T , F , p }
\box_if_exist:p:c 26737
\box_if_exist:c 26738 \prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c
{ TF , T , F , p }
\box_if_exist:NTF 26739
\box_if_exist:cTF

```

Copies of the cs functions defined in l3basics.

42.3 Measuring and setting box dimensions

Accessing the height, depth, and width of a $\langle box \rangle$ register.

```

\box_ht:N 26740 \cs_new_eq:NN \box_ht:N \tex_ht:D
\box_dp:N 26741 \cs_new_eq:NN \box_dp:N \tex_dp:D
\box_wd:N 26742 \cs_new_eq:NN \box_wd:N \tex_wd:D
\box_ht:c 26743 \cs_generate_variant:Nn \box_ht:N { c }
\box_dp:c 26744 \cs_generate_variant:Nn \box_dp:N { c }
\box_wd:c 26745 \cs_generate_variant:Nn \box_wd:N { c }

\box_set_ht:Nn 26746 \cs_new_protected:Npn \box_set_dp:Nn #1#2
\box_set_ht:cn {
\box_gset_ht:Nn 26747 {
\box_gset_ht:cn 26748 \tex_setbox:D #1 = \tex_copy:D #1
\box_set_dp:Nn 26749 \box_dp:N #1 \__box_dim_eval:n {#2}
\box_set_dp:cn 26750 }
\box_gset_dp:Nn 26751 \cs_generate_variant:Nn \box_set_dp:Nn { c }
\box_gset_dp:cn 26752 \cs_new_protected:Npn \box_gset_dp:Nn #1#2
\box_set_wd:Nn 26753 { \box_dp:N #1 \__box_dim_eval:n {#2} }
\box_set_wd:cn 26754 \cs_generate_variant:Nn \box_gset_dp:Nn { c }
\box_gset_wd:Nn 26755 \cs_new_protected:Npn \box_set_ht:Nn #1#2
\box_gset_wd:cn 26756 {
\box_ht:N #1 \__box_dim_eval:n {#2}
\box_set_dp:Nn #1 \__box_dim_eval:n {#2}
\box_gset_dp:Nn #1 \__box_dim_eval:n {#2}
\box_set_wd:Nn #1 \__box_dim_eval:n {#2}
\box_gset_wd:Nn #1 \__box_dim_eval:n {#2}
\box_gset_wd:cn 26759 }
\box_set_dp:Nn #1 \__box_dim_eval:n {#2}
\box_gset_dp:Nn #1 \__box_dim_eval:n {#2}
\box_set_wd:Nn #1 \__box_dim_eval:n {#2}
\box_gset_wd:Nn #1 \__box_dim_eval:n {#2}
\box_gset_wd:cn 26760 }
\box_set_dp:Nn #1 \__box_dim_eval:n {#2}
\box_gset_dp:Nn #1 \__box_dim_eval:n {#2}
\box_set_wd:Nn #1 \__box_dim_eval:n {#2}
\box_gset_wd:Nn #1 \__box_dim_eval:n {#2}
\box_gset_wd:cn 26763 }
\box_set_dp:Nn #1 \__box_dim_eval:n {#2}
\box_gset_dp:Nn #1 \__box_dim_eval:n {#2}
\box_set_wd:Nn #1 \__box_dim_eval:n {#2}
\box_gset_wd:Nn #1 \__box_dim_eval:n {#2}
\box_gset_wd:cn 26766 }
\box_set_dp:Nn #1 \__box_dim_eval:n {#2}
\box_gset_dp:Nn #1 \__box_dim_eval:n {#2}
\box_set_wd:Nn #1 \__box_dim_eval:n {#2}
\box_gset_wd:Nn #1 \__box_dim_eval:n {#2}
\box_gset_wd:cn 26769 }
\box_set_dp:Nn #1 \__box_dim_eval:n {#2}
\box_gset_dp:Nn #1 \__box_dim_eval:n {#2}
\box_set_wd:Nn #1 \__box_dim_eval:n {#2}
\box_gset_wd:Nn #1 \__box_dim_eval:n {#2}
\box_gset_wd:cn 26772 }

```

Setting the size whilst respecting local scope requires copying; the same issue does not come up when working globally. When debugging, the dimension expression #2 is surrounded by parentheses to catch early termination.

42.4 Using boxes

Using a $\langle box \rangle$. These are just \TeX primitives with meaningful names.

```

\box_use_drop:N 26773 \cs_new_eq:NN \box_use_drop:N \tex_box:D
\box_use_drop:c 26774 \cs_new_eq:NN \box_use:N \tex_copy:D
\box_use:N       26775 \cs_generate_variant:Nn \box_use_drop:N { c }
\box_use:c      26776 \cs_generate_variant:Nn \box_use:N { c }

```

Move box material in different directions. When debugging, the dimension expression #1 is surrounded by parentheses to catch early termination.

```

\box_move_left:nn 26777 \cs_new_protected:Npn \box_move_left:nn #1#2
\box_move_right:nn 26778 { \tex_moveleft:D \_box_dim_eval:n {#1} #2 }
\box_move_up:nn   26779 \cs_new_protected:Npn \box_move_right:nn #1#2
\box_move_down:nn 26780 { \tex_moveright:D \_box_dim_eval:n {#1} #2 }
                  26781 \cs_new_protected:Npn \box_move_up:nn #1#2
                  26782 { \tex_raise:D \_box_dim_eval:n {#1} #2 }
                  26783 \cs_new_protected:Npn \box_move_down:nn #1#2
                  26784 { \tex_lower:D \_box_dim_eval:n {#1} #2 }

```

42.5 Box conditionals

The primitives for testing if a $\langle box \rangle$ is empty/void or which type of box it is.

```

\if_hbox:N 26785 \cs_new_eq:NN \if_hbox:N \tex_ifhbox:D
\if_vbox:N 26786 \cs_new_eq:NN \if_vbox:N \tex_ifvbox:D
\if_box_empty:N 26787 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D

```

```

\box_if_horizontal_p:N 26788 \prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
\box_if_horizontal_p:c 26789 { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_horizontal:N $\underline{TF}$  26790 \prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
\box_if_horizontal:c $\underline{TF}$  26791 { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_vertical_p:N 26792 \prg_generate_conditional_variant:Nnn \box_if_horizontal:N
\box_if_vertical_p:c 26793 { c } { p , T , F , TF }
\box_if_vertical:N $\underline{TF}$  26794 \prg_generate_conditional_variant:Nnn \box_if_vertical:N
\box_if_vertical:c $\underline{TF}$  26795 { c } { p , T , F , TF }

```

Testing if a $\langle box \rangle$ is empty/void.

```

\box_if_empty_p:N 26796 \prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
\box_if_empty_p:c 26797 { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_empty:N $\underline{TF}$  26798 \prg_generate_conditional_variant:Nnn \box_if_empty:N
\box_if_empty:c $\underline{TF}$  26799 { c } { p , T , F , TF }

```

(End definition for $\backslash box_new:N$ and others. These functions are documented on page 240.)

42.6 The last box inserted

```

\box_set_to_last:N Set a box to the previous box.
\box_set_to_last:c 26800 \cs_new_protected:Npn \box_set_to_last:N #1
\box_gset_to_last:N 26801 { \tex_setbox:D #1 \tex_lastbox:D }
\box_gset_to_last:c 26802 \cs_new_protected:Npn \box_gset_to_last:N #1
                  26803 { \tex_global:D \tex_setbox:D #1 \tex_lastbox:D }
                  26804 \cs_generate_variant:Nn \box_set_to_last:N { c }
                  26805 \cs_generate_variant:Nn \box_gset_to_last:N { c }

```


(End definition for `\box_set_to_last:N` and `\box_gset_to_last:N`. These functions are documented on page 242.)

42.7 Constant boxes

`\c_empty_box` A box we never use.

```
26806 \box_new:N \c_empty_box
```

(End definition for `\c_empty_box`. This variable is documented on page 242.)

42.8 Scratch boxes

`\l_tmpa_box` Scratch boxes.

```
\l_tmpb_box 26807 \box_new:N \l_tmpa_box
```

```
\g_tmpa_box 26808 \box_new:N \l_tmpb_box
```

```
\g_tmpb_box 26809 \box_new:N \g_tmpa_box
```

```
26810 \box_new:N \g_tmpb_box
```

(End definition for `\l_tmpa_box` and others. These variables are documented on page 243.)

42.9 Viewing box contents

TeX's `\showbox` is not really that helpful in many cases, and it is also inconsistent with other L^AT_EX3 show functions as it does not actually shows material in the terminal. So we provide a richer set of functionality.

`\box_show:N` Essentially a wrapper around the internal function, but evaluating the breadth and depth arguments now outside the group.

`\box_show:c`

`\box_show:Nnn`

`\box_show:cnn`

```
26811 \cs_new_protected:Npn \box_show:N #1
26812 { \box_show:Nnn #1 \c_max_int \c_max_int }
26813 \cs_generate_variant:Nn \box_show:N { c }
26814 \cs_new_protected:Npn \box_show:Nnn #1#2#3
26815 { \__box_show:NNff 1 #1 { \int_eval:n {#2} } { \int_eval:n {#3} } }
26816 \cs_generate_variant:Nn \box_show:Nnn { c }
```

(End definition for `\box_show:N` and `\box_show:Nnn`. These functions are documented on page 243.)

`\box_log:N` Getting TeX to write to the log without interruption the run is done by altering the interaction mode. For that, the ϵ -TeX extensions are needed.

`\box_log:c`

`\box_log:Nnn`

`\box_log:cnn`

`__box_log:nNnn`

```
26817 \cs_new_protected:Npn \box_log:N #1
26818 { \box_log:Nnn #1 \c_max_int \c_max_int }
26819 \cs_generate_variant:Nn \box_log:N { c }
26820 \cs_new_protected:Npn \box_log:Nnn
26821 { \exp_args:No \__box_log:nNnn { \tex_the:D \tex_interactionmode:D } }
26822 \cs_new_protected:Npn \__box_log:nNnn #1#2#3#4
26823 {
26824   \int_set:Nn \tex_interactionmode:D { 0 }
26825   \__box_show:NNff 0 #2 { \int_eval:n {#3} } { \int_eval:n {#4} }
26826   \int_set:Nn \tex_interactionmode:D {#1}
26827 }
26828 \cs_generate_variant:Nn \box_log:Nnn { c }
```

(End definition for `\box_log:N`, `\box_log:Nnn`, and `__box_log:nNnn`. These functions are documented on page 243.)

`__box_show:NNnn` The internal auxiliary to actually do the output uses a group to deal with breadth and
`__box_show:NNff` depth values. The `\use:n` here gives better output appearance. Setting `\tracingonline`
and `\errorcontextlines` is used to control what appears in the terminal.

```

26829 \cs_new_protected:Npn \__box_show:NNnn #1#2#3#4
26830 {
26831   \box_if_exist:NTF #2
26832   {
26833     \group_begin:
26834       \int_set:Nn \tex_showboxbreadth:D {#3}
26835       \int_set:Nn \tex_showboxdepth:D {#4}
26836       \int_set:Nn \tex_tracingonline:D {#1}
26837       \int_set:Nn \tex_errorcontextlines:D { -1 }
26838       \tex_showbox:D \use:n {#2}
26839     \group_end:
26840   }
26841   {
26842     \__kernel_msg_error:nxx { kernel } { variable-not-defined }
26843     { \token_to_str:N #2 }
26844   }
26845 }
26846 \cs_generate_variant:Nn \__box_show:NNnn { NNff }

```

(End definition for `__box_show:NNnn`.)

42.10 Horizontal mode boxes

`\hbox:n` (The test suite for this command, and others in this file, is `m3box002.lvt`.)
Put a horizontal box directly into the input stream.

```

26847 \cs_new_protected:Npn \hbox:n #1
26848 { \tex_hbox:D \scan_stop: { \color_group_begin: #1 \color_group_end: } }

```

(End definition for `\hbox:n`. This function is documented on page 243.)

```

\hbox_set:Nn
\hbox_set:cn
\hbox_gset:Nn
\hbox_gset:cn
26849 \cs_new_protected:Npn \hbox_set:Nn #1#2
26850 {
26851   \tex_setbox:D #1 \tex_hbox:D
26852   { \color_group_begin: #2 \color_group_end: }
26853 }
26854 \cs_new_protected:Npn \hbox_gset:Nn #1#2
26855 {
26856   \tex_global:D \tex_setbox:D #1 \tex_hbox:D
26857   { \color_group_begin: #2 \color_group_end: }
26858 }
26859 \cs_generate_variant:Nn \hbox_set:Nn { c }
26860 \cs_generate_variant:Nn \hbox_gset:Nn { c }

```

(End definition for `\hbox_set:Nn` and `\hbox_gset:Nn`. These functions are documented on page 244.)

`\hbox_set_to_wd:Nnn` Storing material in a horizontal box with a specified width. Again, put the dimension
`\hbox_set_to_wd:cn` expression in parentheses when debugging.

```

\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cn
26861 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
26862 {
26863   \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}

```

```

26864         { \color_group_begin: #3 \color_group_end: }
26865     }
26866 \cs_new_protected:Npn \hbox_gset_to_wd:Nnn #1#2#3
26867 {
26868     \tex_global:D \tex_setbox:D #1 \tex_hbox:D to \_box_dim_eval:n {#2}
26869     { \color_group_begin: #3 \color_group_end: }
26870 }
26871 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
26872 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnn { c }

```

(End definition for `\hbox_set_to_wd:Nnn` and `\hbox_gset_to_wd:Nnn`. These functions are documented on page 244.)

`\hbox_set:Nw` Storing material in a horizontal box. This type is useful in environment definitions.

```

\hbox_set:cw 26873 \cs_new_protected:Npn \hbox_set:Nw #1
\hbox_gset:Nw 26874 {
\hbox_gset:cw 26875     \tex_setbox:D #1 \tex_hbox:D
\hbox_set_end: 26876     \c_group_begin_token
\hbox_gset_end: 26877     \color_group_begin:
26878 }
26879 \cs_new_protected:Npn \hbox_gset:Nw #1
26880 {
26881     \tex_global:D \tex_setbox:D #1 \tex_hbox:D
26882     \c_group_begin_token
26883     \color_group_begin:
26884 }
26885 \cs_generate_variant:Nn \hbox_set:Nw { c }
26886 \cs_generate_variant:Nn \hbox_gset:Nw { c }
26887 \cs_new_protected:Npn \hbox_set_end:
26888 {
26889     \color_group_end:
26890     \c_group_end_token
26891 }
26892 \cs_new_eq:NN \hbox_gset_end: \hbox_set_end:

```

(End definition for `\hbox_set:Nw` and others. These functions are documented on page 244.)

`\hbox_set_to_wd:Nnw` Combining the above ideas.

```

\hbox_set_to_wd:cw 26893 \cs_new_protected:Npn \hbox_set_to_wd:Nnw #1#2
\hbox_gset_to_wd:Nnw 26894 {
\hbox_gset_to_wd:cw 26895     \tex_setbox:D #1 \tex_hbox:D to \_box_dim_eval:n {#2}
26896     \c_group_begin_token
26897     \color_group_begin:
26898 }
26899 \cs_new_protected:Npn \hbox_gset_to_wd:Nnw #1#2
26900 {
26901     \tex_global:D \tex_setbox:D #1 \tex_hbox:D to \_box_dim_eval:n {#2}
26902     \c_group_begin_token
26903     \color_group_begin:
26904 }
26905 \cs_generate_variant:Nn \hbox_set_to_wd:Nnw { c }
26906 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnw { c }

```

(End definition for `\hbox_set_to_wd:Nnw` and `\hbox_gset_to_wd:Nnw`. These functions are documented on page 244.)

\hbox_to_wd:nn Put a horizontal box directly into the input stream.

```

\hbox_to_zero:n 26907 \cs_new_protected:Npn \hbox_to_wd:nn #1#2
                26908 {
                26909     \tex_hbox:D to \__box_dim_eval:n {#1}
                26910     { \color_group_begin: #2 \color_group_end: }
                26911 }
                26912 \cs_new_protected:Npn \hbox_to_zero:n #1
                26913 {
                26914     \tex_hbox:D to \c_zero_dim
                26915     { \color_group_begin: #1 \color_group_end: }
                26916 }

```

(End definition for \hbox_to_wd:nn and \hbox_to_zero:n. These functions are documented on page 244.)

\hbox_overlap_left:n Put a zero-sized box with the contents pushed against one side (which makes it stick out
\hbox_overlap_right:n on the other) directly into the input stream.

```

26917 \cs_new_protected:Npn \hbox_overlap_left:n #1
26918 { \hbox_to_zero:n { \tex_hss:D #1 } }
26919 \cs_new_protected:Npn \hbox_overlap_right:n #1
26920 { \hbox_to_zero:n { #1 \tex_hss:D } }

```

(End definition for \hbox_overlap_left:n and \hbox_overlap_right:n. These functions are documented on page 244.)

\hbox_unpack:N Unpacking a box and if requested also clear it.

```

\hbox_unpack:c 26921 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
\hbox_unpack_drop:N 26922 \cs_new_eq:NN \hbox_unpack_drop:N \tex_unhbox:D
\hbox_unpack_drop:c 26923 \cs_generate_variant:Nn \hbox_unpack:N { c }
26924 \cs_generate_variant:Nn \hbox_unpack_drop:N { c }

```

(End definition for \hbox_unpack:N and \hbox_unpack_drop:N. These functions are documented on page 244.)

42.11 Vertical mode boxes

TeX ends these boxes directly with the internal *end_graf* routine. This means that there is no \par at the end of vertical boxes unless we insert one. Thus all vertical boxes include a \par just before closing the color group.

\vbox:n The following test files are used for this code: m3box003.lvt.

The following test files are used for this code: m3box003.lvt.

\vbox_top:n Put a vertical box directly into the input stream.

```

26925 \cs_new_protected:Npn \vbox:n #1
26926 { \tex_vbox:D { \color_group_begin: #1 \par \color_group_end: } }
26927 \cs_new_protected:Npn \vbox_top:n #1
26928 { \tex_vtop:D { \color_group_begin: #1 \par \color_group_end: } }

```

(End definition for \vbox:n and \vbox_top:n. These functions are documented on page 245.)

`\vbox_to_ht:nn` Put a vertical box directly into the input stream.

```

\vbox_to_zero:n 26929 \cs_new_protected:Npn \vbox_to_ht:nn #1#2
\vbox_to_ht:nn 26930 {
\vbox_to_zero:n 26931 \tex_vbox:D to \__box_dim_eval:n {#1}
26932 { \color_group_begin: #2 \par \color_group_end: }
26933 }
26934 \cs_new_protected:Npn \vbox_to_zero:n #1
26935 {
26936 \tex_vbox:D to \c_zero_dim
26937 { \color_group_begin: #1 \par \color_group_end: }
26938 }

```

(End definition for `\vbox_to_ht:nn` and others. These functions are documented on page 245.)

`\vbox_set:Nn` Storing material in a vertical box with a natural height.

```

\vbox_set:cn 26939 \cs_new_protected:Npn \vbox_set:Nn #1#2
\vbox_gset:Nn 26940 {
\vbox_gset:cn 26941 \tex_setbox:D #1 \tex_vbox:D
26942 { \color_group_begin: #2 \par \color_group_end: }
26943 }
26944 \cs_new_protected:Npn \vbox_gset:Nn #1#2
26945 {
26946 \tex_global:D \tex_setbox:D #1 \tex_vbox:D
26947 { \color_group_begin: #2 \par \color_group_end: }
26948 }
26949 \cs_generate_variant:Nn \vbox_set:Nn { c }
26950 \cs_generate_variant:Nn \vbox_gset:Nn { c }

```

(End definition for `\vbox_set:Nn` and `\vbox_gset:Nn`. These functions are documented on page 245.)

`\vbox_set_top:Nn` Storing material in a vertical box with a natural height and reference point at the baseline of the first object in the box.

```

\vbox_set_top:cn 26951 \cs_new_protected:Npn \vbox_set_top:Nn #1#2
\vbox_gset_top:Nn 26952 {
\vbox_gset_top:cn 26953 \tex_setbox:D #1 \tex_vtop:D
26954 { \color_group_begin: #2 \par \color_group_end: }
26955 }
26956 \cs_new_protected:Npn \vbox_gset_top:Nn #1#2
26957 {
26958 \tex_global:D \tex_setbox:D #1 \tex_vtop:D
26959 { \color_group_begin: #2 \par \color_group_end: }
26960 }
26961 \cs_generate_variant:Nn \vbox_set_top:Nn { c }
26962 \cs_generate_variant:Nn \vbox_gset_top:Nn { c }

```

(End definition for `\vbox_set_top:Nn` and `\vbox_gset_top:Nn`. These functions are documented on page 245.)

`\vbox_set_to_ht:Nnn` Storing material in a vertical box with a specified height.

```

\vbox_set_to_ht:cnn 26963 \cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3
\vbox_gset_to_ht:Nnn 26964 {
\vbox_gset_to_ht:cnn 26965 \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}
26966 { \color_group_begin: #3 \par \color_group_end: }
26967 }

```

```

26968 \cs_new_protected:Npn \vbox_gset_to_ht:Nnn #1#2#3
26969 {
26970   \tex_global:D \tex_setbox:D #1 \tex_vbox:D to \_box_dim_eval:n {#2}
26971   { \color_group_begin: #3 \par \color_group_end: }
26972 }
26973 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn { c }
26974 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnn { c }

```

(End definition for `\vbox_set_to_ht:Nnn` and `\vbox_gset_to_ht:Nnn`. These functions are documented on page 245.)

`\vbox_set:Nw` Storing material in a vertical box. This type is useful in environment definitions.
`\vbox_set:cw`
`\vbox_gset:Nw`
`\vbox_gset:cw`
`\vbox_set_end:`
`\vbox_gset_end:`

```

26975 \cs_new_protected:Npn \vbox_set:Nw #1
26976 {
26977   \tex_setbox:D #1 \tex_vbox:D
26978   \c_group_begin_token
26979   \color_group_begin:
26980 }
26981 \cs_new_protected:Npn \vbox_gset:Nw #1
26982 {
26983   \tex_global:D \tex_setbox:D #1 \tex_vbox:D
26984   \c_group_begin_token
26985   \color_group_begin:
26986 }
26987 \cs_generate_variant:Nn \vbox_set:Nw { c }
26988 \cs_generate_variant:Nn \vbox_gset:Nw { c }
26989 \cs_new_protected:Npn \vbox_set_end:
26990 {
26991   \par
26992   \color_group_end:
26993   \c_group_end_token
26994 }
26995 \cs_new_eq:NN \vbox_gset_end: \vbox_set_end:

```

(End definition for `\vbox_set:Nw` and others. These functions are documented on page 246.)

`\vbox_set_to_ht:Nnw` A combination of the above ideas.

```

\vbox_set_to_ht:cnw 26996 \cs_new_protected:Npn \vbox_set_to_ht:Nnw #1#2
\vbox_gset_to_ht:Nnw 26997 {
\vbox_gset_to_ht:cnw 26998   \tex_setbox:D #1 \tex_vbox:D to \_box_dim_eval:n {#2}
26999   \c_group_begin_token
27000   \color_group_begin:
27001 }
27002 \cs_new_protected:Npn \vbox_gset_to_ht:Nnw #1#2
27003 {
27004   \tex_global:D \tex_setbox:D #1 \tex_vbox:D to \_box_dim_eval:n {#2}
27005   \c_group_begin_token
27006   \color_group_begin:
27007 }
27008 \cs_generate_variant:Nn \vbox_set_to_ht:Nnw { c }
27009 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnw { c }

```

(End definition for `\vbox_set_to_ht:Nnw` and `\vbox_gset_to_ht:Nnw`. These functions are documented on page 246.)

\vbox_unpack:N Unpacking a box and if requested also clear it.

\vbox_unpack:c 27010 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D

\vbox_unpack_drop:N 27011 \cs_new_eq:NN \vbox_unpack_drop:N \tex_unvbox:D

\vbox_unpack_drop:c 27012 \cs_generate_variant:Nn \vbox_unpack:N { c }

27013 \cs_generate_variant:Nn \vbox_unpack_drop:N { c }

(End definition for \vbox_unpack:N and \vbox_unpack_drop:N. These functions are documented on page 246.)

\vbox_set_split_to_ht:NNn Splitting a vertical box in two.

\vbox_set_split_to_ht:cNn 27014 \cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3

\vbox_set_split_to_ht:Ncn 27015 { \tex_setbox:D #1 \tex_vsplit:D #2 to _box_dim_eval:n {#3} }

\vbox_set_split_to_ht:ccn 27016 \cs_generate_variant:Nn \vbox_set_split_to_ht:NNn { c , Nc , cc }

\vbox_gset_split_to_ht:NNn 27017 \cs_new_protected:Npn \vbox_gset_split_to_ht:NNn #1#2#3

\vbox_gset_split_to_ht:cNn 27018 {

\vbox_gset_split_to_ht:Ncn 27019 \tex_global:D \tex_setbox:D #1

\vbox_gset_split_to_ht:ccn 27020 \tex_vsplit:D #2 to _box_dim_eval:n {#3}

27021 }

27022 \cs_generate_variant:Nn \vbox_gset_split_to_ht:NNn { c , Nc , cc }

(End definition for \vbox_set_split_to_ht:NNn and \vbox_gset_split_to_ht:NNn. These functions are documented on page 246.)

42.12 Affine transformations

\l__box_angle_fp When rotating boxes, the angle itself may be needed by the engine-dependent code. This is done using the fp module so that the value is tidied up properly.

27023 \fp_new:N \l__box_angle_fp

(End definition for \l__box_angle_fp.)

\l__box_cos_fp These are used to hold the calculated sine and cosine values while carrying out a rotation.

\l__box_sin_fp 27024 \fp_new:N \l__box_cos_fp

27025 \fp_new:N \l__box_sin_fp

(End definition for \l__box_cos_fp and \l__box_sin_fp.)

\l__box_top_dim These are the positions of the four edges of a box before manipulation.

\l__box_bottom_dim 27026 \dim_new:N \l__box_top_dim

\l__box_left_dim 27027 \dim_new:N \l__box_bottom_dim

\l__box_right_dim 27028 \dim_new:N \l__box_left_dim

27029 \dim_new:N \l__box_right_dim

(End definition for \l__box_top_dim and others.)

\l__box_top_new_dim These are the positions of the four edges of a box after manipulation.

\l__box_bottom_new_dim 27030 \dim_new:N \l__box_top_new_dim

\l__box_left_new_dim 27031 \dim_new:N \l__box_bottom_new_dim

\l__box_right_new_dim 27032 \dim_new:N \l__box_left_new_dim

27033 \dim_new:N \l__box_right_new_dim

(End definition for \l__box_top_new_dim and others.)

\l__box_internal_box Scratch space, but also needed by some parts of the driver.

27034 \box_new:N \l__box_internal_box

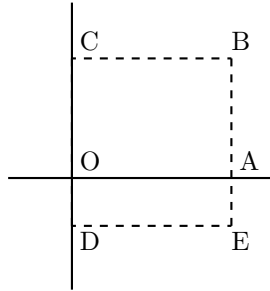


Figure 1: Co-ordinates of a box prior to rotation.

(End definition for `\l_box_internal_box`.)

`\box_rotate:Nn` Rotation of a box starts with working out the relevant sine and cosine. The actual rotation is in an auxiliary to keep the flow slightly clearer

`\box_grotate:Nn`

`\box_grotate:cn`

`__box_rotate:NnN`

`__box_rotate:N`

`__box_rotate_xdir:nnN`

`__box_rotate_ydir:nnN`

`__box_rotate_quadrant_one:`

`__box_rotate_quadrant_two:`

`__box_rotate_quadrant_three:`

`__box_rotate_quadrant_four:`

```

27035 \cs_new_protected:Npn \box_rotate:Nn #1#2
27036 { \__box_rotate:NnN #1 {#2} \hbox_set:Nn }
27037 \cs_generate_variant:Nn \box_rotate:Nn { c }
27038 \cs_new_protected:Npn \box_grotate:Nn #1#2
27039 { \__box_rotate:NnN #1 {#2} \hbox_gset:Nn }
27040 \cs_generate_variant:Nn \box_grotate:Nn { c }
27041 \cs_new_protected:Npn \__box_rotate:NnN #1#2#3
27042 {
27043   #3 #1
27044   {
27045     \fp_set:Nn \l_box_angle_fp {#2}
27046     \fp_set:Nn \l_box_sin_fp { sind ( \l_box_angle_fp ) }
27047     \fp_set:Nn \l_box_cos_fp { cosd ( \l_box_angle_fp ) }
27048     \__box_rotate:N #1
27049   }
27050 }

```

The edges of the box are then recorded: the left edge is always at zero. Rotation of the four edges then takes place: this is most efficiently done on a quadrant by quadrant basis.

```

27051 \cs_new_protected:Npn \__box_rotate:N #1
27052 {
27053   \dim_set:Nn \l_box_top_dim { \box_ht:N #1 }
27054   \dim_set:Nn \l_box_bottom_dim { -\box_dp:N #1 }
27055   \dim_set:Nn \l_box_right_dim { \box_wd:N #1 }
27056   \dim_zero:N \l_box_left_dim

```

The next step is to work out the x and y coordinates of vertices of the rotated box in relation to its original coordinates. The box can be visualized with vertices B , C , D and E is illustrated (Figure 1). The vertex O is the reference point on the baseline, and in this implementation is also the centre of rotation. The formulae are, for a point P and angle α :

$$\begin{aligned}
 P'_x &= P_x - O_x \\
 P'_y &= P_y - O_y \\
 P''_x &= (P'_x \cos(\alpha)) - (P'_y \sin(\alpha)) \\
 P''_y &= (P'_x \sin(\alpha)) + (P'_y \cos(\alpha)) \\
 P'''_x &= P''_x + O_x + L_x \\
 P'''_y &= P''_y + O_y
 \end{aligned}$$

The “extra” horizontal translation L_x at the end is calculated so that the leftmost point of the resulting box has x -coordinate 0. This is desirable as \TeX boxes must have the reference point at the left edge of the box. (As O is always $(0,0)$, this part of the calculation is omitted here.)

```

27057 \fp_compare:nNnTF \l__box_sin_fp > \c_zero_fp
27058 {
27059     \fp_compare:nNnTF \l__box_cos_fp > \c_zero_fp
27060     { \__box_rotate_quadrant_one: }
27061     { \__box_rotate_quadrant_two: }
27062 }
27063 {
27064     \fp_compare:nNnTF \l__box_cos_fp < \c_zero_fp
27065     { \__box_rotate_quadrant_three: }
27066     { \__box_rotate_quadrant_four: }
27067 }

```

The position of the box edges are now known, but the box at this stage be misplaced relative to the current \TeX reference point. So the content of the box is moved such that the reference point of the rotated box is in the same place as the original.

```

27068 \hbox_set:Nn \l__box_internal_box { \box_use:N #1 }
27069 \hbox_set:Nn \l__box_internal_box
27070 {
27071     \tex_kern:D -\l__box_left_new_dim
27072     \hbox:n
27073     {
27074         \__box_backend_rotate:Nn
27075         \l__box_internal_box
27076         \l__box_angle_fp
27077     }
27078 }

```

Tidy up the size of the box so that the material is actually inside the bounding box. The result can then be used to reset the original box.

```

27079 \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
27080 \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
27081 \box_set_wd:Nn \l__box_internal_box
27082 { \l__box_right_new_dim - \l__box_left_new_dim }
27083 \box_use_drop:N \l__box_internal_box
27084 }

```

These functions take a general point $(\#1,\#2)$ and rotate its location about the origin, using the previously-set sine and cosine values. Each function gives only one component of the location of the updated point. This is because for rotation of a box each step needs only one value, and so performance is gained by avoiding working out both x' and y' at the same time. Contrast this with the equivalent function in the `l3coffins` module, where both parts are needed.

```

27085 \cs_new_protected:Npn \__box_rotate_xdir:nnN #1#2#3
27086 {
27087     \dim_set:Nn #3
27088     {
27089         \fp_to_dim:n
27090         {
27091             \l__box_cos_fp * \dim_to_fp:n {#1}
27092             - \l__box_sin_fp * \dim_to_fp:n {#2}

```

```

27093     }
27094   }
27095 }
27096 \cs_new_protected:Npn \__box_rotate_ydir:nnN #1#2#3
27097 {
27098   \dim_set:Nn #3
27099   {
27100     \fp_to_dim:n
27101     {
27102       \l__box_sin_fp * \dim_to_fp:n {#1}
27103       + \l__box_cos_fp * \dim_to_fp:n {#2}
27104     }
27105   }
27106 }

```

Rotation of the edges is done using a different formula for each quadrant. In every case, the top and bottom edges only need the resulting y -values, whereas the left and right edges need the x -values. Each case is a question of picking out which corner ends up at with the maximum top, bottom, left and right value. Doing this by hand means a lot less calculating and avoids lots of comparisons.

```

27107 \cs_new_protected:Npn \__box_rotate_quadrant_one:
27108 {
27109   \__box_rotate_ydir:nnN \l__box_right_dim \l__box_top_dim
27110   \l__box_top_new_dim
27111   \__box_rotate_ydir:nnN \l__box_left_dim \l__box_bottom_dim
27112   \l__box_bottom_new_dim
27113   \__box_rotate_xdir:nnN \l__box_left_dim \l__box_top_dim
27114   \l__box_left_new_dim
27115   \__box_rotate_xdir:nnN \l__box_right_dim \l__box_bottom_dim
27116   \l__box_right_new_dim
27117 }
27118 \cs_new_protected:Npn \__box_rotate_quadrant_two:
27119 {
27120   \__box_rotate_ydir:nnN \l__box_right_dim \l__box_bottom_dim
27121   \l__box_top_new_dim
27122   \__box_rotate_ydir:nnN \l__box_left_dim \l__box_top_dim
27123   \l__box_bottom_new_dim
27124   \__box_rotate_xdir:nnN \l__box_right_dim \l__box_top_dim
27125   \l__box_left_new_dim
27126   \__box_rotate_xdir:nnN \l__box_left_dim \l__box_bottom_dim
27127   \l__box_right_new_dim
27128 }
27129 \cs_new_protected:Npn \__box_rotate_quadrant_three:
27130 {
27131   \__box_rotate_ydir:nnN \l__box_left_dim \l__box_bottom_dim
27132   \l__box_top_new_dim
27133   \__box_rotate_ydir:nnN \l__box_right_dim \l__box_top_dim
27134   \l__box_bottom_new_dim
27135   \__box_rotate_xdir:nnN \l__box_right_dim \l__box_bottom_dim
27136   \l__box_left_new_dim
27137   \__box_rotate_xdir:nnN \l__box_left_dim \l__box_top_dim
27138   \l__box_right_new_dim
27139 }
27140 \cs_new_protected:Npn \__box_rotate_quadrant_four:

```

```

27141 {
27142   \__box_rotate_ydir:nnN \l__box_left_dim \l__box_top_dim
27143   \l__box_top_new_dim
27144   \__box_rotate_ydir:nnN \l__box_right_dim \l__box_bottom_dim
27145   \l__box_bottom_new_dim
27146   \__box_rotate_xdir:nnN \l__box_left_dim \l__box_bottom_dim
27147   \l__box_left_new_dim
27148   \__box_rotate_xdir:nnN \l__box_right_dim \l__box_top_dim
27149   \l__box_right_new_dim
27150 }

```

(End definition for `\box_rotate:Nn` and others. These functions are documented on page 250.)

`\l__box_scale_x_fp` Scaling is potentially-different in the two axes.
`\l__box_scale_y_fp`

```

27151 \fp_new:N \l__box_scale_x_fp
27152 \fp_new:N \l__box_scale_y_fp

```

(End definition for `\l__box_scale_x_fp` and `\l__box_scale_y_fp`.)

`\box_resize_to_wd_and_ht_plus_dp:Nnn` Resizing a box starts by working out the various dimensions of the existing box.

```

\box_resize_to_wd_and_ht_plus_dp:cmn
\box_gresize_to_wd_and_ht_plus_dp:Nnn
\box_gresize_to_wd_and_ht_plus_dp:cmn
\__box_resize_to_wd_and_ht_plus_dp:NnnN
\__box_resize_set_corners:N
  \__box_resize:N
  \__box_resize:NNN
27153 \cs_new_protected:Npn \box_resize_to_wd_and_ht_plus_dp:Nnn #1#2#3
27154 {
27155   \__box_resize_to_wd_and_ht_plus_dp:NnnN #1 {#2} {#3}
27156   \hbox_set:Nn
27157 }
27158 \cs_generate_variant:Nn \box_resize_to_wd_and_ht_plus_dp:Nnn { c }
27159 \cs_new_protected:Npn \box_gresize_to_wd_and_ht_plus_dp:Nnn #1#2#3
27160 {
27161   \__box_resize_to_wd_and_ht_plus_dp:NnnN #1 {#2} {#3}
27162   \hbox_gset:Nn
27163 }
27164 \cs_generate_variant:Nn \box_gresize_to_wd_and_ht_plus_dp:Nnn { c }
27165 \cs_new_protected:Npn \__box_resize_to_wd_and_ht_plus_dp:NnnN #1#2#3#4
27166 {
27167   #4 #1
27168   {
27169     \__box_resize_set_corners:N #1

```

The x -scaling and resulting box size is easy enough to work out: the dimension is that given as #2, and the scale is simply the new width divided by the old one.

```

27170   \fp_set:Nn \l__box_scale_x_fp
27171   { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }

```

The y -scaling needs both the height and the depth of the current box.

```

27172   \fp_set:Nn \l__box_scale_y_fp
27173   {
27174     \dim_to_fp:n {#3}
27175     / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
27176   }

```

Hand off to the auxiliary which does the rest of the work.

```

27177   \__box_resize:N #1
27178 }
27179 }
27180 \cs_new_protected:Npn \__box_resize_set_corners:N #1
27181 {

```

```

27182 \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
27183 \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
27184 \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
27185 \dim_zero:N \l__box_left_dim
27186 }

```

With at least one real scaling to do, the next phase is to find the new edge co-ordinates. In the x direction this is relatively easy: just scale the right edge. In the y direction, both dimensions have to be scaled, and this again needs the absolute scale value. Once that is all done, the common resize/rescale code can be employed.

```

27187 \cs_new_protected:Npn \__box_resize:N #1
27188 {
27189   \__box_resize:NNN \l__box_right_new_dim
27190   \l__box_scale_x_fp \l__box_right_dim
27191   \__box_resize:NNN \l__box_bottom_new_dim
27192   \l__box_scale_y_fp \l__box_bottom_dim
27193   \__box_resize:NNN \l__box_top_new_dim
27194   \l__box_scale_y_fp \l__box_top_dim
27195   \__box_resize_common:N #1
27196 }
27197 \cs_new_protected:Npn \__box_resize:NNN #1#2#3
27198 {
27199   \dim_set:Nn #1
27200   { \fp_to_dim:n { \fp_abs:n { #2 } * \dim_to_fp:n { #3 } } }
27201 }

```

(End definition for `\box_resize_to_wd_and_ht_plus_dp:Nnn` and others. These functions are documented on page 249.)

```

\box_resize_to_ht:Nn Scaling to a (total) height or to a width is a simplified version of the main resizing
\box_resize_to_ht:cn operation, with the scale simply copied between the two parts. The internal auxiliary is
\box_gresize_to_ht:Nn called using the scaling value twice, as the sign for both parts is needed (as this allows
\box_gresize_to_ht:cn the same internal code to be used as for the general case).
\__box_resize_to_ht:NnN
\box_resize_to_ht_plus_dp:Nn
\box_resize_to_ht_plus_dp:cn
\box_gresize_to_ht_plus_dp:Nn
\box_gresize_to_ht_plus_dp:cn
\__box_resize_to_ht_plus_dp:NnN
\box_resize_to_wd:Nn
\box_resize_to_wd:cn
\box_gresize_to_wd:Nn
\box_gresize_to_wd:cn
\__box_resize_to_wd:NnN
\box_resize_to_wd_and_ht:Nnn
\box_gresize_to_wd_and_ht:Nnn
\box_gresize_to_wd_and_ht:cn
\__box_resize_to_wd_ht:NnnN
27202 \cs_new_protected:Npn \box_resize_to_ht:Nn #1#2
27203 { \__box_resize_to_ht:NnN #1 {#2} \hbox_set:Nn }
27204 \cs_generate_variant:Nn \box_resize_to_ht:Nn { c }
27205 \cs_new_protected:Npn \box_gresize_to_ht:Nn #1#2
27206 { \__box_resize_to_ht:NnN #1 {#2} \hbox_gset:Nn }
27207 \cs_generate_variant:Nn \box_gresize_to_ht:Nn { c }
27208 \cs_new_protected:Npn \__box_resize_to_ht:NnN #1#2#3
27209 {
27210   #3 #1
27211   {
27212     \__box_resize_set_corners:N #1
27213     \fp_set:Nn \l__box_scale_y_fp
27214     {
27215       \dim_to_fp:n {#2}
27216       / \dim_to_fp:n { \l__box_top_dim }
27217     }
27218     \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
27219     \__box_resize:N #1
27220   }
27221 }
27222 \cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2

```

```

27223 { \_box_resize_to_ht_plus_dp:NnN #1 {#2} \hbox_set:Nn }
27224 \cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
27225 \cs_new_protected:Npn \box_gresize_to_ht_plus_dp:Nn #1#2
27226 { \_box_resize_to_ht_plus_dp:NnN #1 {#2} \hbox_gset:Nn }
27227 \cs_generate_variant:Nn \box_gresize_to_ht_plus_dp:Nn { c }
27228 \cs_new_protected:Npn \_box_resize_to_ht_plus_dp:NnN #1#2#3
27229 {
27230   \hbox_set:Nn #1
27231   {
27232     \_box_resize_set_corners:N #1
27233     \fp_set:Nn \l__box_scale_y_fp
27234     {
27235       \dim_to_fp:n {#2}
27236       / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
27237     }
27238     \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
27239     \_box_resize:N #1
27240   }
27241 }
27242 \cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
27243 { \_box_resize_to_wd:NnN #1 {#2} \hbox_set:Nn }
27244 \cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
27245 \cs_new_protected:Npn \box_gresize_to_wd:Nn #1#2
27246 { \_box_resize_to_wd:NnN #1 {#2} \hbox_gset:Nn }
27247 \cs_generate_variant:Nn \box_gresize_to_wd:Nn { c }
27248 \cs_new_protected:Npn \_box_resize_to_wd:NnN #1#2#3
27249 {
27250   #3 #1
27251   {
27252     \_box_resize_set_corners:N #1
27253     \fp_set:Nn \l__box_scale_x_fp
27254     { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
27255     \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
27256     \_box_resize:N #1
27257   }
27258 }
27259 \cs_new_protected:Npn \box_resize_to_wd_and_ht:Nnn #1#2#3
27260 { \_box_resize_to_wd_and_ht:NnnN #1 {#2} {#3} \hbox_set:Nn }
27261 \cs_generate_variant:Nn \box_resize_to_wd_and_ht:Nnn { c }
27262 \cs_new_protected:Npn \box_gresize_to_wd_and_ht:Nnn #1#2#3
27263 { \_box_resize_to_wd_and_ht:NnnN #1 {#2} {#3} \hbox_gset:Nn }
27264 \cs_generate_variant:Nn \box_gresize_to_wd_and_ht:Nnn { c }
27265 \cs_new_protected:Npn \_box_resize_to_wd_and_ht:NnnN #1#2#3#4
27266 {
27267   #4 #1
27268   {
27269     \_box_resize_set_corners:N #1
27270     \fp_set:Nn \l__box_scale_x_fp
27271     { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
27272     \fp_set:Nn \l__box_scale_y_fp
27273     {
27274       \dim_to_fp:n {#3}
27275       / \dim_to_fp:n { \l__box_top_dim }
27276     }

```

```

27277         \_box_resize:N #1
27278     }
27279 }

```

(End definition for `\box_resize_to_ht:Nn` and others. These functions are documented on page 248.)

`\box_scale:Nnn` When scaling a box, setting the scaling itself is easy enough. The new dimensions are also relatively easy to find, allowing only for the need to keep them positive in all cases. `\box_scale:cnn` Once that is done then after a check for the trivial scaling a hand-off can be made to the common code. The code here is split into two as this allows sharing with the auto-resizing functions. `\box_gscale:Nnn` `\box_gscale:cnn` `_box_scale:NnnN` `_box_scale:N`

```

27280 \cs_new_protected:Npn \box_scale:Nnn #1#2#3
27281 { \_box_scale:NnnN #1 {#2} {#3} \hbox_set:Nn }
27282 \cs_generate_variant:Nn \box_scale:Nnn { c }
27283 \cs_new_protected:Npn \box_gscale:Nnn #1#2#3
27284 { \_box_scale:NnnN #1 {#2} {#3} \hbox_gset:Nn }
27285 \cs_generate_variant:Nn \box_gscale:Nnn { c }
27286 \cs_new_protected:Npn \_box_scale:NnnN #1#2#3#4
27287 {
27288     #4 #1
27289     {
27290         \fp_set:Nn \l__box_scale_x_fp {#2}
27291         \fp_set:Nn \l__box_scale_y_fp {#3}
27292         \_box_scale:N #1
27293     }
27294 }
27295 \cs_new_protected:Npn \_box_scale:N #1
27296 {
27297     \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
27298     \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
27299     \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
27300     \dim_zero:N \l__box_left_dim
27301     \dim_set:Nn \l__box_top_new_dim
27302     { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
27303     \dim_set:Nn \l__box_bottom_new_dim
27304     { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
27305     \dim_set:Nn \l__box_right_new_dim
27306     { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
27307     \_box_resize_common:N #1
27308 }

```

(End definition for `\box_scale:Nnn` and others. These functions are documented on page 250.)

`\box_autosize_to_wd_and_ht:Nnn` Although autosizing a box uses dimensions, it has more in common in implementation with scaling. As such, most of the real work here is done elsewhere. `\box_autosize_to_wd_and_ht:cnn`

```

\box_gautosize_to_wd_and_ht:Nnn
\box_gautosize_to_wd_and_ht:cnn
\box_autosize_to_wd_and_ht_plus_dp:Nnn
\box_autosize_to_wd_and_ht_plus_dp:cnn
\box_gautosize_to_wd_and_ht_plus_dp:Nnn
\box_gautosize_to_wd_and_ht_plus_dp:cnn
\_box_autosize:NnnnN
27309 \cs_new_protected:Npn \box_autosize_to_wd_and_ht:Nnn #1#2#3
27310 { \_box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 } \hbox_set:Nn }
27311 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht:Nnn { c }
27312 \cs_new_protected:Npn \box_gautosize_to_wd_and_ht:Nnn #1#2#3
27313 { \_box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 } \hbox_gset:Nn }
27314 \cs_generate_variant:Nn \box_gautosize_to_wd_and_ht:Nnn { c }
27315 \cs_new_protected:Npn \box_autosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
27316 {
27317     \_box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 }

```

```

27318     \hbox_set:Nn
27319   }
27320 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht_plus_dp:Nnn { c }
27321 \cs_new_protected:Npn \box_gautosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
27322 {
27323   \__box_autosize:NnnN #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 }
27324   \hbox_gset:Nn
27325 }
27326 \cs_generate_variant:Nn \box_gautosize_to_wd_and_ht_plus_dp:Nnn { c }
27327 \cs_new_protected:Npn \__box_autosize:NnnN #1#2#3#4#5
27328 {
27329   #5 #1
27330   {
27331     \fp_set:Nn \l__box_scale_x_fp { ( #2 ) / \box_wd:N #1 }
27332     \fp_set:Nn \l__box_scale_y_fp { ( #3 ) / ( #4 ) }
27333     \fp_compare:nNnTF \l__box_scale_x_fp > \l__box_scale_y_fp
27334       { \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp }
27335       { \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp }
27336     \__box_scale:N #1
27337   }
27338 }

```

(End definition for `\box_autosize_to_wd_and_ht:Nnn` and others. These functions are documented on page 248.)

`__box_resize_common:N` The main resize function places its input into a box which start off with zero width, and includes the handles for engine rescaling.

```

27339 \cs_new_protected:Npn \__box_resize_common:N #1
27340 {
27341   \hbox_set:Nn \l__box_internal_box
27342   {
27343     \__box_backend_scale:Nnn
27344     #1
27345     \l__box_scale_x_fp
27346     \l__box_scale_y_fp
27347   }

```

The new height and depth can be applied directly.

```

27348   \fp_compare:nNnTF \l__box_scale_y_fp > \c_zero_fp
27349   {
27350     \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
27351     \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
27352   }
27353   {
27354     \box_set_dp:Nn \l__box_internal_box { \l__box_top_new_dim }
27355     \box_set_ht:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
27356   }

```

Things are not quite as obvious for the width, as the reference point needs to remain unchanged. For positive scaling factors resizing the box is all that is needed. However, for case of a negative scaling the material must be shifted such that the reference point ends up in the right place.

```

27357   \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
27358   {
27359     \hbox_to_wd:nn { \l__box_right_new_dim }

```

```

27360         {
27361             \tex_kern:D \l__box_right_new_dim
27362             \box_use_drop:N \l__box_internal_box
27363             \tex_hss:D
27364         }
27365     }
27366     {
27367         \box_set_wd:Nn \l__box_internal_box { \l__box_right_new_dim }
27368         \hbox:n
27369         {
27370             \tex_kern:D \c_zero_dim
27371             \box_use_drop:N \l__box_internal_box
27372             \tex_hss:D
27373         }
27374     }
27375 }

```

(End definition for `__box_resize_common:N`.)

```
27376 \</package>
```

43 l3coffins Implementation

```
27377 \*package>
```

```
27378 \@@=coffin>
```

43.1 Coffins: data structures and general variables

`\l__coffin_internal_box` Scratch variables.

```

\l__coffin_internal_dim 27379 \box_new:N \l__coffin_internal_box
\l__coffin_internal_tl 27380 \dim_new:N \l__coffin_internal_dim
27381 \tl_new:N \l__coffin_internal_tl

```

(End definition for `\l__coffin_internal_box`, `\l__coffin_internal_dim`, and `\l__coffin_internal_tl`.)

`\c__coffin_corners_prop` The “corners”; of a coffin define the real content, as opposed to the \TeX bounding box. They all start off in the same place, of course.

```

27382 \prop_const_from_keyval:Nn \c__coffin_corners_prop
27383 {
27384     tl = { Opt } { Opt } ,
27385     tr = { Opt } { Opt } ,
27386     bl = { Opt } { Opt } ,
27387     br = { Opt } { Opt } ,
27388 }

```

(End definition for `\c__coffin_corners_prop`.)

`\c__coffin_poles_prop` Pole positions are given for horizontal, vertical and reference-point based values.

```

27389 \prop_const_from_keyval:Nn \c__coffin_poles_prop
27390 {
27391     l  = { Opt } { Opt } { Opt } { 1000pt } ,
27392     hc = { Opt } { Opt } { Opt } { 1000pt } ,
27393     r  = { Opt } { Opt } { Opt } { 1000pt } ,

```



```

27394     b = { 0pt } { 0pt } { 1000pt } { 0pt } ,
27395     vc = { 0pt } { 0pt } { 1000pt } { 0pt } ,
27396     t  = { 0pt } { 0pt } { 1000pt } { 0pt } ,
27397     B  = { 0pt } { 0pt } { 1000pt } { 0pt } ,
27398     H  = { 0pt } { 0pt } { 1000pt } { 0pt } ,
27399     T  = { 0pt } { 0pt } { 1000pt } { 0pt } ,
27400 }

```

(End definition for \c__coffin_poles_prop.)

\l__coffin_slope_A_fp Used for calculations of intersections.

```

\l__coffin_slope_B_fp    27401 \fp_new:N \l__coffin_slope_A_fp
                           27402 \fp_new:N \l__coffin_slope_B_fp

```

(End definition for \l__coffin_slope_A_fp and \l__coffin_slope_B_fp.)

\l__coffin_error_bool For propagating errors so that parts of the code can work around them.

```

27403 \bool_new:N \l__coffin_error_bool

```

(End definition for \l__coffin_error_bool.)

\l__coffin_offset_x_dim The offset between two sets of coffin handles when typesetting. These values are corrected from those requested in an alignment for the positions of the handles.

```

\l__coffin_offset_y_dim    27404 \dim_new:N \l__coffin_offset_x_dim
                           27405 \dim_new:N \l__coffin_offset_y_dim

```

(End definition for \l__coffin_offset_x_dim and \l__coffin_offset_y_dim.)

\l__coffin_pole_a_tl Needed for finding the intersection of two poles.

```

\l__coffin_pole_b_tl    27406 \tl_new:N \l__coffin_pole_a_tl
                           27407 \tl_new:N \l__coffin_pole_b_tl

```

(End definition for \l__coffin_pole_a_tl and \l__coffin_pole_b_tl.)

\l__coffin_x_dim For calculating intersections and so forth.

```

\l__coffin_y_dim    27408 \dim_new:N \l__coffin_x_dim
\l__coffin_x_prime_dim 27409 \dim_new:N \l__coffin_y_dim
\l__coffin_y_prime_dim 27410 \dim_new:N \l__coffin_x_prime_dim
                           27411 \dim_new:N \l__coffin_y_prime_dim

```

(End definition for \l__coffin_x_dim and others.)

43.2 Basic coffin functions

There are a number of basic functions needed for creating coffins and placing material in them. This all relies on the following data structures.

__coffin_to_value:N Coffins are a two-part structure and we rely on the internal nature of box allocation to make everything work. As such, we need an interface to turn coffin identifiers into numbers. For the purposes here, the signature allowed is N despite the nature of the underlying primitive.

```

27412 \cs_new_eq:NN \__coffin_to_value:N \tex_number:D

```

(End definition for __coffin_to_value:N.)

`\coffin_if_exist_p:N` Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. A cleaner way to handle this is provided here: both the box and the coffin structure are checked.

```

\coffin_if_exist_p:c
\coffin_if_exist:NTF
\coffin_if_exist:cTF
27413 \prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }
27414 {
27415     \cs_if_exist:NTF #1
27416     {
27417         \cs_if_exist:cTF { coffin ~ \__coffin_to_value:N #1 ~ poles }
27418         { \prg_return_true: }
27419         { \prg_return_false: }
27420     }
27421     { \prg_return_false: }
27422 }
27423 \prg_generate_conditional_variant:Nnn \coffin_if_exist:N
27424 { c } { p , T , F , TF }

```

(End definition for `\coffin_if_exist:NTF`. This function is documented on page 251.)

`__coffin_if_exist:NT` Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. So a wrapper is provided to deal with this correctly, issuing an error on erroneous use.

```

27425 \cs_new_protected:Npn \__coffin_if_exist:NT #1#2
27426 {
27427     \coffin_if_exist:NTF #1
27428     { #2 }
27429     {
27430         \_kernel_msg_error:nxn { kernel } { unknown-coffin }
27431         { \token_to_str:N #1 }
27432     }
27433 }

```

(End definition for `__coffin_if_exist:NT`.)

`\coffin_clear:N` Clearing coffins means emptying the box and resetting all of the structures.

```

\coffin_clear:c
\coffin_gclear:N
\coffin_gclear:c
27434 \cs_new_protected:Npn \coffin_clear:N #1
27435 {
27436     \__coffin_if_exist:NT #1
27437     {
27438         \box_clear:N #1
27439         \__coffin_reset_structure:N #1
27440     }
27441 }
27442 \cs_generate_variant:Nn \coffin_clear:N { c }
27443 \cs_new_protected:Npn \coffin_gclear:N #1
27444 {
27445     \__coffin_if_exist:NT #1
27446     {
27447         \box_gclear:N #1
27448         \__coffin_greset_structure:N #1
27449     }
27450 }
27451 \cs_generate_variant:Nn \coffin_gclear:N { c }

```

(End definition for `\coffin_clear:N` and `\coffin_gclear:N`. These functions are documented on page 251.)

\coffin_new:N Creating a new coffin means making the underlying box and adding the data structures. The **\debug_suspend:** and **\debug_resume:** functions prevent **\prop_gclear_new:c** from writing useless information to the log file.

```

27452 \cs_new_protected:Npn \coffin_new:N #1
27453 {
27454   \box_new:N #1
27455   \debug_suspend:
27456   \prop_gclear_new:c { coffin ~ \__coffin_to_value:N #1 ~ corners }
27457   \prop_gclear_new:c { coffin ~ \__coffin_to_value:N #1 ~ poles }
27458   \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
27459     \c__coffin_corners_prop
27460   \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
27461     \c__coffin_poles_prop
27462   \debug_resume:
27463 }
27464 \cs_generate_variant:Nn \coffin_new:N { c }

```

(End definition for **\coffin_new:N**. This function is documented on page 251.)

\hcoffin_set:Nn Horizontal coffins are relatively easy: set the appropriate box, reset the structures then update the handle positions.

```

\hcoffin_set:cn
\hcoffin_gset:Nn
\hcoffin_gset:cn
27465 \cs_new_protected:Npn \hcoffin_set:Nn #1#2
27466 {
27467   \__coffin_if_exist:NT #1
27468   {
27469     \hbox_set:Nn #1
27470     {
27471       \color_ensure_current:
27472       #2
27473     }
27474     \__coffin_update:N #1
27475   }
27476 }
27477 \cs_generate_variant:Nn \hcoffin_set:Nn { c }
27478 \cs_new_protected:Npn \hcoffin_gset:Nn #1#2
27479 {
27480   \__coffin_if_exist:NT #1
27481   {
27482     \hbox_gset:Nn #1
27483     {
27484       \color_ensure_current:
27485       #2
27486     }
27487     \__coffin_gupdate:N #1
27488   }
27489 }
27490 \cs_generate_variant:Nn \hcoffin_gset:Nn { c }

```

(End definition for **\hcoffin_set:Nn** and **\hcoffin_gset:Nn**. These functions are documented on page 251.)

\vcoffin_set:Nnn Setting vertical coffins is more complex. First, the material is typeset with a given width. The default handles and poles are set as for a horizontal coffin, before finding the top baseline using a temporary box. No **\color_ensure_current:** here as that would add a

```

\__coffin_set_vertical:NnnNN
\__coffin_set_vertical_aux:

```

whatsit to the start of the vertical box and mess up the location of the T pole (see *T_EX by Topic* for discussion of the `\vtop` primitive, used to do the measuring).

```

27491 \cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
27492 {
27493   \__coffin_set_vertical:NnnNN #1 {#2} {#3}
27494   \vbox_set:Nn \__coffin_update:N
27495 }
27496 \cs_generate_variant:Nn \vcoffin_set:Nnn { c }
27497 \cs_new_protected:Npn \vcoffin_gset:Nnn #1#2#3
27498 {
27499   \__coffin_set_vertical:NnnNN #1 {#2} {#3}
27500   \vbox_gset:Nn \__coffin_gupdate:N
27501 }
27502 \cs_generate_variant:Nn \vcoffin_gset:Nnn { c }
27503 \cs_new_protected:Npn \__coffin_set_vertical:NnnNN #1#2#3#4#5
27504 {
27505   \__coffin_if_exist:NT #1
27506   {
27507     #4 #1
27508     {
27509       \dim_set:Nn \tex_hsize:D {#2}
27510       \__coffin_set_vertical_aux:
27511       #3
27512     }
27513     #5 #1
27514     \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
27515     \__coffin_set_pole:Nnx #1 { T }
27516     {
27517       { Opt }
27518       {
27519         \dim_eval:n
27520         { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
27521       }
27522       { 1000pt }
27523       { Opt }
27524     }
27525     \box_clear:N \l__coffin_internal_box
27526   }
27527 }
27528 \cs_new_protected:Npx \__coffin_set_vertical_aux:
27529 {
27530   \cs_if_exist:NT \linewidth
27531   { \dim_set_eq:NN \linewidth \tex_hsize:D }
27532   \cs_if_exist:NT \columnwidth
27533   { \dim_set_eq:NN \columnwidth \tex_hsize:D }
27534 }

```

(End definition for `\vcoffin_set:Nnn` and others. These functions are documented on page 252.)

<pre> \hcoffin_set:Nw \hcoffin_set:cw \hcoffin_gset:Nw \hcoffin_gset:cw \hcoffin_set_end: \hcoffin_gset_end: </pre>	<p>These are the “begin”/“end” versions of the above: watch the grouping!</p> <pre> 27535 \cs_new_protected:Npn \hcoffin_set:Nw #1 27536 { 27537 __coffin_if_exist:NT #1 27538 { </pre>
---	--

```

27539         \hbox_set:Nw #1 \color_ensure_current:
27540         \cs_set_protected:Npn \hcoffin_set_end:
27541         {
27542             \hbox_set_end:
27543             \__coffin_update:N #1
27544         }
27545     }
27546 }
27547 \cs_generate_variant:Nn \hcoffin_set:Nw { c }
27548 \cs_new_protected:Npn \hcoffin_gset:Nw #1
27549 {
27550     \__coffin_if_exist:NT #1
27551     {
27552         \hbox_gset:Nw #1 \color_ensure_current:
27553         \cs_set_protected:Npn \hcoffin_gset_end:
27554         {
27555             \hbox_gset_end:
27556             \__coffin_gupdate:N #1
27557         }
27558     }
27559 }
27560 \cs_generate_variant:Nn \hcoffin_gset:Nw { c }
27561 \cs_new_protected:Npn \hcoffin_set_end: { }
27562 \cs_new_protected:Npn \hcoffin_gset_end: { }

```

(End definition for `\hcoffin_set:Nw` and others. These functions are documented on page 252.)

```

\vcoffin_set:Nnw The same for vertical coffins.
\vcoffin_set:cnw 27563 \cs_new_protected:Npn \vcoffin_set:Nnw #1#2
\vcoffin_gset:Nnw 27564 {
\vcoffin_gset:cnw 27565     \__coffin_set_vertical:NnNNNNw #1 {#2} \vbox_set:Nw
\__coffin_set_vertical:NnNNNNw 27566     \vcoffin_set_end:
\vcoffin_set_end: 27567     \vbox_set_end: \__coffin_update:N
\vcoffin_gset_end: 27568 }
27569 \cs_generate_variant:Nn \vcoffin_set:Nnw { c }
27570 \cs_new_protected:Npn \vcoffin_gset:Nnw #1#2
27571 {
27572     \__coffin_set_vertical:NnNNNNw #1 {#2} \vbox_gset:Nw
27573     \vcoffin_gset_end:
27574     \vbox_gset_end: \__coffin_gupdate:N
27575 }
27576 \cs_generate_variant:Nn \vcoffin_gset:Nnw { c }
27577 \cs_new_protected:Npn \__coffin_set_vertical:NnNNNNw #1#2#3#4#5#6
27578 {
27579     \__coffin_if_exist:NT #1
27580     {
27581         #3 #1
27582         \dim_set:Nn \tex_hsize:D {#2}
27583         \__coffin_set_vertical_aux:
27584         \cs_set_protected:Npn #4
27585         {
27586             #5
27587             #6 #1
27588             \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }

```

```

27589         \__coffin_set_pole:Nnx #1 { T }
27590         {
27591             { Opt }
27592             {
27593                 \dim_eval:n
27594                 { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
27595             }
27596             { 1000pt }
27597             { Opt }
27598         }
27599         \box_clear:N \l__coffin_internal_box
27600     }
27601 }
27602 }
27603 \cs_new_protected:Npn \vcoffin_set_end: { }
27604 \cs_new_protected:Npn \vcoffin_gset_end: { }

```

(End definition for \vcoffin_set:Nnw and others. These functions are documented on page 252.)

```

\coffin_set_eq:NN Setting two coffins equal is just a wrapper around other functions.
\coffin_set_eq:Nc 27605 \cs_new_protected:Npn \coffin_set_eq:NN #1#2
\coffin_set_eq:cN 27606 {
\coffin_set_eq:cc 27607     \__coffin_if_exist:NT #1
\coffin_gset_eq:NN 27608     {
\coffin_gset_eq:Nc 27609         \box_set_eq:NN #1 #2
\coffin_gset_eq:cN 27610         \prop_set_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ corners }
\coffin_gset_eq:cc 27611             { coffin ~ \__coffin_to_value:N #2 ~ corners }
27612         \prop_set_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ poles }
27613             { coffin ~ \__coffin_to_value:N #2 ~ poles }
27614     }
27615 }
27616 \cs_generate_variant:Nn \coffin_set_eq:NN { c , Nc , cc }
27617 \cs_new_protected:Npn \coffin_gset_eq:NN #1#2
27618 {
27619     \__coffin_if_exist:NT #1
27620     {
27621         \box_gset_eq:NN #1 #2
27622         \prop_gset_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ corners }
27623             { coffin ~ \__coffin_to_value:N #2 ~ corners }
27624         \prop_gset_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ poles }
27625             { coffin ~ \__coffin_to_value:N #2 ~ poles }
27626     }
27627 }
27628 \cs_generate_variant:Nn \coffin_gset_eq:NN { c , Nc , cc }

```

(End definition for \coffin_set_eq:NN and \coffin_gset_eq:NN. These functions are documented on page 251.)

```

\c_empty_coffin Special coffins: these cannot be set up earlier as they need \coffin_new:N. The empty
\l__coffin_aligned_coffin coffin is set as a box as the full coffin-setting system needs some material which is not
\l__coffin_aligned_internal_coffin yet available. The empty coffin is created entirely by hand: not everything is in place yet.
27629 \coffin_new:N \c_empty_coffin
27630 \coffin_new:N \l__coffin_aligned_coffin
27631 \coffin_new:N \l__coffin_aligned_internal_coffin

```

(End definition for `\c_empty_coffin`, `\l__coffin_aligned_coffin`, and `\l__coffin_aligned_internal_coffin`. This variable is documented on page 255.)

```
\l_tmpa_coffin The usual scratch space.
\l_tmpb_coffin 27632 \coffin_new:N \l_tmpa_coffin
\g_tmpa_coffin 27633 \coffin_new:N \l_tmpb_coffin
\g_tmpb_coffin 27634 \coffin_new:N \g_tmpa_coffin
                27635 \coffin_new:N \g_tmpb_coffin
```

(End definition for `\l_tmpa_coffin` and others. These variables are documented on page 255.)

43.3 Measuring coffins

`\coffin_dp:N` Coffins are just boxes when it comes to measurement. However, semantically a separate set of functions are required.

```
\coffin_dp:c 27636 \cs_new_eq:NN \coffin_dp:N \box_dp:N
\coffin_ht:c 27637 \cs_new_eq:NN \coffin_dp:c \box_dp:c
\coffin_ht:N 27638 \cs_new_eq:NN \coffin_ht:N \box_ht:N
\coffin_wd:c 27639 \cs_new_eq:NN \coffin_ht:c \box_ht:c
\coffin_wd:N 27640 \cs_new_eq:NN \coffin_wd:N \box_wd:N
                27641 \cs_new_eq:NN \coffin_wd:c \box_wd:c
```

(End definition for `\coffin_dp:N`, `\coffin_ht:N`, and `\coffin_wd:N`. These functions are documented on page 254.)

43.4 Coffins: handle and pole management

`__coffin_get_pole:NnN` A simple wrapper around the recovery of a coffin pole, with some error checking and recovery built-in.

```
27642 \cs_new_protected:Npn \__coffin_get_pole:NnN #1#2#3
27643 {
27644   \prop_get:cnNF
27645   { coffin ~ \__coffin_to_value:N #1 ~ poles } {#2} #3
27646   {
27647     \__kernel_msg_error:nxxx { kernel } { unknown-coffin-pole }
27648     { \exp_not:n {#2} } { \token_to_str:N #1 }
27649     \tl_set:Nn #3 { { Opt } { Opt } { Opt } { Opt } }
27650   }
27651 }
```

(End definition for `__coffin_get_pole:NnN`.)

`__coffin_reset_structure:N` Resetting the structure is a simple copy job.

```
\__coffin_greset_structure:N 27652 \cs_new_protected:Npn \__coffin_reset_structure:N #1
                             27653 {
                             27654   \prop_set_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
                             27655   \c__coffin_corners_prop
                             27656   \prop_set_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
                             27657   \c__coffin_poles_prop
                             27658 }
                             27659 \cs_new_protected:Npn \__coffin_greset_structure:N #1
                             27660 {
                             27661   \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
                             27662   \c__coffin_corners_prop
```

```

27663     \prop_gset_eq:cN { coffin ~ \_coffin_to_value:N #1 ~ poles }
27664     \c__coffin_poles_prop
27665 }

```

(End definition for _coffin_reset_structure:N and _coffin_greset_structure:N.)

`\coffin_set_horizontal_pole:Nnn` Setting the pole of a coffin at the user/designer level requires a bit more care. The idea here is to provide a reasonable interface to the system, then to do the setting with full expansion. The three-argument version is used internally to do a direct setting.

```

\coffin_set_horizontal_pole:cnn
\coffin_gset_horizontal_pole:Nnn
\coffin_gset_horizontal_pole:cnn
\_coffin_set_horizontal_pole:NnnN
\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnn
\coffin_gset_vertical_pole:Nnn
\coffin_gset_vertical_pole:cnn
\_coffin_set_vertical_pole:NnnN
\_coffin_set_pole:Nnn
\_coffin_set_pole:Nnx

27666 \cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
27667 { \_coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_put:cnx }
27668 \cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
27669 \cs_new_protected:Npn \coffin_gset_horizontal_pole:Nnn #1#2#3
27670 { \_coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_gput:cnx }
27671 \cs_generate_variant:Nn \coffin_gset_horizontal_pole:Nnn { c }
27672 \cs_new_protected:Npn \_coffin_set_horizontal_pole:NnnN #1#2#3#4
27673 {
27674   \_coffin_if_exist:NT #1
27675   {
27676     #4 { coffin ~ \_coffin_to_value:N #1 ~ poles }
27677     {#2}
27678     {
27679       { Opt } { \dim_eval:n {#3} }
27680       { 1000pt } { Opt }
27681     }
27682   }
27683 }
27684 \cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
27685 { \_coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_put:cnx }
27686 \cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
27687 \cs_new_protected:Npn \coffin_gset_vertical_pole:Nnn #1#2#3
27688 { \_coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_gput:cnx }
27689 \cs_generate_variant:Nn \coffin_gset_vertical_pole:Nnn { c }
27690 \cs_new_protected:Npn \_coffin_set_vertical_pole:NnnN #1#2#3#4
27691 {
27692   \_coffin_if_exist:NT #1
27693   {
27694     #4 { coffin ~ \_coffin_to_value:N #1 ~ poles }
27695     {#2}
27696     {
27697       { \dim_eval:n {#3} } { Opt }
27698       { Opt } { 1000pt }
27699     }
27700   }
27701 }
27702 \cs_new_protected:Npn \_coffin_set_pole:Nnn #1#2#3
27703 {
27704   \prop_put:cnn { coffin ~ \_coffin_to_value:N #1 ~ poles }
27705   {#2} {#3}
27706 }
27707 \cs_generate_variant:Nn \_coffin_set_pole:Nnn { Nnx }

```

(End definition for \coffin_set_horizontal_pole:Nnn and others. These functions are documented on page 252.)


```

\__coffin_update:N
\__coffin_gupdate:N
27708 \cs_new_protected:Npn \__coffin_update:N #1
27709 {
27710     \__coffin_reset_structure:N #1
27711     \__coffin_update_corners:N #1
27712     \__coffin_update_poles:N #1
27713 }
27714 \cs_new_protected:Npn \__coffin_gupdate:N #1
27715 {
27716     \__coffin_greset_structure:N #1
27717     \__coffin_gupdate_corners:N #1
27718     \__coffin_gupdate_poles:N #1
27719 }

```

Simple shortcuts.

(End definition for `__coffin_update:N` and `__coffin_gupdate:N`.)

```

\__coffin_update_corners:N
\__coffin_gupdate_corners:N
\__coffin_update_corners:NN
\__coffin_update_corners:NNN
27720 \cs_new_protected:Npn \__coffin_update_corners:N #1
27721 { \__coffin_update_corners:NN #1 \prop_put:Nnx }
27722 \cs_new_protected:Npn \__coffin_gupdate_corners:N #1
27723 { \__coffin_update_corners:NN #1 \prop_gput:Nnx }
27724 \cs_new_protected:Npn \__coffin_update_corners:NN #1#2
27725 {
27726     \exp_args:Nc \__coffin_update_corners:NNN
27727     { coffin ~ \__coffin_to_value:N #1 ~ corners }
27728     #1 #2
27729 }
27730 \cs_new_protected:Npn \__coffin_update_corners:NNN #1#2#3
27731 {
27732     #3 #1
27733     { tl }
27734     { { Opt } { \dim_eval:n { \box_ht:N #2 } } }
27735     #3 #1
27736     { tr }
27737     {
27738         { \dim_eval:n { \box_wd:N #2 } }
27739         { \dim_eval:n { \box_ht:N #2 } }
27740     }
27741     #3 #1
27742     { bl }
27743     { { Opt } { \dim_eval:n { -\box_dp:N #2 } } }
27744     #3 #1
27745     { br }
27746     {
27747         { \dim_eval:n { \box_wd:N #2 } }
27748         { \dim_eval:n { -\box_dp:N #2 } }
27749     }
27750 }

```

Updating the corners of a coffin is straight-forward as at this stage there can be no rotation. So the corners of the content are just those of the underlying \TeX box.

(End definition for `__coffin_update_corners:N` and others.)

```

\__coffin_update_poles:N
\__coffin_gupdate_poles:N
\__coffin_update_poles:NN
\__coffin_update_poles:NNN

```

This function is called when a coffin is set, and updates the poles to reflect the nature of size of the box. Thus this function only alters poles where the default position is

dependent on the size of the box. It also does not set poles which are relevant only to vertical coffins.

```

27751 \cs_new_protected:Npn \__coffin_update_poles:N #1
27752 { \__coffin_update_poles:NN #1 \prop_put:Nnx }
27753 \cs_new_protected:Npn \__coffin_gupdate_poles:N #1
27754 { \__coffin_update_poles:NN #1 \prop_gput:Nnx }
27755 \cs_new_protected:Npn \__coffin_update_poles:NN #1#2
27756 {
27757   \exp_args:Nc \__coffin_update_poles:NNN
27758     { coffin ~ \__coffin_to_value:N #1 ~ poles }
27759   #1 #2
27760 }
27761 \cs_new_protected:Npn \__coffin_update_poles:NNN #1#2#3
27762 {
27763   #3 #1 { hc }
27764   {
27765     { \dim_eval:n { 0.5 \box_wd:N #2 } }
27766     { Opt } { Opt } { 1000pt }
27767   }
27768   #3 #1 { r }
27769   {
27770     { \dim_eval:n { \box_wd:N #2 } }
27771     { Opt } { Opt } { 1000pt }
27772   }
27773   #3 #1 { vc }
27774   {
27775     { Opt }
27776     { \dim_eval:n { ( \box_ht:N #2 - \box_dp:N #2 ) / 2 } }
27777     { 1000pt }
27778     { Opt }
27779   }
27780   #3 #1 { t }
27781   {
27782     { Opt }
27783     { \dim_eval:n { \box_ht:N #2 } }
27784     { 1000pt }
27785     { Opt }
27786   }
27787   #3 #1 { b }
27788   {
27789     { Opt }
27790     { \dim_eval:n { -\box_dp:N #2 } }
27791     { 1000pt }
27792     { Opt }
27793   }
27794 }

```

(End definition for __coffin_update_poles:N and others.)

43.5 Coffins: calculation of pole intersections

The lead off in finding intersections is to recover the two poles and then hand off to the auxiliary for the actual calculation. There may of course not be an intersection, for which an error trap is needed.

```

27795 \cs_new_protected:Npn \__coffin_calculate_intersection:Nnn #1#2#3
27796 {
27797   \__coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
27798   \__coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
27799   \bool_set_false:N \l__coffin_error_bool
27800   \exp_last_two_unbraced:Noo
27801     \__coffin_calculate_intersection:nnnnnnnn
27802     \l__coffin_pole_a_tl \l__coffin_pole_b_tl
27803   \bool_if:NT \l__coffin_error_bool
27804     {
27805       \__kernel_msg_error:nn { kernel } { no-pole-intersection }
27806       \dim_zero:N \l__coffin_x_dim
27807       \dim_zero:N \l__coffin_y_dim
27808     }
27809 }

```

The two poles passed here each have four values (as dimensions), (a, b, c, d) and (a', b', c', d') . These are arguments 1–4 and 5–8, respectively. In both cases a and b are the co-ordinates of a point on the pole and c and d define the direction of the pole. Finding the intersection depends on the directions of the poles, which are given by d/c and d'/c' . However, if one of the poles is either horizontal or vertical then one or more of c, d, c' and d' are zero and a special case is needed.

```

27810 \cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnnnn
27811   #1#2#3#4#5#6#7#8
27812 {
27813   \dim_compare:nNnTF {#3} = \c_zero_dim

```

The case where the first pole is vertical. So the x -component of the interaction is at a . There is then a test on the second pole: if it is also vertical then there is an error.

```

27814   {
27815     \dim_set:Nn \l__coffin_x_dim {#1}
27816     \dim_compare:nNnTF {#7} = \c_zero_dim
27817       { \bool_set_true:N \l__coffin_error_bool }

```

The second pole may still be horizontal, in which case the y -component of the intersection is b' . If not,

$$y = \frac{d'}{c'}(a - a') + b'$$

with the x -component already known to be $\#1$.

```

27818   {
27819     \dim_set:Nn \l__coffin_y_dim
27820       {
27821         \dim_compare:nNnTF {#8} = \c_zero_dim
27822           {#6}
27823         {
27824           \fp_to_dim:n
27825             {
27826               ( \dim_to_fp:n {#8} / \dim_to_fp:n {#7} )
27827               * ( \dim_to_fp:n {#1} - \dim_to_fp:n {#5} )
27828               + \dim_to_fp:n {#6}
27829             }
27830         }
27831       }
27832   }
27833 }

```

If the first pole is not vertical then it may be horizontal. If so, then the procedure is essentially the same as that already done but with the x - and y -components interchanged.

```

27834     {
27835         \dim_compare:nNnTF {#4} = \c_zero_dim
27836         {
27837             \dim_set:Nn \l__coffin_y_dim {#2}
27838             \dim_compare:nNnTF {#8} = { \c_zero_dim }
27839             { \bool_set_true:N \l__coffin_error_bool }
27840         }

```

Now we deal with the case where the second pole may be vertical, or if not we have

$$x = \frac{c'}{d'}(b - b') + a'$$

which is again handled by the same auxiliary.

```

27841         \dim_set:Nn \l__coffin_x_dim
27842         {
27843             \dim_compare:nNnTF {#7} = \c_zero_dim
27844             {#5}
27845             {
27846                 \fp_to_dim:n
27847                 {
27848                     ( \dim_to_fp:n {#7} / \dim_to_fp:n {#8} )
27849                     * ( \dim_to_fp:n {#4} - \dim_to_fp:n {#6} )
27850                     + \dim_to_fp:n {#5}
27851                 }
27852             }
27853         }
27854     }
27855 }

```

The first pole is neither horizontal nor vertical. To avoid even more complexity, we now work out both slopes and pass to an auxiliary.

```

27856     {
27857         \use:x
27858         {
27859             \__coffin_calculate_intersection:nnnnnn
27860             { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
27861             { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
27862         }
27863         {#1} {#2} {#5} {#6}
27864     }
27865 }
27866 }

```

Assuming the two poles are not parallel, then the intersection point is found in two steps. First we find the x -value with

$$x = \frac{sa - s'a' - b + b'}{s - s'}$$

and then finding the y -value with

$$y = s(x - a) + b$$

```

27867 \cs_set_protected:Npn \__coffin_calculate_intersection:nnnnnn #1#2#3#4#5#6
27868 {
27869   \fp_compare:nNnTF {#1} = {#2}
27870   { \bool_set_true:N \l__coffin_error_bool }
27871   {
27872     \dim_set:Nn \l__coffin_x_dim
27873     {
27874       \fp_to_dim:n
27875       {
27876         (
27877           #1 * \dim_to_fp:n {#3}
27878           - #2 * \dim_to_fp:n {#5}
27879           - \dim_to_fp:n {#4}
27880           + \dim_to_fp:n {#6}
27881         )
27882         /
27883         ( #1 - #2 )
27884       }
27885     }
27886     \dim_set:Nn \l__coffin_y_dim
27887     {
27888       \fp_to_dim:n
27889       {
27890         #1 * ( \l__coffin_x_dim - \dim_to_fp:n {#3} )
27891         + \dim_to_fp:n {#4}
27892       }
27893     }
27894   }
27895 }

```

(End definition for `__coffin_calculate_intersection:Nnn`, `__coffin_calculate_intersection:nnnnnnnn`, and `__coffin_calculate_intersection:nnnnnn`.)

43.6 Affine transformations

`\l__coffin_sin_fp` Used for rotations to get the sine and cosine values.
`\l__coffin_cos_fp`

```

27896 \fp_new:N \l__coffin_sin_fp
27897 \fp_new:N \l__coffin_cos_fp

```

(End definition for `\l__coffin_sin_fp` and `\l__coffin_cos_fp`.)

`\l__coffin_bounding_prop` A property list for the bounding box of a coffin. This is only needed during the rotation, so there is just the one.

```

27898 \prop_new:N \l__coffin_bounding_prop

```

(End definition for `\l__coffin_bounding_prop`.)

`\l__coffin_corners_prop` Used to avoid needing to track scope for intermediate steps.
`\l__coffin_poles_prop`

```

27899 \prop_new:N \l__coffin_corners_prop
27900 \prop_new:N \l__coffin_poles_prop

```

(End definition for `\l__coffin_corners_prop` and `\l__coffin_poles_prop`.)

`\l__coffin_bounding_shift_dim` The shift of the bounding box of a coffin from the real content.

```

27901 \dim_new:N \l__coffin_bounding_shift_dim

```

(End definition for \l__coffin_bounding_shift_dim.)

\l__coffin_left_corner_dim These are used to hold maxima for the various corner values: these thus define the
 \l__coffin_right_corner_dim minimum size of the bounding box after rotation.
 \l__coffin_bottom_corner_dim
 \l__coffin_top_corner_dim

```
27902 \dim_new:N \l__coffin_left_corner_dim
27903 \dim_new:N \l__coffin_right_corner_dim
27904 \dim_new:N \l__coffin_bottom_corner_dim
27905 \dim_new:N \l__coffin_top_corner_dim
```

(End definition for \l__coffin_left_corner_dim and others.)

\coffin_rotate:Nn Rotating a coffin requires several steps which can be conveniently run together. The sine
\coffin_rotate:cn and cosine of the angle in degrees are computed. This is then used to set \l__coffin_
\coffin_grotate:Nn sin_fp and \l__coffin_cos_fp, which are carried through unchanged for the rest of
\coffin_grotate:cn the procedure.
 __coffin_rotate:NnNNN

```
27906 \cs_new_protected:Npn \coffin_rotate:Nn #1#2
27907 { \__coffin_rotate:NnNNN #1 {#2} \box_rotate:Nn \prop_set_eq:cn \hbox_set:Nn }
27908 \cs_generate_variant:Nn \coffin_rotate:Nn { c }
27909 \cs_new_protected:Npn \coffin_grotate:Nn #1#2
27910 { \__coffin_rotate:NnNNN #1 {#2} \box_grotate:Nn \prop_gset_eq:cn \hbox_gset:Nn }
27911 \cs_generate_variant:Nn \coffin_grotate:Nn { c }
27912 \cs_new_protected:Npn \__coffin_rotate:NnNNN #1#2#3#4#5
27913 {
27914   \fp_set:Nn \l__coffin_sin_fp { sind ( #2 ) }
27915   \fp_set:Nn \l__coffin_cos_fp { cosd ( #2 ) }
```

Use a local copy of the property lists to avoid needing to pass the name and scope around.

```
27916 \prop_set_eq:Nc \l__coffin_corners_prop
27917 { coffin ~ \__coffin_to_value:N #1 ~ corners }
27918 \prop_set_eq:Nc \l__coffin_poles_prop
27919 { coffin ~ \__coffin_to_value:N #1 ~ poles }
```

The corners and poles of the coffin can now be rotated around the origin. This is best achieved using mapping functions.

```
27920 \prop_map_inline:Nn \l__coffin_corners_prop
27921 { \__coffin_rotate_corner:Nnnn #1 {##1} ##2 }
27922 \prop_map_inline:Nn \l__coffin_poles_prop
27923 { \__coffin_rotate_pole:Nnnnnn #1 {##1} ##2 }
```

The bounding box of the coffin needs to be rotated, and to do this the corners have to be found first. They are then rotated in the same way as the corners of the coffin material itself.

```
27924 \__coffin_set_bounding:N #1
27925 \prop_map_inline:Nn \l__coffin_bounding_prop
27926 { \__coffin_rotate_bounding:nnn {##1} ##2 }
```

At this stage, there needs to be a calculation to find where the corners of the content and the box itself will end up.

```
27927 \__coffin_find_corner_maxima:N #1
27928 \__coffin_find_bounding_shift:
27929 #3 #1 {#2}
```

The correction of the box position itself takes place here. The idea is that the bounding box for a coffin is tight up to the content, and has the reference point at the bottom-left. The x -direction is handled by moving the content by the difference in the positions of the bounding box and the content left edge. The y -direction is dealt with by moving the box down by any depth it has acquired. The internal box is used here to allow for the next step.

```

27930 \hbox_set:Nn \l__coffin_internal_box
27931 {
27932   \tex_kern:D
27933   \dim_eval:n
27934     { \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim }
27935   \exp_stop_f:
27936   \box_move_down:nn { \l__coffin_bottom_corner_dim }
27937   { \box_use:N #1 }
27938 }

```

If there have been any previous rotations then the size of the bounding box will be bigger than the contents. This can be corrected easily by setting the size of the box to the height and width of the content. As this operation requires setting box dimensions and these transcend grouping, the safe way to do this is to use the internal box and to reset the result into the target box.

```

27939 \box_set_ht:Nn \l__coffin_internal_box
27940   { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
27941 \box_set_dp:Nn \l__coffin_internal_box { 0pt }
27942 \box_set_wd:Nn \l__coffin_internal_box
27943   { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
27944 #5 #1 { \box_use_drop:N \l__coffin_internal_box }

```

The final task is to move the poles and corners such that they are back in alignment with the box reference point.

```

27945 \prop_map_inline:Nn \l__coffin_corners_prop
27946   { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
27947 \prop_map_inline:Nn \l__coffin_poles_prop
27948   { \__coffin_shift_pole:Nnnnnn #1 {##1} ##2 }

```

Update the coffin data.

```

27949 #4 { coffin ~ \__coffin_to_value:N #1 ~ corners }
27950   \l__coffin_corners_prop
27951 #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
27952   \l__coffin_poles_prop
27953 }

```

(End definition for \coffin_rotate:Nn, \coffin_grotate:Nn, and __coffin_rotate:NnNNN. These functions are documented on page 253.)

`__coffin_set_bounding:N` The bounding box corners for a coffin are easy enough to find: this is the same code as for the corners of the material itself, but using a dedicated property list.

```

27954 \cs_new_protected:Npn \__coffin_set_bounding:N #1
27955 {
27956   \prop_put:Nnx \l__coffin_bounding_prop { tl }
27957   { { 0pt } { \dim_eval:n { \box_ht:N #1 } } }
27958   \prop_put:Nnx \l__coffin_bounding_prop { tr }
27959   {
27960     { \dim_eval:n { \box_wd:N #1 } }

```

```

27961         { \dim_eval:n { \box_ht:N #1 } }
27962     }
27963     \dim_set:Nn \l__coffin_internal_dim { -\box_dp:N #1 }
27964     \prop_put:Nnx \l__coffin_bounding_prop { bl }
27965     { { Opt } { \dim_use:N \l__coffin_internal_dim } }
27966     \prop_put:Nnx \l__coffin_bounding_prop { br }
27967     {
27968         { \dim_eval:n { \box_wd:N #1 } }
27969         { \dim_use:N \l__coffin_internal_dim }
27970     }
27971 }

```

(End definition for __coffin_set_bounding:N.)

__coffin_rotate_bounding:nnn Rotating the position of the corner of the coffin is just a case of treating this as a vector from the reference point. The same treatment is used for the corners of the material itself and the bounding box.

```

27972 \cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
27973 {
27974     \__coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
27975     \prop_put:Nnx \l__coffin_bounding_prop {#1}
27976     { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
27977 }
27978 \cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
27979 {
27980     \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
27981     \prop_put:Nnx \l__coffin_corners_prop {#2}
27982     { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
27983 }

```

(End definition for __coffin_rotate_bounding:nnn and __coffin_rotate_corner:Nnnn.)

__coffin_rotate_pole:Nnnnnn Rotating a single pole simply means shifting the co-ordinate of the pole and its direction. The rotation here is about the bottom-left corner of the coffin.

```

27984 \cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
27985 {
27986     \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
27987     \__coffin_rotate_vector:nnNN {#5} {#6}
27988     \l__coffin_x_prime_dim \l__coffin_y_prime_dim
27989     \prop_put:Nnx \l__coffin_poles_prop {#2}
27990     {
27991         { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
27992         { \dim_use:N \l__coffin_x_prime_dim }
27993         { \dim_use:N \l__coffin_y_prime_dim }
27994     }
27995 }

```

(End definition for __coffin_rotate_pole:Nnnnnn.)

__coffin_rotate_vector:nnNN A rotation function, which needs only an input vector (as dimensions) and an output space. The values \l__coffin_cos_fp and \l__coffin_sin_fp should previously have been set up correctly. Working this way means that the floating point work is kept to a minimum: for any given rotation the sin and cosine values do no change, after all.

```

27996 \cs_new_protected:Npn \__coffin_rotate_vector:nnNN #1#2#3#4

```



```

27997 {
27998   \dim_set:Nn #3
27999   {
28000     \fp_to_dim:n
28001     {
28002       \dim_to_fp:n {#1} * \l__coffin_cos_fp
28003       - \dim_to_fp:n {#2} * \l__coffin_sin_fp
28004     }
28005   }
28006   \dim_set:Nn #4
28007   {
28008     \fp_to_dim:n
28009     {
28010       \dim_to_fp:n {#1} * \l__coffin_sin_fp
28011       + \dim_to_fp:n {#2} * \l__coffin_cos_fp
28012     }
28013   }
28014 }

```

(End definition for `__coffin_rotate_vector:nnNN`.)

`__coffin_find_corner_maxima:N`
`__coffin_find_corner_maxima_aux:nn`

The idea here is to find the extremities of the content of the coffin. This is done by looking for the smallest values for the bottom and left corners, and the largest values for the top and right corners. The values start at the maximum dimensions so that the case where all are positive or all are negative works out correctly.

```

28015 \cs_new_protected:Npn \__coffin_find_corner_maxima:N #1
28016 {
28017   \dim_set:Nn \l__coffin_top_corner_dim { -\c_max_dim }
28018   \dim_set:Nn \l__coffin_right_corner_dim { -\c_max_dim }
28019   \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
28020   \dim_set:Nn \l__coffin_left_corner_dim { \c_max_dim }
28021   \prop_map_inline:Nn \l__coffin_corners_prop
28022   { \__coffin_find_corner_maxima_aux:nn ##2 }
28023 }
28024 \cs_new_protected:Npn \__coffin_find_corner_maxima_aux:nn #1#2
28025 {
28026   \dim_set:Nn \l__coffin_left_corner_dim
28027   { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
28028   \dim_set:Nn \l__coffin_right_corner_dim
28029   { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
28030   \dim_set:Nn \l__coffin_bottom_corner_dim
28031   { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
28032   \dim_set:Nn \l__coffin_top_corner_dim
28033   { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
28034 }

```

(End definition for `__coffin_find_corner_maxima:N` and `__coffin_find_corner_maxima_aux:nn`.)

`__coffin_find_bounding_shift:`
`__coffin_find_bounding_shift_aux:nn`

The approach to finding the shift for the bounding box is similar to that for the corners. However, there is only one value needed here and a fixed input property list, so things are a bit clearer.

```

28035 \cs_new_protected:Npn \__coffin_find_bounding_shift:
28036 {
28037   \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }

```

```

28038     \prop_map_inline:Nn \l__coffin_bounding_prop
28039     { \__coffin_find_bounding_shift_aux:nn ##2 }
28040   }
28041   \cs_new_protected:Npn \__coffin_find_bounding_shift_aux:nn #1#2
28042   {
28043     \dim_set:Nn \l__coffin_bounding_shift_dim
28044     { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
28045   }

```

(End definition for `__coffin_find_bounding_shift:` and `__coffin_find_bounding_shift_aux:nn`.)

`__coffin_shift_corner:Nnnn` `__coffin_shift_pole:Nnnnnn` Shifting the corners and poles of a coffin means subtracting the appropriate values from the x - and y -components. For the poles, this means that the direction vector is unchanged.

```

28046   \cs_new_protected:Npn \__coffin_shift_corner:Nnnn #1#2#3#4
28047   {
28048     \prop_put:Nnx \l__coffin_corners_prop {#2}
28049     {
28050       { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
28051       { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
28052     }
28053   }
28054   \cs_new_protected:Npn \__coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
28055   {
28056     \prop_put:Nnx \l__coffin_poles_prop {#2}
28057     {
28058       { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
28059       { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
28060       {#5} {#6}
28061     }
28062   }

```

(End definition for `__coffin_shift_corner:Nnnn` and `__coffin_shift_pole:Nnnnnn`.)

`\l__coffin_scale_x_fp` `\l__coffin_scale_y_fp` Storage for the scaling factors in x and y , respectively.

```

28063   \fp_new:N \l__coffin_scale_x_fp
28064   \fp_new:N \l__coffin_scale_y_fp

```

(End definition for `\l__coffin_scale_x_fp` and `\l__coffin_scale_y_fp`.)

`\l__coffin_scaled_total_height_dim` `\l__coffin_scaled_width_dim` When scaling, the values given have to be turned into absolute values.

```

28065   \dim_new:N \l__coffin_scaled_total_height_dim
28066   \dim_new:N \l__coffin_scaled_width_dim

```

(End definition for `\l__coffin_scaled_total_height_dim` and `\l__coffin_scaled_width_dim`.)

`\coffin_resize:Nnn` `\coffin_resize:cnn` `\coffin_gresize:Nnn` `\coffin_gresize:cnn` Resizing a coffin begins by setting up the user-friendly names for the dimensions of the coffin box. The new sizes are then turned into scale factor. This is the same operation as takes place for the underlying box, but that operation is grouped and so the same calculation is done here.

```

\__coffin_resize:NnnNN
28067   \cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
28068   {
28069     \__coffin_resize:NnnNN #1 {#2} {#3}
28070     \box_resize_to_wd_and_ht_plus_dp:Nnn

```

```

28071     \prop_set_eq:cN
28072   }
28073   \cs_generate_variant:Nn \coffin_resize:Nnn { c }
28074   \cs_new_protected:Npn \coffin_gresize:Nnn #1#2#3
28075   {
28076     \__coffin_resize:NnnNN #1 {#2} {#3}
28077     \box_gresize_to_wd_and_ht_plus_dp:Nnn
28078     \prop_gset_eq:cN
28079   }
28080   \cs_generate_variant:Nn \coffin_gresize:Nnn { c }
28081   \cs_new_protected:Npn \__coffin_resize:NnnNN #1#2#3#4#5
28082   {
28083     \fp_set:Nn \l__coffin_scale_x_fp
28084     { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
28085     \fp_set:Nn \l__coffin_scale_y_fp
28086     {
28087       \dim_to_fp:n {#3}
28088       / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
28089     }
28090     #4 #1 {#2} {#3}
28091     \__coffin_resize_common:NnnN #1 {#2} {#3} #5
28092   }

```

(End definition for `\coffin_resize:Nnn`, `\coffin_gresize:Nnn`, and `__coffin_resize:NnnNN`. These functions are documented on page 253.)

`__coffin_resize_common:NnnN` The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```

28093   \cs_new_protected:Npn \__coffin_resize_common:NnnN #1#2#3#4
28094   {
28095     \prop_set_eq:Nc \l__coffin_corners_prop
28096     { coffin ~ \__coffin_to_value:N #1 ~ corners }
28097     \prop_set_eq:Nc \l__coffin_poles_prop
28098     { coffin ~ \__coffin_to_value:N #1 ~ poles }
28099     \prop_map_inline:Nn \l__coffin_corners_prop
28100     { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }
28101     \prop_map_inline:Nn \l__coffin_poles_prop
28102     { \__coffin_scale_pole:Nnnnnn #1 {##1} ##2 }

```

Negative x -scaling values place the poles in the wrong location: this is corrected here.

```

28103     \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
28104     {
28105       \prop_map_inline:Nn \l__coffin_corners_prop
28106       { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
28107       \prop_map_inline:Nn \l__coffin_poles_prop
28108       { \__coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
28109     }
28110     #4 { coffin ~ \__coffin_to_value:N #1 ~ corners }
28111     \l__coffin_corners_prop
28112     #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
28113     \l__coffin_poles_prop
28114   }

```

(End definition for `__coffin_resize_common:NnnN`.)

`\coffin_scale:Nnn` For scaling, the opposite calculation is done to find the new dimensions for the coffin.
`\coffin_scale:cnn` Only the total height is needed, as this is the shift required for corners and poles. The
`\coffin_gscale:Nnn` scaling is done the T_EX way as this works properly with floating point values without
`\coffin_gscale:cnn` needing to use the fp module.
`\coffin_scale:NnnNN`

```

28115 \cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
28116 { \__coffin_scale:NnnNN #1 {#2} {#3} \box_scale:Nnn \prop_set_eq:cN }
28117 \cs_generate_variant:Nn \coffin_scale:Nnn { c }
28118 \cs_new_protected:Npn \coffin_gscale:Nnn #1#2#3
28119 { \__coffin_scale:NnnNN #1 {#2} {#3} \box_gscale:Nnn \prop_gset_eq:cN }
28120 \cs_generate_variant:Nn \coffin_gscale:Nnn { c }
28121 \cs_new_protected:Npn \__coffin_scale:NnnNN #1#2#3#4#5
28122 {
28123   \fp_set:Nn \l__coffin_scale_x_fp {#2}
28124   \fp_set:Nn \l__coffin_scale_y_fp {#3}
28125   #4 #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }
28126   \dim_set:Nn \l__coffin_internal_dim
28127     { \coffin_ht:N #1 + \coffin_dp:N #1 }
28128   \dim_set:Nn \l__coffin_scaled_total_height_dim
28129     { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_internal_dim }
28130   \dim_set:Nn \l__coffin_scaled_width_dim
28131     { -\fp_abs:n { \l__coffin_scale_x_fp } \coffin_wd:N #1 }
28132   \__coffin_resize_common:NnnN #1
28133     { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
28134   #5
28135 }

```

(End definition for `\coffin_scale:Nnn`, `\coffin_gscale:Nnn`, and `\coffin_scale:NnnNN`. These functions are documented on page 253.)

`__coffin_scale_vector:nnNN` This functions scales a vector from the origin using the pre-set scale factors in *x* and *y*. This is a much less complex operation than rotation, and as a result the code is a lot clearer.

```

28136 \cs_new_protected:Npn \__coffin_scale_vector:nnNN #1#2#3#4
28137 {
28138   \dim_set:Nn #3
28139     { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
28140   \dim_set:Nn #4
28141     { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
28142 }

```

(End definition for `__coffin_scale_vector:nnNN`.)

`__coffin_scale_corner:Nnnn` Scaling both corners and poles is a simple calculation using the preceding vector scaling.
`__coffin_scale_pole:Nnnnnn`

```

28143 \cs_new_protected:Npn \__coffin_scale_corner:Nnnn #1#2#3#4
28144 {
28145   \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
28146   \prop_put:Nnx \l__coffin_corners_prop {#2}
28147     { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
28148 }
28149 \cs_new_protected:Npn \__coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
28150 {
28151   \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
28152   \prop_put:Nnx \l__coffin_poles_prop {#2}
28153   {

```

```

28154         { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
28155         {#5} {#6}
28156     }
28157 }

```

(End definition for `__coffin_scale_corner:Nnnn` and `__coffin_scale_pole:Nnnnnn`.)

`__coffin_x_shift_corner:Nnnn`
`__coffin_x_shift_pole:Nnnnnn`

These functions correct for the x displacement that takes place with a negative horizontal scaling.

```

28158 \cs_new_protected:Npn \__coffin_x_shift_corner:Nnnn #1#2#3#4
28159 {
28160     \prop_put:Nnx \l__coffin_corners_prop {#2}
28161     {
28162         { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
28163     }
28164 }
28165 \cs_new_protected:Npn \__coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
28166 {
28167     \prop_put:Nnx \l__coffin_poles_prop {#2}
28168     {
28169         { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
28170         {#5} {#6}
28171     }
28172 }

```

(End definition for `__coffin_x_shift_corner:Nnnn` and `__coffin_x_shift_pole:Nnnnnn`.)

43.7 Aligning and typesetting of coffins

`\coffin_join:NnnNnnnn`
`\coffin_join:cnnNnnnn`
`\coffin_join:Nnncnnnn`
`\coffin_join:cnncnnnn`

This command joins two coffins, using a horizontal and vertical pole from each coffin and making an offset between the two. The result is stored as the as a third coffin, which has all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

```

\coffin_gjoin:NnnNnnnn
\coffin_gjoin:Nnncnnnn
\coffin_gjoin:cnncnnnn
\__coffin_join:NnnNnnnnN
28173 \cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
28174 {
28175     \__coffin_join:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
28176     \coffin_set_eq:NN
28177 }
28178 \cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cnnc }
28179 \cs_new_protected:Npn \coffin_gjoin:NnnNnnnn #1#2#3#4#5#6#7#8
28180 {
28181     \__coffin_join:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
28182     \coffin_gset_eq:NN
28183 }
28184 \cs_generate_variant:Nn \coffin_gjoin:NnnNnnnn { c , Nnnc , cnnc }
28185 \cs_new_protected:Npn \__coffin_join:NnnNnnnnN #1#2#3#4#5#6#7#8#9
28186 {
28187     \__coffin_align:NnnNnnnnN
28188     #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin

```

Correct the placement of the reference point. If the x -offset is negative then the reference point of the second box is to the left of that of the first, which is corrected using a kern. On the right side the first box might stick out, which would show up if it is wider than the sum of the x -offset and the width of the second box. So a second kern may be needed.

```

28189 \hbox_set:Nn \l__coffin_aligned_coffin
28190 {
28191   \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
28192   { \tex_kern:D -\l__coffin_offset_x_dim }
28193   \hbox_unpack:N \l__coffin_aligned_coffin
28194   \dim_set:Nn \l__coffin_internal_dim
28195   { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
28196   \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
28197   { \tex_kern:D -\l__coffin_internal_dim }
28198 }

```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```

28199 \__coffin_reset_structure:N \l__coffin_aligned_coffin
28200 \prop_clear:c
28201 {
28202   coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
28203   \c_space_tl corners
28204 }
28205 \__coffin_update_poles:N \l__coffin_aligned_coffin

```

The structures of the parent coffins are now transferred to the new coffin, which requires that the appropriate offsets are applied. That then depends on whether any shift was needed.

```

28206 \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
28207 {
28208   \__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
28209   \__coffin_offset_poles:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
28210   \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
28211   \__coffin_offset_corners:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
28212 }
28213 {
28214   \__coffin_offset_poles:Nnn #1 { Opt } { Opt }
28215   \__coffin_offset_poles:Nnn #4
28216   { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
28217   \__coffin_offset_corners:Nnn #1 { Opt } { Opt }
28218   \__coffin_offset_corners:Nnn #4
28219   { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
28220 }
28221 \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
28222 #9 #1 \l__coffin_aligned_coffin
28223 }

```

(End definition for `\coffin_join:NnnNnnnn`, `\coffin_gjoin:NnnNnnnn`, and `__coffin_join:NnnNnnnnN`. These functions are documented on page 253.)

```

\coffin_attach:NnnNnnnn
\coffin_attach:cnnNnnnn
\coffin_attach:Nnnncnnnn
\coffin_attach:cnncnnnn
\coffin_gattach:NnnNnnnn
\coffin_gattach:cnnNnnnn
\coffin_gattach:Nnnncnnnn
\coffin_gattach:cnncnnnn
\__coffin_attach:NnnNnnnnN
  \__coffin_attach_mark:NnnNnnnn

```

A more simple version of the above, as it simply uses the size of the first coffin for the new one. This means that the work here is rather simplified compared to the above code. The function used when marking a position is hear also as it is similar but without the structure updates.

```

28224 \cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
28225 {
28226   \__coffin_attach:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
28227   \coffin_set_eq:NN
28228 }

```

```

28229 \cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnnc }
28230 \cs_new_protected:Npn \coffin_gattach:NnnNnnnn #1#2#3#4#5#6#7#8
28231 {
28232   \__coffin_attach:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
28233   \coffin_gset_eq:NN
28234 }
28235 \cs_generate_variant:Nn \coffin_gattach:NnnNnnnn { c , Nnnc , cnnc }
28236 \cs_new_protected:Npn \__coffin_attach:NnnNnnnnN #1#2#3#4#5#6#7#8#9
28237 {
28238   \__coffin_align:NnnNnnnnN
28239   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
28240   \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
28241   \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
28242   \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
28243   \__coffin_reset_structure:N \l__coffin_aligned_coffin
28244   \prop_set_eq:cc
28245   {
28246     coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
28247     \c_space_tl corners
28248   }
28249   { coffin ~ \__coffin_to_value:N #1 ~ corners }
28250   \__coffin_update_poles:N \l__coffin_aligned_coffin
28251   \__coffin_offset_poles:Nnn #1 { Opt } { Opt }
28252   \__coffin_offset_poles:Nnn #4
28253   { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
28254   \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
28255   \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
28256 }
28257 \cs_new_protected:Npn \__coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
28258 {
28259   \__coffin_align:NnnNnnnnN
28260   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
28261   \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
28262   \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
28263   \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
28264   \box_set_eq:NN #1 \l__coffin_aligned_coffin
28265 }

```

(End definition for \coffin_attach:NnnNnnnn and others. These functions are documented on page 253.)

`__coffin_align:NnnNnnnnN` The internal function aligns the two coffins into a third one, but performs no corrections on the resulting coffin poles. The process begins by finding the points of intersection for the poles for each of the input coffins. Those for the first coffin are worked out after those for the second coffin, as this allows the ‘primed’ storage area to be used for the second coffin. The ‘real’ box offsets are then calculated, before using these to re-box the input coffins. The default poles are then set up, but the final result depends on how the bounding box is being handled.

```

28266 \cs_new_protected:Npn \__coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
28267 {
28268   \__coffin_calculate_intersection:Nnn #4 {#5} {#6}
28269   \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
28270   \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
28271   \__coffin_calculate_intersection:Nnn #1 {#2} {#3}

```

```

28272 \dim_set:Nn \l__coffin_offset_x_dim
28273 { \l__coffin_x_dim - \l__coffin_x_prime_dim + #7 }
28274 \dim_set:Nn \l__coffin_offset_y_dim
28275 { \l__coffin_y_dim - \l__coffin_y_prime_dim + #8 }
28276 \hbox_set:Nn \l__coffin_aligned_internal_coffin
28277 {
28278   \box_use:N #1
28279   \tex_kern:D -\box_wd:N #1
28280   \tex_kern:D \l__coffin_offset_x_dim
28281   \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #4 }
28282 }
28283 \coffin_set_eq:NN #9 \l__coffin_aligned_internal_coffin
28284 }

```

(End definition for __coffin_align:NnnNnnnnN.)

__coffin_offset_poles:Nnn
 __coffin_offset_pole:Nnnnnnn

Transferring structures from one coffin to another requires that the positions are updated by the offset between the two coffins. This is done by mapping to the property list of the source coffins, moving as appropriate and saving to the new coffin data structures. The test for a - means that the structures from the parent coffins are uniquely labelled and do not depend on the order of alignment. The pay off for this is that - should not be used in coffin pole or handle names, and that multiple alignments do not result in a whole set of values.

```

28285 \cs_new_protected:Npn \__coffin_offset_poles:Nnn #1#2#3
28286 {
28287   \prop_map_inline:cn { coffin ~ \__coffin_to_value:N #1 ~ poles }
28288   { \__coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {#2} {#3} }
28289 }
28290 \cs_new_protected:Npn \__coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
28291 {
28292   \dim_set:Nn \l__coffin_x_dim { #3 + #7 }
28293   \dim_set:Nn \l__coffin_y_dim { #4 + #8 }
28294   \tl_if_in:nnTF {#2} { - }
28295   { \tl_set:Nn \l__coffin_internal_tl { {#2} } }
28296   { \tl_set:Nn \l__coffin_internal_tl { { #1 - #2 } } }
28297   \exp_last_unbraced:NNo \__coffin_set_pole:Nnx \l__coffin_aligned_coffin
28298   { \l__coffin_internal_tl }
28299   {
28300     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
28301     {#5} {#6}
28302   }
28303 }

```

(End definition for __coffin_offset_poles:Nnn and __coffin_offset_pole:Nnnnnnn.)

__coffin_offset_corners:Nnn
 __coffin_offset_corner:Nnnnn

Saving the offset corners of a coffin is very similar, except that there is no need to worry about naming: every corner can be saved here as order is unimportant.

```

28304 \cs_new_protected:Npn \__coffin_offset_corners:Nnn #1#2#3
28305 {
28306   \prop_map_inline:cn { coffin ~ \__coffin_to_value:N #1 ~ corners }
28307   { \__coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
28308 }
28309 \cs_new_protected:Npn \__coffin_offset_corner:Nnnnn #1#2#3#4#5#6
28310 {

```



```

28311 \prop_put:cnx
28312 {
28313   coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
28314   \c_space_tl corners
28315 }
28316 { #1 - #2 }
28317 {
28318   { \dim_eval:n { #3 + #5 } }
28319   { \dim_eval:n { #4 + #6 } }
28320 }
28321 }

```

(End definition for __coffin_offset_corners:Nnn and __coffin_offset_corner:Nnnnn.)

```

\__coffin_update_vertical_poles:NNN
\__coffin_update_T:nnnnnnnnN
\__coffin_update_B:nnnnnnnnN

```

The T and B poles need to be recalculated after alignment. These functions find the larger absolute value for the poles, but this is of course only logical when the poles are horizontal.

```

28322 \cs_new_protected:Npn \__coffin_update_vertical_poles:NNN #1#2#3
28323 {
28324   \__coffin_get_pole:NnN #3 { #1 -T } \l__coffin_pole_a_tl
28325   \__coffin_get_pole:NnN #3 { #2 -T } \l__coffin_pole_b_tl
28326   \exp_last_two_unbraced:Noo \__coffin_update_T:nnnnnnnnN
28327   \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
28328   \__coffin_get_pole:NnN #3 { #1 -B } \l__coffin_pole_a_tl
28329   \__coffin_get_pole:NnN #3 { #2 -B } \l__coffin_pole_b_tl
28330   \exp_last_two_unbraced:Noo \__coffin_update_B:nnnnnnnnN
28331   \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
28332 }
28333 \cs_new_protected:Npn \__coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
28334 {
28335   \dim_compare:nNnTF {#2} < {#6}
28336   {
28337     \__coffin_set_pole:Nnx #9 { T }
28338     { { Opt } {#6} { 1000pt } { Opt } }
28339   }
28340   {
28341     \__coffin_set_pole:Nnx #9 { T }
28342     { { Opt } {#2} { 1000pt } { Opt } }
28343   }
28344 }
28345 \cs_new_protected:Npn \__coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
28346 {
28347   \dim_compare:nNnTF {#2} < {#6}
28348   {
28349     \__coffin_set_pole:Nnx #9 { B }
28350     { { Opt } {#2} { 1000pt } { Opt } }
28351   }
28352   {
28353     \__coffin_set_pole:Nnx #9 { B }
28354     { { Opt } {#6} { 1000pt } { Opt } }
28355   }
28356 }

```

(End definition for __coffin_update_vertical_poles:NNN, __coffin_update_T:nnnnnnnnN, and __coffin_update_B:nnnnnnnnN.)

`\c__coffin_empty_coffin` An empty-but-horizontal coffin.

```

28357 \coffin_new:N \c__coffin_empty_coffin
28358 \tex_setbox:D \c__coffin_empty_coffin = \tex_hbox:D { }

(End definition for \c__coffin_empty_coffin.)

```

`\coffin_typeset:Nnnnn` Typesetting a coffin means aligning it with the current position, which is done using a coffin with no content at all. As well as aligning to the empty coffin, there is also a need to leave vertical mode, if necessary.

`\coffin_typeset:cnnnn`

```

28359 \cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
28360 {
28361   \mode_leave_vertical:
28362   \__coffin_align:NnnNnnnnN \c__coffin_empty_coffin { H } { 1 }
28363   #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
28364   \box_use_drop:N \l__coffin_aligned_coffin
28365 }
28366 \cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }

(End definition for \coffin_typeset:Nnnnn. This function is documented on page 254.)

```

43.8 Coffin diagnostics

`\l__coffin_display_coffin` Used for printing coffins with data structures attached.

```

\l__coffin_display_coord_coffin 28367 \coffin_new:N \l__coffin_display_coffin
\l__coffin_display_pole_coffin 28368 \coffin_new:N \l__coffin_display_coord_coffin
28369 \coffin_new:N \l__coffin_display_pole_coffin

```

(End definition for `\l__coffin_display_coffin`, `\l__coffin_display_coord_coffin`, and `\l__coffin_display_pole_coffin`.)

`\l__coffin_display_handles_prop` This property list is used to print coffin handles at suitable positions. The offsets are expressed as multiples of the basic offset value, which therefore acts as a scale-factor.

```

28370 \prop_new:N \l__coffin_display_handles_prop
28371 \prop_put:Nnn \l__coffin_display_handles_prop { tl }
28372 { { b } { r } { -1 } { 1 } }
28373 \prop_put:Nnn \l__coffin_display_handles_prop { thc }
28374 { { b } { hc } { 0 } { 1 } }
28375 \prop_put:Nnn \l__coffin_display_handles_prop { tr }
28376 { { b } { l } { 1 } { 1 } }
28377 \prop_put:Nnn \l__coffin_display_handles_prop { vcl }
28378 { { vc } { r } { -1 } { 0 } }
28379 \prop_put:Nnn \l__coffin_display_handles_prop { vhc }
28380 { { vc } { hc } { 0 } { 0 } }
28381 \prop_put:Nnn \l__coffin_display_handles_prop { vcr }
28382 { { vc } { l } { 1 } { 0 } }
28383 \prop_put:Nnn \l__coffin_display_handles_prop { bl }
28384 { { t } { r } { -1 } { -1 } }
28385 \prop_put:Nnn \l__coffin_display_handles_prop { bhc }
28386 { { t } { hc } { 0 } { -1 } }
28387 \prop_put:Nnn \l__coffin_display_handles_prop { br }
28388 { { t } { l } { 1 } { -1 } }
28389 \prop_put:Nnn \l__coffin_display_handles_prop { Tl }
28390 { { t } { r } { -1 } { -1 } }
28391 \prop_put:Nnn \l__coffin_display_handles_prop { Thc }

```

```

28392 { { t } { hc } { 0 } { -1 } }
28393 \prop_put:Nnn \l__coffin_display_handles_prop { Tr }
28394 { { t } { l } { 1 } { -1 } }
28395 \prop_put:Nnn \l__coffin_display_handles_prop { Hl }
28396 { { vc } { r } { -1 } { 1 } }
28397 \prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
28398 { { vc } { hc } { 0 } { 1 } }
28399 \prop_put:Nnn \l__coffin_display_handles_prop { Hr }
28400 { { vc } { l } { 1 } { 1 } }
28401 \prop_put:Nnn \l__coffin_display_handles_prop { Bl }
28402 { { b } { r } { -1 } { -1 } }
28403 \prop_put:Nnn \l__coffin_display_handles_prop { Bhc }
28404 { { b } { hc } { 0 } { -1 } }
28405 \prop_put:Nnn \l__coffin_display_handles_prop { Br }
28406 { { b } { l } { 1 } { -1 } }

```

(End definition for \l__coffin_display_handles_prop.)

`\l__coffin_display_offset_dim` The standard offset for the label from the handle position when displaying handles.

```

28407 \dim_new:N \l__coffin_display_offset_dim
28408 \dim_set:Nn \l__coffin_display_offset_dim { 2pt }

```

(End definition for \l__coffin_display_offset_dim.)

`\l__coffin_display_x_dim` As the intersections of poles have to be calculated to find which ones to print, there is
`\l__coffin_display_y_dim` a need to avoid repetition. This is done by saving the intersection into two dedicated
values.

```

28409 \dim_new:N \l__coffin_display_x_dim
28410 \dim_new:N \l__coffin_display_y_dim

```

(End definition for \l__coffin_display_x_dim and \l__coffin_display_y_dim.)

`\l__coffin_display_poles_prop` A property list for printing poles: various things need to be deleted from this to get a
“nice” output.

```

28411 \prop_new:N \l__coffin_display_poles_prop

```

(End definition for \l__coffin_display_poles_prop.)

`\l__coffin_display_font_tl` Stores the settings used to print coffin data: this keeps things flexible.

```

28412 \tl_new:N \l__coffin_display_font_tl
28413 \cs_if_exist:NTF \AtBeginDocument
28414 { \AtBeginDocument }
28415 { \use:n }
28416 {
28417   \tl_set:Nx \l__coffin_display_font_tl
28418   {
28419     \cs_if_exist:NT \sffamily { \exp_not:N \sffamily }
28420     \cs_if_exist:NT \tiny { \exp_not:N \tiny }
28421   }
28422 }

```

(End definition for \l__coffin_display_font_tl.)

`__coffin_color:n` Calls `\color`, and otherwise does nothing if `\color` is not defined.

```

28423 \cs_if_exist:NTF \AtBeginDocument
28424 { \AtBeginDocument }
28425 { \use:n }
28426 {
28427   \cs_new_protected:Npx \__coffin_color:n #1
28428   {
28429     \cs_if_exist:NTF \color_select:n
28430     { \color_select:n {#1} }
28431     {
28432       \cs_if_exist:NT \color
28433       { \exp_not:N \color {#1} }
28434     }
28435   }
28436 }

```

(End definition for `__coffin_color:n`.)

`__coffin_rule:nn` Abstract out creation of rules here until there is a higher-level interface.

```

28437 \cs_new_protected:Npx \__coffin_rule:nn #1#2
28438 {
28439   \cs_if_exist:NTF \rule
28440   { \exp_not:N \rule {#1} {#2} }
28441   { \hbox:n { \tex_vrule:D width #1 height #2 \scan_stop: } }
28442 }

```

(End definition for `__coffin_rule:nn`.)

`\coffin_mark_handle:Nnnn` Marking a single handle is relatively easy. The standard attachment function is used, meaning that there are two calculations for the location. However, this is likely to be okay given the load expected. Contrast with the more optimised version for showing all handles which comes next.

`\coffin_mark_handle:cnnn`
`__coffin_mark_handle_aux:nnnnNnn`

```

28443 \cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
28444 {
28445   \hcoffin_set:Nn \l__coffin_display_pole_coffin
28446   {
28447     \__coffin_color:n {#4}
28448     \__coffin_rule:nn { 1pt } { 1pt }
28449   }
28450   \__coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
28451   \l__coffin_display_pole_coffin { hc } { vc } { Opt } { Opt }
28452   \hcoffin_set:Nn \l__coffin_display_coord_coffin
28453   {
28454     \__coffin_color:n {#4}
28455     \l__coffin_display_font_tl
28456     ( \tl_to_str:n { #2 , #3 } )
28457   }
28458   \prop_get:NnN \l__coffin_display_handles_prop
28459   { #2 #3 } \l__coffin_internal_tl
28460   \quark_if_no_value:NTF \l__coffin_internal_tl
28461   {
28462     \prop_get:NnN \l__coffin_display_handles_prop
28463     { #3 #2 } \l__coffin_internal_tl
28464     \quark_if_no_value:NTF \l__coffin_internal_tl

```

```

28465     {
28466         \__coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
28467         \l__coffin_display_coord_coffin { l } { vc }
28468         { 1pt } { 0pt }
28469     }
28470     {
28471         \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
28472         \l__coffin_internal_tl #1 {#2} {#3}
28473     }
28474 }
28475 {
28476     \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
28477     \l__coffin_internal_tl #1 {#2} {#3}
28478 }
28479 }
28480 \cs_new_protected:Npn \__coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
28481 {
28482     \__coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
28483     \l__coffin_display_coord_coffin {#1} {#2}
28484     { #3 \l__coffin_display_offset_dim }
28485     { #4 \l__coffin_display_offset_dim }
28486 }
28487 \cs_generate_variant:Nn \coffin_mark_handle:Nnnn { c }

```

(End definition for \coffin_mark_handle:Nnnn and __coffin_mark_handle_aux:nnnnNnn. This function is documented on page 254.)

\coffin_display_handles:Nn
\coffin_display_handles:cn
 __coffin_display_handles_aux:nnnnnn
 __coffin_display_handles_aux:nnnn
 __coffin_display_attach:Nnnnn

Printing the poles starts by removing any duplicates, for which the H poles is used as the definitive version for the baseline and bottom. Two loops are then used to find the combinations of handles for all of these poles. This is done such that poles are removed during the loops to avoid duplication.

```

28488 \cs_new_protected:Npn \coffin_display_handles:Nn #1#2
28489 {
28490     \hcoffin_set:Nn \l__coffin_display_pole_coffin
28491     {
28492         \__coffin_color:n {#2}
28493         \__coffin_rule:nn { 1pt } { 1pt }
28494     }
28495     \prop_set_eq:Nc \l__coffin_display_poles_prop
28496     { coffin ~ \__coffin_to_value:N #1 ~ poles }
28497     \__coffin_get_pole:NnN #1 { H } \l__coffin_pole_a_tl
28498     \__coffin_get_pole:NnN #1 { T } \l__coffin_pole_b_tl
28499     \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
28500     { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
28501     \__coffin_get_pole:NnN #1 { B } \l__coffin_pole_b_tl
28502     \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
28503     { \prop_remove:Nn \l__coffin_display_poles_prop { B } }
28504     \coffin_set_eq:NN \l__coffin_display_coffin #1
28505     \prop_map_inline:Nn \l__coffin_display_poles_prop
28506     {
28507         \prop_remove:Nn \l__coffin_display_poles_prop {##1}
28508         \__coffin_display_handles_aux:nnnnnn {##1} ##2 {#2}
28509     }
28510     \box_use_drop:N \l__coffin_display_coffin

```

```
28511 }
```

For each pole there is a check for an intersection, which here does not give an error if none is found. The successful values are stored and used to align the pole coffin with the main coffin for output. The positions are recovered from the preset list if available.

```
28512 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnnn #1#2#3#4#5#6
28513 {
28514   \prop_map_inline:Nn \l__coffin_display_poles_prop
28515   {
28516     \bool_set_false:N \l__coffin_error_bool
28517     \__coffin_calculate_intersection:nnnnnnn {#2} {#3} {#4} {#5} ##2
28518     \bool_if:NF \l__coffin_error_bool
28519     {
28520       \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
28521       \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
28522       \__coffin_display_attach:Nnnnn
28523       \l__coffin_display_pole_coffin { hc } { vc }
28524       { Opt } { Opt }
28525       \hcoffin_set:Nn \l__coffin_display_coord_coffin
28526       {
28527         \__coffin_color:n {#6}
28528         \l__coffin_display_font_tl
28529         ( \tl_to_str:n { #1 , ##1 } )
28530       }
28531       \prop_get:NnN \l__coffin_display_handles_prop
28532       { #1 ##1 } \l__coffin_internal_tl
28533       \quark_if_no_value:NTF \l__coffin_internal_tl
28534       {
28535         \prop_get:NnN \l__coffin_display_handles_prop
28536         { ##1 #1 } \l__coffin_internal_tl
28537         \quark_if_no_value:NTF \l__coffin_internal_tl
28538         {
28539           \__coffin_display_attach:Nnnnn
28540           \l__coffin_display_coord_coffin { l } { vc }
28541           { 1pt } { Opt }
28542         }
28543         {
28544           \exp_last_unbraced:No
28545           \__coffin_display_handles_aux:nnnn
28546           \l__coffin_internal_tl
28547         }
28548       }
28549       {
28550         \exp_last_unbraced:No \__coffin_display_handles_aux:nnnn
28551         \l__coffin_internal_tl
28552       }
28553     }
28554   }
28555 }
28556 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
28557 {
28558   \__coffin_display_attach:Nnnnn
28559   \l__coffin_display_coord_coffin {#1} {#2}
28560   { #3 \l__coffin_display_offset_dim }
```

```

28561     { #4 \l__coffin_display_offset_dim }
28562   }
28563 \cs_generate_variant:Nn \coffin_display_handles:Nn { c }

28564 \cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5
28565 {
28566   \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
28567   \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
28568   \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
28569   \dim_set:Nn \l__coffin_offset_x_dim
28570     { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
28571   \dim_set:Nn \l__coffin_offset_y_dim
28572     { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
28573   \hbox_set:Nn \l__coffin_aligned_coffin
28574     {
28575     \box_use:N \l__coffin_display_coffin
28576     \tex_kern:D -\box_wd:N \l__coffin_display_coffin
28577     \tex_kern:D \l__coffin_offset_x_dim
28578     \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
28579   }
28580   \box_set_ht:Nn \l__coffin_aligned_coffin
28581     { \box_ht:N \l__coffin_display_coffin }
28582   \box_set_dp:Nn \l__coffin_aligned_coffin
28583     { \box_dp:N \l__coffin_display_coffin }
28584   \box_set_wd:Nn \l__coffin_aligned_coffin
28585     { \box_wd:N \l__coffin_display_coffin }
28586   \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
28587 }

```

(End definition for \coffin_display_handles:Nn and others. This function is documented on page 254.)

```

\coffin_show_structure:N
\coffin_show_structure:c
\coffin_log_structure:N
\coffin_log_structure:c
\__coffin_show_structure:NN

```

For showing the various internal structures attached to a coffin in a way that keeps things relatively readable. If there is no apparent structure then the code complains.

```

28588 \cs_new_protected:Npn \coffin_show_structure:N
28589   { \__coffin_show_structure:NN \msg_show:nnxxxx }
28590 \cs_generate_variant:Nn \coffin_show_structure:N { c }
28591 \cs_new_protected:Npn \coffin_log_structure:N
28592   { \__coffin_show_structure:NN \msg_log:nnxxxx }
28593 \cs_generate_variant:Nn \coffin_log_structure:N { c }
28594 \cs_new_protected:Npn \__coffin_show_structure:NN #1#2
28595 {
28596   \__coffin_if_exist:NT #2
28597   {
28598     #1 { LaTeX / kernel } { show-coffin }
28599     { \token_to_str:N #2 }
28600     {
28601       \iow_newline: >~ ht ~~~ \dim_eval:n { \coffin_ht:N #2 }
28602       \iow_newline: >~ dp ~~~ \dim_eval:n { \coffin_dp:N #2 }
28603       \iow_newline: >~ wd ~~~ \dim_eval:n { \coffin_wd:N #2 }
28604     }
28605   }

```

```

28606         \prop_map_function:cN
28607         { coffin ~ \__coffin_to_value:N #2 ~ poles }
28608         \msg_show_item_unbraced:nn
28609     }
28610 { }
28611 }
28612 }

```

(End definition for `\coffin_show_structure:N`, `\coffin_log_structure:N`, and `__coffin_show_structure:NN`. These functions are documented on page 254.)

43.9 Messages

```

28613 \__kernel_msg_new:nnnn { kernel } { no-pole-intersection }
28614 { No~intersection~between~coffin~poles. }
28615 {
28616     LaTeX~was~asked~to~find~the~intersection~between~two~poles,~
28617     but~they~do~not~have~a~unique~meeting~point:~
28618     the~value~(0pt,~0pt)~will~be~used.
28619 }
28620 \__kernel_msg_new:nnnn { kernel } { unknown-coffin }
28621 { Unknown~coffin~'#1'. }
28622 { The~coffin~'#1'~was~never~defined. }
28623 \__kernel_msg_new:nnnn { kernel } { unknown-coffin-pole }
28624 { Pole~'#1'~unknown~for~coffin~'#2'. }
28625 {
28626     LaTeX~was~asked~to~find~a~typesetting~pole~for~a~coffin,~
28627     but~either~the~coffin~does~not~exist~or~the~pole~name~is~wrong.
28628 }
28629 \__kernel_msg_new:nnn { kernel } { show-coffin }
28630 {
28631     Size~of~coffin~#1 : #2 \\
28632     Poles~of~coffin~#1 : #3 .
28633 }
28634 </package>

```

44 l3color-base Implementation

```

28635 <*package>
28636 <@@=color>

```

`\l__color_current_tl` The color currently active for foreground (text, *etc.*) material. This is stored in the form of a color model followed by one or more values. There are four pre-defined models, three of which take numerical values in the range [0, 1]:

- `gray` `<gray>` Grayscale color with the `<gray>` value running from 0 (fully black) to 1 (fully white)
- `cmk` `<cyan>` `<magenta>` `<yellow>` `<black>`
- `rgb` `<red>` `<green>` `<blue>`

Notice that the value are separated by spaces. There is a fourth pre-defined model using a string value and a numerical one:

- **spot** $\langle name \rangle$ $\langle tint \rangle$ A pre-defined spot color, where the $\langle name \rangle$ should be a pre-defined string color name and the $\langle tint \rangle$ should be in the range $[0, 1]$.

Additional models may be created to allow mixing of spot colors. The number of data entries these require will depend on the number of colors to be mixed.

T_EXhackers note: The content of `\l__color_current_tl` comprises two brace groups, the first containing the color model and the second containing the value(s) applicable in that model.

(End definition for \l__color_current_tl.)

\color_group_begin: Grouping for color is the same as using the basic `\group_begin:` and `\group_end:` functions. However, for semantic reasons, they are renamed here.

```
28637 \cs_new_eq:NN \color_group_begin: \group_begin:
28638 \cs_new_eq:NN \color_group_end: \group_end:
```

(End definition for \color_group_begin: and \color_group_end:.. These functions are documented on page 256.)

\color_ensure_current: A driver-independent wrapper for setting the foreground color to the current color “now”.

```
28639 \cs_new_protected:Npn \color_ensure_current:
28640 {
28641   \__color_backend_pickup:N \l__color_current_tl
28642   \__color_select:N \l__color_current_tl
28643 }
```

(End definition for \color_ensure_current:.. This function is documented on page 256.)

\s__color_stop Internal scan marks.

```
28644 \scan_new:N \s__color_stop
```

(End definition for \s__color_stop.)

__color_select:N Take an internal color specification and pass it to the driver. This code is needed to ensure the current color but will also be used by the higher-level experimental material.

__color_select:nn

```
28645 \cs_new_protected:Npn \__color_select:N #1
28646 { \exp_after:wN \__color_select:nn #1 }
28647 \cs_new_protected:Npn \__color_select:nn #1#2
28648 { \use:c { __color_backend_select_ #1 :n } {#2} }
```

(End definition for __color_select:N and __color_select:nn.)

\l__color_current_tl The current color, with the model and

```
28649 \tl_new:N \l__color_current_tl
28650 \tl_set:Nn \l__color_current_tl { { gray } { 0 } }
```

(End definition for \l__color_current_tl.)

```
28651 \</package>
```

45 l3luatex implementation

28652 $\langle *package \rangle$

45.1 Breaking out to Lua

28653 $\langle *tex \rangle$

28654 $\langle @@=lua \rangle$

$\backslash_lua_escape:n$ Copies of primitives.
 $\backslash_lua_now:n$ 28655 $\backslash cs_new_eq:NN \backslash_lua_escape:n \backslash tex_luaescapestring:D$
 $\backslash_lua_shipout:n$ 28656 $\backslash cs_new_eq:NN \backslash_lua_now:n \backslash tex_directlua:D$
28657 $\backslash cs_new_eq:NN \backslash_lua_shipout:n \backslash tex_latelua:D$

(End definition for $\backslash_lua_escape:n$, $\backslash_lua_now:n$, and $\backslash_lua_shipout:n$.)

These functions are set up in l3str for bootstrapping: we want to replace them with a “proper” version at this stage, so clean up.

28658 $\backslash cs_undefine:N \backslash lua_escape:e$

28659 $\backslash cs_undefine:N \backslash lua_now:e$

$\backslash lua_now:n$ Wrappers around the primitives. As with engines other than LuaTeX these have to be
 $\backslash lua_now:e$ macros, we give them the same status in all cases. When LuaTeX is not in use, simply
 $\backslash lua_shipout_e:n$ give an error message/
 $\backslash lua_shipout:n$

$\backslash lua_escape:n$ 28660 $\backslash cs_new:Npn \backslash lua_now:e \#1 \{ \backslash_lua_now:n \{ \#1 \} \}$
 $\backslash lua_escape:e$ 28661 $\backslash cs_new:Npn \backslash lua_now:n \#1 \{ \backslash lua_now:e \{ \exp_not:n \{ \#1 \} \} \}$
28662 $\backslash cs_new_protected:Npn \backslash lua_shipout_e:n \#1 \{ \backslash_lua_shipout:n \{ \#1 \} \}$
28663 $\backslash cs_new_protected:Npn \backslash lua_shipout:n \#1$
28664 $\{ \backslash lua_shipout_e:n \{ \exp_not:n \{ \#1 \} \} \}$
28665 $\backslash cs_new:Npn \backslash lua_escape:e \#1 \{ \backslash_lua_escape:n \{ \#1 \} \}$
28666 $\backslash cs_new:Npn \backslash lua_escape:n \#1 \{ \backslash lua_escape:e \{ \exp_not:n \{ \#1 \} \} \}$
28667 $\backslash sys_if_engine_luatex:F$
28668 $\{$
28669 $\backslash clist_map_inline:nn$
28670 $\{$
28671 $\backslash lua_escape:n , \backslash lua_escape:e ,$
28672 $\backslash lua_now:n , \backslash lua_now:e$
28673 $\}$
28674 $\{$
28675 $\backslash cs_set:Npn \#1 \##1$
28676 $\{$
28677 $\backslash_kernel_msg_expandable_error:nnn$
28678 $\{ kernel \} \{ luatex-required \} \{ \#1 \}$
28679 $\}$
28680 $\}$
28681 $\backslash clist_map_inline:nn$
28682 $\{ \backslash lua_shipout_e:n , \backslash lua_shipout:n \}$
28683 $\{$
28684 $\backslash cs_set_protected:Npn \#1 \##1$
28685 $\{$
28686 $\backslash_kernel_msg_error:nnn$
28687 $\{ kernel \} \{ luatex-required \} \{ \#1 \}$
28688 $\}$
28689 $\}$
28690 $\}$

(End definition for $\backslash lua_now:n$ and others. These functions are documented on page 257.)

45.2 Messages

```
28691 \__kernel_msg_new:nnnn { kernel } { luatex-required }
28692 { LuaTeX~engine~not~in~use!~Ignoring~#1. }
28693 {
28694     The~feature~you~are~using~is~only~available~
28695     with~the~LuaTeX~engine.~LaTeX3~ignored~'~#1'.
28696 }
28697 </tex>
```

45.3 Lua functions for internal use

```
28698 <*lua>
```

Most of the emulation of pdfTeX here is based heavily on Heiko Oberdiek's pdfTeX-cmds package.

13kernel Create a table for the kernel's own use.

```
28699 13kernel = 13kernel or { }
```

(End definition for 13kernel. This function is documented on page 258.)

Local copies of global tables.

```
28700 local io      = io
28701 local kpse    = kpse
28702 local lfs     = lfs
28703 local math    = math
28704 local md5     = md5
28705 local os      = os
28706 local string  = string
28707 local tex     = tex
28708 local texio   = texio
28709 local tonumber = tonumber
28710 local unicode = unicode
```

Local copies of standard functions.

```
28711 local abs      = math.abs
28712 local byte     = string.byte
28713 local floor    = math.floor
28714 local format   = string.format
28715 local gsub     = string.gsub
28716 local lfs_attr = lfs.attributes
28717 local md5_sum  = md5.sum
28718 local open     = io.open
28719 local os_clock = os.clock
28720 local os_date  = os.date
28721 local os_exec  = os.execute
28722 local setcatcode = tex.setcatcode
28723 local sprint   = tex.sprint
28724 local cprint   = tex.cprint
28725 local write    = tex.write
28726 local write_nl = texio.write_nl
```

Newer ConTeXt releases replace the unicode library by utf and since Lua 5.3 we can even use the Lua standard utf8 library.

```
28727 local utf8_char = (utf8 and utf8.char) or (utf and utf.char) or unicode.utf8.char
```

Deal with ConT_EXt: doesn't use kpse library.

```
28728 local kpse_find = (resolvers and resolvers.findfile) or kpse.find_file
```

escapehex An internal auxiliary to convert a string to the matching hex escape. This works on a byte basis: extension to handled UTF-8 input is covered in pdf_{tex}cmds but is not currently required here.

```
28729 local function escapehex(str)
28730   write((gsub(str, ".",
28731     function (ch) return format("%02X", byte(ch)) end)))
28732 end
```

(End definition for escapehex.)

l3kernel.charcat Creating arbitrary chars using tex.cprint.

```
28733 local charcat
28734 function charcat(charcode, catcode)
28735   cprint(catcode, utf8_char(charcode))
28736 end
28737 l3kernel.charcat = charcat
```

(End definition for l3kernel.charcat. This function is documented on page 258.)

l3kernel.elapsedtime Simple timing set up: give the result from the system clock in scaled seconds.

l3kernel.resettimer

```
28738 local base_time = 0
28739 local function elapsedtime()
28740   local val = (os_clock() - base_time) * 65536 + 0.5
28741   if val > 2147483647 then
28742     val = 2147483647
28743   end
28744   write(format("%d", floor(val)))
28745 end
28746 l3kernel.elapsedtime = elapsedtime
28747 local function resettimer()
28748   base_time = os_clock()
28749 end
28750 l3kernel.resettimer = resettimer
```

(End definition for l3kernel.elapsedtime and l3kernel.resettimer. These functions are documented on page 258.)

l3kernel.filedump Similar comments here to the next function: read the file in binary mode to avoid any line-end weirdness.

```
28751 local function filedump(name, offset, length)
28752   local file = kpse_find(name, "tex", true)
28753   if file then
28754     local length = tonumber(length) or lfs_attr(file, "size")
28755     local offset = tonumber(offset) or 0
28756     local f = open(file, "rb")
28757     if f then
28758       if offset > 0 then
28759         f:seek("set", offset)
28760       end
28761       local data = f:read(length)
28762       escapehex(data)

```

```

28763         f:close()
28764     end
28765 end
28766 end
28767 l3kernel.filedump = filedump

```

(End definition for l3kernel.filedump. This function is documented on page 258.)

l3kernel.filemdfivesum Read an entire file and hash it: the hash function itself is a built-in. As Lua is byte-based there is no work needed here in terms of UTF-8 (see `pdfdoccmds` and how it handles strings that have passed through LuaTeX). The file is read in binary mode so that no line ending normalisation occurs.

```

28768 local function filemdfivesum(name)
28769     local file = kpse_find(name, "tex", true)
28770     if file then
28771         local f = open(file, "rb")
28772         if f then
28773             local data = f:read("*a")
28774             escapehex(md5_sum(data))
28775             f:close()
28776         end
28777     end
28778 end
28779 l3kernel.filemdfivesum = filemdfivesum

```

(End definition for l3kernel.filemdfivesum. This function is documented on page 258.)

l3kernel.filemoddate See procedure `makepdftime` in `utils.c` of pdfTeX.

```

28780 local function filemoddate(name)
28781     local file = kpse_find(name, "tex", true)
28782     if file then
28783         local date = lfs_attr(file, "modification")
28784         if date then
28785             local d = os_date("!*t", date)
28786             if d.sec >= 60 then
28787                 d.sec = 59
28788             end
28789             local u = os_date("!*t", date)
28790             local off = 60 * (d.hour - u.hour) + d.min - u.min
28791             if d.year ~= u.year then
28792                 if d.year > u.year then
28793                     off = off + 1440
28794                 else
28795                     off = off - 1440
28796                 end
28797             elseif d.yday ~= u.yday then
28798                 if d.yday > u.yday then
28799                     off = off + 1440
28800                 else
28801                     off = off - 1440
28802                 end
28803             end
28804             local timezone
28805             if off == 0 then

```

```

28806         timezone = "Z"
28807     else
28808         local hours = floor(off / 60)
28809         local mins = abs(off - hours * 60)
28810         timezone = format("%+03d", hours)
28811         .. "" .. format("%02d", mins) .. ""
28812     end
28813     write("D:"
28814         .. format("%04d", d.year)
28815         .. format("%02d", d.month)
28816         .. format("%02d", d.day)
28817         .. format("%02d", d.hour)
28818         .. format("%02d", d.min)
28819         .. format("%02d", d.sec)
28820         .. timezone)
28821     end
28822 end
28823 end
28824 l3kernel.filemoddate = filemoddate

```

(End definition for l3kernel.filemoddate. This function is documented on page 258.)

l3kernel.filesize A simple disk lookup.

```

28825 local function filesize(name)
28826     local file = kpse_find(name, "tex", true)
28827     if file then
28828         local size = lfs_attr(file, "size")
28829         if size then
28830             write(size)
28831         end
28832     end
28833 end
28834 l3kernel.filesize = filesize

```

(End definition for l3kernel.filesize. This function is documented on page 258.)

l3kernel.strcmp String comparison which gives the same results as pdfTeX's `\pdfstrcmp`, although the ordering should likely not be relied upon!

```

28835 local function strcmp(A, B)
28836     if A == B then
28837         write("0")
28838     elseif A < B then
28839         write("-1")
28840     else
28841         write("1")
28842     end
28843 end
28844 l3kernel.strcmp = strcmp

```

(End definition for l3kernel.strcmp. This function is documented on page 258.)

l3kernel.shellescape Replicating the pdfTeX log interaction for shell escape.

```

28845 local function shellescape(cmd)
28846     local status,msg = os_exec(cmd)
28847     if status == nil then

```

```

28848     write_nl("log","runsystem(" .. cmd .. ")...(" .. msg .. ")\n")
28849   elseif status == 0 then
28850     write_nl("log","runsystem(" .. cmd .. ")...executed\n")
28851   else
28852     write_nl("log","runsystem(" .. cmd .. ")...failed " .. (msg or "") .. "\n")
28853   end
28854 end
28855 l3kernel.shellescape = shellescape

```

(End definition for `l3kernel.shellescape`. This function is documented on page [258](#).)

```

28856 </lua>
28857 </package>

```

46 l3unicode implementation

```

28858 (*package>
28859 @@@=char>

```

Case changing both for strings and “text” requires data from the Unicode Consortium. Some of this is build in to the format (as `\lccode` and `\uccode` values) but this covers only the simple one-to-one situations and does not fully handle for example case folding.

As only the data needs to remain at the end of this process, everything is set up inside a group. The only thing that is outside is creating a stream: they are global anyway and it is best to force a stream for all engines. For performance reasons, some of the code here is very low-level: the material is read during loading `expl3` in package mode.

```

28860 \ior_new:N \g__char_data_ior
28861 \bool_lazy_or:nnTF { \sys_if_engine luatex_p: } { \sys_if_engine xetex_p: }
28862 {
28863   \group_begin:

```

Access the primitive but suppress further expansion: active chars are otherwise an issue.

```

28864   \cs_set:Npn \__char_generate_char:n #1
28865   { \tex_detokenize:D \tex_expandafter:D { \tex_Uchar:D " #1 } }

```

A fast local implementation for generating characters; the chars may be active, so we prevent further expansion.

```

28866   \cs_set:Npx \__char_generate:n #1
28867   {
28868     \exp_not:N \tex_unexpanded:D \exp_not:N \exp_after:wN
28869     {
28870       \sys_if_engine luatex:TF
28871       {
28872         \exp_not:N \tex_directlua:D
28873         {
28874           l3kernel.charcat
28875           (
28876             \exp_not:N \tex_number:D #1 ,
28877             \exp_not:N \tex_the:D \tex_catcode:D #1
28878           )
28879         }
28880       }

```

```

28881         {
28882             \exp_not:N \tex_Ucharcat:D
28883             #1 ~
28884             \tex_catcode:D #1 ~
28885         }
28886     }
28887 }

```

Parse the main Unicode data file for two things. First, we want the titlecase exceptions: the one-to-one lower- and uppercase mappings it contains are all be covered by the T_EX data. Second, we need normalization data: at present, just the canonical NFD mappings. Those all yield either one or two codepoints, so the split is relatively easy.

```

28888 \ior_open:Nn \g__char_data_ior { UnicodeData.txt }
28889 \cs_set_protected:Npn \__char_data_auxi:w
28890   #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ;
28891   {
28892     \tl_if_blank:nF {#6}
28893     {
28894       \tl_if_head_eq_charcode:nNF {#6} < % >
28895       { \__char_data_auxii:w #1 ; #6 ~ \q_stop }
28896     }
28897     \__char_data_auxiii:w #1 ;
28898   }
28899 \cs_set_protected:Npn \__char_data_auxii:w #1 ; #2 ~ #3 \q_stop
28900 {
28901   \tl_const:cx
28902   { c__char_nfd_ \__char_generate_char:n {#1} _tl }
28903   {
28904     \__char_generate:n { "#2 }
28905     \tl_if_blank:nF {#3}
28906     { \__char_generate:n { "#3 } }
28907   }
28908 }
28909 \cs_set_protected:Npn \__char_data_auxiii:w
28910   #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ~ \q_stop
28911   {
28912     \cs_set_nopar:Npn \l__char_tmpa_tl {#7}
28913     \reverse_if:N \if_meaning:w \l__char_tmpa_tl \c_empty_tl
28914     \cs_set_nopar:Npn \l__char_tmpb_tl {#5}
28915     \reverse_if:N \if_meaning:w \l__char_tmpa_tl \l__char_tmpb_tl
28916     \tl_const:cx
28917     { c__char_titlecase_ \__char_generate_char:n {#1} _tl }
28918     { \__char_generate:n { "#7 } }
28919     \fi:
28920     \fi:
28921   }
28922 \group_begin:
28923   \char_set_catcode_space:n { ‘ \ }%
28924   \ior_map_variable:NNn \g__char_data_ior \l__char_tmpa_tl
28925   {%
28926     \if_meaning:w \l__char_tmpa_tl \c_space_tl
28927     \exp_after:wN \ior_map_break:
28928     \fi:
28929     \exp_after:wN \__char_data_auxi:w \l__char_tmpa_tl \q_stop

```



```

28930        }%
28931     \group_end:
28932     \ior_close:N \g__char_data_ior

```

The other data files all use C-style comments so we have to worry about # tokens (and reading as strings). The set up for case folding is in two parts. For the basic (core) mappings, folding is the same as lower casing in most positions so only store the differences. For the more complex foldings, always store the result, splitting up the two or three code points in the input as required.

```

28933     \ior_open:Nn \g__char_data_ior { CaseFolding.txt }
28934     \cs_set_protected:Npn \__char_data_auxi:w #1 ;~ #2 ;~ #3 ; #4 \q_stop
28935     {
28936         \if:w \tl_head:n { #2 ? } C
28937         \reverse_if:N \if_int_compare:w
28938             \char_value_lccode:n {"#1} = "#3 ~
28939             \tl_const:cx
28940                 { c__char_foldcase_ \__char_generate_char:n {#1} _tl }
28941                 { \__char_generate:n { "#3 } }
28942         \fi:
28943     \else:
28944         \if:w \tl_head:n { #2 ? } F
28945         \__char_data_auxii:w #1 ~ #3 ~ \q_stop
28946         \fi:
28947     \fi:
28948     }
28949     \cs_set_protected:Npn \__char_data_auxii:w #1 ~ #2 ~ #3 ~ #4 \q_stop
28950     {
28951         \tl_const:cx { c__char_foldcase_ \__char_generate_char:n {#1} _tl }
28952         {
28953             \__char_generate:n { "#2 }
28954             \__char_generate:n { "#3 }
28955             \tl_if_blank:nF {#4}
28956                 { \__char_generate:n { \int_value:w "#4 } }
28957         }
28958     }
28959     \ior_str_map_inline:Nn \g__char_data_ior
28960     {
28961         \reverse_if:N \if:w \c_hash_str \tl_head:w #1 \c_hash_str \q_stop
28962         \__char_data_auxi:w #1 \q_stop
28963         \fi:
28964     }
28965     \ior_close:N \g__char_data_ior

```

For upper- and lowercasing special situations, there is a bit more to do as we also have title casing to consider, plus we need to stop part-way through the file.

```

28966     \ior_open:Nn \g__char_data_ior { SpecialCasing.txt }
28967     \cs_set_protected:Npn \__char_data_auxi:w
28968         #1 ;~ #2 ;~ #3 ;~ #4 ; #5 \q_stop
28969     {
28970         \use:n { \__char_data_auxii:w #1 ~ lower ~ #2 ~ } ~ \q_stop
28971         \use:n { \__char_data_auxii:w #1 ~ upper ~ #4 ~ } ~ \q_stop
28972         \str_if_eq:nnF {#3} {#4}
28973             { \use:n { \__char_data_auxii:w #1 ~ title ~ #3 ~ } ~ \q_stop }
28974     }
28975     \cs_set_protected:Npn \__char_data_auxii:w

```

```

28976     #1 ~ #2 ~ #3 ~ #4 ~ #5 \q_stop
28977     {
28978         \tl_if_empty:nF {#4}
28979         {
28980             \tl_const:cx { c__char_ #2 case_ \__char_generate_char:n {#1} _tl }
28981             {
28982                 \__char_generate:n { "#3 }
28983                 \__char_generate:n { "#4 }
28984                 \tl_if_blank:nF {#5}
28985                 { \__char_generate:n { "#5 } }
28986             }
28987         }
28988     }
28989     \ior_str_map_inline:Nn \g__char_data_ior
28990     {
28991         \str_if_eq:eeTF
28992         { \tl_head:w #1 \c_hash_str \q_stop }
28993         { \c_hash_str }
28994         {
28995             \str_if_eq:eeT
28996             {#1}
28997             { \c_hash_str \c_space_tl Conditional-Mappings }
28998             { \ior_map_break: }
28999         }
29000         { \__char_data_auxi:w #1 \q_stop }
29001     }
29002     \ior_close:N \g__char_data_ior
29003     \group_end:
29004 }

```

For the 8-bit engines, the above is skipped but there is still some set up required. As case changing can only be applied to bytes, and they have to be in the ASCII range, we define a series of data stores to represent them, and the data are used such that only these are ever case-changed. We do open and close one file to force allocation of a read: this keeps all engines in line.

```

29005 {
29006     \group_begin:
29007     \cs_set_protected:Npn \__char_tmp:NN #1#2
29008     {
29009         \quark_if_recursion_tail_stop:N #2
29010         \tl_const:cn { c__char_uppercase_ #2 _tl } {#1}
29011         \tl_const:cn { c__char_lowercase_ #1 _tl } {#2}
29012         \tl_const:cn { c__char_foldcase_ #1 _tl } {#2}
29013         \__char_tmp:NN
29014     }
29015     \__char_tmp:NN
29016     AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
29017     ? \q_recursion_tail \q_recursion_stop
29018     \ior_open:Nn \g__char_data_ior { UnicodeData.txt }
29019     \ior_close:N \g__char_data_ior
29020     \group_end:
29021 }
29022 \</package>

```

47 l3text implementation

29023 `*package`

29024 `\@@=text`

47.1 Internal auxiliaries

`\s__text_stop` Internal scan marks.

29025 `\scan_new:N \s__text_stop`

(End definition for `\s__text_stop`.)

`\q__text_nil` Internal quarks.

29026 `\quark_new:N \q__text_nil`

(End definition for `\q__text_nil`.)

`__text_quark_if_nil_p:n` Branching quark conditional.

`__text_quark_if_nil:nTF` 29027 `__kernel_quark_new_conditional:Nn __text_quark_if_nil:n { TF }`

(End definition for `__text_quark_if_nil:nTF`.)

`\q__text_recursion_tail` Internal recursion quarks.

`\q__text_recursion_stop` 29028 `\quark_new:N \q__text_recursion_tail`

29029 `\quark_new:N \q__text_recursion_stop`

(End definition for `\q__text_recursion_tail` and `\q__text_recursion_stop`.)

`__text_use_i_delimit_by_q_recursion_stop:nw` Functions to gobble up to a quark.

29030 `\cs_new:Npn __text_use_i_delimit_by_q_recursion_stop:nw`

29031 `#1 #2 \q__text_recursion_stop {#1}`

(End definition for `__text_use_i_delimit_by_q_recursion_stop:nw`.)

`__text_if_recursion_tail_stop_do:Nn` Functions to query recursion quarks.

29032 `__kernel_quark_new_test:N __text_if_recursion_tail_stop_do:Nn`

(End definition for `__text_if_recursion_tail_stop_do:Nn`.)

47.2 Utilities

`__text_token_to_explicit:N` The idea here is to take a token and ensure that if it's an implicit char, we output the explicit version. Otherwise, the token needs to be unchanged. First, we have to split between control sequences and everything else.

`__text_token_to_explicit_char:N`

`__text_token_to_explicit_cs:N`

`__text_token_to_explicit_cs_aux:N`

`__text_token_to_explicit:n`

`__text_token_to_explicit_auxi:w`

`__text_token_to_explicit_auxii:w`

`__text_token_to_explicit_auxiii:w`

29033 `\group_begin:`

29034 `\char_set_catcode_active:n { 0 }`

29035 `\cs_new:Npn __text_token_to_explicit:N #1`

29036 `{`

29037 `\if_catcode:w \exp_not:N #1`

29038 `\if_catcode:w \scan_stop: \exp_not:N #1`

29039 `\scan_stop:`

29040 `\else:`

29041 `\exp_not:N ^^@`

29042 `\fi:`

29043 `\exp_after:wN __text_token_to_explicit_cs:N`

```

29044     \else:
29045         \exp_after:wN \_text_token_to_explicit_char:N
29046     \fi:
29047     #1
29048 }
29049 \group_end:

```

For control sequences, we can check for macros versus other cases using `\if_meaning:w`, then explicitly check for `\chardef` and `\mathchardef`.

```

29050 \cs_new:Npn \_text_token_to_explicit_cs:N #1
29051 {
29052     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
29053     \exp_after:wN \use:nn \exp_after:wN
29054     \_text_token_to_explicit_cs_aux:N
29055     \else:
29056         \exp_after:wN \exp_not:n
29057     \fi:
29058     {#1}
29059 }
29060 \cs_new:Npn \_text_token_to_explicit_cs_aux:N #1
29061 {
29062     \bool_lazy_or:nnTF
29063     { \token_if_chardef_p:N #1 }
29064     { \token_if_mathchardef_p:N #1 }
29065     {
29066         \char_generate:nn {#1}
29067         { \char_value_catcode:n {#1} }
29068     }
29069     {#1}
29070 }

```

For character tokens, we need to filter out the implicit characters from those that are explicit. That's done here, then if necessary we work out the category code and generate the char. To avoid issues with alignment tabs, that one is done by elimination rather than looking up the code explicitly. The trick with finding the charcode is that the `TeX` messages are either the `<something>` character `<char>` or the `<type>` `<char>`.

```

29071 \cs_new:Npn \_text_token_to_explicit_char:N #1
29072 {
29073     \if:w
29074         \if_catcode:w ^ \exp_args:No \str_tail:n { \token_to_str:N #1 } ^
29075         \token_to_str:N #1 #1
29076     \else:
29077         AB
29078     \fi:
29079     \exp_after:wN \exp_not:n
29080     \else:
29081         \exp_after:wN \_text_token_to_explicit:n
29082     \fi:
29083     {#1}
29084 }
29085 \cs_new:Npn \_text_token_to_explicit:n #1
29086 {
29087     \exp_after:wN \_text_token_to_explicit_auxi:w
29088     \int_value:w

```

```

29089 \if_catcode:w \c_group_begin_token #1 1 \else:
29090 \if_catcode:w \c_group_end_token #1 2 \else:
29091 \if_catcode:w \c_math_toggle_token #1 3 \else:
29092 \if_catcode:w ## #1 6 \else:
29093 \if_catcode:w ^ #1 7 \else:
29094 \if_catcode:w \c_math_subscript_token #1 8 \else:
29095 \if_catcode:w \c_space_token #1 10 \else:
29096 \if_catcode:w A #1 11 \else:
29097 \if_catcode:w + #1 12 \else:
29098 4 \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi:
29099 \exp_after:wN ;
29100 \token_to_meaning:N #1 \s__text_stop
29101 }
29102 \cs_new:Npn \__text_token_to_explicit_auxi:w #1 ; #2 \s__text_stop
29103 {
29104 \char_generate:nn
29105 {
29106 \if_int_compare:w #1 < 9 \exp_stop_f:
29107 \exp_after:wN \__text_token_to_explicit_auxii:w
29108 \else:
29109 \exp_after:wN \__text_token_to_explicit_auxiii:w
29110 \fi:
29111 #2
29112 }
29113 {#1}
29114 }
29115 \exp_last_unbraced:NNNNo \cs_new:Npn \__text_token_to_explicit_auxii:w
29116 #1 { \tl_to_str:n { character ~ } } { ' }
29117 \cs_new:Npn \__text_token_to_explicit_auxiii:w #1 ~ #2 ~ { ' }

```

(End definition for __text_token_to_explicit:N and others.)

__text_char_catcode:N An idea from l3char: we need to get the category code of a specific token, not the general case.

```

29118 \cs_new:Npn \__text_char_catcode:N #1
29119 {
29120 \if_catcode:w \exp_not:N #1 \c_math_toggle_token
29121 3
29122 \else:
29123 \if_catcode:w \exp_not:N #1 \c_alignment_token
29124 4
29125 \else:
29126 \if_catcode:w \exp_not:N #1 \c_math_superscript_token
29127 7
29128 \else:
29129 \if_catcode:w \exp_not:N #1 \c_math_subscript_token
29130 8
29131 \else:
29132 \if_catcode:w \exp_not:N #1 \c_space_token
29133 10
29134 \else:
29135 \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
29136 11
29137 \else:

```

```

29138             \if_catcode:w \exp_not:N #1 \c_catcode_other_token
29139                 12
29140             \else:
29141                 13
29142             \fi:
29143         \fi:
29144     \fi:
29145 \fi:
29146 \fi:
29147 \fi:
29148 \fi:
29149 }

```

(End definition for `_text_char_catcode:N`.)

`_text_if_expandable:NTF` Test for tokens that make sense to expand here: that is more restrictive than the engine view.

```

29150 \prg_new_conditional:Npnn \_text_if_expandable:N #1 { T , F , TF }
29151 {
29152     \token_if_expandable:NTF #1
29153     {
29154         \bool_lazy_any:nTF
29155         {
29156             { \token_if_protected_macro_p:N      #1 }
29157             { \token_if_protected_long_macro_p:N #1 }
29158             { \token_if_eq_meaning_p:NN \q_text_recursion_tail #1 }
29159         }
29160         { \prg_return_false: }
29161         { \prg_return_true: }
29162     }
29163     { \prg_return_false: }
29164 }

```

(End definition for `_text_if_expandable:NTF`.)

47.3 Configuration variables

`\l_text_accents_tl` `\l_text_letterlike_tl` Special cases for accents and letter-like symbols, which in some cases will need to be converted further.

```

29165 \tl_new:N \l_text_accents_tl
29166 \tl_set:Nn \l_text_accents_tl
29167 { \‘ \’ \^ \~ \= \u \. \" \r \H \V \d \c \k \b \t }
29168 \tl_new:N \l_text_letterlike_tl
29169 \tl_set:Nn \l_text_letterlike_tl
29170 {
29171     \AA \aa
29172     \AE \ae
29173     \DH \dh
29174     \DJ \dj
29175     \IJ \ij
29176     \L \l
29177     \NG \ng
29178     \O \o
29179     \OE \oe

```

```

29180 \SS \ss
29181 \TH \th
29182 }

```

(End definition for `\l_text_accents_tl` and `\l_text_letterlike_tl`. These variables are documented on page 263.)

`\l_text_case_exclude_arg_tl` Non-text arguments.

```

29183 \tl_new:N \l_text_case_exclude_arg_tl
29184 \tl_set:Nn \l_text_case_exclude_arg_tl { \begin \cite \end \label \ref }

```

(End definition for `\l_text_case_exclude_arg_tl`. This variable is documented on page 263.)

`\l_text_math_arg_tl` Math mode as arguments.

```

29185 \tl_new:N \l_text_math_arg_tl
29186 \tl_set:Nn \l_text_math_arg_tl { \ensuremath }

```

(End definition for `\l_text_math_arg_tl`. This variable is documented on page 263.)

`\l_text_math_delims_tl` Paired math mode delimiters.

```

29187 \tl_new:N \l_text_math_delims_tl
29188 \tl_set:Nn \l_text_math_delims_tl { $ $ \ ( \ ) }

```

(End definition for `\l_text_math_delims_tl`. This variable is documented on page 263.)

`\l_text_expand_exclude_tl` Commands which need not to expand.

```

29189 \tl_new:N \l_text_expand_exclude_tl
29190 \tl_set:Nn \l_text_expand_exclude_tl
29191 { \begin \cite \end \label \ref }

```

(End definition for `\l_text_expand_exclude_tl`. This variable is documented on page 263.)

`\l__text_math_mode_tl` Used to control math mode output: internal as there is a dedicated setter.

```

29192 \tl_new:N \l__text_math_mode_tl

```

(End definition for `\l__text_math_mode_tl`.)

47.4 Expansion to formatted text

`\c__text_chardef_space_token` Markers for implicit char handling.

```

\c__text_mathchardef_space_token 29193 \tex_chardef:D \c__text_chardef_space_token = '\ %
\c__text_chardef_group_begin_token 29194 \tex_mathchardef:D \c__text_mathchardef_space_token = '\ %
\c__text_mathchardef_group_begin_token 29195 \tex_chardef:D \c__text_chardef_group_begin_token = '\{ % '\}
\c__text_chardef_group_end_token 29196 \tex_mathchardef:D \c__text_mathchardef_group_begin_token = '\{ % '\} '\{
\c__text_mathchardef_group_end_token 29197 \tex_chardef:D \c__text_chardef_group_end_token = '\} % '\{
29198 \tex_mathchardef:D \c__text_mathchardef_group_end_token = '\} %

```

(End definition for `\c__text_chardef_space_token` and others.)

```

\text_expand:n
  \__text_expand:n
    \__text_expand_result:n
    \__text_expand_store:n
    \__text_expand_store:o
    \__text_expand_store:nw
    \__text_expand_end:w
    \__text_expand_loop:w
    \__text_expand_group:n
    \__text_expand_space:w
    \__text_expand_N_type:N
\__text_expand_N_type_auxi:N
  \__text_expand_N_type_auxii:N
  \__text_expand_N_type_auxiii:N
  \__text_expand_math_search:NNN
\__text_expand_math_loop:Nw
  \__text_expand_math_N_type:NN
\__text_expand_math_group:Nn
\__text_expand_math_space:Nw
  \__text_expand_implicit:N
  \__text_expand_explicit:N
  \__text_expand_exclude:N
  \__text_expand_exclude:nN
  \__text_expand_exclude:NN
  \__text_expand_exclude:Nn
\__text_expand_letterlike:N
\__text_expand_letterlike:NN
  \__text_expand_cs:N
  \__text_expand_encoding:N
  \__text_expand_encoding_escape:N
  \__text_expand_protect:N
  \__text_expand_protect:nN
  \__text_expand_protect:Nw
  \__text_expand_replace:N
  \__text_expand_replace:n
  \__text_expand_cs_expand:N
  \__text_expand_noexpand:nn

```

After precautions against & tokens, start a simple loop: that of course means that “text” cannot contain the two recursion quarks. The loop here must be f-type expandable; we have arbitrary user commands which might be protected *and* take arguments, and if the expansion code is used in a typesetting context, that will otherwise explode. (The same issue applies more clearly to case changing: see the example there.)

```

29199 \cs_new:Npn \text_expand:n #1
29200 {
29201   \__kernel_exp_not:w \exp_after:wN
29202   {
29203     \exp:w
29204     \__text_expand:n {#1}
29205   }
29206 }
29207 \cs_new:Npn \__text_expand:n #1
29208 {
29209   \group_align_safe_begin:
29210   \__text_expand_loop:w #1
29211   \q__text_recursion_tail \q__text_recursion_stop
29212   \__text_expand_result:n { }
29213 }

```

The approach to making the code f-type expandable is to use a marker result token and to shuffle the collected tokens

```

29214 \cs_new:Npn \__text_expand_store:n #1
29215 { \__text_expand_store:nw {#1} }
29216 \cs_generate_variant:Nn \__text_expand_store:n { o }
29217 \cs_new:Npn \__text_expand_store:nw #1#2 \__text_expand_result:n #3
29218 { #2 \__text_expand_result:n { #3 #1 } }
29219 \cs_new:Npn \__text_expand_end:w #1 \__text_expand_result:n #2
29220 {
29221   \group_align_safe_end:
29222   \exp_end:
29223   #2
29224 }

```

The main loop is a standard “tl action”; groups are handled recursively, while spaces are just passed through. Thus all of the action is in handling N-type tokens.

```

29225 \cs_new:Npn \__text_expand_loop:w #1 \q__text_recursion_stop
29226 {
29227   \tl_if_head_is_N_type:nTF {#1}
29228   { \__text_expand_N_type:N }
29229   {
29230     \tl_if_head_is_group:nTF {#1}
29231     { \__text_expand_group:n }
29232     { \__text_expand_space:w }
29233   }
29234   #1 \q__text_recursion_stop
29235 }
29236 \cs_new:Npn \__text_expand_group:n #1
29237 {
29238   \__text_expand_store:o
29239   {
29240     \exp_after:wN
29241     {

```



```

29242         \exp:w
29243         \__text_expand:n {#1}
29244     }
29245 }
29246 \__text_expand_loop:w
29247 }
29248 \exp_last_unbraced:NNo \cs_new:Npn \__text_expand_space:w \c_space_tl
29249 {
29250     \__text_expand_store:n { ~ }
29251     \__text_expand_loop:w
29252 }

```

Before we get into the real work, we have to watch out for problematic implicit characters: spaces and grouping tokens. Converting these to explicit characters later would lead to real issues as they are *not* N-type. A space is the easy case, so it's dealt with first: just insert the explicit token and continue the loop.

```

29253 \cs_new:Npx \__text_expand_N_type:N #1
29254 {
29255     \exp_not:N \__text_if_recursion_tail_stop_do:Nn #1
29256     { \exp_not:N \__text_expand_end:w }
29257     \exp_not:N \bool_lazy_any:nTF
29258     {
29259         { \exp_not:N \token_if_eq_meaning_p:NN #1 \c_space_token }
29260         {
29261             \exp_not:N \token_if_eq_meaning_p:NN #1
29262             \c__text_chardef_space_token
29263         }
29264         {
29265             \exp_not:N \token_if_eq_meaning_p:NN #1
29266             \c__text_mathchardef_space_token
29267         }
29268     }
29269     { \exp_not:N \__text_expand_space:w \c_space_tl }
29270     { \exp_not:N \__text_expand_N_type_auxi:N #1 }
29271 }

```

Implicit {/} offer two issues. First, the token could be an implicit brace character: we need to avoid turning that into a brace group, so filter out the cases manually. Then we handle the case where an implicit group is present. That is done in an “open-ended” way: there's the possibility the closing token is hidden somewhere.

```

29272 \cs_new:Npn \__text_expand_N_type_auxi:N #1
29273 {
29274     \bool_lazy_or:nnTF
29275     { \token_if_eq_meaning_p:NN #1 \c__text_chardef_group_begin_token }
29276     { \token_if_eq_meaning_p:NN #1 \c__text_mathchardef_group_begin_token }
29277     {
29278         \__text_expand_store:o \c_left_brace_str
29279         \__text_expand_loop:w
29280     }
29281     {
29282         \bool_lazy_or:nnTF
29283         { \token_if_eq_meaning_p:NN #1 \c__text_chardef_group_end_token }
29284         { \token_if_eq_meaning_p:NN #1 \c__text_mathchardef_group_end_token }
29285         {

```

```

29286         \_text_expand_store:o \c_right_brace_str
29287         \_text_expand_loop:w
29288     }
29289     { \_text_expand_N_type_auxii:N #1 }
29290 }
29291 }
29292 \cs_new:Npn \_text_expand_N_type_auxii:N #1
29293 {
29294     \token_if_eq_meaning:NNTF #1 \c_group_begin_token
29295     {
29296         { \if_false: } \fi:
29297         \_text_expand_loop:w
29298     }
29299     {
29300         \token_if_eq_meaning:NNTF #1 \c_group_end_token
29301         {
29302             \if_false: { \fi: }
29303             \_text_expand_loop:w
29304         }
29305         { \_text_expand_N_type_auxiii:N #1 }
29306     }
29307 }

```

The first step in dealing with N-type tokens is to look for math mode material: that needs to be left alone. The starting function has to be split into two as we need `\quark_if_recursion_tail_stop:N` first before we can trigger the search. We then look for matching pairs of delimiters, allowing for the case where math mode starts but does not end. Within math mode, we simply pass all the tokens through unchanged, just checking the N-type ones against the end marker.

```

29308 \cs_new:Npn \_text_expand_N_type_auxiii:N #1
29309 {
29310     \exp_after:wN \_text_expand_math_search:NNN
29311     \exp_after:wN #1 \l_text_math_delims_tl
29312     \q_text_recursion_tail \q_text_recursion_tail
29313     \q_text_recursion_stop
29314 }
29315 \cs_new:Npn \_text_expand_math_search:NNN #1#2#3
29316 {
29317     \_text_if_recursion_tail_stop_do:Nn #2
29318     { \_text_expand_explicit:N #1 }
29319     \token_if_eq_meaning:NNTF #1 #2
29320     {
29321         \_text_use_i_delimit_by_q_recursion_stop:nw
29322         {
29323             \_text_expand_store:n {#1}
29324             \_text_expand_math_loop:Nw #3
29325         }
29326     }
29327     { \_text_expand_math_search:NNN #1 }
29328 }
29329 \cs_new:Npn \_text_expand_math_loop:Nw #1#2 \q_text_recursion_stop
29330 {
29331     \tl_if_head_is_N_type:nTF {#2}
29332     { \_text_expand_math_N_type:NN }

```

```

29333     {
29334         \tl_if_head_is_group:nTF {#2}
29335         { \__text_expand_math_group:Nn }
29336         { \__text_expand_math_space:Nw }
29337     }
29338     #1#2 \q__text_recursion_stop
29339 }
29340 \cs_new:Npn \__text_expand_math_N_type:NN #1#2
29341 {
29342     \__text_if_recursion_tail_stop_do:Nn #2
29343     { \__text_expand_end:w }
29344     \__text_expand_store:n {#2}
29345     \token_if_eq_meaning:NNTF #2 #1
29346     { \__text_expand_loop:w }
29347     { \__text_expand_math_loop:Nw #1 }
29348 }
29349 \cs_new:Npn \__text_expand_math_group:Nn #1#2
29350 {
29351     \__text_expand_store:n { {#2} }
29352     \__text_expand_math_loop:Nw #1
29353 }
29354 \exp_after:wN \cs_new:Npn \exp_after:wN \__text_expand_math_space:Nw
29355 \exp_after:wN # \exp_after:wN 1 \c_space_tl
29356 {
29357     \__text_expand_store:n { ~ }
29358     \__text_expand_math_loop:Nw #1
29359 }

```

At this stage, either we have a control sequence or a simple character: split and handle.

```

29360 \cs_new:Npn \__text_expand_explicit:N #1
29361 {
29362     \token_if_cs:NNTF #1
29363     { \__text_expand_exclude:N #1 }
29364     {
29365         \__text_expand_store:n {#1}
29366         \__text_expand_loop:w
29367     }
29368 }
29369 % Next we exclude math commands: this is mainly as there \emph{might} be an
29370 % \cs{ensuremath}. We also handle accents, which are basically the same issue
29371 % but are kept separate for semantic reasons.
29372 % \begin{macrocode}
29373 \cs_new:Npn \__text_expand_exclude:N #1
29374 {
29375     \exp_args:Ne \__text_expand_exclude:nN
29376     {
29377         \exp_not:V \l_text_math_arg_tl
29378         \exp_not:V \l_text_accents_tl
29379         \exp_not:V \l_text_expand_exclude_tl
29380     }
29381     #1
29382 }
29383 \cs_new:Npn \__text_expand_exclude:nN #1#2
29384 {
29385     \__text_expand_exclude:NN #2 #1

```

```

29386     \q__text_recursion_tail \q__text_recursion_stop
29387   }
29388 \cs_new:Npn \__text_expand_exclude:Nn #1#2
29389 {
29390   \__text_if_recursion_tail_stop_do:Nn #2
29391   { \__text_expand_letterlike:N #1 }
29392   \cs_if_eq:NNTF #2 #1
29393   {
29394     \__text_use_i_delimit_by_q_recursion_stop:nw
29395     { \__text_expand_exclude:Nn #1 }
29396   }
29397   { \__text_expand_exclude:Nn #1 }
29398 }
29399 \cs_new:Npn \__text_expand_exclude:Nn #1#2
29400 {
29401   \__text_expand_store:n { #1 {#2} }
29402   \__text_expand_loop:w
29403 }

```

Another list of exceptions: these ones take no arguments so are easier to handle.

```

29404 \cs_new:Npn \__text_expand_letterlike:N #1
29405 {
29406   \exp_after:wN \__text_expand_letterlike:NN \exp_after:wN
29407   #1 \l_text_letterlike_tl
29408   \q__text_recursion_tail \q__text_recursion_stop
29409 }
29410 \cs_new:Npn \__text_expand_letterlike:NN #1#2
29411 {
29412   \__text_if_recursion_tail_stop_do:Nn #2
29413   { \__text_expand_cs:N #1 }
29414   \cs_if_eq:NNTF #2 #1
29415   {
29416     \__text_use_i_delimit_by_q_recursion_stop:nw
29417     {
29418       \__text_expand_store:n {#1}
29419       \__text_expand_loop:w
29420     }
29421   }
29422   { \__text_expand_letterlike:NN #1 }
29423 }

```

L^AT_EX 2_ε’s `\protect` makes life interesting. Where possible, we simply remove it and replace with the “parent” command; of course, the `\protect` might be explicit, in which case we need to leave it alone if it’s required.

```

29424 \cs_new:Npx \__text_expand_cs:N #1
29425 {
29426   \exp_not:N \str_if_eq:nnTF {#1} { \exp_not:N \protect }
29427   { \exp_not:N \__text_expand_protect:N }
29428   {
29429     \cs_if_exist:cTF { @current@cmd }
29430     { \exp_not:N \__text_expand_encoding:N #1 }
29431     { \exp_not:N \__text_expand_replace:N #1 }
29432   }
29433 }
29434 \cs_new:Npn \__text_expand_protect:N #1

```

```

29435 {
29436   \exp_args:Ne \__text_expand_protect:nN
29437   { \cs_to_str:N #1 } #1
29438 }
29439 \cs_new:Npn \__text_expand_protect:nN #1#2
29440 { \__text_expand_protect:Nw #2 #1 \q__text_nil #1 ~ \q__text_nil \q__text_nil \s__text_stop
29441 \cs_new:Npn \__text_expand_protect:Nw #1 #2 ~ \q__text_nil #3 \q__text_nil #4 \s__text_stop
29442 {
29443   \__text_quark_if_nil:nTF {#4}
29444   {
29445     \cs_if_exist:cTF {#2}
29446     { \exp_args:Ne \__text_expand_store:n { \exp_not:c {#2} } }
29447     { \__text_expand_store:n { \protect #1 } }
29448   }
29449   { \__text_expand_store:n { \protect #1 } }
29450   \__text_expand_loop:w
29451 }

```

Deal with encoding-specific commands

```

29452 \cs_new:Npn \__text_expand_encoding:N #1
29453 {
29454   \cs_if_eq:NNTF #1 \@current@cmd
29455   { \exp_after:wN \__text_expand_loop:w \__text_expand_encoding_escape:NN }
29456   {
29457     \cs_if_eq:NNTF #1 \@changed@cmd
29458     {
29459       \exp_after:wN \__text_expand_loop:w
29460       \__text_expand_encoding_escape:NN
29461     }
29462     { \__text_expand_replace:N #1 }
29463   }
29464 }
29465 \cs_new:Npn \__text_expand_encoding_escape:NN #1#2 { \exp_not:n {#1} }

```

See if there is a dedicated replacement, and if there is, insert it.

```

29466 \cs_new:Npn \__text_expand_replace:N #1
29467 {
29468   \bool_lazy_and:nnTF
29469   { \cs_if_exist_p:c { l__text_expand_ \token_to_str:N #1 _tl } }
29470   {
29471     \bool_lazy_or_p:nn
29472     { \token_if_cs_p:N #1 }
29473     { \token_if_active_p:N #1 }
29474   }
29475   {
29476     \exp_args:Nv \__text_expand_replace:n
29477     { l__text_expand_ \token_to_str:N #1 _tl }
29478   }
29479   { \__text_expand_cs_expand:N #1 }
29480 }
29481 \cs_new:Npn \__text_expand_replace:n #1 { \__text_expand_loop:w #1 }

```

Finally, expand any macros which can be: this then loops back around to deal with what they produce. The only issue is if the token is `\exp_not:n`, as that must apply to the following balanced text. There might be an `\exp_after:wN` there, so we check for it.

```

29482 \cs_new:Npn \__text_expand_cs_expand:N #1
29483 {
29484   \__text_if_expandable:NTF #1
29485   {
29486     \token_if_eq_meaning:NNTF #1 \exp_not:n
29487     { \__text_expand_noexpand:w }
29488     { \exp_after:wN \__text_expand_loop:w #1 }
29489   }
29490   {
29491     \__text_expand_store:n {#1}
29492     \__text_expand_loop:w
29493   }
29494 }
29495 \cs_new:Npn \__text_expand_noexpand:w #1#
29496 { \__text_expand_noexpand:nn {#1} }
29497 \cs_new:Npn \__text_expand_noexpand:nn #1#2
29498 {
29499   #1 \__text_expand_store:n #1 {#2}
29500   \__text_expand_loop:w
29501 }

```

(End definition for `\text_expand:n` and others. This function is documented on page 260.)

`\text_declare_expand_equivalent:Nn`
`\text_declare_expand_equivalent:cn`

Create equivalents to allow replacement.

```

29502 \cs_new_protected:Npn \text_declare_expand_equivalent:Nn #1#2
29503 {
29504   \tl_clear_new:c { l__text_expand_ \token_to_str:N #1 _tl }
29505   \tl_set:cn { l__text_expand_ \token_to_str:N #1 _tl } {#2}
29506 }
29507 \cs_generate_variant:Nn \text_declare_expand_equivalent:Nn { c }

```

(End definition for `\text_declare_expand_equivalent:Nn`. This function is documented on page 260.)

29508 `\</package>`

48 l3text-case implementation

29509 `*package>`

29510 `<@@=text>`

48.1 Case changing

`\l_text_titlecase_check_letter_bool`

Needed to determine the route used in titlecasing.

```

29511 \bool_new:N \l_text_titlecase_check_letter_bool
29512 \bool_set_true:N \l_text_titlecase_check_letter_bool

```

(End definition for `\l_text_titlecase_check_letter_bool`. This variable is documented on page 263.)

`\text_lowercase:n`
`\text_uppercase:n`
`\text_titlecase:n`
`\text_titlecase_first:n`
`\text_lowercase:nn`
`\text_uppercase:nn`
`\text_titlecase:nn`
`\text_titlecase_first:nn`

The user level functions here are all wrappers around the internal functions for case changing.

```

29513 \cs_new:Npn \text_lowercase:n #1
29514 { \__text_change_case:nnn { lower } { } {#1} }
29515 \cs_new:Npn \text_uppercase:n #1
29516 { \__text_change_case:nnn { upper } { } {#1} }

```

```

29517 \cs_new:Npn \text_titlecase:n #1
29518 { \__text_change_case:nnn { title } { } {#1} }
29519 \cs_new:Npn \text_titlecase_first:n #1
29520 { \__text_change_case:nnn { titleonly } { } {#1} }
29521 \cs_new:Npn \text_lowercase:nn #1#2
29522 { \__text_change_case:nnn { lower } {#1} {#2} }
29523 \cs_new:Npn \text_uppercase:nn #1#2
29524 { \__text_change_case:nnn { upper } {#1} {#2} }
29525 \cs_new:Npn \text_titlecase:nn #1#2
29526 { \__text_change_case:nnn { title } {#1} {#2} }
29527 \cs_new:Npn \text_titlecase_first:nn #1#2
29528 { \__text_change_case:nnn { titleonly } {#1} {#2} }

```

(End definition for `\text_lowercase:n` and others. These functions are documented on page 262.)

```

\__text_change_case:nnn
\__text_change_case_aux:nnn
\__text_change_case_store:n
\__text_change_case_store:o
\__text_change_case_store:V
\__text_change_case_store:v
\__text_change_case_store:e
\__text_change_case_store:nw
\__text_change_case_result:n
\__text_change_case_end:w
\__text_change_case_loop:nnw
\__text_change_case_break:w
\__text_change_case_group_lower:nnn
\__text_change_case_group_upper:nnn
\__text_change_case_group_title:nnn
\__text_change_case_group_titleonly:nnn
\__text_change_case_space:nnw
\__text_change_case_N_type:nnN
\__text_change_case_N_type_aux:nnN
\__text_change_case_N_type:nnnN
\__text_change_case_math_search:nnNN
\__text_change_case_math_loop:nnNw
\__text_change_case_math_N_type:nnNN
\__text_change_case_math_group:nnNn
\__text_change_case_math_space:nnNw
\__text_change_case_cs_check:nnN
\__text_change_case_exclude:nnN
\__text_change_case_exclude:nnnN
\__text_change_case_exclude:nnNN
\__text_change_case_exclude:nnNn
\__text_change_case_letterlike_lower:nnN
\__text_change_case_letterlike_upper:nnN
\__text_change_case_letterlike_title:nnN
\__text_change_case_letterlike_titleonly:nnN
\__text_change_case_letterlike:nnnnN
\__text_change_case_char_lower:nnN
\__text_change_case_char_upper:nnN
\__text_change_case_lower_sigma:nnnN
\__text_change_case_lower_sigma:nnNw
\__text_change_case_lower_sigma:NnnN
\__text_change_case_char_title:nnN
\__text_change_case_char_titleonly:nnN
\__text_change_case_char_title:nN
\__text_change_case_char_title:nnnN
\__text_change_case_char:nnnN
\__text_change_case_char:nnnnN

```

As for the expansion code, the business end of case changing is the handling of N-type tokens. First, we expand the input fully (so the loops here don't need to worry about awkward look-aheads and the like). Then we split into the different paths.

The code here needs to be f-type expandable to deal with the situation where case changing is applied in running text. There, we might have case changing as a document command and the text containing other non-expandable document commands.

```

\cs_set_eq:NN \MakeLowercase \text_lowercase
...
\MakeLowercase{\enquote*{A} text}

```

If we use an e-type expansion and wrap each token in `\exp_not:n`, that would explode: the document command grabs `\exp_not:n` as an argument, and things go badly wrong. So we have to wrap the entire result in exactly one `\exp_not:n`, or rather in the kernel version.

```

29529 \cs_new:Npn \__text_change_case:nnn #1#2#3
29530 {
29531   \__kernel_exp_not:w \exp_after:wN
29532   {
29533     \exp:w
29534     \exp_args:Ne \__text_change_case_aux:nnn
29535     { \text_expand:n {#3} }
29536     {#1} {#2}
29537   }
29538 }
29539 \cs_new:Npn \__text_change_case_aux:nnn #1#2#3
29540 {
29541   \group_align_safe_begin:
29542   \__text_change_case_loop:nnw {#2} {#3} #1
29543   \q__text_recursion_tail \q__text_recursion_stop
29544   \__text_change_case_result:n { }
29545 }

```

As for expansion, collect up the tokens for future use.

```

29546 \cs_new:Npn \__text_change_case_store:n #1
29547 { \__text_change_case_store:nw {#1} }
29548 \cs_generate_variant:Nn \__text_change_case_store:n { o , e , V , v }
29549 \cs_new:Npn \__text_change_case_store:nw #1#2 \__text_change_case_result:n #3
29550 { #2 \__text_change_case_result:n { #3 #1 } }

```

```

29551 \cs_new:Npn \__text_change_case_end:w #1 \__text_change_case_result:n #2
29552 {
29553   \group_align_safe_end:
29554   \exp_end:
29555   #2
29556 }

```

The main loop is the standard `tl` action type.

```

29557 \cs_new:Npn \__text_change_case_loop:nw #1#2#3 \q__text_recursion_stop
29558 {
29559   \tl_if_head_is_N_type:nTF {#3}
29560   { \__text_change_case_N_type:nnN }
29561   {
29562     \tl_if_head_is_group:nTF {#3}
29563     { \use:c { \__text_change_case_group_ #1 :nnn } }
29564     { \__text_change_case_space:nw }
29565   }
29566   {#1} {#2} #3 \q__text_recursion_stop
29567 }
29568 \cs_new:Npn \__text_change_case_break:w #1 \q__text_recursion_tail \q__text_recursion_stop
29569 {
29570   \__text_change_case_store:n {#1}
29571   \__text_change_case_end:w
29572 }

```

For a group, we *could* worry about whether this contains a character or not. However, that would make life very complex for little gain: exactly what a first character is is rather weakly-defined anyway. So if there is a group, we simply assume that a character has been seen, and for title case we switch to the “rest of the tokens” situation. To avoid having too much testing, we use a two-step process here to allow the titlecase functions to be separate.

```

29573 \cs_new:Npn \__text_change_case_group_lower:nnn #1#2#3
29574 {
29575   \__text_change_case_store:o
29576   {
29577     \exp_after:wN
29578     {
29579       \exp:w
29580       \__text_change_case_aux:nnn {#3} {#1} {#2}
29581     }
29582   }
29583   \__text_change_case_loop:nw {#1} {#2}
29584 }
29585 \cs_new_eq:NN \__text_change_case_group_upper:nnn
29586 \__text_change_case_group_lower:nnn
29587 \cs_new:Npn \__text_change_case_group_title:nnn #1#2#3
29588 {
29589   \__text_change_case_store:o
29590   {
29591     \exp_after:wN
29592     {
29593       \exp:w
29594       \__text_change_case_aux:nnn {#3} {#1} {#2}
29595     }

```



```

29596     }
29597     \__text_change_case_loop:nnw { lower } {#2}
29598   }
29599   \cs_new:Npn \__text_change_case_group_titleonly:nnn #1#2#3
29600   {
29601     \__text_change_case_store:o
29602     {
29603       \exp_after:wN
29604       {
29605         \exp:w
29606         \__text_change_case_aux:nnn {#3} {#1} {#2}
29607       }
29608     }
29609     \__text_change_case_break:w
29610   }
29611   \use:x
29612   {
29613     \cs_new:Npn \exp_not:N \__text_change_case_space:nnw ##1##2 \c_space_tl
29614   }
29615   {
29616     \__text_change_case_store:n { ~ }
29617     \__text_change_case_loop:nnw {#1} {#2}
29618   }

```

The first step of handling N-type tokens is to filter out the end-of-loop. That has to be done separately from the first real step as otherwise we pick up the wrong delimiter. The loop here is the same as the `expand` one, just passing the additional data long. If no close-math token is found then the final clean-up is forced (i.e. there is no assumption of “well-behaved” input in terms of math mode).

```

29619   \cs_new:Npn \__text_change_case_N_type:nnN #1#2#3
29620   {
29621     \__text_if_recursion_tail_stop_do:Nn #3
29622     { \__text_change_case_end:w }
29623     \__text_change_case_N_type_aux:nnN {#1} {#2} #3
29624   }
29625   \cs_new:Npn \__text_change_case_N_type_aux:nnN #1#2#3
29626   {
29627     \exp_args:NV \__text_change_case_N_type:nnnN
29628     \l_text_math_delims_tl {#1} {#2} #3
29629   }
29630   \cs_new:Npn \__text_change_case_N_type:nnnN #1#2#3#4
29631   {
29632     \__text_change_case_math_search:nnNN {#2} {#3} #4 #1
29633     \q__text_recursion_tail \q__text_recursion_tail
29634     \q__text_recursion_stop
29635   }
29636   \cs_new:Npn \__text_change_case_math_search:nnNN #1#2#3#4#5
29637   {
29638     \__text_if_recursion_tail_stop_do:Nn #4
29639     { \__text_change_case_cs_check:nnN {#1} {#2} #3 }
29640     \token_if_eq_meaning:NNTF #3 #4
29641     {
29642       \__text_use_i_delimit_by_q_recursion_stop:nw
29643     }

```

```

29644         \_text_change_case_store:n {#3}
29645         \_text_change_case_math_loop:nnNw {#1} {#2} #5
29646     }
29647 }
29648 { \_text_change_case_math_search:nnNN {#1} {#2} #3 }
29649 }
29650 \cs_new:Npn \_text_change_case_math_loop:nnNw #1#2#3#4 \q_text_recursion_stop
29651 {
29652     \tl_if_head_is_N_type:nTF {#4}
29653     { \_text_change_case_math_N_type:nnNN }
29654     {
29655         \tl_if_head_is_group:nTF {#4}
29656         { \_text_change_case_math_group:nnNn }
29657         { \_text_change_case_math_space:nnNw }
29658     }
29659     {#1} {#2} #3 #4 \q_text_recursion_stop
29660 }
29661 \cs_new:Npn \_text_change_case_math_N_type:nnNN #1#2#3#4
29662 {
29663     \_text_if_recursion_tail_stop_do:Nn #4
29664     { \_text_change_case_end:w }
29665     \_text_change_case_store:n {#4}
29666     \token_if_eq_meaning:NNTF #4 #3
29667     { \_text_change_case_loop:nnw {#1} {#2} }
29668     { \_text_change_case_math_loop:nnNw {#1} {#2} #3 }
29669 }
29670 \cs_new:Npn \_text_change_case_math_group:nnNn #1#2#3#4
29671 {
29672     \_text_change_case_store:n { {#4} }
29673     \_text_change_case_math_loop:nnNw {#1} {#2} #3
29674 }
29675 \use:x
29676 {
29677     \cs_new:Npn \exp_not:N \_text_change_case_math_space:nnNw ##1##2##3
29678     \c_space_tl
29679 }
29680 {
29681     \_text_change_case_store:n { ~ }
29682     \_text_change_case_math_loop:nnNw {#1} {#2} #3
29683 }

```

Once potential math-mode cases are filtered out the next stage is to test if the token grabbed is a control sequence: the two routes the code may take are then very different.

```

29684 \cs_new:Npn \_text_change_case_cs_check:nnN #1#2#3
29685 {
29686     \token_if_cs:NNTF #3
29687     { \_text_change_case_exclude:nnN }
29688     { \use:c { \_text_change_case_char_ #1 :nnN } }
29689     {#1} {#2} #3
29690 }

```

To deal with a control sequence there is first a need to test if it is on the list which indicate that case changing should be skipped. That's done using a loop as for the other special cases. If a hit is found then the argument is grabbed and passed through as-is.

```

29691 \cs_new:Npn \_text_change_case_exclude:nnN #1#2#3

```

```

29692 {
29693   \exp_args:Ne \__text_change_case_exclude:nnnN
29694   {
29695     \exp_not:V \l_text_math_arg_tl
29696     \exp_not:V \l_text_case_exclude_arg_tl
29697   }
29698   {#1} {#2} #3
29699 }
29700 \cs_new:Npn \__text_change_case_exclude:nnnN #1#2#3#4
29701 {
29702   \__text_change_case_exclude:nnNN {#2} {#3} #4 #1
29703   \q__text_recursion_tail \q__text_recursion_stop
29704 }
29705 \cs_new:Npn \__text_change_case_exclude:nnNN #1#2#3#4
29706 {
29707   \__text_if_recursion_tail_stop_do:Nn #4
29708   { \use:c { \__text_change_case_letterlike_ #1 :nnN } {#1} {#2} #3 }
29709   \cs_if_eq:NNTF #3 #4
29710   {
29711     \__text_use_i_delimit_by_q_recursion_stop:nw
29712     { \__text_change_case_exclude:nnNn {#1} {#2} #3 }
29713   }
29714   { \__text_change_case_exclude:nnNN {#1} {#2} #3 }
29715 }
29716 \cs_new:Npn \__text_change_case_exclude:nnNn #1#2#3#4
29717 {
29718   \__text_change_case_store:n { #3 {#4} }
29719   \__text_change_case_loop:nnw {#1} {#2}
29720 }

```

Letter-like commands may still be present: they are set up using a simple lookup approach, so can easily be handled with no loop. If there is no hit, we are at the end of the process: we loop around. Letter-like chars are all available only in upper- and lowercase, so titlecasing maps to the uppercase version.

```

29721 \cs_new:Npn \__text_change_case_letterlike_lower:nnN #1#2#3
29722 { \__text_change_case_letterlike:nnnnN {#1} {#1} {#1} {#2} #3 }
29723 \cs_new_eq:NN \__text_change_case_letterlike_upper:nnN
29724   \__text_change_case_letterlike_lower:nnN
29725 \cs_new:Npn \__text_change_case_letterlike_title:nnN #1#2#3
29726 { \__text_change_case_letterlike:nnnnN { upper } { lower } {#1} {#2} #3 }
29727 \cs_new:Npn \__text_change_case_letterlike_titleonly:nnN #1#2#3
29728 { \__text_change_case_letterlike:nnnnN { upper } { end } {#1} {#2} #3 }
29729 \cs_new:Npn \__text_change_case_letterlike:nnnnN #1#2#3#4#5
29730 {
29731   \cs_if_exist:cTF { c__text_ #1 case_ \token_to_str:N #5 _tl }
29732   {
29733     \__text_change_case_store:v
29734     { c__text_ #1 case_ \token_to_str:N #5 _tl }
29735     \use:c { \__text_change_case_char_next_ #2 :nn } {#2} {#4}
29736   }
29737   {
29738     \__text_change_case_store:n {#5}
29739     \cs_if_exist:cTF
29740     {

```

```

29741         c__text_
29742         \str_if_eq:nnTF {#1} { lower } { upper } { lower }
29743         case_ \token_to_str:N #5 _tl
29744     }
29745     { \use:c { __text_change_case_char_next_ #2 :nn } {#2} {#4} }
29746     { \__text_change_case_loop:nnw {#3} {#4} }
29747 }
29748 }

```

For upper- and lowercase changes, once we get to this stage there are only a couple of questions remaining: is there a language-specific mapping and is there the special case of a terminal sigma. If not, then we pass to a simple character mapping.

```

29749 \cs_new:Npx \__text_change_case_char_lower:nnN #1#2#3
29750 {
29751     \exp_not:N \cs_if_exist_use:cF { __text_change_case_lower_ #2 :nnN }
29752     {
29753         \bool_lazy_or:nnTF
29754         { \sys_if_engine luatex_p: }
29755         { \sys_if_engine xetex_p: }
29756         { \exp_not:N \__text_change_case_lower_sigma:nnnN }
29757         { \exp_not:N \__text_change_case_char:nnnN }
29758     }
29759     {#1} {#1} {#2} #3
29760 }
29761 \cs_new:Npn \__text_change_case_char_upper:nnN #1#2#3
29762 {
29763     \cs_if_exist_use:cF { __text_change_case_upper_ #2 :nnnN }
29764     { \__text_change_case_char:nnnN }
29765     {#1} {#1} {#2} #3
29766 }

```

If the current character is an uppercase sigma, then a check is made on the next item in the input. If it is N-type and not a control sequence then there is a look-ahead phase: the logic here is simply based on letters. The one exception is Dutch: see below.

```

29767 \bool_lazy_or:nnT
29768 { \sys_if_engine luatex_p: }
29769 { \sys_if_engine xetex_p: }
29770 {
29771     \cs_new:Npn \__text_change_case_lower_sigma:nnnN #1#2#3#4
29772     {
29773         \int_compare:nNnTF { '#4 } = { "03A3 }
29774         { \__text_change_case_lower_sigma:nnNw {#2} {#3} #4 }
29775         { \__text_change_case_char:nnnN {#1} {#2} {#3} #4 }
29776     }
29777     \cs_new:Npn \__text_change_case_lower_sigma:nnNw #1#2#3#4 \q__text_recursion_stop
29778     {
29779         \tl_if_head_is_N_type:nTF {#4}
29780         { \__text_change_case_lower_sigma:NnnN #3 }
29781         {
29782             \__text_change_case_store:e
29783             { \char_generate:nn { "03C2 } { \__text_char_catcode:N #3 } }
29784             \__text_change_case_loop:nnw
29785         }
29786         {#1} {#2} #4 \q__text_recursion_stop
29787     }

```

```

29788 \cs_new:Npn \__text_change_case_lower_sigma:NnnN #1#2#3#4
29789 {
29790   \__text_change_case_store:e
29791   {
29792     \token_if_letter:NTF #4
29793     { \char_generate:nn { "03C3 } { \__text_char_catcode:N #1 } }
29794     { \char_generate:nn { "03C2 } { \__text_char_catcode:N #1 } }
29795   }
29796   \__text_change_case_loop:nnw {#2} {#3} #4
29797 }
29798 }

```

For titlecasing, we need to fully expand the new character to see if it is a letter (or active) But that means looking ahead in the 8-bit case, so we have to grab the required tokens up-front. Life is a lot easier for Unicode \TeX 's, where we just have one token to worry about. The one wrinkle here is that for look-ahead we'd get into trouble: luckily, only Dutch has that issue.

```

29799 \cs_new:Npx \__text_change_case_char_title:nnN #1#2#3
29800 {
29801   \exp_not:N \bool_if:NTF \l_text_titlecase_check_letter_bool
29802   {
29803     \bool_lazy_or:nnTF
29804     { \sys_if_engine luatex_p: }
29805     { \sys_if_engine xetex_p: }
29806     { \exp_not:N \token_if_letter:NTF #3 }
29807     {
29808       \exp_not:N \bool_lazy_or:nnTF
29809       { \exp_not:N \token_if_letter_p:N #3 }
29810       { \exp_not:N \token_if_active_p:N #3 }
29811     }
29812     { \exp_not:N \use:c { __text_change_case_char_ #1 :nN } }
29813     { \exp_not:N \__text_change_case_char_title:nnnN { title } {#1} }
29814   }
29815   { \exp_not:N \use:c { __text_change_case_char_ #1 :nN } }
29816   {#2} #3
29817 }
29818 \cs_new_eq:NN \__text_change_case_char_titleonly:nnN
29819 \__text_change_case_char_title:nnN
29820 \cs_new:Npn \__text_change_case_char_title:nN #1#2
29821 { \__text_change_case_char_title:nnnN { title } { lower } {#1} #2 }
29822 \cs_new:Npn \__text_change_case_char_titleonly:nN #1#2
29823 { \__text_change_case_char_title:nnnN { title } { end } {#1} #2 }
29824 \cs_new:Npn \__text_change_case_char_title:nnnN #1#2#3#4
29825 {
29826   \cs_if_exist_use:cF { __text_change_case_title_ #3 :nnnN }
29827   {
29828     \cs_if_exist_use:cF { __text_change_case_upper_ #3 :nnnN }
29829     { \__text_change_case_char:nnnN }
29830   }
29831   {#1} {#2} {#3} #4
29832 }

```

For Unicode engines we can handle all characters directly. However, for the 8-bit engines the aim is to deal with (a subset of) Unicode (UTF-8) input. They deal with that by making the upper half of the range active, so we look for that and if found work out how

many UTF-8 octets there are to deal with. Those can then be grabbed to reconstruct the full Unicode character, which is then used in a lookup. (As will become obvious below, there is no intention here of covering all of Unicode.)

```

29833 \cs_new:Npn \__text_change_case_char:nnnN #1#2#3#4
29834 {
29835   \token_if_active:NTF #4
29836   { \__text_change_case_store:n {#4} }
29837   {
29838     \__text_change_case_store:e
29839     { \use:c { char_ #1 case :N } #4 }
29840   }
29841   \use:c { __text_change_case_char_next_ #2 :nn } {#2} {#3}
29842 }
29843 \bool_lazy_or:nnF
29844 { \sys_if_engine luatex_p: }
29845 { \sys_if_engine xetex_p: }
29846 {
29847   \cs_new_eq:NN \__text_change_case_char_aux:nnnN
29848   \__text_change_case_char:nnnN
29849   \cs_gset:Npn \__text_change_case_char:nnnN #1#2#3#4
29850   {
29851     \int_compare:nNnTF { '#4 } > { "80 }
29852     {
29853       \int_compare:nNnTF { '#4 } < { "EO }
29854       { \__text_change_case_char_UTFviii:nnnNN }
29855       {
29856         \int_compare:nNnTF { '#4 } < { "F0 }
29857         { \__text_change_case_char_UTFviii:nnnNNN }
29858         { \__text_change_case_char_UTFviii:nnnNNNN }
29859       }
29860       {#1} {#2} {#3} #4
29861     }
29862     { \__text_change_case_char_aux:nnnN {#1} {#2} {#3} #4 }
29863   }
29864   \cs_new:Npn \__text_change_case_char_UTFviii:nnnNN #1#2#3#4#5
29865   { \__text_change_case_char_UTFviii:nnnn {#1} {#2} {#3} {#4#5} }
29866   \cs_new:Npn \__text_change_case_char_UTFviii:nnnNNN #1#2#3#4#5#6
29867   { \__text_change_case_char_UTFviii:nnnn {#1} {#2} {#3} {#4#5#6} }
29868   \cs_new:Npn \__text_change_case_char_UTFviii:nnnNNNN #1#2#3#4#5#6#7
29869   { \__text_change_case_char_UTFviii:nnnn {#1} {#2} {#3} {#4#5#6#7} }
29870   \cs_new:Npn \__text_change_case_char_UTFviii:nnnn #1#2#3#4
29871   {
29872     \cs_if_exist:cTF { c__text_ #1 case_ \tl_to_str:n {#4} _tl }
29873     {
29874       \__text_change_case_store:v
29875       { c__text_ #1 case_ \tl_to_str:n {#4} _tl }
29876     }
29877     { \__text_change_case_store:n {#4} }
29878     \use:c { __text_change_case_char_next_ #2 :nn } {#2} {#3}
29879   }
29880 }
29881 \cs_new:Npn \__text_change_case_char_next_lower:nn #1#2
29882 { \__text_change_case_loop:nw {#1} {#2} }
29883 \cs_new_eq:NN \__text_change_case_char_next_upper:nn

```

```

29884 \__text_change_case_char_next_lower:nn
29885 \cs_new_eq:NN \__text_change_case_char_next_title:nn
29886 \__text_change_case_char_next_lower:nn
29887 \cs_new_eq:NN \__text_change_case_char_next_titleonly:nn
29888 \__text_change_case_char_next_lower:nn
29889 \cs_new:Npn \__text_change_case_char_next_end:nn #1#2
29890 { \__text_change_case_break:w }

```

(End definition for __text_change_case:nnn and others.)

A simple alternative version for German.

```

\__text_change_case_upper_de-alt:nnnN
\__text_change_case_upper_de-alt:nnnNN

```

```

29891 \bool_lazy_or:nnTF
29892 { \sys_if_engine luatex_p: }
29893 { \sys_if_engine xetex_p: }
29894 {
29895   \cs_new:cpn { __text_change_case_upper_de-alt:nnnN } #1#2#3#4
29896   {
29897     \int_compare:nNnTF { '#4 } = { "00DF }
29898     {
29899       \__text_change_case_store:e
29900       { \char_generate:nn { "1E9E } { \__text_char_catcode:N #4 } }
29901       \use:c { __text_change_case_char_next_ #2 :nn }
29902       {#2} {#3}
29903     }
29904     { \__text_change_case_char:nnnN {#1} {#2} {#3} #4 }
29905   }
29906 }
29907 {
29908   \cs_new:cpx { __text_change_case_upper_de-alt:nnnN } #1#2#3#4
29909   {
29910     \exp_not:N \int_compare:nNnTF { '#4 } = { "00C3 }
29911     {
29912       \exp_not:c { __text_change_case_upper_de-alt:nnnNN }
29913       {#1} {#2} {#3} #4
29914     }
29915     { \exp_not:N \__text_change_case_char:nnnN {#1} {#2} {#3} #4 }
29916   }
29917   \cs_new:cpn { __text_change_case_upper_de-alt:nnnNN } #1#2#3#4#5
29918   {
29919     \int_compare:nNnTF { '#5 } = { "009F }
29920     {
29921       \__text_change_case_store:V \c__text_grosses_Eszett_tl
29922       \use:c { __text_change_case_char_next_ #2 :nn } {#2} {#3}
29923     }
29924     { \__text_change_case_char:nnnN {#1} {#2} {#3} #4#5 }
29925   }
29926 }

```

(End definition for __text_change_case_upper_de-alt:nnnN and __text_change_case_upper_de-alt:nnnNN.)

```

\__text_change_case_upper_el:nnnN
\__text_change_case_upper_el:nnnN
\__text_change_case_upper_el_aux:nnnN
\__text_change_case_upper_el_loop:nnw
\__text_change_case_upper_el:nnN
\__text_change_case_if_greek:nTF

```

For Greek uppcasing, we need to know if characters *in the Greek range* have accents. That means doing a NFD conversion first, then starting a search. As described by the Unicode CLDR, Greek accents need to be found *after* any U+0308 (dieresis) and are done in two groups to allow for the canonical ordering.

```

29927 \bool_lazy_or:nnT
29928 { \sys_if_engine luatex_p: }
29929 { \sys_if_engine xetex_p: }
29930 {
29931   \cs_new:Npn \__text_change_case_upper_el:nnnN #1#2#3#4
29932   {
29933     \__text_change_case_if_greek:nTF { '#4 }
29934     {
29935       \exp_args:Ne \__text_change_case_upper_el:nnnn
29936       { \char_to_nfd:N #4 } {#1} {#2} {#3}
29937     }
29938     { \__text_change_case_char:nnnN {#1} {#2} {#3} #4 }
29939   }
29940   \cs_new:Npn \__text_change_case_upper_el:nnnn #1#2#3#4
29941   { \__text_change_case_upper_el_aux:nnnN {#2} {#3} {#4} #1 }
29942   \cs_new:Npn \__text_change_case_upper_el_aux:nnnN #1#2#3#4
29943   {
29944     \__text_change_case_store:e { \use:c { char_ #1 case:N } #4 }
29945     \__text_change_case_upper_el_loop:nnw {#2} {#3}
29946   }
29947   \cs_new:Npn \__text_change_case_upper_el_loop:nnw
29948   #1#2#3 \q__text_recursion_stop
29949   {
29950     \tl_if_head_is_N_type:nTF {#3}
29951     { \__text_change_case_upper_el:nnN }
29952     { \__text_change_case_loop:nnw }
29953     {#1} {#2} #3 \q__text_recursion_stop
29954   }

```

In addition to the Greek accents, we list three cases here where an accent outside the Greek range has a nfd that would make it equivalent. That includes U+0344, which has to insert U+0308.

```

29955   \cs_new:Npn \__text_change_case_upper_el:nnN #1#2#3
29956   {
29957     \token_if_cs:NTF #3
29958     { \__text_change_case_loop:nnw {#1} {#2} #3 }
29959     {
29960       \int_compare:nNnTF { '#3 } = { "0308 }
29961       {
29962         \__text_change_case_store:n {#3}
29963         \__text_change_case_upper_el_loop:nnw {#1} {#2}
29964       }
29965       {
29966         \bool_lazy_any:nTF
29967         {
29968           { \int_compare_p:nNn { '#3 } = { "0300 } }
29969           { \int_compare_p:nNn { '#3 } = { "0301 } }
29970           { \int_compare_p:nNn { '#3 } = { "0304 } }
29971           { \int_compare_p:nNn { '#3 } = { "0306 } }
29972           { \int_compare_p:nNn { '#3 } = { "0308 } }
29973           { \int_compare_p:nNn { '#3 } = { "0313 } }
29974           { \int_compare_p:nNn { '#3 } = { "0314 } }
29975           { \int_compare_p:nNn { '#3 } = { "0342 } }
29976           { \int_compare_p:nNn { '#3 } = { "0340 } }

```



```

29977         { \int_compare_p:nNn { '#3 } = { "0341 } }
29978         { \int_compare_p:nNn { '#3 } = { "0343 } }
29979     }
29980     { \__text_change_case_upper_el_loop:nnw {#1} {#2} }
29981     {
29982         \int_compare:nNnTF { '#3 } = { "0344 }
29983         {
29984             \__text_change_case_store:e
29985             {
29986                 \char_generate:nn { "0308 }
29987                 { \__text_char_catcode:N #3 }
29988             }
29989             \__text_change_case_upper_el_loop:nnw {#1} {#2}
29990         }
29991         {
29992             \int_compare:nNnTF { '#3 } = { "0345 }
29993             { \__text_change_case_loop:nnw {#1} {#2} }
29994             { \__text_change_case_loop:nnw {#1} {#2} #3 }
29995         }
29996     }
29997 }
29998 }
29999 }
30000 \prg_new_conditional:Npnn \__text_change_case_if_greek:n #1 { TF }
30001 {
30002     \if_int_compare:w #1 < "0370 \exp_stop_f:
30003     \prg_return_false:
30004     \else:
30005         \if_int_compare:w #1 > "03FF \exp_stop_f:
30006         \if_int_compare:w #1 < "1F00 \exp_stop_f:
30007         \prg_return_false:
30008         \else:
30009             \if_int_compare:w #1 > "1FFF \exp_stop_f:
30010             \prg_return_false:
30011             \else:
30012                 \prg_return_true:
30013             \fi:
30014             \fi:
30015             \else:
30016                 \prg_return_true:
30017             \fi:
30018             \fi:
30019         }
30020     }

```

(End definition for __text_change_case_upper_el:nnnN and others.)

__text_change_case_title_el:nnnN Titlecasing retains accents, but to prevent the uppercasing code from kicking in, there has to be an explicit function here.

```

30021 \bool_lazy_or:nnT
30022 { \sys_if_engine luatex_p: }
30023 { \sys_if_engine xetex_p: }
30024 {
30025     \cs_new:Npn \__text_change_case_title_el:nnnN #1#2#3#4

```

```

30026     { \_text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30027   }

```

(End definition for _text_change_case_title_el:nnnN.)

```

\_text_change_cases_lower_lt:nnnN
\_text_change_cases_lower_lt_auxi:nnnN
\_text_change_cases_lower_lt_auxii:nnnN
  \_text_change_case_lower_lt:nnw
  \_text_change_case_lower_lt:nnN

```

For Lithuanian, the issue to be dealt with is dots over lower case letters: these should be present if there is another accent. The first step is a simple match attempt: look for the three uppercase accented letters which should gain a dot-above char in their lowercase form.

```

30028 \bool_lazy_or:nnT
30029   { \sys_if_engine luatex_p: }
30030   { \sys_if_engine xetex_p: }
30031   {
30032     \cs_new:Npn \_text_change_case_lower_lt:nnnN #1#2#3#4
30033       {
30034         \exp_args:Ne \_text_change_case_lower_lt_auxi:nnnN
30035           {
30036             \int_case:nn { '#4 }
30037               {
30038                 { "00CC } { "0300 }
30039                 { "00CD } { "0301 }
30040                 { "0128 } { "0303 }
30041               }
30042             }
30043           {#2} {#3} #4
30044         }

```

If there was a hit, output the result with the dot-above and move on. Otherwise, look for one of the three letters that can take a combining accent: I, J and I-ogonek.

```

30045     \cs_new:Npn \_text_change_case_lower_lt_auxi:nnnN #1#2#3#4
30046       {
30047         \tl_if_blank:nTF {#1}
30048           {
30049             \exp_args:Ne \_text_change_case_lower_lt_auxii:nnnN
30050               {
30051                 \int_case:nn { '#4 }
30052                   {
30053                     { "0049 } { "0069 }
30054                     { "004A } { "006A }
30055                     { "012E } { "012F }
30056                   }
30057                 }
30058               {#2} {#3} #4
30059             }
30060           {
30061             \_text_change_case_store:e
30062               {
30063                 \char_generate:nn { "0069 } { \_text_char_catcode:N #4 }
30064                 \char_generate:nn { "0307 } { \_text_char_catcode:N #4 }
30065                 \char_generate:nn {#1} { \_text_char_catcode:N #4 }
30066               }
30067             \_text_change_case_loop:nnw {#2} {#3}
30068           }
30069         }

```

Again, branch depending on a hit. If there is one, we output the character then need to look for a combining accent: as usual, we need to be aware of the loop situation.

```

30070 \cs_new:Npn \__text_change_case_lower_lt_auxii:nnnN #1#2#3#4
30071 {
30072   \tl_if_blank:nTF {#1}
30073   { \__text_change_case_lower_sigma:nnnN {#2} {#2} {#3} #4 }
30074   {
30075     \__text_change_case_store:e
30076     { \char_generate:nn {#1} { \__text_char_catcode:N #4 } }
30077     \__text_change_case_lower_lt:nnw {#2} {#3}
30078   }
30079 }
30080 \cs_new:Npn \__text_change_case_lower_lt:nnw #1#2#3 \q__text_recursion_stop
30081 {
30082   \tl_if_head_is_N_type:nTF {#3}
30083   { \__text_change_case_lower_lt:nnN }
30084   { \__text_change_case_loop:nnw }
30085   {#1} {#2} #3 \q__text_recursion_stop
30086 }
30087 \cs_new:Npn \__text_change_case_lower_lt:nnN #1#2#3
30088 {
30089   \bool_lazy_and:nnT
30090   { ! \token_if_cs_p:N #3 }
30091   {
30092     \bool_lazy_any_p:n
30093     {
30094       { \int_compare_p:nNn { '#3 } = { "0300 } }
30095       { \int_compare_p:nNn { '#3 } = { "0301 } }
30096       { \int_compare_p:nNn { '#3 } = { "0303 } }
30097     }
30098   }
30099   {
30100     \__text_change_case_store:e
30101     { \char_generate:nn { "0307 } { \__text_char_catcode:N #3 } }
30102   }
30103   \__text_change_case_loop:nnw {#1} {#2} #3
30104 }
30105 }

```

(End definition for `__text_change_cases_lower_lt:nnnN` and others.)

The uppercasing version: first find i/j/i-ogonek, then look for the combining char: drop it if present.

```

\__text_change_cases_upper_lt:nnnN
\__text_change_cases_upper_lt_aux:nnnN
\__text_change_case_upper_lt:nnw
\__text_change_case_upper_lt:nnN
30106 \bool_lazy_or:nnT
30107 { \sys_if_engine luatex_p: }
30108 { \sys_if_engine xetex_p: }
30109 {
30110   \cs_new:Npn \__text_change_case_upper_lt:nnnN #1#2#3#4
30111   {
30112     \exp_args:Ne \__text_change_case_upper_lt_aux:nnnN
30113     {
30114       \int_case:nn { '#4 }
30115       {
30116         { "0069 } { "0049 }

```

```

30117         { "006A } { "004A }
30118         { "012F } { "012E }
30119     }
30120 }
30121     {#2} {#3} #4
30122 }
30123 \cs_new:Npn \__text_change_case_upper_lt_aux:nnnN #1#2#3#4
30124 {
30125     \tl_if_blank:nTF {#1}
30126     { \__text_change_case_char:nnnN { upper } {#2} {#3} #4 }
30127     {
30128         \__text_change_case_store:e
30129         { \char_generate:nn {#1} { \__text_char_catcode:N #4 } }
30130         \__text_change_case_upper_lt:nnw {#2} {#3}
30131     }
30132 }
30133 \cs_new:Npn \__text_change_case_upper_lt:nnw #1#2#3 \q__text_recursion_stop
30134 {
30135     \tl_if_head_is_N_type:nTF {#3}
30136     { \__text_change_case_upper_lt:nnN }
30137     { \use:c { __text_change_case_char_next_ #1 :nn } }
30138     {#1} {#2} #3 \q__text_recursion_stop
30139 }
30140 \cs_new:Npn \__text_change_case_upper_lt:nnN #1#2#3
30141 {
30142     \bool_lazy_and:nnTF
30143     { ! \token_if_cs_p:N #3 }
30144     { \int_compare_p:nNn { '#3 } = { "0307 } }
30145     { \use:c { __text_change_case_char_next_ #1 :nn } {#1} {#2} }
30146     { \use:c { __text_change_case_char_next_ #1 :nn } {#1} {#2} #3 }
30147 }
30148 }

```

(End definition for __text_change_cases_upper_lt:nnnN and others.)

__text_change_case_title_nl:nnnN For Dutch, there is a single look-ahead test for ij when title casing. If the appropriate letters are found, produce IJ and gobble the j/J.

```

\__text_change_case_title_nl:nnw
\__text_change_case_title_nl:nnN
30149 \cs_new:Npn \__text_change_case_title_nl:nnnN #1#2#3#4
30150 {
30151     \bool_lazy_or:nnTF
30152     { \int_compare_p:nNn { '#4 } = { "0049 } }
30153     { \int_compare_p:nNn { '#4 } = { "0069 } }
30154     {
30155         \__text_change_case_store:e
30156         { \char_generate:nn { "0049 } { \__text_char_catcode:N #4 } }
30157         \__text_change_case_title_nl:nnw {#2} {#3}
30158     }
30159     { \__text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30160 }
30161 \cs_new:Npn \__text_change_case_title_nl:nnw #1#2#3 \q__text_recursion_stop
30162 {
30163     \tl_if_head_is_N_type:nTF {#3}
30164     { \__text_change_case_title_nl:nnN }
30165     { \use:c { __text_change_case_char_next_ #1 :nn } }

```

```

30166         {#1} {#2} #3 \q__text_recursion_stop
30167     }
30168 \cs_new:Npn \__text_change_case_title_nl:nnN #1#2#3
30169 {
30170     \bool_lazy_and:nnTF
30171     { ! \token_if_cs_p:N #3 }
30172     {
30173         \bool_lazy_or_p:nn
30174         { \int_compare_p:nNn { '#3 } = { "004A } }
30175         { \int_compare_p:nNn { '#3 } = { "006A } }
30176     }
30177     {
30178         \__text_change_case_store:e
30179         { \char_generate:nn { "004A } { \__text_char_catcode:N #3 } }
30180         \use:c { __text_change_case_char_next_ #1 :nn } {#1} {#2}
30181     }
30182     { \use:c { __text_change_case_char_next_ #1 :nn } {#1} {#2} #3 }
30183 }

```

(End definition for __text_change_case_title_nl:nnnN, __text_change_case_title_nl:nnw, and __text_change_case_title_nl:nnN.)

__text_change_case_lower_tr:nnnN
 __text_change_case_lower_tr:nnNw
 __text_change_case_lower_tr:NnnN
 __text_change_case_lower_tr:nnnN

The Turkic languages need special treatment for dotted-i and dotless-i. The lower casing rule can be expressed in terms of searching first for either a dotless-I or a dotted-I. In the latter case the mapping is easy, but in the former there is a second stage search.

```

30184 \bool_lazy_or:nnTF
30185 { \sys_if_engine luatex_p: }
30186 { \sys_if_engine xetex_p: }
30187 {
30188     \cs_new:Npn \__text_change_case_lower_tr:nnnN #1#2#3#4
30189     {
30190         \int_compare:nNnTF { '#4 } = { "0049 }
30191         { \__text_change_case_lower_tr:nnNw {#1} {#3} #4 }
30192         {
30193             \int_compare:nNnTF { '#4 } = { "0130 }
30194             {
30195                 \__text_change_case_store:e
30196                 { \char_generate:nn { "0069 } { \__text_char_catcode:N #4 } }
30197                 \__text_change_case_loop:nnw {#1} {#3}
30198             }
30199             { \__text_change_case_lower_sigma:nnnN {#1} {#2} {#3} #4 }
30200         }
30201     }

```

After a dotless-I there may be a dot-above character. If there is then a dotted-i should be produced, otherwise output a dotless-i. When the combination is found both the dotless-I and the dot-above char have to be removed from the input.

```

30202 \cs_new:Npn \__text_change_case_lower_tr:nnNw #1#2#3#4 \q__text_recursion_stop
30203 {
30204     \tl_if_head_is_N_type:nTF {#4}
30205     { \__text_change_case_lower_tr:NnnN #3 }
30206     {
30207         \__text_change_case_store:e
30208         { \char_generate:nn { "0131 } { \__text_char_catcode:N #3 } }

```

```

30209         \_text_change_case_loop:nnw
30210     }
30211     {#1} {#2} #4 \q_text_recursion_stop
30212 }
30213 \cs_new:Npn \_text_change_case_lower_tr:NnnN #1#2#3#4
30214 {
30215     \bool_lazy_or:nnTF
30216     { \token_if_cs_p:N #4 }
30217     { ! \int_compare_p:nNn { '#4 } = { "0307 } }
30218     {
30219         \_text_change_case_store:e
30220         { \char_generate:nn { "0131 } { \_text_char_catcode:N #1 } }
30221         \_text_change_case_loop:nnw {#2} {#3} #4
30222     }
30223     {
30224         \_text_change_case_store:e
30225         { \char_generate:nn { "0069 } { \_text_char_catcode:N #1 } }
30226         \_text_change_case_loop:nnw {#2} {#3}
30227     }
30228 }
30229 }

```

For 8-bit engines, dot-above is not available so there is a simple test for an upper-case I. Then we can look for the UTF-8 representation of an upper case dotted-I without the combining char. If it's not there, preserve the UTF-8 sequence as-is. With 8bit engines, we cannot completely preserve category codes, so we have to make some assumptions: output a “normal” i for the dotted case. As the original character here is catcode-13, we have to make a choice about handling of i: generate a “normal” one.

```

30230 {
30231     \cs_new:Npn \_text_change_case_lower_tr:nnnN #1#2#3#4
30232     {
30233         \int_compare:nNnTF { '#4 } = { "0049 }
30234         {
30235             \_text_change_case_store:V \c_text_dotless_i_tl
30236             \_text_change_case_loop:nnw {#1} {#3}
30237         }
30238         {
30239             \int_compare:nNnTF { '#4 } = { "00C4 }
30240             { \_text_change_case_lower_tr:nnnN {#1} {#2} {#3} #4 }
30241             { \_text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30242         }
30243     }
30244     \cs_new:Npn \_text_change_case_lower_tr:nnnNN #1#2#3#4#5
30245     {
30246         \int_compare:nNnTF { '#5 } = { "00B0 }
30247         {
30248             \_text_change_case_store:e
30249             {
30250                 \char_generate:nn { "0069 }
30251                 { \char_value_catcode:n { "0069 } }
30252             }
30253             \_text_change_case_loop:nnw {#1} {#3}
30254         }
30255         { \_text_change_case_char:nnnN {#1} {#2} {#3} #4#5 }

```

```

30256     }
30257 }

```

(End definition for `_text_change_case_lower_tr:nnnN` and others.)

`_text_change_case_upper_tr:nnnN`

Uppercasing is easier: just one exception with no context.

```

30258 \cs_new:Npx \_text_change_case_upper_tr:nnnN #1#2#3#4
30259 {
30260   \exp_not:N \int_compare:nNnTF { '#4 } = { "0069 }
30261   {
30262     \bool_lazy_or:nnTF
30263     { \sys_if_engine luatex_p: }
30264     { \sys_if_engine xetex_p: }
30265     {
30266       \exp_not:N \_text_change_case_store:e
30267       {
30268         \exp_not:N \char_generate:nn { "0130 }
30269         { \exp_not:N \_text_char_catcode:N #4 }
30270       }
30271     }
30272     {
30273       \exp_not:N \_text_change_case_store:V
30274       \exp_not:N \c__text_dotted_I_tl
30275     }
30276     \exp_not:N \use:c { \_text_change_case_char_next_ #2 :nn } {#2} {#3}
30277   }
30278   { \exp_not:N \_text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30279 }

```

(End definition for `_text_change_case_upper_tr:nnnN`.)

`_text_change_case_lower_az:nnnN`

Straight copies.

`_text_change_case_upper_az:nnnN`

```

30280 \cs_new_eq:NN \_text_change_case_lower_az:nnnN
30281   \_text_change_case_lower_tr:nnnN
30282 \cs_new_eq:NN \_text_change_case_upper_az:nnnN
30283   \_text_change_case_upper_tr:nnnN

```

(End definition for `_text_change_case_lower_az:nnnN` and `_text_change_case_upper_az:nnnN`.)

48.2 Case changing data for 8-bit engines

`\c__text_dotless_i_tl`

`\c__text_dotted_I_tl`

`\c__text_i_ogonek_tl`

`\c__text_I_ogonek_tl`

`\c__text_grosses_Eszett_tl`

For cases where there is an 8-bit option in the T1 font set up, a variant is provided in both cases.

```

30284 \group_begin:
30285   \bool_lazy_or:nnF
30286   { \sys_if_engine luatex_p: }
30287   { \sys_if_engine xetex_p: }
30288   {
30289     \cs_set_protected:Npn \_text_tmp:w #1#2
30290     {
30291       \group_begin:
30292       \cs_set_protected:Npn \_text_tmp:w ##1##2##3##4
30293       {
30294         \tl_const:Nx #1

```

```

30295         {
30296             \exp_after:wN \exp_after:wN \exp_after:wN
30297             \exp_not:N \char_generate:nn {##1} { 13 }
30298             \exp_after:wN \exp_after:wN \exp_after:wN
30299             \exp_not:N \char_generate:nn {##2} { 13 }
30300             \tl_if_blank:nF {##3}
30301             {
30302                 \exp_after:wN \exp_after:wN \exp_after:wN
30303                 \exp_not:N \char_generate:nn {##3} { 13 }
30304             }
30305         }
30306     }
30307     \use:x
30308     { \__text_tmp:w \char_to_utfviii_bytes:n { "#2 } }
30309 \group_end:
30310 }
30311 \__text_tmp:w \c__text_dotless_i_tl      { 0131 }
30312 \__text_tmp:w \c__text_dotted_I_tl      { 0130 }
30313 \__text_tmp:w \c__text_i_ogonek_tl      { 012F }
30314 \__text_tmp:w \c__text_I_ogonek_tl      { 012E }
30315 \__text_tmp:w \c__text_grosses_Eszett_tl { 1E9E }
30316 }
30317 \group_end:

```

(End definition for \c__text_dotless_i_tl and others.)

For 8-bit engines we now need to define the case-change data for the multi-octet mappings. These need a list of what code points are doable in T1 so the list is hard coded (there's no saving in loading the mappings dynamically). All of the straight-forward ones have two octets, so that is taken as read.

```

30318 \group_begin:
30319 \bool_lazy_or:nnF
30320 { \sys_if_engine_luatex_p: }
30321 { \sys_if_engine_xetex_p: }
30322 {
30323     \cs_set_protected:Npn \__text_loop:nn #1#2
30324     {
30325         \quark_if_recursion_tail_stop:n {#1}
30326         \use:x
30327         {
30328             \__text_tmp:w
30329             \char_to_utfviii_bytes:n { "#1 }
30330             \char_to_utfviii_bytes:n { "#2 }
30331         }
30332         \__text_loop:nn
30333     }
30334     \cs_set_protected:Npn \__text_tmp:nnnn #1#2#3#4#5
30335     {
30336         \tl_const:cx
30337         {
30338             c__text_ #1 case_
30339             \char_generate:nn {#2} { 12 }
30340             \char_generate:nn {#3} { 12 }
30341             _tl
30342         }

```



```

30343         {
30344             \exp_after:wN \exp_after:wN \exp_after:wN
30345             \exp_not:N \char_generate:nn {#4} { 13 }
30346             \exp_after:wN \exp_after:wN \exp_after:wN
30347             \exp_not:N \char_generate:nn {#5} { 13 }
30348         }
30349     }
30350 \cs_set_protected:Npn \__text_tmp:w #1#2#3#4#5#6#7#8
30351 {
30352     \tl_const:cx
30353     {
30354         c__text_lowercase_
30355         \char_generate:nn {#1} { 12 }
30356         \char_generate:nn {#2} { 12 }
30357         _tl
30358     }
30359     {
30360         \exp_after:wN \exp_after:wN \exp_after:wN
30361         \exp_not:N \char_generate:nn {#5} { 13 }
30362         \exp_after:wN \exp_after:wN \exp_after:wN
30363         \exp_not:N \char_generate:nn {#6} { 13 }
30364     }
30365     \__text_tmp:nnnn { upper } {#5} {#6} {#1} {#2}
30366     \__text_tmp:nnnn { title } {#5} {#6} {#1} {#2}
30367 }
30368 \__text_loop:nn
30369 { 00C0 } { 00E0 }
30370 { 00C1 } { 00E1 }
30371 { 00C2 } { 00E2 }
30372 { 00C3 } { 00E3 }
30373 { 00C4 } { 00E4 }
30374 { 00C5 } { 00E5 }
30375 { 00C6 } { 00E6 }
30376 { 00C7 } { 00E7 }
30377 { 00C8 } { 00E8 }
30378 { 00C9 } { 00E9 }
30379 { 00CA } { 00EA }
30380 { 00CB } { 00EB }
30381 { 00CC } { 00EC }
30382 { 00CD } { 00ED }
30383 { 00CE } { 00EE }
30384 { 00CF } { 00EF }
30385 { 00D0 } { 00F0 }
30386 { 00D1 } { 00F1 }
30387 { 00D2 } { 00F2 }
30388 { 00D3 } { 00F3 }
30389 { 00D4 } { 00F4 }
30390 { 00D5 } { 00F5 }
30391 { 00D6 } { 00F6 }
30392 { 00D8 } { 00F8 }
30393 { 00D9 } { 00F9 }
30394 { 00DA } { 00FA }
30395 { 00DB } { 00FB }
30396 { 00DC } { 00FC }

```

30397	{ 00DD }	{ 00FD }
30398	{ 00DE }	{ 00FE }
30399	{ 0100 }	{ 0101 }
30400	{ 0102 }	{ 0103 }
30401	{ 0104 }	{ 0105 }
30402	{ 0106 }	{ 0107 }
30403	{ 0108 }	{ 0109 }
30404	{ 010A }	{ 010B }
30405	{ 010C }	{ 010D }
30406	{ 010E }	{ 010F }
30407	{ 0110 }	{ 0111 }
30408	{ 0112 }	{ 0113 }
30409	{ 0114 }	{ 0115 }
30410	{ 0116 }	{ 0117 }
30411	{ 0118 }	{ 0119 }
30412	{ 011A }	{ 011B }
30413	{ 011C }	{ 011D }
30414	{ 011E }	{ 011F }
30415	{ 0120 }	{ 0121 }
30416	{ 0122 }	{ 0123 }
30417	{ 0124 }	{ 0125 }
30418	{ 0128 }	{ 0129 }
30419	{ 012A }	{ 012B }
30420	{ 012C }	{ 012D }
30421	{ 012E }	{ 012F }
30422	{ 0132 }	{ 0133 }
30423	{ 0134 }	{ 0135 }
30424	{ 0136 }	{ 0137 }
30425	{ 0139 }	{ 013A }
30426	{ 013B }	{ 013C }
30427	{ 013E }	{ 013F }
30428	{ 0141 }	{ 0142 }
30429	{ 0143 }	{ 0144 }
30430	{ 0145 }	{ 0146 }
30431	{ 0147 }	{ 0148 }
30432	{ 014A }	{ 014B }
30433	{ 014C }	{ 014D }
30434	{ 014E }	{ 014F }
30435	{ 0150 }	{ 0151 }
30436	{ 0152 }	{ 0153 }
30437	{ 0154 }	{ 0155 }
30438	{ 0156 }	{ 0157 }
30439	{ 0158 }	{ 0159 }
30440	{ 015A }	{ 015B }
30441	{ 015C }	{ 015D }
30442	{ 015E }	{ 015F }
30443	{ 0160 }	{ 0161 }
30444	{ 0162 }	{ 0163 }
30445	{ 0164 }	{ 0165 }
30446	{ 0168 }	{ 0169 }
30447	{ 016A }	{ 016B }
30448	{ 016C }	{ 016D }
30449	{ 016E }	{ 016F }
30450	{ 0170 }	{ 0171 }

30451	{ 0172 }	{ 0173 }
30452	{ 0174 }	{ 0175 }
30453	{ 0176 }	{ 0177 }
30454	{ 0178 }	{ 00FF }
30455	{ 0179 }	{ 017A }
30456	{ 017B }	{ 017C }
30457	{ 017D }	{ 017E }
30458	{ 01CD }	{ 01CE }
30459	{ 01CF }	{ 01D0 }
30460	{ 01D1 }	{ 01D2 }
30461	{ 01D3 }	{ 01D4 }
30462	{ 01E2 }	{ 01E3 }
30463	{ 01E6 }	{ 01E7 }
30464	{ 01E8 }	{ 01E9 }
30465	{ 01EA }	{ 01EB }
30466	{ 01F4 }	{ 01F5 }
30467	{ 0218 }	{ 0219 }
30468	{ 021A }	{ 021B }

Add T2 (Cyrillic) as this is doable using a classical \MakeUppercase approach.

30469	{ 0400 }	{ 0450 }
30470	{ 0401 }	{ 0451 }
30471	{ 0402 }	{ 0452 }
30472	{ 0403 }	{ 0453 }
30473	{ 0404 }	{ 0454 }
30474	{ 0405 }	{ 0455 }
30475	{ 0406 }	{ 0456 }
30476	{ 0407 }	{ 0457 }
30477	{ 0408 }	{ 0458 }
30478	{ 0409 }	{ 0459 }
30479	{ 040A }	{ 045A }
30480	{ 040B }	{ 045B }
30481	{ 040C }	{ 045C }
30482	{ 040D }	{ 045D }
30483	{ 040E }	{ 045E }
30484	{ 040F }	{ 045F }
30485	{ 0410 }	{ 0430 }
30486	{ 0411 }	{ 0431 }
30487	{ 0412 }	{ 0432 }
30488	{ 0413 }	{ 0433 }
30489	{ 0414 }	{ 0434 }
30490	{ 0415 }	{ 0435 }
30491	{ 0416 }	{ 0436 }
30492	{ 0417 }	{ 0437 }
30493	{ 0418 }	{ 0438 }
30494	{ 0419 }	{ 0439 }
30495	{ 041A }	{ 043A }
30496	{ 041B }	{ 043B }
30497	{ 041C }	{ 043C }
30498	{ 041D }	{ 043D }
30499	{ 041E }	{ 043E }
30500	{ 041F }	{ 043F }
30501	{ 0420 }	{ 0440 }
30502	{ 0421 }	{ 0441 }
30503	{ 0422 }	{ 0442 }

```

30504      { 0423 } { 0443 }
30505      { 0424 } { 0444 }
30506      { 0425 } { 0445 }
30507      { 0426 } { 0446 }
30508      { 0427 } { 0447 }
30509      { 0428 } { 0448 }
30510      { 0429 } { 0449 }
30511      { 042A } { 044A }
30512      { 042B } { 044B }
30513      { 042C } { 044C }
30514      { 042D } { 044D }
30515      { 042E } { 044E }
30516      { 042F } { 044F }

```

Core Greek support: there may need to be a little more work here to deal completely with accents.

```

30517      { 0391 } { 03B1 }
30518      { 0392 } { 03B2 }
30519      { 0393 } { 03B3 }
30520      { 0394 } { 03B4 }
30521      { 0395 } { 03B5 }
30522      { 0396 } { 03B6 }
30523      { 0397 } { 03B7 }
30524      { 0398 } { 03B8 }
30525      { 0399 } { 03B9 }
30526      { 039A } { 03BA }
30527      { 039B } { 03BB }
30528      { 039C } { 03BC }
30529      { 039D } { 03BD }
30530      { 039E } { 03BE }
30531      { 039F } { 03BF }
30532      { 03A0 } { 03C0 }
30533      { 03A1 } { 03C1 }
30534      { 03A3 } { 03C3 }
30535      { 03A4 } { 03C4 }
30536      { 03A5 } { 03C5 }
30537      { 03A6 } { 03C6 }
30538      { 03A7 } { 03C7 }
30539      { 03A8 } { 03C8 }
30540      { 03A9 } { 03C9 }
30541      { 03D8 } { 03D9 }
30542      { 03DA } { 03DB }
30543      { 03DC } { 03DD }
30544      { 03DE } { 03DF }
30545      { 03E0 } { 03E1 }
30546      \q_recursion_tail ?
30547      \q_recursion_stop
30548      \cs_set_protected:Npn \__text_tmp:w #1#2#3
30549      {
30550          \group_begin:
30551              \cs_set_protected:Npn \__text_tmp:w ##1##2##3##4
30552              {
30553                  \tl_const:cx
30554                  {

```

```

30555         c__text_ #3 case_
30556         \char_generate:nn {##1} { 12 }
30557         \char_generate:nn {##2} { 12 }
30558         _tl
30559     }
30560     {#2}
30561 }
30562 \use:x
30563 { \__text_tmp:w \char_to_utfviii_bytes:n { "#1 } }
30564 \group_end:
30565 }
30566 \__text_tmp:w { 00DF } { SS } { upper }
30567 \__text_tmp:w { 00DF } { Ss } { title }
30568 \__text_tmp:w { 0131 } { I } { upper }
30569 }
30570 \group_end:

```

The (fixed) look-up mappings for letter-like control sequences.

```

30571 \group_begin:
30572 \cs_set_protected:Npn \__text_change_case_setup:NN #1#2
30573 {
30574     \quark_if_recursion_tail_stop:N #1
30575     \tl_const:cn { c__text_lowercase_ \token_to_str:N #1 _tl }
30576     { #2 }
30577     \tl_const:cn { c__text_uppercase_ \token_to_str:N #2 _tl }
30578     { #1 }
30579     \__text_change_case_setup:NN
30580 }
30581 \__text_change_case_setup:NN
30582 \AA \aa
30583 \AE \ae
30584 \DH \dh
30585 \DJ \dj
30586 \IJ \ij
30587 \L \l
30588 \NG \ng
30589 \O \o
30590 \OE \oe
30591 \SS \ss
30592 \TH \th
30593 \q_recursion_tail ?
30594 \q_recursion_stop
30595 \tl_const:cn { c__text_uppercase_ \token_to_str:N \i _tl } { I }
30596 \tl_const:cn { c__text_uppercase_ \token_to_str:N \j _tl } { J }
30597 \group_end:

```

To deal with possible encoding-specific extensions to `\@uclclist`, we check at the end of the preamble. This will therefore only apply to L^AT_EX 2_ε package mode.

```

30598 \cs_if_exist:cT { \@uclclist }
30599 {
30600     \AtBeginDocument
30601     {
30602         \group_begin:
30603         \cs_set_protected:Npn \__text_change_case_setup:Nn #1#2
30604         {

```

```

30605         \quark_if_recursion_tail_stop:N #1
30606         \tl_if_single_token:nT {#2}
30607         {
30608             \cs_if_exist:cF
30609             { c__text_uppercase_ \token_to_str:N #1 _tl }
30610             {
30611                 \tl_const:cn
30612                 { c__text_uppercase_ \token_to_str:N #1 _tl }
30613                 { #2 }
30614             }
30615             \cs_if_exist:cF
30616             { c__text_lowercase_ \token_to_str:N #2 _tl }
30617             {
30618                 \tl_const:cn
30619                 { c__text_lowercase_ \token_to_str:N #2 _tl }
30620                 { #1 }
30621             }
30622         }
30623         \__text_change_case_setup:Nn
30624     }
30625     \exp_after:wN \__text_change_case_setup:Nn \@uclclist
30626     \q_recursion_tail ?
30627     \q_recursion_stop
30628 \group_end:
30629 }
30630 }
30631 \</package>

```

49 l3text-purify implementation

```

30632 \*package>
30633 \@@=text>

```

49.1 Purifying text

_text_if_recursion_tail_stop:N Functions to query recursion quarks.

```
30634 \__kernel_quark_new_test:N \__text_if_recursion_tail_stop:N
```

(End definition for __text_if_recursion_tail_stop:N.)

\text_purify:n

As in the other parts of the module, we start off with a standard “action” loop, with expansion applied up-front.

```

\__text_purify:n
\__text_purify_store:n
\__text_purify_store:nw
\__text_purify_end:w
\__text_purify_loop:w
\__text_purify_group:n
\__text_purify_space:w
\__text_purify_N_type:N
\__text_purify_N_type_aux:N
\__text_purify_math_search:NNN
\__text_purify_math_start:NNw
\__text_purify_math_store:n
\__text_purify_math_store:nw
\__text_purify_math_end:w
\__text_purify_math_loop:NNw
\__text_purify_math_N_type:NNN
\__text_purify_math_group:NNn
\__text_purify_math_space:NNw
\__text_purify_math_cmd:N
\__text_purify_math_cmd:NN
\__text_purify_math_cmd:Nn

```

```

30635 \cs_new:Npn \text_purify:n #1
30636 {
30637     \__kernel_exp_not:w \exp_after:wN
30638     {
30639         \exp:w
30640         \exp_args:Ne \__text_purify:n
30641         { \text_expand:n {#1} }
30642     }
30643 }
30644 \cs_new:Npn \__text_purify:n #1
30645 {

```

```

30646 \group_align_safe_begin:
30647 \__text_purify_loop:w #1
30648 \q__text_recursion_tail \q__text_recursion_stop
30649 \__text_purify_result:n { }
30650 }

```

As for expansion, collect up the tokens for future use.

```

30651 \cs_new:Npn \__text_purify_store:n #1
30652 { \__text_purify_store:nw {#1} }
30653 \cs_new:Npn \__text_purify_store:nw #1#2 \__text_purify_result:n #3
30654 { #2 \__text_purify_result:n { #3 #1 } }
30655 \cs_new:Npn \__text_purify_end:w #1 \__text_purify_result:n #2
30656 {
30657 \group_align_safe_end:
30658 \exp_end:
30659 #2
30660 }

```

The main loop is a standard “tl action”. Unlike the expansion or case changing, here any groups have to be run inline. Most of the business end is as before in the N-type token processing.

```

30661 \cs_new:Npn \__text_purify_loop:w #1 \q__text_recursion_stop
30662 {
30663 \tl_if_head_is_N_type:nTF {#1}
30664 { \__text_purify_N_type:N }
30665 {
30666 \tl_if_head_is_group:nTF {#1}
30667 { \__text_purify_group:n }
30668 { \__text_purify_space:w }
30669 }
30670 #1 \q__text_recursion_stop
30671 }
30672 \cs_new:Npn \__text_purify_group:n #1 { \__text_purify_loop:w #1 }
30673 \exp_last_unbraced:NNo \cs_new:Npn \__text_purify_space:w \c_space_tl
30674 {
30675 \__text_purify_store:n { ~ }
30676 \__text_purify_loop:w
30677 }

```

The first part of handling math mode is exactly the same as in the other functions: look for a start-of-math mode token and if found start a new loop tracking the closing token.

```

30678 \cs_new:Npn \__text_purify_N_type:N #1
30679 {
30680 \__text_if_recursion_tail_stop_do:Nn #1 { \__text_purify_end:w }
30681 \__text_purify_N_type_aux:N #1
30682 }
30683 \cs_new:Npn \__text_purify_N_type_aux:N #1
30684 {
30685 \exp_after:wN \__text_purify_math_search:NNN
30686 \exp_after:wN #1 \l_text_math_delims_tl
30687 \q__text_recursion_tail ?
30688 \q__text_recursion_stop
30689 }
30690 \cs_new:Npn \__text_purify_math_search:NNN #1#2#3
30691 {

```

```

30692     \_text_if_recursion_tail_stop_do:Nn #2
30693     { \_text_purify_math_cmd:N #1 }
30694     \token_if_eq_meaning:NNTF #1 #2
30695     {
30696         \_text_use_i_delimit_by_q_recursion_stop:nw
30697         { \_text_purify_math_start:NNw #2 #3 }
30698     }
30699     { \_text_purify_math_search:NNN #1 }
30700 }
30701 \cs_new:Npn \_text_purify_math_start:NNw #1#2#3 \q_text_recursion_stop
30702 {
30703     \_text_purify_math_loop:NNw #1#2#3 \q_text_recursion_stop
30704     \_text_purify_math_result:n { }
30705 }
30706 \cs_new:Npn \_text_purify_math_store:n #1
30707 { \_text_purify_math_store:nw {#1} }
30708 \cs_new:Npn \_text_purify_math_store:nw #1#2 \_text_purify_math_result:n #3
30709 { #2 \_text_purify_math_result:n { #3 #1 } }
30710 \cs_new:Npn \_text_purify_math_end:w #1 \_text_purify_math_result:n #2
30711 {
30712     \_text_purify_store:n { $ #2 $ }
30713     \_text_purify_loop:w #1
30714 }
30715 \cs_new:Npn \_text_purify_math_stop:Nw #1 \_text_purify_math_result:n #2
30716 {
30717     \_text_purify_store:n {#1#2}
30718     \_text_purify_end:w
30719 }
30720 \cs_new:Npn \_text_purify_math_loop:NNw #1#2#3 \q_text_recursion_stop
30721 {
30722     \tl_if_head_is_N_type:nTF {#3}
30723     { \_text_purify_math_N_type:NNN }
30724     {
30725         \tl_if_head_is_group:nTF {#3}
30726         { \_text_purify_math_group:NNn }
30727         { \_text_purify_math_space:NNw }
30728     }
30729     #1#2#3 \q_text_recursion_stop
30730 }
30731 \cs_new:Npn \_text_purify_math_N_type:NNN #1#2#3
30732 {
30733     \_text_if_recursion_tail_stop_do:Nn #3
30734     { \_text_purify_math_stop:Nw #1 }
30735     \token_if_eq_meaning:NNTF #3 #2
30736     { \_text_purify_math_end:w }
30737     {
30738         \_text_purify_math_store:n {#3}
30739         \_text_purify_math_loop:NNw #1#2
30740     }
30741 }
30742 \cs_new:Npn \_text_purify_math_group:NNn #1#2#3
30743 {
30744     \_text_purify_math_store:n { {#3} }
30745     \_text_purify_math_loop:NNw #1#2

```



```

30746 }
30747 \exp_after:wN \cs_new:Npn \exp_after:wN \__text_purify_math_space:NNw
30748 \exp_after:wN # \exp_after:wN 1
30749 \exp_after:wN # \exp_after:wN 2 \c_space_tl
30750 {
30751   \__text_purify_math_store:n { ~ }
30752   \__text_purify_math_loop:NNw #1#2
30753 }

```

Then handle math mode as an argument: same outcomes, different input syntax.

```

30754 \cs_new:Npn \__text_purify_math_cmd:N #1
30755 {
30756   \exp_after:wN \__text_purify_math_cmd:NN \exp_after:wN #1
30757   \l_text_math_arg_tl \q__text_recursion_tail \q__text_recursion_stop
30758 }
30759 \cs_new:Npn \__text_purify_math_cmd:NN #1#2
30760 {
30761   \__text_if_recursion_tail_stop_do:Nn #2
30762   { \__text_purify_replace:N #1 }
30763   \cs_if_eq:NNTF #2 #1
30764   {
30765     \__text_use_i_delimit_by_q_recursion_stop:nw
30766     { \__text_purify_math_cmd:n }
30767   }
30768   { \__text_purify_math_cmd:NN #1 }
30769 }
30770 \cs_new:Npn \__text_purify_math_cmd:n #1
30771 { \__text_purify_math_end:w \__text_purify_math_result:n {#1} }

```

For N-type tokens, we first look for a string-context replacement before anything else: this can therefore cover anything. Assuming we don't find one, check to see if we can expand control sequences: if not, they have to be dropped. We also allow for L^AT_EX 2_ε \protect: there's an assumption that we don't have \protect { \oops } or similar, but that's also in the expansion code and seems like a reasonable balance.

```

30772 \cs_new:Npn \__text_purify_replace:N #1
30773 {
30774   \bool_lazy_and:nnTF
30775   { \cs_if_exist_p:c { l__text_purify_ \token_to_str:N #1 _tl } }
30776   {
30777     \bool_lazy_or_p:nn
30778     { \token_if_cs_p:N #1 }
30779     { \token_if_active_p:N #1 }
30780   }
30781   {
30782     \exp_args:Nv \__text_purify_replace:n
30783     { l__text_purify_ \token_to_str:N #1 _tl }
30784   }
30785   {
30786     \token_if_cs:NTF #1
30787     { \__text_purify_expand:N #1 }
30788     {
30789       \exp_args:Ne \__text_purify_store:n
30790       { \__text_token_to_explicit:N #1 }
30791       \__text_purify_loop:w
30792     }
30793   }
30794 }

```

```

30793     }
30794   }
30795   \cs_new:Npn \__text_purify_replace:n #1 { \__text_purify_loop:w #1 }
30796   \cs_new:Npn \__text_purify_expand:N #1
30797   {
30798     \str_if_eq:nnTF {#1} { \protect }
30799     { \__text_purify_protect:N }
30800     {
30801       \__text_if_expandable:NTF #1
30802       { \exp_after:wN \__text_purify_loop:w #1 }
30803       { \__text_purify_loop:w }
30804     }
30805   }
30806   \cs_new:Npn \__text_purify_protect:N #1
30807   {
30808     \__text_if_recursion_tail_stop_do:Nn #1 { \__text_purify_end:w }
30809     \__text_purify_loop:w
30810   }

```

(End definition for `\text_purify:n` and others. This function is documented on page 263.)

`\text_declare_purify_equivalent:Nn`

`\text_declare_purify_equivalent:Nx`

```

30811 \cs_new_protected:Npn \text_declare_purify_equivalent:Nn #1#2
30812 {
30813   \tl_clear_new:c { l__text_purify_ \token_to_str:N #1 _tl }
30814   \tl_set:cn { l__text_purify_ \token_to_str:N #1 _tl } {#2}
30815 }
30816 \cs_generate_variant:Nn \text_declare_purify_equivalent:Nn { Nx }

```

(End definition for `\text_declare_purify_equivalent:Nn`. This function is documented on page 263.)

Now pre-define a range of standard commands that need dedicated definitions in purified text. First handle font-related stuff: all of this needs to be disabled.

```

30817 \tl_map_inline:nn
30818 {
30819   \fontencoding
30820   \fontfamily
30821   \fontseries
30822   \fontshape
30823 }
30824 { \text_declare_purify_equivalent:Nn #1 { \use_none:n } }
30825 \text_declare_purify_equivalent:Nn \fontsize { \use_none:n }
30826 \text_declare_purify_equivalent:Nn \selectfont { }
30827 \text_declare_purify_equivalent:Nn \usefont { \use_none:n }
30828 \tl_map_inline:nn
30829 {
30830   \emph
30831   \text
30832   \textnormal
30833   \textrm
30834   \textsf
30835   \texttt
30836   \textbf
30837   \textmd
30838   \textit

```

```

30839     \textsl
30840     \textup
30841     \textsc
30842     \textulc
30843   }
30844   { \text_declare_purify_equivalent:Nn #1 { \use:n } }
30845 \tl_map_inline:nn
30846 {
30847   \normalfont
30848   \rmfamily
30849   \sffamily
30850   \ttfamily
30851   \bfseries
30852   \mdseries
30853   \itshape
30854   \scshape
30855   \slshape
30856   \upshape
30857   \em
30858   \Huge
30859   \LARGE
30860   \Large
30861   \footnotesize
30862   \huge
30863   \large
30864   \normalsize
30865   \scriptsize
30866   \small
30867   \tiny
30868 }
30869 { \text_declare_purify_equivalent:Nn #1 { } }

```

Environments have to be handled by pure expansion.

```

30870 \text_declare_purify_equivalent:Nn \begin { \use:c }
30871 \text_declare_purify_equivalent:Nn \end { \use:c }

```

Some common symbols and similar ideas.

```

30872 \text_declare_purify_equivalent:Nn \\\ { }
30873 \tl_map_inline:nn
30874 { \{ \} \# \$ \% \_ }
30875 { \text_declare_purify_equivalent:Nx #1 { \cs_to_str:N #1 } }

```

Cross-referencing.

```

30876 \text_declare_purify_equivalent:Nn \label { \use_none:n }

```

Spaces.

```

30877 \group_begin:
30878 \char_set_catcode_active:N \~
30879 \use:n
30880 {
30881   \group_end:
30882   \text_declare_purify_equivalent:Nx ~ { \c_space_tl }
30883 }
30884 \text_declare_purify_equivalent:Nn \nobreakspace { ~ }
30885 \text_declare_purify_equivalent:Nn \ { ~ }
30886 \text_declare_purify_equivalent:Nn \, { ~ }

```

49.2 Accent and letter-like data for purifying text

In contrast to case changing, both 8-bit and Unicode engines need information for text purification to handle accents and letter-like functions: these all need to be removed. However, the results are of course engine-dependent.

For the letter-like commands, life is relatively easy: they are all simply added as standard exceptions. The only oddity is `\SS`, which gets converted to two letters. (At some stage an alternative version can presumably be added to `babel` or similar.)

```

30887 \bool_lazy_or:nnTF
30888 { \sys_if_engine_luatex_p: }
30889 { \sys_if_engine_xetex_p: }
30890 {
30891   \cs_set_protected:Npn \__text_loop:Nn #1#2
30892   {
30893     \quark_if_recursion_tail_stop:N #1
30894     \text_declare_purify_equivalent:Nx #1
30895     {
30896       \char_generate:nn { "#2 }
30897       { \char_value_catcode:n { "#2 } }
30898     }
30899     \__text_loop:Nn
30900   }
30901 }
30902 {
30903   \cs_set_protected:Npn \__text_loop:Nn #1#2
30904   {
30905     \quark_if_recursion_tail_stop:N #1
30906     \text_declare_purify_equivalent:Nx #1
30907     {
30908       \exp_args:Ne \__text_tmp:n
30909       { \char_to_utfviii_bytes:n { "#2 } }
30910     }
30911     \__text_loop:Nn
30912   }
30913   \cs_set:Npn \__text_tmp:n #1 { \__text_tmp:nnnn #1 }
30914   \cs_set:Npn \__text_tmp:nnnn #1#2#3#4
30915   {
30916     \exp_after:wN \exp_after:wN \exp_after:wN
30917     \exp_not:N \char_generate:nn {#1} { 13 }
30918     \exp_after:wN \exp_after:wN \exp_after:wN
30919     \exp_not:N \char_generate:nn {#2} { 13 }
30920   }
30921 }
30922 \__text_loop:Nn
30923 \AA { 00C5 }
30924 \AE { 00C6 }
30925 \DH { 00D0 }
30926 \DJ { 0110 }
30927 \IJ { 0132 }
30928 \L { 0141 }
30929 \NG { 014A }
30930 \O { 00D8 }
30931 \OE { 0152 }
30932 \TH { 00DE }

```

```

30933 \aa { 00E5 }
30934 \ae { 00E6 }
30935 \dh { 00F0 }
30936 \dj { 0111 }
30937 \i { 0131 }
30938 \j { 0237 }
30939 \ij { 0132 }
30940 \l { 0142 }
30941 \ng { 014B }
30942 \o { 00F8 }
30943 \oe { 0153 }
30944 \ss { 00DF }
30945 \th { 00FE }
30946 \q_recursion_tail ?
30947 \q_recursion_stop
30948 \text_declare_purify_equivalent:Nn \SS { SS }

```

`__text_purify_accent:NN` Accent LICR handling is a little more complex. Accents may exist as pre-composed codepoints or as independent glyphs. The former are all saved as single token lists, whilst for the latter the combining accent needs to be re-ordered compared to the character it applies to.

```

30949 \cs_new:Npn \__text_purify_accent:NN #1#2
30950 {
30951   \cs_if_exist:cTF
30952     { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
30953     {
30954       \exp_not:v
30955         { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
30956     }
30957     {
30958       \exp_not:n {#2}
30959       \exp_not:v { c__text_purify_ \token_to_str:N #1 _tl }
30960     }
30961 }
30962 \tl_map_inline:Nn \l_text_accents_tl
30963 { \text_declare_purify_equivalent:Nn #1 { \__text_purify_accent:NN #1 } }

```

First set up the combining accents.

```

30964 \group_begin:
30965   \cs_set_protected:Npn \__text_loop:Nn #1#2
30966   {
30967     \quark_if_recursion_tail_stop:N #1
30968     \tl_const:cx { c__text_purify_ \token_to_str:N #1 _tl }
30969     { \__text_tmp:n {#2} }
30970     \__text_loop:Nn
30971   }
30972   \bool_lazy_or:nnTF
30973     { \sys_if_engine luatex_p: }
30974     { \sys_if_engine xetex_p: }
30975     {
30976       \cs_set:Npn \__text_tmp:n #1
30977       {
30978         \char_generate:nn { "#1 }
30979         { \char_value_catcode:n { "#1 } }

```

```

30980     }
30981   }
30982   {
30983     \cs_set:Npn \__text_tmp:n #1
30984     {
30985       \exp_args:Ne \__text_tmp_aux:n
30986       { \char_to_utfviii_bytes:n { "#1 } }
30987     }
30988     \cs_set:Npn \__text_tmp_aux:n #1 { \__text_tmp:nnnn #1 }
30989     \cs_set:Npn \__text_tmp:nnnn #1#2#3#4
30990     {
30991       \exp_after:wN \exp_after:wN \exp_after:wN
30992       \exp_not:N \char_generate:nn {#1} { 13 }
30993       \exp_after:wN \exp_after:wN \exp_after:wN
30994       \exp_not:N \char_generate:nn {#2} { 13 }
30995     }
30996   }
30997   \__text_loop:Nn
30998   \‘ { 0300 }
30999   \’ { 0301 }
31000   \^ { 0302 }
31001   \~ { 0303 }
31002   \= { 0304 }
31003   \u { 0306 }
31004   \. { 0307 }
31005   \" { 0308 }
31006   \r { 030A }
31007   \H { 030B }
31008   \v { 030C }
31009   \d { 0323 }
31010   \c { 0327 }
31011   \k { 0328 }
31012   \b { 0331 }
31013   \t { 0361 }
31014   \q_recursion_tail { }
31015   \q_recursion_stop

```

Now we handle the pre-composed accents: the list here is taken from `puenc.def`. All of the precomposed cases take a single letter as their second argument. We do not try to cover the case where an accent is added to a “real” dotless-i or -j, or a æ/Æ. Rather, we assume that if the UTF-8 character is used, it will have the real accent character too.

```

31016   \cs_set_protected:Npn \__text_loop:NNn #1#2#3
31017   {
31018     \quark_if_recursion_tail_stop:N #1
31019     \tl_const:cx
31020     { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
31021     { \__text_tmp:n {#3} }
31022     \__text_loop:NNn
31023   }
31024   \__text_loop:NNn
31025   \‘ A { 00C0 }
31026   \’ A { 00C1 }
31027   \^ A { 00C2 }
31028   \~ A { 00C3 }

```

31029	\ " A	{ 00C4 }
31030	\ r A	{ 00C5 }
31031	\ c C	{ 00C7 }
31032	\ ' E	{ 00C8 }
31033	\ ' E	{ 00C9 }
31034	\ ^ E	{ 00CA }
31035	\ " E	{ 00CB }
31036	\ ' I	{ 00CC }
31037	\ ' I	{ 00CD }
31038	\ ^ I	{ 00CE }
31039	\ " I	{ 00CF }
31040	\ ~ N	{ 00D1 }
31041	\ ' O	{ 00D2 }
31042	\ ' O	{ 00D3 }
31043	\ ^ O	{ 00D4 }
31044	\ ~ O	{ 00D5 }
31045	\ " O	{ 00D6 }
31046	\ ' U	{ 00D9 }
31047	\ ' U	{ 00DA }
31048	\ ^ U	{ 00DB }
31049	\ " U	{ 00DC }
31050	\ ' Y	{ 00DD }
31051	\ ' a	{ 00E0 }
31052	\ ' a	{ 00E1 }
31053	\ ^ a	{ 00E2 }
31054	\ ~ a	{ 00E3 }
31055	\ " a	{ 00E4 }
31056	\ r a	{ 00E5 }
31057	\ c c	{ 00E7 }
31058	\ ' e	{ 00E8 }
31059	\ ' e	{ 00E9 }
31060	\ ^ e	{ 00EA }
31061	\ " e	{ 00EB }
31062	\ ' i	{ 00EC }
31063	\ ' \i	{ 00EC }
31064	\ ' i	{ 00ED }
31065	\ ' \i	{ 00ED }
31066	\ ^ i	{ 00EE }
31067	\ ^ \i	{ 00EE }
31068	\ " i	{ 00EF }
31069	\ " \i	{ 00EF }
31070	\ ~ n	{ 00F1 }
31071	\ ' o	{ 00F2 }
31072	\ ' o	{ 00F3 }
31073	\ ^ o	{ 00F4 }
31074	\ ~ o	{ 00F5 }
31075	\ " o	{ 00F6 }
31076	\ ' u	{ 00F9 }
31077	\ ' u	{ 00FA }
31078	\ ^ u	{ 00FB }
31079	\ " u	{ 00FC }
31080	\ ' y	{ 00FD }
31081	\ " y	{ 00FF }
31082	\ = A	{ 0100 }

31083	\= a	{ 0101 }
31084	\u A	{ 0102 }
31085	\u a	{ 0103 }
31086	\k A	{ 0104 }
31087	\k a	{ 0105 }
31088	\' C	{ 0106 }
31089	\' c	{ 0107 }
31090	\^ C	{ 0108 }
31091	\^ c	{ 0109 }
31092	\. C	{ 010A }
31093	\. c	{ 010B }
31094	\v C	{ 010C }
31095	\v c	{ 010D }
31096	\v D	{ 010E }
31097	\v d	{ 010F }
31098	\= E	{ 0112 }
31099	\= e	{ 0113 }
31100	\u E	{ 0114 }
31101	\u e	{ 0115 }
31102	\. E	{ 0116 }
31103	\. e	{ 0117 }
31104	\k E	{ 0118 }
31105	\k e	{ 0119 }
31106	\v E	{ 011A }
31107	\v e	{ 011B }
31108	\^ G	{ 011C }
31109	\^ g	{ 011D }
31110	\u G	{ 011E }
31111	\u g	{ 011F }
31112	\. G	{ 0120 }
31113	\. g	{ 0121 }
31114	\c G	{ 0122 }
31115	\c g	{ 0123 }
31116	\^ H	{ 0124 }
31117	\^ h	{ 0125 }
31118	\~ I	{ 0128 }
31119	\~ i	{ 0129 }
31120	\~ \i	{ 0129 }
31121	\= I	{ 012A }
31122	\= i	{ 012B }
31123	\= \i	{ 012B }
31124	\u I	{ 012C }
31125	\u i	{ 012D }
31126	\u \i	{ 012D }
31127	\k I	{ 012E }
31128	\k i	{ 012F }
31129	\k \i	{ 012F }
31130	\. I	{ 0130 }
31131	\^ J	{ 0134 }
31132	\^ j	{ 0135 }
31133	\^ \j	{ 0135 }
31134	\c K	{ 0136 }
31135	\c k	{ 0137 }
31136	\' L	{ 0139 }

31137	\' l	{ 013A }
31138	\c L	{ 013B }
31139	\c l	{ 013C }
31140	\v L	{ 013D }
31141	\v l	{ 013E }
31142	\. L	{ 013F }
31143	\. l	{ 0140 }
31144	\' N	{ 0143 }
31145	\' n	{ 0144 }
31146	\c N	{ 0145 }
31147	\c n	{ 0146 }
31148	\v N	{ 0147 }
31149	\v n	{ 0148 }
31150	\= O	{ 014C }
31151	\= o	{ 014D }
31152	\u O	{ 014E }
31153	\u o	{ 014F }
31154	\H O	{ 0150 }
31155	\H o	{ 0151 }
31156	\' R	{ 0154 }
31157	\' r	{ 0155 }
31158	\c R	{ 0156 }
31159	\c r	{ 0157 }
31160	\v R	{ 0158 }
31161	\v r	{ 0159 }
31162	\' S	{ 015A }
31163	\' s	{ 015B }
31164	\^ S	{ 015C }
31165	\^ s	{ 015D }
31166	\c S	{ 015E }
31167	\c s	{ 015F }
31168	\v S	{ 0160 }
31169	\v s	{ 0161 }
31170	\c T	{ 0162 }
31171	\c t	{ 0163 }
31172	\v T	{ 0164 }
31173	\v t	{ 0165 }
31174	\~ U	{ 0168 }
31175	\~ u	{ 0169 }
31176	\= U	{ 016A }
31177	\= u	{ 016B }
31178	\u U	{ 016C }
31179	\u u	{ 016D }
31180	\r U	{ 016E }
31181	\r u	{ 016F }
31182	\H U	{ 0170 }
31183	\H u	{ 0171 }
31184	\k U	{ 0172 }
31185	\k u	{ 0173 }
31186	\^ W	{ 0174 }
31187	\^ w	{ 0175 }
31188	\^ Y	{ 0176 }
31189	\^ y	{ 0177 }
31190	\" Y	{ 0178 }

```

31191      \' Z { 0179 }
31192      \' z { 017A }
31193      \. Z { 017B }
31194      \. z { 017C }
31195      \v Z { 017D }
31196      \v z { 017E }
31197      \v A { 01CD }
31198      \v a { 01CE }
31199      \v I { 01CF }
31200      \v \i { 01D0 }
31201      \v i { 01D0 }
31202      \v O { 01D1 }
31203      \v o { 01D2 }
31204      \v U { 01D3 }
31205      \v u { 01D4 }
31206      \v G { 01E6 }
31207      \v g { 01E7 }
31208      \v K { 01E8 }
31209      \v k { 01E9 }
31210      \k O { 01EA }
31211      \k o { 01EB }
31212      \v \j { 01F0 }
31213      \v j { 01F0 }
31214      \' G { 01F4 }
31215      \' g { 01F5 }
31216      \' N { 01F8 }
31217      \' n { 01F9 }
31218      \' \AE { 01FC }
31219      \' \ae { 01FD }
31220      \' \O { 01FE }
31221      \' \o { 01FF }
31222      \v H { 021E }
31223      \v h { 021F }
31224      \. A { 0226 }
31225      \. a { 0227 }
31226      \c E { 0228 }
31227      \c e { 0229 }
31228      \. O { 022E }
31229      \. o { 022F }
31230      \= Y { 0232 }
31231      \= y { 0233 }
31232      \q_recursion_tail ? { }
31233      \q_recursion_stop
31234 \group_end:

```

(End definition for _text_purify_accent:NN.)

```

31235 \</package>

```

50 l3legacy Implementation

```

31236 \<*package>
31237 \<@@=legacy>

```

```

\legacy_if_p:n A friendly wrapper.
\legacy_if:nTF
31238 \prg_new_conditional:Npnn \legacy_if:n #1 { p , T , F , TF }
31239 {
31240     \exp_args:Nc \if_meaning:w { if#1 } \iftrue
31241     \prg_return_true:
31242     \else:
31243     \prg_return_false:
31244     \fi:
31245 }

(End definition for \legacy_if:nTF. This function is documented on page 264.)

31246 \>/package>

```

51 l3candidates Implementation

```

31247 \>*package>

```

51.1 Additions to l3box

```

31248 \>@@=box>

```

51.1.1 Viewing part of a box

```

\box_clip:N A wrapper around the driver-dependent code.
\box_clip:c
\box_gclip:N
\box_gclip:c
31249 \cs_new_protected:Npn \box_clip:N #1
31250 { \hbox_set:Nn #1 { \_box_backend_clip:N #1 } }
31251 \cs_generate_variant:Nn \box_clip:N { c }
31252 \cs_new_protected:Npn \box_gclip:N #1
31253 { \hbox_gset:Nn #1 { \_box_backend_clip:N #1 } }
31254 \cs_generate_variant:Nn \box_gclip:N { c }

```

(End definition for `\box_clip:N` and `\box_gclip:N`. These functions are documented on page 265.)

```

\box_set_trim:Nnnnn Trimming from the left- and right-hand edges of the box is easy: kern the appropriate
\box_set_trim:cnnnn parts off each side.
\box_gset_trim:Nnnnn
\box_gset_trim:cnnnn
\_box_set_trim:NnnnnN
31255 \cs_new_protected:Npn \box_set_trim:Nnnnn #1#2#3#4#5
31256 { \_box_set_trim:NnnnnN #1 {#2} {#3} {#4} {#5} \box_set_eq:NN }
31257 \cs_generate_variant:Nn \box_set_trim:Nnnnn { c }
31258 \cs_new_protected:Npn \box_gset_trim:Nnnnn #1#2#3#4#5
31259 { \_box_set_trim:NnnnnN #1 {#2} {#3} {#4} {#5} \box_gset_eq:NN }
31260 \cs_generate_variant:Nn \box_gset_trim:Nnnnn { c }
31261 \cs_new_protected:Npn \_box_set_trim:NnnnnN #1#2#3#4#5#6
31262 {
31263     \hbox_set:Nn \l\_box_internal_box
31264     {
31265         \tex_kern:D - \_box_dim_eval:n {#2}
31266         \box_use:N #1
31267         \tex_kern:D - \_box_dim_eval:n {#4}
31268     }

```

For the height and depth, there is a need to watch the baseline is respected. Material always has to stay on the correct side, so trimming has to check that there is enough material to trim. First, the bottom edge. If there is enough depth, simply set the depth, or if not move down so the result is zero depth. `\box_move_down:nn` is used in both

cases so the resulting box always contains a `\lower` primitive. The internal box is used here as it allows safe use of `\box_set_dp:Nn`.

```

31269 \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
31270 {
31271   \hbox_set:Nn \l__box_internal_box
31272   {
31273     \box_move_down:nn \c_zero_dim
31274     { \box_use_drop:N \l__box_internal_box }
31275   }
31276   \box_set_dp:Nn \l__box_internal_box { \box_dp:N #1 - (#3) }
31277 }
31278 {
31279   \hbox_set:Nn \l__box_internal_box
31280   {
31281     \box_move_down:nn { (#3) - \box_dp:N #1 }
31282     { \box_use_drop:N \l__box_internal_box }
31283   }
31284   \box_set_dp:Nn \l__box_internal_box \c_zero_dim
31285 }

```

Same thing, this time from the top of the box.

```

31286 \dim_compare:nNnTF { \box_ht:N \l__box_internal_box } > {#5}
31287 {
31288   \hbox_set:Nn \l__box_internal_box
31289   {
31290     \box_move_up:nn \c_zero_dim
31291     { \box_use_drop:N \l__box_internal_box }
31292   }
31293   \box_set_ht:Nn \l__box_internal_box
31294   { \box_ht:N \l__box_internal_box - (#5) }
31295 }
31296 {
31297   \hbox_set:Nn \l__box_internal_box
31298   {
31299     \box_move_up:nn { (#5) - \box_ht:N \l__box_internal_box }
31300     { \box_use_drop:N \l__box_internal_box }
31301   }
31302   \box_set_ht:Nn \l__box_internal_box \c_zero_dim
31303 }
31304 #6 #1 \l__box_internal_box
31305 }

```

(End definition for `\box_set_trim:Nnnnn`, `\box_gset_trim:Nnnnn`, and `__box_set_trim:NnnnnN`. These functions are documented on page 266.)

`\box_set_viewport:Nnnnn`
`\box_set_viewport:cnnnn`
`\box_gset_viewport:Nnnnn`
`\box_gset_viewport:cnnnn`
`__box_viewport:NnnnnN`

The same general logic as for the trim operation, but with absolute dimensions. As a result, there are some things to watch out for in the vertical direction.

```

31306 \cs_new_protected:Npn \box_set_viewport:Nnnnn #1#2#3#4#5
31307 { \__box_set_viewport:NnnnnN #1 {#2} {#3} {#4} {#5} \box_set_eq:NN }
31308 \cs_generate_variant:Nn \box_set_viewport:Nnnnn { c }
31309 \cs_new_protected:Npn \box_gset_viewport:Nnnnn #1#2#3#4#5
31310 { \__box_set_viewport:NnnnnN #1 {#2} {#3} {#4} {#5} \box_gset_eq:NN }
31311 \cs_generate_variant:Nn \box_gset_viewport:Nnnnn { c }
31312 \cs_new_protected:Npn \__box_set_viewport:NnnnnN #1#2#3#4#5#6

```

```

31313 {
31314   \hbox_set:Nn \l__box_internal_box
31315   {
31316     \tex_kern:D - \__box_dim_eval:n {#2}
31317     \box_use:N #1
31318     \tex_kern:D \__box_dim_eval:n { #4 - \box_wd:N #1 }
31319   }
31320   \dim_compare:nNnTF {#3} < \c_zero_dim
31321   {
31322     \hbox_set:Nn \l__box_internal_box
31323     {
31324       \box_move_down:nn \c_zero_dim
31325       { \box_use_drop:N \l__box_internal_box }
31326     }
31327     \box_set_dp:Nn \l__box_internal_box { - \__box_dim_eval:n {#3} }
31328   }
31329   {
31330     \hbox_set:Nn \l__box_internal_box
31331     { \box_move_down:nn {#3} { \box_use_drop:N \l__box_internal_box } }
31332     \box_set_dp:Nn \l__box_internal_box \c_zero_dim
31333   }
31334   \dim_compare:nNnTF {#5} > \c_zero_dim
31335   {
31336     \hbox_set:Nn \l__box_internal_box
31337     {
31338       \box_move_up:nn \c_zero_dim
31339       { \box_use_drop:N \l__box_internal_box }
31340     }
31341     \box_set_ht:Nn \l__box_internal_box
31342     {
31343       (#5)
31344       \dim_compare:nNnT {#3} > \c_zero_dim
31345       { - (#3) }
31346     }
31347   }
31348   {
31349     \hbox_set:Nn \l__box_internal_box
31350     {
31351       \box_move_up:nn { - \__box_dim_eval:n {#5} }
31352       { \box_use_drop:N \l__box_internal_box }
31353     }
31354     \box_set_ht:Nn \l__box_internal_box \c_zero_dim
31355   }
31356   #6 #1 \l__box_internal_box
31357 }

```

(End definition for `\box_set_viewport:Nnnnn`, `\box_gset_viewport:Nnnnn`, and `__box_viewport:Nnnnn`.
These functions are documented on page 266.)

51.2 Additions to l3flag

```
31358 <@@=flag>
```

`\flag_raise_if_clear:n` It might be faster to just call the “trap” function in all cases but conceptually the function name suggests we should only run it if the flag is zero in case the “trap” made customizable

in the future.

```

31359 \cs_new:Npn \flag_raise_if_clear:n #1
31360 {
31361   \if_cs_exist:w flag~#1-0 \cs_end:
31362   \else:
31363     \cs:w flag~#1 \cs_end: 0 ;
31364   \fi:
31365 }
```

(End definition for `\flag_raise_if_clear:n`. This function is documented on page 267.)

51.3 Additions to `l3msg`

```

31366 (@@=msg)
```

```

\msg_show_eval:Nn
\msg_log_eval:Nn
\__msg_show_eval:nnN
```

A short-hand used for `\int_show:n` and similar functions that passes to `\tl_show:n` the result of applying `#1` (a function such as `\int_eval:n`) to the expression `#2`. The use of `f`-expansion ensures that `#1` is expanded in the scope in which the show command is called, rather than in the group created by `\iow_wrap:nnnN`. This is only important for expressions involving the `\currentgrouplevel` or `\currentgrouptype`. On the other hand we want the expression to be converted to a string with the usual escape character, hence within the wrapping code.

```

31367 \cs_new_protected:Npn \msg_show_eval:Nn #1#2
31368 { \exp_args:Nf \__msg_show_eval:nnN { #1 {#2} } {#2} \tl_show:n }
31369 \cs_new_protected:Npn \msg_log_eval:Nn #1#2
31370 { \exp_args:Nf \__msg_show_eval:nnN { #1 {#2} } {#2} \tl_log:n }
31371 \cs_new_protected:Npn \__msg_show_eval:nnN #1#2#3 { #3 { #2 = #1 } }
```

(End definition for `\msg_show_eval:Nn`, `\msg_log_eval:Nn`, and `__msg_show_eval:nnN`. These functions are documented on page 268.)

```

\msg_show_item:n
\msg_show_item_unbraced:n
\msg_show_item:nn
\msg_show_item_unbraced:nn
```

Each item in the variable is formatted using one of the following functions. We cannot use `\` and so on because these short-hands cannot be used inside the arguments of messages, only when defining the messages.

```

31372 \cs_new:Npx \msg_show_item:n #1
31373 { \iow_newline: > ~ \c_space_tl \exp_not:N \tl_to_str:n { {#1} } }
31374 \cs_new:Npx \msg_show_item_unbraced:n #1
31375 { \iow_newline: > ~ \c_space_tl \exp_not:N \tl_to_str:n {#1} }
31376 \cs_new:Npx \msg_show_item:nn #1#2
31377 {
31378   \iow_newline: > \use:nn { ~ } { ~ }
31379   \exp_not:N \tl_to_str:n { {#1} }
31380   \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
31381   \exp_not:N \tl_to_str:n { {#2} }
31382 }
31383 \cs_new:Npx \msg_show_item_unbraced:nn #1#2
31384 {
31385   \iow_newline: > \use:nn { ~ } { ~ }
31386   \exp_not:N \tl_to_str:n {#1}
31387   \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
31388   \exp_not:N \tl_to_str:n {#2}
31389 }
```

(End definition for `\msg_show_item:n` and others. These functions are documented on page 268.)

51.4 Additions to l3prg

31390 <@@=bool>

\bool_set_inverse:N Set to false or true locally or globally.
\bool_set_inverse:c
\bool_gset_inverse:N
\bool_gset_inverse:c

```

31391 \cs_new_protected:Npn \bool_set_inverse:N #1
31392 { \bool_if:NTF #1 { \bool_set_false:N } { \bool_set_true:N } #1 }
31393 \cs_generate_variant:Nn \bool_set_inverse:N { c }
31394 \cs_new_protected:Npn \bool_gset_inverse:N #1
31395 { \bool_if:NTF #1 { \bool_gset_false:N } { \bool_gset_true:N } #1 }
31396 \cs_generate_variant:Nn \bool_gset_inverse:N { c }

```

(End definition for `\bool_set_inverse:N` and `\bool_gset_inverse:N`. These functions are documented on page 268.)

\s__bool_mark Internal scan marks.
\s__bool_stop

```

31397 \scan_new:N \s__bool_mark
31398 \scan_new:N \s__bool_stop

```

(End definition for `\s__bool_mark` and `\s__bool_stop`.)

\bool_case_true:n For boolean cases the overall idea is the same as for `\tl_case:nn(TF)` as described in l3tl.
\bool_case_true:nTF
\bool_case_false:n
\bool_case_false:nTF

```

31399 \cs_new:Npn \bool_case_true:nTF
31400 { \exp:w \__bool_case:NnTF \c_true_bool }
31401 \cs_new:Npn \bool_case_true:nT #1#2
31402 { \exp:w \__bool_case:NnTF \c_true_bool {#1} {#2} { } }
31403 \cs_new:Npn \bool_case_true:nF #1
31404 { \exp:w \__bool_case:NnTF \c_true_bool {#1} { } }
31405 \cs_new:Npn \bool_case_true:n #1
31406 { \exp:w \__bool_case:NnTF \c_true_bool {#1} { } { } }
31407 \cs_new:Npn \bool_case_false:nTF
31408 { \exp:w \__bool_case:NnTF \c_false_bool }
31409 \cs_new:Npn \bool_case_false:nT #1#2
31410 { \exp:w \__bool_case:NnTF \c_false_bool {#1} {#2} { } }
31411 \cs_new:Npn \bool_case_false:nF #1
31412 { \exp:w \__bool_case:NnTF \c_false_bool {#1} { } }
31413 \cs_new:Npn \bool_case_false:n #1
31414 { \exp:w \__bool_case:NnTF \c_false_bool {#1} { } { } }
31415 \cs_new:Npn \__bool_case:NnTF #1#2#3#4
31416 {
31417   \bool_if:NTF #1 \__bool_case_true:w \__bool_case_false:w
31418   #2 #1 { } \s__bool_mark {#3} \s__bool_mark {#4} \s__bool_stop
31419 }
31420 \cs_new:Npn \__bool_case_true:w #1#2
31421 {
31422   \bool_if:nTF {#1}
31423   { \__bool_case_end:nw {#2} }
31424   { \__bool_case_true:w }
31425 }
31426 \cs_new:Npn \__bool_case_false:w #1#2
31427 {
31428   \bool_if:nTF {#1}
31429   { \__bool_case_false:w }
31430   { \__bool_case_end:nw {#2} }

```

```

31431 }
31432 \cs_new:Npn \__bool_case_end:nw #1#2#3 \s__bool_mark #4#5 \s__bool_stop
31433 { \exp_end: #1 #4 }

```

(End definition for `\bool_case_true:nTF` and others. These functions are documented on page 269.)

51.5 Additions to `l3prop`

```

31434 <@@=prop>

```

`_prop_use_i_delimit_by_s_stop:nw` Functions to gobble up to a scan mark.

```

31435 \cs_new:Npn \__prop_use_i_delimit_by_s_stop:nw #1 #2 \s__prop_stop {#1}

```

(End definition for `__prop_use_i_delimit_by_s_stop:nw`.)

```

\prop_rand_key_value:N
\prop_rand_key_value:c
  \__prop_rand_item:w

```

Contrarily to `clist`, `seq` and `tl`, there is no function to get an item of a `prop` given an integer between 1 and the number of items, so we write the appropriate code. There is no bounds checking because `\int_rand:nn` is always within bounds. The initial `\int_value:w` is stopped by the first `\s__prop` in #1.

```

31436 \cs_new:Npn \prop_rand_key_value:N #1
31437 {
31438   \prop_if_empty:NF #1
31439   {
31440     \exp_after:wN \__prop_rand_item:w
31441     \int_value:w \int_rand:nn { 1 } { \prop_count:N #1 }
31442     #1 \s__prop_stop
31443   }
31444 }
31445 \cs_generate_variant:Nn \prop_rand_key_value:N { c }
31446 \cs_new:Npn \__prop_rand_item:w #1 \s__prop \__prop_pair:wn #2 \s__prop #3
31447 {
31448   \int_compare:nNnF {#1} > 1
31449   { \__prop_use_i_delimit_by_s_stop:nw { \exp_not:n { {#2} {#3} } } }
31450   \exp_after:wN \__prop_rand_item:w
31451   \int_value:w \int_eval:n { #1 - 1 } \s__prop
31452 }

```

(End definition for `\prop_rand_key_value:N` and `__prop_rand_item:w`. This function is documented on page 269.)

51.6 Additions to `l3seq`

```

31453 <@@=seq>

```

```

\seq_mapthread_function:NNN
\seq_mapthread_function:NcN
\seq_mapthread_function:cNN
\seq_mapthread_function:ccN
  \__seq_mapthread_function:wNN
  \__seq_mapthread_function:wNw
  \__seq_mapthread_function:Nnnwnn

```

The idea is to first expand both sequences, adding the usual `{ ? \prg_break: } { }` to the end of each one. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first tokens of #2 and #5, which for items in both sequences are `\s__seq` `__seq_item:n`. The function to be mapped are then be applied to the two entries. When the code hits the end of one of the sequences, the break material stops the entire loop and tidy up. This avoids needing to find the count of the two sequences, or worrying about which is longer.

```

31454 \cs_new:Npn \seq_mapthread_function:NNN #1#2#3
31455 { \exp_after:wN \__seq_mapthread_function:wNN #2 \s__seq_stop #1 #3 }
31456 \cs_new:Npn \__seq_mapthread_function:wNN \s__seq #1 \s__seq_stop #2#3

```



```

31457 {
31458   \exp_after:wN \__seq_mapthread_function:wNw #2 \s__seq_stop #3
31459   #1 { ? \prg_break: } { }
31460   \prg_break_point:
31461 }
31462 \cs_new:Npn \__seq_mapthread_function:wNw \s__seq #1 \s__seq_stop #2
31463 {
31464   \__seq_mapthread_function:Nnnwnn #2
31465   #1 { ? \prg_break: } { }
31466   \s__seq_stop
31467 }
31468 \cs_new:Npn \__seq_mapthread_function:Nnnwnn #1#2#3#4 \s__seq_stop #5#6
31469 {
31470   \use_none:n #2
31471   \use_none:n #5
31472   #1 {#3} {#6}
31473   \__seq_mapthread_function:Nnnwnn #1 #4 \s__seq_stop
31474 }
31475 \cs_generate_variant:Nn \seq_mapthread_function:NNN { Nc , c , cc }

```

(End definition for `\seq_mapthread_function:NNN` and others. This function is documented on page 270.)

`\seq_set_filter:NNn` Similar to `\seq_map_inline:Nn`, without a `\prg_break_point:` because the user's code is performed within the evaluation of a boolean expression, and skipping out of that would break horribly. The `__seq_wrap_item:n` function inserts the relevant `__seq_item:n` without expansion in the input stream, hence in the x-expanding assignment.

`\seq_gset_filter:NNn`

`__seq_set_filter:NNNn`

```

31476 \cs_new_protected:Npn \seq_set_filter:NNn
31477 { \__seq_set_filter:NNNn \tl_set:Nx }
31478 \cs_new_protected:Npn \seq_gset_filter:NNn
31479 { \__seq_set_filter:NNNn \tl_gset:Nx }
31480 \cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
31481 {
31482   \__seq_push_item_def:n { \bool_if:nT {#4} { \__seq_wrap_item:n {##1} } }
31483   #1 #2 { #3 }
31484   \__seq_pop_item_def:
31485 }

```

(End definition for `\seq_set_filter:NNn`, `\seq_gset_filter:NNn`, and `__seq_set_filter:NNNn`. These functions are documented on page 270.)

`\seq_set_from_inline_x:Nnn` Set `__seq_item:n` then map it using the loop code.

```

\seq_gset_from_inline_x:Nnn
\__seq_set_from_inline_x:NNnn
31486 \cs_new_protected:Npn \seq_set_from_inline_x:Nnn
31487 { \__seq_set_from_inline_x:NNnn \tl_set:Nx }
31488 \cs_new_protected:Npn \seq_gset_from_inline_x:Nnn
31489 { \__seq_set_from_inline_x:NNnn \tl_gset:Nx }
31490 \cs_new_protected:Npn \__seq_set_from_inline_x:NNnn #1#2#3#4
31491 {
31492   \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
31493   #1 #2 { \s__seq #3 \__seq_item:n }
31494   \__seq_pop_item_def:
31495 }

```

(End definition for `\seq_set_from_inline_x:Nnn`, `\seq_gset_from_inline_x:Nnn`, and `__seq_set_from_inline_x:NNnn`. These functions are documented on page 270.)

`\seq_set_from_function:NnN`
`\seq_gset_from_function:NnN`

Reuse `\seq_set_from_inline_x:Nnn`.

```

31496 \cs_new_protected:Npn \seq_set_from_function:NnN #1#2#3
31497 { \seq_set_from_inline_x:Nnn #1 {#2} { #3 {##1} } }
31498 \cs_new_protected:Npn \seq_gset_from_function:NnN #1#2#3
31499 { \seq_gset_from_inline_x:Nnn #1 {#2} { #3 {##1} } }

```

(End definition for `\seq_set_from_function:NnN` and `\seq_gset_from_function:NnN`. These functions are documented on page 270.)

51.7 Additions to l3sys

31500 `<@@=sys>`

`\c_sys_engine_version_str`

Various different engines, various different ways to extract the data!

```

31501 \str_const:Nx \c_sys_engine_version_str
31502 {
31503   \str_case:on \c_sys_engine_str
31504   {
31505     { pdftex }
31506     {
31507       \fp_eval:n { round(\int_use:N \tex_pdftexversion:D / 100 , 2) }
31508       .
31509       \tex_pdftexrevision:D
31510     }
31511     { ptex }
31512     {
31513       \cs_if_exist:NT \tex_ptexversion:D
31514       {
31515         p
31516         \int_use:N \tex_ptexversion:D
31517         .
31518         \int_use:N \tex_ptexminorversion:D
31519         \tex_ptexrevision:D
31520         -
31521         \int_use:N \tex_epTeXversion:D
31522       }
31523     }
31524     { luatex }
31525     {
31526       \fp_eval:n { round(\int_use:N \tex_luatexversion:D / 100, 2) }
31527       .
31528       \tex_luatexrevision:D
31529     }
31530     { uptex }
31531     {
31532       \cs_if_exist:NT \tex_ptexversion:D
31533       {
31534         p
31535         \int_use:N \tex_ptexversion:D
31536         .
31537         \int_use:N \tex_ptexminorversion:D
31538         \tex_ptexrevision:D
31539         -
31540         u

```

```

31541             \int_use:N \tex_uptexversion:D
31542             \tex_uptexrevision:D
31543             -
31544             \int_use:N \tex_epTeXversion:D
31545         }
31546     }
31547     { xetex }
31548     {
31549         \int_use:N \tex_XeTeXversion:D
31550         \tex_XeTeXrevision:D
31551     }
31552 }
31553 }

```

(End definition for `\c_sys_engine_version_str`. This variable is documented on page 271.)

51.8 Additions to `l3file`

```

31554 <@@=ior>
\ior_shell_open:Nn \__ior_shell_open:nN

```

Actually much easier than either the standard open or input versions! When calling `__kernel_ior_open:Nn` the file the pipe is added to signal a shell command, but the quotes are not added yet—they are added later by `__kernel_file_name_quote:n`.

```

31555 \cs_new_protected:Npn \ior_shell_open:Nn #1#2
31556 {
31557     \sys_if_shell:TF
31558     { \exp_args:No \__ior_shell_open:nN { \tl_to_str:n {#2} } #1 }
31559     { \__kernel_msg_error:nn { kernel } { pipe-failed } }
31560 }
31561 \cs_new_protected:Npn \__ior_shell_open:nN #1#2
31562 {
31563     \tl_if_in:nnTF {#1} { " }
31564     {
31565         \__kernel_msg_error:nnx
31566         { kernel } { quote-in-shell } {#1}
31567     }
31568     { \__kernel_ior_open:Nn #2 { |#1 } }
31569 }
31570 \__kernel_msg_new:nnnn { kernel } { pipe-failed }
31571 { Cannot~run~piped~system~commands. }
31572 {
31573     LaTeX~tried~to~call~a~system~process~but~this~was~not~possible.\\
31574     Try~the~"--shell-escape"~(or~"--enable-pipes")~option.
31575 }

```

(End definition for `\ior_shell_open:Nn` and `__ior_shell_open:nN`. This function is documented on page 267.)

51.9 Additions to `l3tl`

51.9.1 Building a token list

```

31576 <@@=tl>

```

Between `\tl_build_begin:N <tl var>` and `\tl_build_end:N <tl var>`, the `<tl var>` has the structure

```
\exp_end: ... \exp_end: \_tl\_build\_last:NNn <assignment> <next tl>
{<left>} <right>
```

where $\langle right \rangle$ is not braced. The “data” it represents is $\langle left \rangle$ followed by the “data” of $\langle next\ tl \rangle$ followed by $\langle right \rangle$. The $\langle next\ tl \rangle$ is a token list variable whose name is that of $\langle tl\ var \rangle$ followed by $'$. There are between 0 and 4 $\backslash exp_end:$ to keep track of when $\langle left \rangle$ and $\langle right \rangle$ should be put into the $\langle next\ tl \rangle$. The $\langle assignment \rangle$ is $\backslash cs_set_nopar:Npx$ if the variable is local, and $\backslash cs_gset_nopar:Npx$ if it is global.

```
\tl\_build\_begin:N First construct the <next tl>: using a prime here conflicts with the usual expl3 convention
\tl\_build\_gbegin:N but we need a name that can be derived from #1 without any external data such as a
\_tl\_build\_begin:NN counter. Empty that <next tl> and setup the structure. The local and global versions
\_tl\_build\_begin:NNN only differ by a single function \cs_(g)set\_nopar:Npx used for all assignments: this is
important because only that function is stored in the <tl var> and <next tl> for subsequent
assignments. In principle \_tl\_build\_begin:NNN could use \tl_(g)clear\_new:N to
empty #1 and make sure it is defined, but logging the definition does not seem useful so
we just do #3 #1 {} to clear it locally or globally as appropriate.
```

```
31577 \cs\_new\_protected:Npn \tl\_build\_begin:N #1
31578 { \_tl\_build\_begin:NN \cs\_set\_nopar:Npx #1 }
31579 \cs\_new\_protected:Npn \tl\_build\_gbegin:N #1
31580 { \_tl\_build\_begin:NN \cs\_gset\_nopar:Npx #1 }
31581 \cs\_new\_protected:Npn \_tl\_build\_begin:NN #1#2
31582 { \exp\_args:Nc \_tl\_build\_begin:NNN { \cs\_to\_str:N #2 ' } #2 #1 }
31583 \cs\_new\_protected:Npn \_tl\_build\_begin:NNN #1#2#3
31584 {
31585   #3 #1 { }
31586   #3 #2
31587   {
31588     \exp\_not:n { \exp\_end: \exp\_end: \exp\_end: \exp\_end: }
31589     \exp\_not:n { \_tl\_build\_last:NNn #3 #1 { } }
31590   }
31591 }
```

(End definition for $\backslash tl_build_begin:N$ and others. These functions are documented on page 272.)

```
\tl\_build\_clear:N The begin and gbegin functions already clear enough to make the token list variable
\tl\_build\_gclear:N effectively empty. Eventually the begin and gbegin functions should check that #1' is
empty or undefined, while the clear and gclear functions ought to empty #1', #1''
and so on, similar to \tl\_build\_end:N. This only affects memory usage.
```

```
31592 \cs\_new\_eq:NN \tl\_build\_clear:N \tl\_build\_begin:N
31593 \cs\_new\_eq:NN \tl\_build\_gclear:N \tl\_build\_gbegin:N
```

(End definition for $\backslash tl_build_clear:N$ and $\backslash tl_build_gclear:N$. These functions are documented on page 272.)

```
\tl\_build\_put\_right:Nn Similar to \tl\_put\_right:Nn, but apply \exp:w to #1. Most of the time this just removes
\tl\_build\_put\_right:Nx one \exp\_end:. When there are none left, \_tl\_build\_last:NNn is expanded instead.
\tl\_build\_gput\_right:Nn It resets the definition of the <tl var> by ending the \exp\_not:n and the definition early.
\tl\_build\_gput\_right:Nx Then it makes sure the <next tl> (its argument #1) is set-up and starts a new definition.
\_tl\_build\_last:NNn Then \_tl\_build\_put:nn and \_tl\_build\_put:nw place the <left> part of the original
\_tl\_build\_put:nn <tl var> as appropriate for the definition of the <next tl> (the <right> part is left in the right
\_tl\_build\_put:nw place without ever becoming a macro argument). We use \exp\_after:wN rather than
some \exp\_args:No to avoid reading arguments that are likely very long token lists. We
```

use `\cs_(g)set_nopar:Npx` rather than `\tl_(g)set:Nx` partly for the same reason and partly because the assignments are interrupted by brace tricks, which implies that the assignment does not simply set the token list to an x-expansion of the second argument.

```

31594 \cs_new_protected:Npn \tl_build_put_right:Nn #1#2
31595 {
31596   \cs_set_nopar:Npx #1
31597   { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 #2 } }
31598 }
31599 \cs_new_protected:Npn \tl_build_put_right:Nx #1#2
31600 {
31601   \cs_set_nopar:Npx #1
31602   { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 } #2 }
31603 }
31604 \cs_new_protected:Npn \tl_build_gput_right:Nn #1#2
31605 {
31606   \cs_gset_nopar:Npx #1
31607   { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 #2 } }
31608 }
31609 \cs_new_protected:Npn \tl_build_gput_right:Nx #1#2
31610 {
31611   \cs_gset_nopar:Npx #1
31612   { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 } #2 }
31613 }
31614 \cs_new_protected:Npn \__tl_build_last:NNn #1#2
31615 {
31616   \if_false: { { \fi:
31617     \exp_end: \exp_end: \exp_end: \exp_end: \exp_end:
31618     \__tl_build_last:NNn #1 #2 { }
31619   }
31620 }
31621 \if_meaning:w \c_empty_tl #2
31622   \__tl_build_begin:NN #1 #2
31623 \fi:
31624 #1 #2
31625 {
31626   \exp_after:wN \exp_not:n \exp_after:wN
31627   {
31628     \exp:w \if_false: } } \fi:
31629   \exp_after:wN \__tl_build_put:nn \exp_after:wN {#2}
31630 }
31631 \cs_new_protected:Npn \__tl_build_put:nn #1#2 { \__tl_build_put:nw {#2} #1 }
31632 \cs_new_protected:Npn \__tl_build_put:nw #1#2 \__tl_build_last:NNn #3#4#5
31633 { #2 \__tl_build_last:NNn #3 #4 { #1 #5 } }

```

(End definition for `\tl_build_put_right:Nn` and others. These functions are documented on page 273.)

<code>\tl_build_put_left:Nn</code> <code>\tl_build_put_left:Nx</code> <code>\tl_build_gput_left:Nn</code> <code>\tl_build_gput_left:Nx</code> <code>__tl_build_put_left:NNn</code>	<p>See <code>\tl_build_put_right:Nn</code> for all the machinery. We could easily provide <code>\tl_build_put_left_right:NNn</code>, by just add the <code>\right</code> material after the <code>\left</code> in the x-expanding assignment.</p> <pre> 31634 \cs_new_protected:Npn \tl_build_put_left:Nn #1 31635 { __tl_build_put_left:NNn \cs_set_nopar:Npx #1 } 31636 \cs_generate_variant:Nn \tl_build_put_left:Nn { Nx } 31637 \cs_new_protected:Npn \tl_build_gput_left:Nn #1 31638 { __tl_build_put_left:NNn \cs_gset_nopar:Npx #1 } </pre>
---	---

```

31639 \cs_generate_variant:Nn \tl_build_gput_left:Nn { Nx }
31640 \cs_new_protected:Npn \__tl_build_put_left:NNn #1#2#3
31641 {
31642     #1 #2
31643     {
31644         \exp_after:wN \exp_not:n \exp_after:wN
31645         {
31646             \exp:w \exp_after:wN \__tl_build_put:nn
31647             \exp_after:wN {#2} {#3}
31648         }
31649     }
31650 }

```

(End definition for `\tl_build_put_left:Nn`, `\tl_build_gput_left:Nn`, and `__tl_build_put_left:NNn`. These functions are documented on page 273.)

`\tl_build_get:NN` The idea is to expand the `<tl var>` then the `<next tl>` and so on, all within an x-expanding assignment, and wrap as appropriate in `\exp_not:n`. The various `<left>` parts are left in the assignment as we go, which enables us to expand the `<next tl>` at the right place. The various `<right>` parts are eventually picked up in one last `\exp_not:n`, with a brace trick to wrap all the `<right>` parts together.

```

31651 \cs_new_protected:Npn \tl_build_get:NN
31652 { \__tl_build_get:NNN \tl_set:Nx }
31653 \cs_new_protected:Npn \__tl_build_get:NNN #1#2#3
31654 { #1 #3 { \if_false: { \fi: \exp_after:wN \__tl_build_get:w #2 } } }
31655 \cs_new:Npn \__tl_build_get:w #1 \__tl_build_last:NNn #2#3#4
31656 {
31657     \exp_not:n {#4}
31658     \if_meaning:w \c_empty_tl #3
31659     \exp_after:wN \__tl_build_get_end:w
31660     \fi:
31661     \exp_after:wN \__tl_build_get:w #3
31662 }
31663 \cs_new:Npn \__tl_build_get_end:w #1#2#3
31664 { \exp_after:wN \exp_not:n \exp_after:wN { \if_false: } \fi: }

```

(End definition for `\tl_build_get:NN` and others. This function is documented on page 273.)

`\tl_build_end:N` Get the data then clear the `<next tl>` recursively until finding an empty one. It is perhaps wasteful to repeatedly use `\cs_to_sr:N`. The local/global scope is checked by `\tl_set:Nx` or `\tl_gset:Nx`.

```

31665 \cs_new_protected:Npn \tl_build_end:N #1
31666 {
31667     \__tl_build_get:NNN \tl_set:Nx #1 #1
31668     \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } \tl_clear:N
31669 }
31670 \cs_new_protected:Npn \tl_build_gend:N #1
31671 {
31672     \__tl_build_get:NNN \tl_gset:Nx #1 #1
31673     \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } \tl_gclear:N
31674 }
31675 \cs_new_protected:Npn \__tl_build_end_loop:NN #1#2
31676 {
31677     \if_meaning:w \c_empty_tl #1

```

```

31678     \exp_after:wN \use_none:nnnnnn
31679     \fi:
31680     #2 #1
31681     \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } #2
31682 }

```

(End definition for `\tl_build_end:N`, `\tl_build_gend:N`, and `__tl_build_end_loop:NN`. These functions are documented on page 273.)

51.9.2 Other additions to `l3tl`

`\tl_range_braced:Nnn` For the braced version `__tl_range_braced:w` sets up `__tl_range_collect_braced:w` which stores items one by one in an argument after the semicolon. The unbraced version is almost identical. The version preserving braces and spaces starts by deleting spaces before the argument to avoid collecting them, and sets up `__tl_range_collect:nn` with a first argument of the form `{ {⟨collected⟩} ⟨tokens⟩ }`, whose head is the collected tokens and whose tail is what remains of the original token list. This form makes it easier to move tokens to the `⟨collected⟩` tokens.

```

\__tl_range_collect_braced:w 31683 \cs_new:Npn \tl_range_braced:Nnn { \exp_args:No \tl_range_braced:nnn }
\__tl_range_unbraced:w       31684 \cs_generate_variant:Nn \tl_range_braced:Nnn { c }
\tl_range_collect_unbraced:w 31685 \cs_new:Npn \tl_range_braced:nnn { \__tl_range:Nnnn \__tl_range_braced:w }
                             31686 \cs_new:Npn \tl_range_unbraced:Nnn
                             31687 { \exp_args:No \tl_range_unbraced:nnn }
                             31688 \cs_generate_variant:Nn \tl_range_unbraced:Nnn { c }
                             31689 \cs_new:Npn \tl_range_unbraced:nnn
                             31690 { \__tl_range:Nnnn \__tl_range_unbraced:w }
                             31691 \cs_new:Npn \__tl_range_braced:w #1 ; #2
                             31692 { \__tl_range_collect_braced:w #1 ; { } #2 }
                             31693 \cs_new:Npn \__tl_range_unbraced:w #1 ; #2
                             31694 { \__tl_range_collect_unbraced:w #1 ; { } #2 }
                             31695 \cs_new:Npn \__tl_range_collect_braced:w #1 ; #2#3
                             31696 {
                             31697   \if_int_compare:w #1 > 1 \exp_stop_f:
                             31698     \exp_after:wN \__tl_range_collect_braced:w
                             31699     \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
                             31700   \fi:
                             31701   { #2 {#3} }
                             31702 }
                             31703 \cs_new:Npn \__tl_range_collect_unbraced:w #1 ; #2#3
                             31704 {
                             31705   \if_int_compare:w #1 > 1 \exp_stop_f:
                             31706     \exp_after:wN \__tl_range_collect_unbraced:w
                             31707     \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
                             31708   \fi:
                             31709   { #2 #3 }
                             31710 }

```

(End definition for `\tl_range_braced:Nnn` and others. These functions are documented on page 272.)

51.10 Additions to `l3token`

`\c_catcode_active_space_tl` While `\char_generate:nn` can produce active characters in some engines it cannot in general. It would be possible to simply change the catcode of space but then the code

would need to avoid all spaces, making it quite unreadable. Instead we use the primitive `\tex_lowercase:D` trick.

```

31711 \group_begin:
31712   \char_set_catcode_active:N *
31713   \char_set_lccode:nn { '*' } { '\ }
31714   \tex_lowercase:D { \tl_const:Nn \c_catcode_active_space_tl { * } }
31715 \group_end:

```

(End definition for `\c_catcode_active_space_tl`. This variable is documented on page 273.)

```

31716 <@@=peek>

```

```

\l__peek_collect_tl

```

```

31717 \tl_new:N \l__peek_collect_tl

```

(End definition for `\l__peek_collect_tl`.)

```

\peek_catcode_collect_inline:Nn Most of the work is done by \__peek_execute_branches_...:, which calls either \__-
\peek_charcode_collect_inline:Nn peek_true:w or \__peek_false:w according to whether the next token \l_peek_token
\peek_meaning_collect_inline:Nn matches the search token (stored in \l__peek_search_token and \l__peek_search-
\__peek_collect:NNn tl). Here, in the true case we run \__peek_collect_true:w, which generally calls
\__peek_collect_true:w \__peek_collect:N to store the peeked token into \l__peek_collect_tl, except in
\__peek_collect_remove:nw special non-N-type cases (begin-group, end-group, or space), where a frozen token is
\__peek_collect:N stored. The true branch calls \__peek_execute_branches_...: to fetch more matching
tokens. Once there are no more, \__peek_false_aux:n closes the safe-align group and
runs the user's inline code.

```

```

31718 \cs_new_protected:Npn \peek_catcode_collect_inline:Nn
31719   { \__peek_collect:NNn \__peek_execute_branches_catcode: }
31720 \cs_new_protected:Npn \peek_charcode_collect_inline:Nn
31721   { \__peek_collect:NNn \__peek_execute_branches_charcode: }
31722 \cs_new_protected:Npn \peek_meaning_collect_inline:Nn
31723   { \__peek_collect:NNn \__peek_execute_branches_meaning: }
31724 \cs_new_protected:Npn \__peek_collect:NNn #1#2#3
31725   {
31726     \group_align_safe_begin:
31727     \cs_set_eq:NN \l__peek_search_token #2
31728     \tl_set:Nn \l__peek_search_tl {#2}
31729     \tl_clear:N \l__peek_collect_tl
31730     \cs_set:Npn \__peek_false:w
31731       { \exp_args:No \__peek_false_aux:n \l__peek_collect_tl }
31732     \cs_set:Npn \__peek_false_aux:n ##1
31733       {
31734         \group_align_safe_end:
31735         #3
31736       }
31737     \cs_set_eq:NN \__peek_true:w \__peek_collect_true:w
31738     \cs_set:Npn \__peek_true_aux:w { \peek_after:Nw #1 }
31739     \__peek_true_aux:w
31740   }
31741 \cs_new_protected:Npn \__peek_collect_true:w
31742   {
31743     \if_case:w
31744       \if_catcode:w \exp_not:N \l_peek_token { 1 \exp_stop_f: \fi:
31745       \if_catcode:w \exp_not:N \l_peek_token { 2 \exp_stop_f: \fi:

```



```

31746     \if_meaning:w \l_peek_token \c_space_token 3 \exp_stop_f: \fi:
31747     0 \exp_stop_f:
31748     \exp_after:wN \__peek_collect:N
31749     \or: \__peek_collect_remove:nw { \c_group_begin_token }
31750     \or: \__peek_collect_remove:nw { \c_group_end_token }
31751     \or: \__peek_collect_remove:nw { ~ }
31752     \fi:
31753   }
31754   \cs_new_protected:Npn \__peek_collect:N #1
31755   {
31756     \tl_put_right:Nn \l__peek_collect_tl {#1}
31757     \__peek_true_aux:w
31758   }
31759   \cs_new_protected:Npn \__peek_collect_remove:nw #1
31760   {
31761     \tl_put_right:Nn \l__peek_collect_tl {#1}
31762     \exp_after:wN \__peek_true_remove:w
31763   }

```

(End definition for `\peek_catcode_collect_inline:Nn` and others. These functions are documented on page 274.)

```

31764 \endpackage

```

52 l3deprecation implementation

```

31765 \begin{package}
31766 \begin{kernel}
31767 \begin{deprecation}

```

52.1 Helpers and variables

`\l_deprecation_grace_period_bool` This is set to `true` when the deprecated command that is being defined is in its grace period, meaning between the time it becomes an error by default and the time 6 months later where even `undo-recent-deprecations` stops restoring it.

```

31768 \bool_new:N \l_deprecation_grace_period_bool

```

(End definition for `\l_deprecation_grace_period_bool`.)

`\s__deprecation_mark` Internal scan marks.

```

\s__deprecation_stop 31769 \scan_new:N \s__deprecation_mark
31770 \scan_new:N \s__deprecation_stop

```

(End definition for `\s__deprecation_mark` and `\s__deprecation_stop`.)

`_deprecation_date_compare:nNnTF` Expects `#1` and `#3` to be dates in the format YYYY-MM-DD (but accepts YYYY or YYYY-MM too, filling in zeros for the missing data). Compares them using `#2` (one of `<`, `=`, `>`).

```

31771 \cs_new:Npn \_deprecation_date_compare:nNnTF #1#2#3
31772 { \_deprecation_date_compare_aux:w #1 -0-0- \s__deprecation_mark #2 #3 -0-0- \s__depreca
31773 \cs_new:Npn \_deprecation_date_compare_aux:w
31774 #1 - #2 - #3 - #4 \s__deprecation_mark #5 #6 - #7 - #8 - #9 \s__deprecation_stop
31775 {
31776   \int_compare:nNnTF {#1} = {#6}

```

```

31777     {
31778         \int_compare:nNnTF {#2} = {#7}
31779         { \int_compare:nNnTF {#3} #5 {#8} }
31780         { \int_compare:nNnTF {#2} #5 {#7} }
31781     }
31782     { \int_compare:nNnTF {#1} #5 {#6} }
31783 }

```

(End definition for `__deprecation_date_compare:nNnTF` and `__deprecation_date_compare_aux:w`.)

`\g__kernel_deprecation_undo_recent_bool`

```

31784 \bool_new:N \g__kernel_deprecation_undo_recent_bool

```

(End definition for `\g__kernel_deprecation_undo_recent_bool`.)

`__deprecation_not_yet_deprecated:nTF`
`__deprecation_minus_six_months:w`

Receives a deprecation $\langle date \rangle$ and runs the `true` (`false`) branch if the `expl3` date is earlier (later) than $\langle date \rangle$. If `undo-recent-deprecations` is used we subtract 6 months to the `expl3` date (equivalently add 6 months to the $\langle date \rangle$). In addition, if the `expl3` date is between $\langle date \rangle$ and $\langle date \rangle$ plus 6 months, `\l__deprecation_grace_period_bool` is set to `true`, otherwise `false`.

```

31785 \cs_new_protected:Npn \__deprecation_not_yet_deprecated:nTF #1
31786 {
31787     \bool_set_false:N \l__deprecation_grace_period_bool
31788     \exp_args:No \__deprecation_date_compare:nNnTF { \ExplLoaderFileDate } < {#1}
31789     { \use_i:nn }
31790     {
31791         \exp_args:Nf \__deprecation_date_compare:nNnTF
31792         {
31793             \exp_after:wN \__deprecation_minus_six_months:w
31794             \ExplLoaderFileDate -0-0- \s__deprecation_stop
31795         } < {#1}
31796         {
31797             \bool_set_true:N \l__deprecation_grace_period_bool
31798             \bool_if:NTF \g__kernel_deprecation_undo_recent_bool
31799         }
31800         { \use_ii:nn }
31801     }
31802 }
31803 \cs_new:Npn \__deprecation_minus_six_months:w #1 - #2 - #3 - #4 \s__deprecation_stop
31804 {
31805     \int_compare:nNnTF {#2} > 6
31806     { #1 - \int_eval:n { #2 - 6 } - #3 }
31807     { \int_eval:n { #1 - 1 } - \int_eval:n { #2 + 6 } - #3 }
31808 }

```

(End definition for `__deprecation_not_yet_deprecated:nTF` and `__deprecation_minus_six_months:w`.)

52.2 Patching definitions to deprecate

`__kernel_patch_deprecation:nnNNpn { $\langle date \rangle$ } { $\langle replacement \rangle$ } $\langle definition \rangle$`
 `$\langle function \rangle$ $\langle parameters \rangle$ { $\langle code \rangle$ }`

defines the $\langle function \rangle$ to produce a warning and run its $\langle code \rangle$, or to produce an error and not run any $\langle code \rangle$, depending on the `expl3` date.

- If the `expl3` date is less than the $\langle date \rangle$ (plus 6 months in case `undo-recent-deprecations` is used) then we define the $\langle function \rangle$ to produce a warning and run its code. The warning is actually suppressed in two cases:
 - if neither `undo-recent-deprecations` nor `enable-debug` are in effect we may be in an end-user’s document so it is suppressed;
 - if the command is expandable then we cannot produce a warning.
- Otherwise, we define the $\langle function \rangle$ to produce an error.

In both cases we additionally make `\debug_on:n {deprecation}` turn the $\langle function \rangle$ into an `\outer` error, and `\debug_off:n {deprecation}` restore whatever the behaviour was without `\debug_on:n {deprecation}`.

In later sections we use the `l3doc` key `deprecated` with a date equal to that $\langle date \rangle$ plus 6 months, so that `l3doc` will complain if we forget to remove the stale $\langle parameters \rangle$ and $\{ \langle code \rangle \}$.

In the explanations below, $\langle definition \rangle$ $\langle function \rangle$ $\langle parameters \rangle$ $\{ \langle code \rangle \}$ or assignments that only differ in the scope of the $\langle definition \rangle$ will be called “the standard definition”.

```

\__kernel_patch_deprecation:nnNNpn
\__deprecation_patch_aux:nnNNnn
\__deprecation_warn_once:nnNnn
\__deprecation_patch_aux:Nn
\__deprecation_just_error:nnNN

```

(The parameter text is grabbed using `#5#`.) The arguments of `__kernel_deprecation_code:nn` are run upon `\debug_on:n {deprecation}` and `\debug_off:n {deprecation}`, respectively. In both scenarios we the $\langle function \rangle$ may be `\outer` so we undefine it with `\tex_let:D` before redefining it, with `__kernel_deprecation_error:Nnn` or with some code added shortly.

Then check the date (taking into account `undo-recent-deprecations`) to see if the command should be deprecated right away (`false` branch of `__deprecation_not_yet_deprecated:nTF`), in which case `__deprecation_just_error:nnNN` makes $\langle function \rangle$ into an error (not `\outer`), ignoring its $\langle parameters \rangle$ and $\langle code \rangle$ completely.

Otherwise distinguish cases where we should give a warning from those where we shouldn’t: warnings can only happen for protected commands, and we only want them if either `undo-recent-deprecations` or `enable-debug` is in force, not for standard users.

```

31809 \cs_new_protected:Npn \__kernel_patch_deprecation:nnNNpn #1#2#3#4#5#
31810 { \__deprecation_patch_aux:nnNNnn {#1} {#2} #3 #4 {#5} }
31811 \cs_new_protected:Npn \__deprecation_patch_aux:nnNNnn #1#2#3#4#5#6
31812 {
31813   \__kernel_deprecation_code:nn
31814   {
31815     \tex_let:D #4 \scan_stop:
31816     \__kernel_deprecation_error:Nnn #4 {#2} {#1}
31817   }
31818   { \tex_let:D #4 \scan_stop: }
31819   \__deprecation_not_yet_deprecated:nTF {#1}
31820   {
31821     \bool_if:nTF
31822     {
31823       \cs_if_eq_p:NN #3 \cs_gset_protected:Npn &&
31824       \__kernel_if_debug:TF
31825       { \c_true_bool } { \g__kernel_deprecation_undo_recent_bool }
31826     }
31827     { \__deprecation_warn_once:nnNnn {#1} {#2} #4 {#5} {#6} }
31828     { \__deprecation_patch_aux:Nn #3 { #4 #5 {#6} } }

```

```

31829     }
31830     { \_deprecation_just_error:nnNN {#1} {#2} #3 #4 }
31831 }

```

In case we want a warning, the *function* is defined to produce such a warning without grabbing any argument, then redefine itself to the standard definition that the *function* should have, with arguments, and call that definition. The `x`-type expansion and `\exp_not:n` avoid needing to double the `#`, which we could not do anyways. We then deal with the code for `\debug_off:n {deprecation}`: presumably someone doing that does not need the warning so we simply do the standard definition.

```

31832 \cs_new_protected:Npn \_deprecation_warn_once:nnNnn #1#2#3#4#5
31833 {
31834     \cs_gset_protected:Npx #3
31835     {
31836         \_kernel_if_debug:TF
31837         {
31838             \exp_not:N \_kernel_msg_warning:nnxxx
31839             { kernel } { deprecated-command }
31840             {#1}
31841             { \token_to_str:N #3 }
31842             { \tl_to_str:n {#2} }
31843         }
31844     }
31845     \exp_not:n { \cs_gset_protected:Npn #3 #4 {#5} }
31846     \exp_not:N #3
31847 }
31848 \_kernel_deprecation_code:nn { }
31849 { \cs_set_protected:Npn #3 #4 {#5} }
31850 }

```

In case we want neither warning nor error, the *function* is given its standard definition. Here `#1` is `\cs_new:Npn` or `\cs_new_protected:Npn` and `#2` is *function* *parameters* *{code}*, so `#1#2` performs the assignment. For `\debug_off:n {deprecation}` we want to use the same assignment but with a different scope, hence the `\cs_if_eq:NNTF` test.

```

31851 \cs_new_protected:Npn \_deprecation_patch_aux:Nn #1#2
31852 {
31853     #1 #2
31854     \cs_if_eq:NNTF #1 \cs_gset_protected:Npn
31855     { \_kernel_deprecation_code:nn { } { \cs_set_protected:Npn #2 } }
31856     { \_kernel_deprecation_code:nn { } { \cs_set:Npn #2 } }
31857 }

```

Finally, if we want an error we reuse the same `_deprecation_patch_aux:Nn` as the previous case. Indeed, we want `\debug_off:n {deprecation}` to make the *function* into an error, just like it is by default. The error is expandable or not, and the last argument of the error message is empty or is `grace` to denote the case where we are in the 6 month grace period, in which case the error message is more detailed.

```

31858 \cs_new_protected:Npn \_deprecation_just_error:nnNN #1#2#3#4
31859 {
31860     \exp_args:NNx \_deprecation_patch_aux:Nn #3
31861     {
31862         \exp_not:N #4
31863         {
31864             \cs_if_eq:NNTF #3 \cs_gset_protected:Npn

```

```

31865         { \exp_not:N \__kernel_msg_error:nnnnnn }
31866         { \exp_not:N \__kernel_msg_expandable_error:nnnnnn }
31867         { kernel } { deprecated-command }
31868         {#1}
31869         { \token_to_str:N #4 }
31870         { \tl_to_str:n {#2} }
31871         { \bool_if:NT \l__deprecation_grace_period_bool { grace } }
31872     }
31873 }
31874 }

```

(End definition for __kernel_patch_deprecation:nnNNpn and others.)

__kernel_deprecation_error:Nnn The \outer definition here ensures the command cannot appear in an argument. Use this auxiliary on all commands that have been removed since 2015.

```

31875 \cs_new_protected:Npn \__kernel_deprecation_error:Nnn #1#2#3
31876 {
31877     \tex_protected:D \tex_outer:D \tex_edef:D #1
31878     {
31879         \exp_not:N \__kernel_msg_expandable_error:nnnnn
31880         { kernel } { deprecated-command }
31881         { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
31882         \exp_not:N \__kernel_msg_error:nnxxx
31883         { kernel } { deprecated-command }
31884         { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
31885     }
31886 }

```

(End definition for __kernel_deprecation_error:Nnn.)

```

31887 \__kernel_msg_new:nnn { kernel } { deprecated-command }
31888 {
31889     \tl_if_blank:nF {#3} { Use~ \tl_trim_spaces:n {#3} ~not~ }
31890     #2~deprecated-on~#1.
31891     \str_if_eq:nnT {#4} { grace }
31892     {
31893         \c_space_tl
31894         For~6~months~after~that~date~one~can~restore~a~deprecated~
31895         command~by~loading~the~expl3~package~with~the~option~
31896         'undo-recent-deprecations'.
31897     }
31898 }

```

52.3 Removed functions

__deprecation_old_protected:Nnn Short-hands for old commands whose definition does not matter anymore, i.e., commands past the grace period.

```

31899 \cs_new_protected:Npn \__deprecation_old_protected:Nnn #1#2#3
31900 {
31901     \__kernel_patch_deprecation:nnNNpn {#3} {#2}
31902     \cs_gset_protected:Npn #1 { }
31903 }
31904 \cs_new_protected:Npn \__deprecation_old:Nnn #1#2#3
31905 {
31906     \__kernel_patch_deprecation:nnNNpn {#3} {#2}

```

```

31907     \cs_gset:Npn #1 { }
31908   }
31909   \__deprecation_old:Nnn \box_resize:Nnn
31910     { \box_resize_to_wd_and_ht_plus_dp:Nnn } { 2019-01-01 }
31911   \__deprecation_old:Nnn \box_use_clear:N
31912     { \box_use_drop:N } { 2019-01-01 }
31913   \__deprecation_old:Nnn \c_job_name_tl
31914     { \c_sys_jobname_str } { 2017-01-01 }
31915   \__deprecation_old:Nnn \c_minus_one
31916     { -1 } { 2019-01-01 }
31917   \__deprecation_old:Nnn \c_zero
31918     { 0 } { 2020-01-01 }
31919   \__deprecation_old:Nnn \c_one
31920     { 1 } { 2020-01-01 }
31921   \__deprecation_old:Nnn \c_two
31922     { 2 } { 2020-01-01 }
31923   \__deprecation_old:Nnn \c_three
31924     { 3 } { 2020-01-01 }
31925   \__deprecation_old:Nnn \c_four
31926     { 4 } { 2020-01-01 }
31927   \__deprecation_old:Nnn \c_five
31928     { 5 } { 2020-01-01 }
31929   \__deprecation_old:Nnn \c_six
31930     { 6 } { 2020-01-01 }
31931   \__deprecation_old:Nnn \c_seven
31932     { 7 } { 2020-01-01 }
31933   \__deprecation_old:Nnn \c_eight
31934     { 8 } { 2020-01-01 }
31935   \__deprecation_old:Nnn \c_nine
31936     { 9 } { 2020-01-01 }
31937   \__deprecation_old:Nnn \c_ten
31938     { 10 } { 2020-01-01 }
31939   \__deprecation_old:Nnn \c_eleven
31940     { 11 } { 2020-01-01 }
31941   \__deprecation_old:Nnn \c_twelve
31942     { 12 } { 2020-01-01 }
31943   \__deprecation_old:Nnn \c_thirteen
31944     { 13 } { 2020-01-01 }
31945   \__deprecation_old:Nnn \c_fourteen
31946     { 14 } { 2020-01-01 }
31947   \__deprecation_old:Nnn \c_fifteen
31948     { 15 } { 2020-01-01 }
31949   \__deprecation_old:Nnn \c_sixteen
31950     { 16 } { 2020-01-01 }
31951   \__deprecation_old:Nnn \c_thirty_two
31952     { 32 } { 2020-01-01 }
31953   \__deprecation_old:Nnn \c_one_hundred
31954     { 100 } { 2020-01-01 }
31955   \__deprecation_old:Nnn \c_two_hundred_fifty_five
31956     { 255 } { 2020-01-01 }
31957   \__deprecation_old:Nnn \c_two_hundred_fifty_six
31958     { 256 } { 2020-01-01 }
31959   \__deprecation_old:Nnn \c_one_thousand
31960     { 1000 } { 2020-01-01 }

```

```

31961 \__deprecation_old:Nnn \c_ten_thousand
31962 { 10000 } { 2020-01-01 }
31963 \__deprecation_old:Nnn \dim_case:nnn
31964 { \dim_case:nnF } { 2015-07-14 }
31965 \__deprecation_old:Nnn \file_add_path:nN
31966 { \file_get_full_name:nN } { 2019-01-01 }
31967 \__deprecation_old_protected:Nnn \file_if_exist_input:nT
31968 { \file_if_exist:nT and~ \file_input:n } { 2018-03-05 }
31969 \__deprecation_old_protected:Nnn \file_if_exist_input:nTF
31970 { \file_if_exist:nT and~ \file_input:n } { 2018-03-05 }
31971 \__deprecation_old:Nnn \file_list:
31972 { \file_log_list: } { 2019-01-01 }
31973 \__deprecation_old:Nnn \file_path_include:n
31974 { \seq_put_right:Nn \l_file_search_path_seq } { 2019-01-01 }
31975 \__deprecation_old:Nnn \file_path_remove:n
31976 { \seq_remove_all:Nn \l_file_search_path_seq } { 2019-01-01 }
31977 \__deprecation_old:Nnn \g_file_current_name_tl
31978 { \g_file_curr_name_str } { 2019-01-01 }
31979 \__deprecation_old:Nnn \int_case:nnn
31980 { \int_case:nnF } { 2015-07-14 }
31981 \__deprecation_old:Nnn \int_from_binary:n
31982 { \int_from_bin:n } { 2016-01-05 }
31983 \__deprecation_old:Nnn \int_from_hexadecimal:n
31984 { \int_from_hex:n } { 2016-01-05 }
31985 \__deprecation_old:Nnn \int_from_octal:n
31986 { \int_from_oct:n } { 2016-01-05 }
31987 \__deprecation_old:Nnn \int_to_binary:n
31988 { \int_to_bin:n } { 2016-01-05 }
31989 \__deprecation_old:Nnn \int_to_hexadecimal:n
31990 { \int_to_hex:n } { 2016-01-05 }
31991 \__deprecation_old:Nnn \int_to_octal:n
31992 { \int_to_oct:n } { 2016-01-05 }
31993 \__deprecation_old_protected:Nnn \ior_get_str:NN
31994 { \ior_str_get:NN } { 2018-03-05 }
31995 \__deprecation_old:Nnn \ior_list_streams:
31996 { \ior_show_list: } { 2019-01-01 }
31997 \__deprecation_old:Nnn \ior_log_streams:
31998 { \ior_log_list: } { 2019-01-01 }
31999 \__deprecation_old:Nnn \iow_list_streams:
32000 { \iow_show_list: } { 2019-01-01 }
32001 \__deprecation_old:Nnn \iow_log_streams:
32002 { \iow_log_list: } { 2019-01-01 }
32003 \__deprecation_old:Nnn \lua_escape_x:n
32004 { \lua_escape:e } { 2020-01-01 }
32005 \__deprecation_old:Nnn \lua_now_x:n
32006 { \lua_now:e } { 2020-01-01 }
32007 \__deprecation_old_protected:Nnn \lua_shipout_x:n
32008 { \lua_shipout_e:n } { 2020-01-01 }
32009 \__deprecation_old:Nnn \luatex_if_engine_p:
32010 { \sys_if_engine luatex_p: } { 2017-01-01 }
32011 \__deprecation_old:Nnn \luatex_if_engine:F
32012 { \sys_if_engine luatex:F } { 2017-01-01 }
32013 \__deprecation_old:Nnn \luatex_if_engine:T
32014 { \sys_if_engine luatex:T } { 2017-01-01 }

```

```

32015 \__deprecation_old:Nnn \luatex_if_engine:TF
32016 { \sys_if_engine luatex:TF } { 2017-01-01 }
32017 \__deprecation_old_protected:Nnn \msg_interrupt:nnn
32018 { [Defined-error-message] } { 2020-01-01 }
32019 \__deprecation_old_protected:Nnn \msg_log:n
32020 { \iow_log:n } { 2020-01-01 }
32021 \__deprecation_old_protected:Nnn \msg_term:n
32022 { \iow_term:n } { 2020-01-01 }
32023 \__deprecation_old:Nnn \pdfTeX_if_engine_p:
32024 { \sys_if_engine pdfTeX_p: } { 2017-01-01 }
32025 \__deprecation_old:Nnn \pdfTeX_if_engine:F
32026 { \sys_if_engine pdfTeX:F } { 2017-01-01 }
32027 \__deprecation_old:Nnn \pdfTeX_if_engine:T
32028 { \sys_if_engine pdfTeX:T } { 2017-01-01 }
32029 \__deprecation_old:Nnn \pdfTeX_if_engine:TF
32030 { \sys_if_engine pdfTeX:TF } { 2017-01-01 }
32031 \__deprecation_old:Nnn \prop_get:cn
32032 { \prop_item:cn } { 2016-01-05 }
32033 \__deprecation_old:Nnn \prop_get:Nn
32034 { \prop_item:Nn } { 2016-01-05 }
32035 \__deprecation_old:Nnn \quark_if_recursion_tail_break:N
32036 { } { 2015-07-14 }
32037 \__deprecation_old:Nnn \quark_if_recursion_tail_break:n
32038 { } { 2015-07-14 }
32039 \__deprecation_old:Nnn \scan_align_safe_stop:
32040 { protected-commands } { 2017-01-01 }
32041 \__deprecation_old:Nnn \sort_ordered:
32042 { \sort_return_same: } { 2019-01-01 }
32043 \__deprecation_old:Nnn \sort_reversed:
32044 { \sort_return_swapped: } { 2019-01-01 }
32045 \__deprecation_old:Nnn \str_case:nnn
32046 { \str_case:nnF } { 2015-07-14 }
32047 \__deprecation_old:Nnn \str_case:onnn
32048 { \str_case:onF } { 2015-07-14 }
32049 \__deprecation_old:Nnn \str_case_x:nn
32050 { \str_case_e:nn } { 2020-01-01 }
32051 \__deprecation_old:Nnn \str_case_x:nnn
32052 { \str_case_e:nnF } { 2015-07-14 }
32053 \__deprecation_old:Nnn \str_case_x:nnT
32054 { \str_case_e:nnT } { 2020-01-01 }
32055 \__deprecation_old:Nnn \str_case_x:nnTF
32056 { \str_case_e:nnTF } { 2020-01-01 }
32057 \__deprecation_old:Nnn \str_case_x:nnF
32058 { \str_case_e:nnF } { 2020-01-01 }
32059 \__deprecation_old:Nnn \str_if_eq_x_p:nn
32060 { \str_if_eq_p:ee } { 2020-01-01 }
32061 \__deprecation_old:Nnn \str_if_eq_x:nnT
32062 { \str_if_eq:eeT } { 2020-01-01 }
32063 \__deprecation_old:Nnn \str_if_eq_x:nnF
32064 { \str_if_eq:eeF } { 2020-01-01 }
32065 \__deprecation_old:Nnn \str_if_eq_x:nnTF
32066 { \str_if_eq:eeTF } { 2020-01-01 }
32067 \__deprecation_old_protected:Nnn \tl_show_analysis:N
32068 { \tl_analysis_show:N } { 2020-01-01 }

```



```

32069 \__deprecation_old_protected:Nnn \tl_show_analysis:n
32070 { \tl_analysis_show:n } { 2020-01-01 }
32071 \__deprecation_old:Nnn \tl_case:cnn
32072 { \tl_case:cnF } { 2015-07-14 }
32073 \__deprecation_old:Nnn \tl_case:Nnn
32074 { \tl_case:NnF } { 2015-07-14 }
32075 \__deprecation_old_protected:Nnn \tl_to_lowercase:n
32076 { \tex_lowercase:D } { 2018-03-05 }
32077 \__deprecation_old_protected:Nnn \tl_to_uppercase:n
32078 { \tex_uppercase:D } { 2018-03-05 }
32079 \__deprecation_old:Nnn \token_new:Nn
32080 { \cs_new_eq:NN } { 2019-01-01 }
32081 \__deprecation_old:Nnn \xetex_if_engine_p:
32082 { \sys_if_engine_xetex_p: } { 2017-01-01 }
32083 \__deprecation_old:Nnn \xetex_if_engine:F
32084 { \sys_if_engine_xetex:F } { 2017-01-01 }
32085 \__deprecation_old:Nnn \xetex_if_engine:T
32086 { \sys_if_engine_xetex:T } { 2017-01-01 }
32087 \__deprecation_old:Nnn \xetex_if_engine:TF
32088 { \sys_if_engine_xetex:TF } { 2017-01-01 }

```

(End definition for __deprecation_old_protected:Nnn and __deprecation_old:Nnn.)

52.4 Loading the patches

When loaded first, the patches are simply read here. Here the deprecation code is loaded with the lower-level __kernel_... macro because we don't want it to flip the \g__sys_deprecation_bool boolean, so that the deprecation code can be re-loaded later (when using undo-recent-deprecations).

```

32089 \group_begin:
32090 \cs_set_protected:Npn \ProvidesExplFile
32091 {
32092   \char_set_catcode_space:n { '\ }
32093   \ProvidesExplFileAux
32094 }
32095 \cs_set_protected:Npx \ProvidesExplFileAux #1#2#3#4
32096 {
32097   \group_end:
32098   \cs_if_exist:NTF \ProvidesFile
32099     { \exp_not:N \ProvidesFile {#1} [ #2~v#3~#4 ] }
32100     { \iow_log:x { File:~#1~#2~v#3~#4 } }
32101 }
32102 \cs_gset_protected:Npn \__kernel_sys_configuration_load:n #1
32103 { \file_input:n { #1 .def } }
32104 \__kernel_sys_configuration_load:n { l3deprecation }
32105 </kernel>
32106 <*patches>

```

Standard file identification.

```

32107 \ProvidesExplFile{l3deprecation.def}{2019-04-06}{L3 Deprecated functions}

```

52.5 Deprecated l3box functions

```

\box_set_eq_clear:NN
\box_set_eq_clear:cN 32108 \__kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \box_set_eq_drop:N }
\box_set_eq_clear:Nc 32109 \cs_gset_protected:Npn \box_set_eq_clear:NN #1#2
\box_set_eq_clear:cc 32110 { \tex_setbox:D #1 \tex_box:D #2 }
\box_gset_eq_clear:NN 32111 \__kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \box_gset_eq_drop:N }
\box_gset_eq_clear:cN 32112 \cs_gset_protected:Npn \box_gset_eq_clear:NN #1#2
\box_gset_eq_clear:Nc 32113 { \tex_global:D \tex_setbox:D #1 \tex_box:D #2 }
\box_gset_eq_clear:cc 32114 \cs_generate_variant:Nn \box_set_eq_clear:NN { c , Nc , cc }
32115 \cs_generate_variant:Nn \box_gset_eq_clear:NN { c , Nc , cc }

(End definition for \box_set_eq_clear:NN and \box_gset_eq_clear:NN.)

```

```

\hbox_unpack_clear:N
\hbox_unpack_clear:c 32116 \__kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \hbox_unpack_drop:N }
32117 \cs_gset_protected:Npn \hbox_unpack_clear:N
32118 { \hbox_unpack_drop:N }
32119 \cs_generate_variant:Nn \hbox_unpack_clear:N { c }

(End definition for \hbox_unpack_clear:N.)

```

```

\vbox_unpack_clear:N
\vbox_unpack_clear:c 32120 \__kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \vbox_unpack_drop:N }
32121 \cs_gset_protected:Npn \vbox_unpack_clear:N
32122 { \vbox_unpack_drop:N }
32123 \cs_generate_variant:Nn \vbox_unpack_clear:N { c }

(End definition for \vbox_unpack_clear:N.)

```

52.6 Deprecated l3str functions

```

\str_lower_case:n
\str_lower_case:f 32124 \__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_lowercase:n }
\str_upper_case:n 32125 \cs_gset:Npn \str_lower_case:n { \str_lowercase:n }
\str_upper_case:f 32126 \__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_lowercase:f }
\str_fold_case:n 32127 \cs_gset:Npn \str_lower_case:f { \str_lowercase:f }
\str_fold_case:V 32128 \__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_uppercase:n }
32129 \cs_gset:Npn \str_upper_case:n { \str_uppercase:n }
32130 \__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_uppercase:f }
32131 \cs_gset:Npn \str_upper_case:f { \str_uppercase:f }
32132 \__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_foldcase:n }
32133 \cs_gset:Npn \str_fold_case:n { \str_foldcase:n }
32134 \__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_foldcase:V }
32135 \cs_gset:Npn \str_fold_case:V { \str_foldcase:V }

(End definition for \str_lower_case:n, \str_upper_case:n, and \str_fold_case:n.)

```

52.7 Deprecated l3seq functions

```

\seq_indexed_map_inline:Nn
\seq_indexed_map_function:NN 32136 \__kernel_patch_deprecation:nnNNpn { 2022-07-01 } { \seq_map_indexed_inline:Nn }
32137 \cs_gset:Npn \seq_indexed_map_inline:Nn { \seq_map_indexed_inline:Nn }
32138 \__kernel_patch_deprecation:nnNNpn { 2022-07-01 } { \seq_map_indexed_function:NN }
32139 \cs_gset:Npn \seq_indexed_map_function:NN { \seq_map_indexed_function:NN }

(End definition for \seq_indexed_map_inline:Nn and \seq_indexed_map_function:NN.)

```

52.7.1 Deprecated l3tl functions

```

32140 <@@=tl>

\tl_set_from_file:Nnn
\tl_set_from_file:cnn
\tl_gset_from_file:Nnn
\tl_gset_from_file:cnn
\tl_set_from_file_x:Nnn
\tl_set_from_file_x:cnn
\tl_gset_from_file_x:Nnn
\tl_gset_from_file_x:cnn

32141 \__kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \file_get:nnN }
32142 \cs_gset_protected:Npn \tl_set_from_file:Nnn #1#2#3
32143 { \file_get:nnN {#3} {#2} #1 }
32144 \cs_generate_variant:Nn \tl_set_from_file:Nnn { c }
32145 \__kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \file_get:nnN }
32146 \cs_gset_protected:Npn \tl_gset_from_file:Nnn #1#2#3
32147 {
32148   \group_begin:
32149     \file_get:nnN {#3} {#2} \l__tl_internal_a_tl
32150     \tl_gset_eq:NN #1 \l__tl_internal_a_tl
32151   \group_end:
32152 }
32153 \cs_generate_variant:Nn \tl_gset_from_file:Nnn { c }
32154 \__kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \file_get:nnN }
32155 \cs_gset_protected:Npn \tl_set_from_file_x:Nnn #1#2#3
32156 {
32157   \group_begin:
32158     \file_get:nnN {#3} {#2} \l__tl_internal_a_tl
32159     #2 \scan_stop:
32160     \tl_set:Nx \l__tl_internal_a_tl { \l__tl_internal_a_tl }
32161     \exp_args:NNNo \group_end:
32162     \tl_set:Nn #1 \l__tl_internal_a_tl
32163   }
32164 \cs_generate_variant:Nn \tl_set_from_file_x:Nnn { c }
32165 \__kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \file_get:nnN }
32166 \cs_gset_protected:Npn \tl_gset_from_file_x:Nnn #1#2#3
32167 {
32168   \group_begin:
32169     \file_get:nnN {#3} {#2} \l__tl_internal_a_tl
32170     #2 \scan_stop:
32171     \tl_gset:Nx #1 { \l__tl_internal_a_tl }
32172   \group_end:
32173 }
32174 \cs_generate_variant:Nn \tl_gset_from_file_x:Nnn { c }

(End definition for \tl_set_from_file:Nnn and others.)

\tl_lower_case:n
\tl_lower_case:nn
\tl_upper_case:n
\tl_upper_case:nn
\tl_mixed_case:n
\tl_mixed_case:nn

32175 \__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_lowercase:n }
32176 \cs_gset:Npn \tl_lower_case:n #1
32177 { \text_lowercase:n {#1} }
32178 \__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_lowercase:nn }
32179 \cs_gset:Npn \tl_lower_case:nn #1#2
32180 { \text_lowercase:nn {#1} {#2} }
32181 \__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_uppercase:n }
32182 \cs_gset:Npn \tl_upper_case:n #1
32183 { \text_uppercase:n {#1} }
32184 \__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_uppercase:nn }
32185 \cs_gset:Npn \tl_upper_case:nn #1#2
32186 { \text_uppercase:nn {#1} {#2} }

```

```

32187 \kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_titlecase:n }
32188 \cs_gset:Npn \tl_mixed_case:n #1
32189 { \text_titlecase:n {#1} }
32190 \kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_titlecase:nn }
32191 \cs_gset:Npn \tl_mixed_case:nn #1#2
32192 { \text_titlecase:nn {#1} {#2} }

```

(End definition for \tl_lower_case:n and others.)

52.8 Deprecated l3token functions

```

\token_get_prefix_spec:N
\token_get_arg_spec:N
\token_get_replacement_spec:N
32193 \kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \cs_prefix_spec:N }
32194 \cs_gset:Npn \token_get_prefix_spec:N { \cs_prefix_spec:N }
32195 \kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \cs_argument_spec:N }
32196 \cs_gset:Npn \token_get_arg_spec:N { \cs_argument_spec:N }
32197 \kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \cs_replacement_spec:N }
32198 \cs_gset:Npn \token_get_replacement_spec:N { \cs_replacement_spec:N }

```

(End definition for \token_get_prefix_spec:N, \token_get_arg_spec:N, and \token_get_replacement_spec:N.)

```

\char_lower_case:N
\char_upper_case:N
\char_mixed_case:Nn
\char_fold_case:N
\char_str_lower_case:N
\char_str_upper_case:N
\char_str_mixed_case:Nn
\char_str_fold_case:N
32199 \kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \char_lowercase:N }
32200 \cs_gset:Npn \char_lower_case:N { \char_lowercase:N }
32201 \kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \char_uppercase:N }
32202 \cs_gset:Npn \char_upper_case:N { \char_uppercase:N }
32203 \kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \char_titlecase:N }
32204 \cs_gset:Npn \char_mixed_case:N { \char_titlecase:N }
32205 \kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \char_foldcase:N }
32206 \cs_gset:Npn \char_fold_case:N { \char_foldcase:N }
32207 \kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \char_str_lowercase:N }
32208 \cs_gset:Npn \char_str_lower_case:N { \char_str_lowercase:N }
32209 \kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \char_str_uppercase:N }
32210 \cs_gset:Npn \char_str_upper_case:N { \char_str_uppercase:N }
32211 \kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \char_str_titlecase:N }
32212 \cs_gset:Npn \char_str_mixed_case:N { \char_str_titlecase:N }
32213 \kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \char_str_foldcase:N }
32214 \cs_gset:Npn \char_str_fold_case:N { \char_str_foldcase:N }

```

(End definition for \char_lower_case:N and others.)

52.9 Deprecated l3file functions

```

\c_term_ior
32215 \kernel_patch_deprecation:nnNNpn { 2021-01-01 } { -1 }
32216 \cs_gset_protected:Npn \c_term_ior { -1 \scan_stop: }

```

(End definition for \c_term_ior.)

```

32217 </patches>
32218 </package>

```

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols	
!	214
\"	10263, 10266, 29167, 31005, 31029, 31035, 31039, 31045, 31049, 31055, 31061, 31068, 31069, 31075, 31079, 31081, 31190
\#	5179, 5488, 10263, 12801, 30874
\\$	5178, 5486, 10263, 10266, 24421, 30874
\%	5180, 5496, 10263, 12803, 30874
\&	5171, 5487, 10263, 10266, 11590
&&	213
\'	29167, 30999, 31026, 31033, 31037, 31042, 31047, 31050, 31052, 31059, 31064, 31065, 31072, 31077, 31080, 31088, 31089, 31136, 31137, 31144, 31145, 31156, 31157, 31162, 31163, 31191, 31192, 31214, 31215, 31218, 31219, 31220, 31221
\(13324, 13542, 13644, 29188
\)	29188
*	214
*	4864, 4887, 10650, 10652, 10656, 10664
**	214
+	213, 214
\,	14524, 30886
-	213, 214
\-	188
\.	29167, 31004, 31092, 31093, 31102, 31103, 31112, 31113, 31130, 31142, 31143, 31193, 31194, 31224, 31225, 31228, 31229
/	214
\/	187
\:	5177
\::	37, 338, 364, 2185, 2186, 2187, 2188, 2189, 2190, 2191, 2193, 2197, 2198, 2201, 2204, 2211, 2214, 2217, 2223, 2310, 2379, 2381, 2386, 2393, 2397, 2400, 2405, 2420, 2461, 2462, 2463, 2464, 2475
\::N	37, 2189, 2310, 2464
\::V	37, 2217
\::V_unbraced	37, 2378
\::c	37, 2191
\::e	37, 2195, 2310
\::e_unbraced	37, 2378, 2420
\::error	157, 12404
\::f	37, 2204, 2463
\::f_unbraced	37, 2378
\::n	37, 375, 2188, 2461, 2464
\::o	37, 2193, 2462
\::o_unbraced	37, 2378, 2461, 2462, 2463, 2464
\::p	37, 338, 2190
\::v	37, 2217
\::v_unbraced	37, 2378
\::x	37, 2211
\::x_unbraced	37, 2378, 2475
<	213
=	213
\=	14525, 29167, 31002, 31082, 31083, 31098, 31099, 31121, 31122, 31123, 31150, 31151, 31176, 31177, 31230, 31231
>	213
?	213
? commands:	
?:	213
\	2124, 5173, 5483, 5723, 5724, 5727, 6049, 6052, 6053, 6054, 6055, 6060, 6066, 6071, 6078, 6235, 6238, 6239, 6240, 6242, 6248, 6253, 6258, 6409, 6416, 10263, 11493, 11511, 11513, 11518, 11519, 11543, 11553, 11560, 11575, 12019, 12027, 12034, 12046, 12047, 12072, 12073, 12080, 12101, 12103, 12104, 12136, 12149, 12150, 12163, 12230, 12263, 12279, 12283, 12288, 12295, 12806, 13849, 15611, 15623, 15629, 16425, 16428, 16429, 16430, 16437, 16440, 16441, 22601, 22603, 22604, 22607, 22609, 22610, 22613, 22615, 22616, 22617, 22621, 22628, 23015, 23018, 23019, 23043, 23044, 23051, 23052, 23493, 24962, 25075, 25076, 25077, 25098, 26397, 26401, 26406, 26440, 26449, 26453, 26458, 26478, 26480, 26481, 26483, 26486, 26488, 26495, 26499, 26502, 26506, 26508, 26512, 26514, 26520, 26522, 26526, 26528, 26532, 26537, 26539, 26581, 26583, 26588, 26590, 26596, 26601, 26602, 26606, 26610, 26620, 26623, 26627, 26628, 26632, 26640, 26697, 28631, 30872, 31573

- $\backslash{}$ 5174, 5484,
 5728, 10263, 12800, 23747, 26401,
 26406, 26453, 26502, 26539, 26628,
 26632, 29195, 29196, 29197, 30874
 $\backslash}$ 5175, 5485, 5728, 10263, 12802, 26400,
 26406, 26503, 26539, 26628, 26632,
 29195, 29196, 29197, 29198, 30874
 \backsim ... 84, 1812, 2505, 5176, 5489, 5842,
 5843, 6176, 6177, 6358, 6359, 6360,
 10263, 10266, 10268, 10274, 10325,
 11597, 12718, 12754, 20305, 23098,
 23171, 23174, 23693, 23698, 23699,
 23700, 23701, 23704, 23715, 23752,
 23806, 23808, 23810, 23812, 23814,
 23816, 24420, 26002, 26005, 26019,
 26022, 26031, 26034, 26037, 26040,
 26054, 26057, 29167, 31000, 31027,
 31034, 31038, 31043, 31048, 31053,
 31060, 31066, 31067, 31073, 31078,
 31090, 31091, 31108, 31109, 31116,
 31117, 31131, 31132, 31133, 31164,
 31165, 31186, 31187, 31188, 31189
 $\hat{}$ 214
 $\backslash_$ 5182, 10263, 10266, 30874
 $\backslash^$.. 29167, 30998, 31025, 31032, 31036,
 31041, 31046, 31051, 31058, 31062,
 31063, 31071, 31076, 31216, 31217
 $\|$ 213
 \backsim 3712, 5181, 5491,
 10263, 10266, 12804, 23737, 23741,
 23747, 29167, 30878, 31001, 31028,
 31040, 31044, 31054, 31070, 31074,
 31118, 31119, 31120, 31174, 31175
- A**
- $\backslash A$ 4865, 4888
 $\backslash AA$ 29171, 30582, 30923
 $\backslash aa$ 29171, 30582, 30933
 $\backslash above$ 189
 $\backslash abovedisplayshortskip$ 190
 $\backslash abovedisplayskip$ 191
 $\backslash abovewithdelims$ 192
 $\backslash abs$ 214
 $\backslash accent$ 193
 $\backslash acos$ 216
 $\backslash acosd$ 216
 $\backslash acot$ 217
 $\backslash acotd$ 217
 $\backslash acsc$ 216
 $\backslash acscd$ 216
 $\backslash adjdemerits$ 194
 $\backslash adjustspacing$ 907
 $\backslash advance$ 195
 $\backslash AE$ 29172, 30583, 30924, 31218
 $\backslash ae$ 29172, 30583, 30934, 31219
 $\backslash afterassignment$ 196
 $\backslash aftergroup$ 197
 $\backslash alignmark$ 782
 $\backslash aligntab$ 783
 $\backslash asec$ 216
 $\backslash ascd$ 216
 $\backslash asin$ 216
 $\backslash asind$ 216
 assert commands:
 $\backslash assert_int:n$ 25238, 26215
 $\backslash atan$ 217
 $\backslash atand$ 217
 $\backslash AtBeginDocument$ 660, 13793,
 28413, 28414, 28423, 28424, 30600
 $\backslash atop$ 198
 $\backslash atopwithdelims$ 199
 $\backslash attribute$ 784
 $\backslash attributedef$ 785
 $\backslash automaticdiscretionary$ 786
 $\backslash automatichyphenmode$ 788
 $\backslash automatichyphenpenalty$ 789
 $\backslash autospacing$ 1106
 $\backslash autoxspacing$ 1107
- B**
- $\backslash b$ 29167, 31012
 $\backslash badness$ 200
 $\backslash baselineskip$ 201
 $\backslash batchmode$ 202
 $\backslash begin$ 227,
 231, 18389, 29184, 29191, 29372, 30870
 begin internal commands:
 $_regex_begin$ 26138
 $\backslash begincsname$ 791
 $\backslash begingroup$. 3, 20, 24, 29, 47, 103, 181, 203
 $\backslash beginL$ 511
 $\backslash beginR$ 512
 $\backslash belowdisplayshortskip$ 204
 $\backslash belowdisplayskip$ 205
 $\backslash bfseries$ 30851
 $\backslash binoppenalty$ 206
 $\backslash bodydir$ 792
 $\backslash bodydirection$ 793
 bool commands:
 $\backslash bool_case_false:n$ 269, 31399

- \bool_case_false:nTF
..... [269](#), [31399](#), [31409](#), [31411](#)
- \bool_case_true:n [269](#), [31399](#)
- \bool_case_true:nTF
..... [269](#), [31399](#), [31401](#), [31403](#)
- \bool_const:Nn [109](#), [8935](#)
- \bool_do_until:Nn [112](#), [9132](#)
- \bool_do_until:nn [113](#), [9138](#)
- \bool_do_while:Nn [112](#), [9132](#)
- \bool_do_while:nn [113](#), [9138](#)
- .bool_gset:N [188](#), [14950](#)
- \bool_gset:Nn [109](#), [8957](#)
- \bool_gset_eq:NN
..... [109](#), [8953](#), [23683](#), [25551](#)
- \bool_gset_false:N
..... [109](#), [5367](#), [5376](#), [8941](#), [25510](#), [31395](#)
- .bool_gset_inverse:N [188](#), [14958](#)
- \bool_gset_inverse:N [268](#), [31391](#)
- \bool_gset_true:N [109](#),
[5357](#), [8941](#), [9327](#), [9333](#), [25561](#), [31395](#)
- \bool_if:NTF [109](#),
[145](#), [1993](#), [5371](#), [5380](#), [8974](#), [9127](#),
[9129](#), [9133](#), [9135](#), [9325](#), [9331](#), [13021](#),
[13028](#), [14701](#), [14917](#), [14926](#), [15116](#),
[15118](#), [15120](#), [15166](#), [15168](#), [15170](#),
[15208](#), [15210](#), [15212](#), [15228](#), [15230](#),
[15232](#), [15271](#), [15312](#), [15331](#), [15333](#),
[15338](#), [15345](#), [15409](#), [15438](#), [15448](#),
[15476](#), [24554](#), [24563](#), [24966](#), [25044](#),
[25062](#), [25091](#), [25188](#), [25408](#), [25416](#),
[26670](#), [26675](#), [27803](#), [28518](#), [29801](#),
[31392](#), [31395](#), [31417](#), [31798](#), [31871](#)
- \bool_if:nTF
[109](#), [111](#), [113](#), [113](#), [113](#), [113](#), [586](#),
[8988](#), [9006](#), [9077](#), [9084](#), [9103](#), [9110](#),
[9119](#), [9140](#), [9149](#), [9153](#), [9162](#), [9231](#),
[25047](#), [31422](#), [31428](#), [31482](#), [31821](#)
- \bool_if_exist:NTF
..... [110](#), [9002](#), [14718](#), [14734](#)
- \bool_if_exist_p:N [110](#), [9002](#)
- \bool_if_p:N [109](#), [8974](#)
- \bool_if_p:n
..... [111](#), [530](#), [8938](#), [8960](#), [8965](#),
[9006](#), [9014](#), [9084](#), [9110](#), [9116](#), [9120](#)
- \bool_lazy_all:nTF
..... [111](#), [111](#), [111](#), [9064](#), [24831](#)
- \bool_lazy_all_p:n [111](#), [9064](#)
- \bool_lazy_and:nnTF [111](#), [111](#), [111](#),
[9081](#), [9425](#), [12619](#), [13123](#), [22446](#),
[29468](#), [30089](#), [30142](#), [30170](#), [30774](#)
- \bool_lazy_and_p:nn .. [111](#), [111](#), [9081](#)
- \bool_lazy_any:nTF
..... [111](#), [112](#), [112](#), [5499](#), [5523](#), [5545](#),
[5712](#), [9090](#), [13439](#), [29154](#), [29257](#), [29966](#)
- \bool_lazy_any_p:n
..... [111](#), [112](#), [9090](#), [13128](#), [30092](#)
- \bool_lazy_or:nnTF
..... [111](#), [112](#), [112](#), [6494](#),
[9107](#), [10635](#), [13316](#), [13350](#), [13396](#),
[13532](#), [22555](#), [23104](#), [28861](#), [29062](#),
[29274](#), [29282](#), [29753](#), [29767](#), [29803](#),
[29808](#), [29843](#), [29891](#), [29927](#), [30021](#),
[30028](#), [30106](#), [30151](#), [30184](#), [30215](#),
[30262](#), [30285](#), [30319](#), [30887](#), [30972](#)
- \bool_lazy_or_p:nn
..... [112](#), [9107](#), [29471](#), [30173](#), [30777](#)
- \bool_log:N [109](#), [8989](#)
- \bool_log:n [110](#), [8983](#)
- \bool_new:N [108](#), [5220](#),
[8933](#), [8998](#), [8999](#), [9000](#), [9001](#), [9321](#),
[9322](#), [12749](#), [14606](#), [14607](#), [14614](#),
[14615](#), [14619](#), [14718](#), [14734](#), [23534](#),
[23961](#), [25467](#), [25468](#), [25470](#), [25471](#),
[25472](#), [27403](#), [29511](#), [31768](#), [31784](#)
- \bool_not_p:n [112](#), [9116](#), [13126](#)
- .bool_set:N [188](#), [14950](#)
- \bool_set:Nn [109](#), [523](#), [8957](#)
- \bool_set_eq:NN [109](#), [8953](#), [23677](#), [25704](#)
- \bool_set_false:N
..... [109](#), [160](#), [8941](#), [12855](#), [12997](#),
[13005](#), [13013](#), [13023](#), [13030](#), [14643](#),
[15110](#), [15111](#), [15112](#), [15162](#), [15163](#),
[15167](#), [15203](#), [15211](#), [15213](#), [15222](#),
[15223](#), [15233](#), [15254](#), [15319](#), [24528](#),
[24733](#), [25446](#), [25523](#), [25537](#), [25583](#),
[25645](#), [27799](#), [28516](#), [31392](#), [31787](#)
- .bool_set_inverse:N [188](#), [14958](#)
- \bool_set_inverse:N [268](#), [31391](#)
- \bool_set_true:N
..... [109](#), [174](#), [8941](#), [12983](#), [14638](#),
[15117](#), [15119](#), [15121](#), [15161](#), [15169](#),
[15171](#), [15204](#), [15205](#), [15209](#), [15224](#),
[15229](#), [15231](#), [15249](#), [15326](#), [24533](#),
[24737](#), [25440](#), [25643](#), [25703](#), [27817](#),
[27839](#), [27870](#), [29512](#), [31392](#), [31797](#)
- \bool_show:N [109](#), [8989](#)
- \bool_show:n [109](#), [8983](#)
- \bool_until_do:Nn [112](#), [9126](#)
- \bool_until_do:nn [113](#), [9138](#)
- \bool_while_do:Nn [112](#), [9126](#)
- \bool_while_do:nn [113](#), [9138](#)
- \bool_xor:nnTF [112](#), [9117](#)
- \bool_xor_p:nn [112](#), [9117](#)
- \c_false_bool .. [22](#), [108](#), [324](#), [357](#),
[524](#), [526](#), [527](#), [527](#), [527](#), [528](#), [1570](#),
[1622](#), [1623](#), [1654](#), [1673](#), [1678](#), [1710](#),
[1729](#), [1930](#), [1937](#), [2838](#), [3112](#), [8933](#),
[8944](#), [8948](#), [9055](#), [9078](#), [9084](#), [9102](#),

- 9242, 24205, 24223, 24437, 24473,
 24772, 24914, 24931, 24987, 25124,
 26317, 31408, 31410, 31412, 31414
 \g_tmpa_bool 110, 8998
 \l_tmpa_bool 110, 8998
 \g_tmpb_bool 110, 8998
 \l_tmpb_bool 110, 8998
 \c_true_bool 22, 108, 324, 389, 524,
 526, 527, 527, 527, 528, 1622, 1654,
 1710, 1728, 1951, 8942, 8946, 9056,
 9057, 9076, 9104, 9110, 9236, 23541,
 23680, 24109, 24169, 24219, 24403,
 24428, 24472, 24479, 24912, 24922,
 24985, 25174, 25346, 25357, 25372,
 25527, 25560, 26148, 26225, 26278,
 31400, 31402, 31404, 31406, 31825
 bool internal commands:
 bool!:Nw 9035
 bool&_0: 9047
 bool&_1: 9047
 bool&_2: 9047
 bool(Nw 9040
 bool)_0: 9047
 bool)_1: 9047
 bool)_2: 9047
 _bool_case:NnTF 31399
 _bool_case_end:nw 31399
 _bool_case_false:w 31399
 _bool_case_true:w 31399
 _bool_choose:NNN .. 9042, 9046, 9047
 _bool_get_next:NN
 . 527, 527, 9022, 9025, 9037, 9043,
 9058, 9059, 9060, 9061, 9062, 9063
 _bool_if_p:n 9014
 _bool_if_p_aux:w 526, 9014
 _bool_if_recursion_tail_stop_-
 do:nn 8973, 9076, 9102
 _bool_lazy_all:n 9064
 _bool_lazy_any:n 9090
 _bool_p:Nw 9045
 _bool_show:NN 8989
 _bool_to_str:n 8983, 8996
 _bool_use_i_delimit_by_q_-
 recursion_stop:nw 8971, 9078, 9104
 bool|_0: 9047
 bool|_1: 9047
 bool|_2: 9047
 \botmark 207
 \botmarks 513
 \box 208
 box commands:
 \box_autosize_to_wd_and_ht:Nnn ..
 248, 27309
 \box_autosize_to_wd_and_ht_plus_-
 dp:Nnn 248, 27309
 \box_clear:N 240,
 240, 26712, 26719, 27438, 27525, 27599
 \box_clear_new:N 240, 26718
 \box_clip:N 265, 266, 266, 31249
 \box_dp:N 241, 16913, 26740,
 26749, 26753, 27054, 27183, 27298,
 27317, 27323, 27636, 27637, 27743,
 27748, 27776, 27790, 27963, 28241,
 28262, 28583, 31269, 31276, 31281
 \box_gautosize_to_wd_and_ht:Nnn .
 248, 27309
 \box_gautosize_to_wd_and_ht_-
 plus_dp:Nnn 248, 27309
 \box_gclear:N 240, 26712, 26721, 27447
 \box_gclear_new:N 240, 26718
 \box_gclip:N 265, 31249
 \box_gresize_to_ht:Nn ... 248, 27202
 \box_gresize_to_ht_plus_dp:Nn ...
 249, 27202
 \box_gresize_to_wd:Nn ... 249, 27202
 \box_gresize_to_wd_and_ht:Nnn ...
 249, 27202
 \box_gresize_to_wd_and_ht_plus_-
 dp:Nnn 249, 27153, 28077
 \box_grotate:Nn ... 250, 27035, 27910
 \box_gscale:Nnn ... 250, 27280, 28119
 \box_gset_dp:Nn 241, 26746
 \box_gset_eq:NN
 240, 26715, 26724, 27621, 31259, 31310
 \box_gset_eq_clear:NN 32108
 \box_gset_eq_drop:N 32111
 \box_gset_eq_drop:NN 247, 26730
 \box_gset_ht:Nn 241, 26746
 \box_gset_to_last:N 242, 26800
 \box_gset_trim:Nnnnn 266, 31255
 \box_gset_viewport:Nnnnn . 266, 31306
 \box_gset_wd:Nn 242, 26746
 \box_ht:N 241, 16912, 26740,
 26758, 26762, 27053, 27182, 27297,
 27310, 27313, 27317, 27323, 27520,
 27594, 27638, 27639, 27734, 27739,
 27776, 27783, 27957, 27961, 28240,
 28261, 28581, 31286, 31294, 31299
 \box_if_empty:NnTF 242, 26796
 \box_if_empty_p:N 242, 26796
 \box_if_exist:NnTF
 240, 26719, 26721, 26736, 26831
 \box_if_exist_p:N 240, 26736
 \box_if_horizontal:NnTF ... 242, 26788
 \box_if_horizontal_p:N ... 242, 26788
 \box_if_vertical:NnTF 242, 26788
 \box_if_vertical_p:N 242, 26788

- `\box_log:N` [243](#), [26817](#)
- `\box_log:Nnn` [243](#), [26817](#)
- `\box_move_down:nn` [241](#), [1170](#), [26777](#),
[27936](#), [31273](#), [31281](#), [31324](#), [31331](#)
- `\box_move_left:nn` [241](#), [26777](#)
- `\box_move_right:nn` [241](#), [26777](#)
- `\box_move_up:nn` [241](#), [26777](#), [28281](#),
[28578](#), [31290](#), [31299](#), [31338](#), [31351](#)
- `\box_new:N` [240](#),
[240](#), [26706](#), [26806](#), [26807](#), [26808](#),
[26809](#), [26810](#), [27034](#), [27379](#), [27454](#)
- `\box_resize:Nnn` [31909](#)
- `\box_resize_to_ht:Nn` [248](#), [27202](#)
- `\box_resize_to_ht_plus_dp:Nn` ...
..... [249](#), [27202](#)
- `\box_resize_to_wd:Nn` [249](#), [27202](#)
- `\box_resize_to_wd_and_ht:Nnn` ...
..... [249](#), [27202](#)
- `\box_resize_to_wd_and_ht_plus_dp:Nnn` ... [249](#), [27153](#), [28070](#), [31910](#)
- `\box_rotate:Nn` [250](#), [27035](#), [27907](#)
- `\box_scale:Nnn` [250](#), [27280](#), [28116](#)
- `\box_set_dp:Nn`
..... [241](#), [1171](#), [26746](#), [27080](#),
[27351](#), [27354](#), [27941](#), [28241](#), [28262](#),
[28582](#), [31276](#), [31284](#), [31327](#), [31332](#)
- `\box_set_eq:NN` . [240](#), [26713](#), [26724](#),
[27609](#), [28264](#), [28586](#), [31256](#), [31307](#)
- `\box_set_eq_clear:NN` [32108](#)
- `\box_set_eq_drop:N` [32108](#)
- `\box_set_eq_drop:NN` [247](#), [26730](#)
- `\box_set_ht:Nn` . [241](#), [26746](#), [27079](#),
[27350](#), [27355](#), [27939](#), [28240](#), [28261](#),
[28580](#), [31293](#), [31302](#), [31341](#), [31354](#)
- `\box_set_to_last:N` [242](#), [26800](#)
- `\box_set_trim:Nnnnn` [266](#), [31255](#)
- `\box_set_viewport:Nnnnn` .. [266](#), [31306](#)
- `\box_set_wd:Nn` . [242](#), [26746](#), [27081](#),
[27367](#), [27942](#), [28242](#), [28263](#), [28584](#)
- `\box_show:N` [243](#), [247](#), [26811](#)
- `\box_show:Nnn` [243](#), [26811](#)
- `\box_use:N` [240](#), [241](#),
[241](#), [26773](#), [27068](#), [27937](#), [28278](#),
[28281](#), [28575](#), [28578](#), [31266](#), [31317](#)
- `\box_use_clear:N` [31911](#)
- `\box_use_drop:N` [247](#), [26773](#),
[27083](#), [27362](#), [27371](#), [27944](#), [28364](#),
[28510](#), [31274](#), [31282](#), [31291](#), [31300](#),
[31325](#), [31331](#), [31339](#), [31352](#), [31912](#)
- `\box_wd:N` [241](#),
[16911](#), [26740](#), [26767](#), [26771](#), [27055](#),
[27184](#), [27299](#), [27331](#), [27640](#), [27641](#),
[27738](#), [27747](#), [27765](#), [27770](#), [27960](#),
[27968](#), [28162](#), [28169](#), [28195](#), [28242](#),
[28263](#), [28279](#), [28576](#), [28585](#), [31318](#)
- `\c_empty_box`
... [240](#), [242](#), [242](#), [26713](#), [26715](#), [26806](#)
- `\g_tmpa_box` [243](#), [26807](#)
- `\l_tmpa_box` [243](#), [26807](#)
- `\g_tmpb_box` [243](#), [26807](#)
- `\l_tmpb_box` [243](#), [26807](#)
- box internal commands:
 - `\l_box_angle_fp`
... [27023](#), [27045](#), [27046](#), [27047](#), [27076](#)
 - `_box_autosize:NnnnN` [27309](#)
 - `_box_backend_clip:N` . [31250](#), [31253](#)
 - `_box_backend_rotate:Nn` [27074](#)
 - `_box_backend_scale:Nnn` [27343](#)
 - `\l_box_bottom_dim` [27026](#),
[27054](#), [27111](#), [27115](#), [27120](#), [27126](#),
[27131](#), [27135](#), [27144](#), [27146](#), [27175](#),
[27183](#), [27192](#), [27236](#), [27298](#), [27304](#)
 - `\l_box_bottom_new_dim`
[27030](#), [27080](#), [27112](#), [27123](#), [27134](#),
[27145](#), [27191](#), [27303](#), [27351](#), [27355](#)
 - `\l_box_cos_fp` [27024](#),
[27047](#), [27059](#), [27064](#), [27091](#), [27103](#)
 - `_box_dim_eval:n`
..... [26703](#), [26749](#), [26753](#), [26758](#),
[26762](#), [26767](#), [26771](#), [26778](#), [26780](#),
[26782](#), [26784](#), [26863](#), [26868](#), [26895](#),
[26901](#), [26909](#), [26931](#), [26965](#), [26970](#),
[26998](#), [27004](#), [27015](#), [27020](#), [31265](#),
[31267](#), [31316](#), [31318](#), [31327](#), [31351](#)
 - `_box_dim_eval:w` [26703](#)
 - `\l_box_internal_box` [27034](#), [27068](#),
[27069](#), [27075](#), [27079](#), [27080](#), [27081](#),
[27083](#), [27341](#), [27350](#), [27351](#), [27354](#),
[27355](#), [27362](#), [27367](#), [27371](#), [31263](#),
[31271](#), [31274](#), [31276](#), [31279](#), [31282](#),
[31284](#), [31286](#), [31288](#), [31291](#), [31293](#),
[31294](#), [31297](#), [31299](#), [31300](#), [31302](#),
[31304](#), [31314](#), [31322](#), [31325](#), [31327](#),
[31330](#), [31331](#), [31332](#), [31336](#), [31339](#),
[31341](#), [31349](#), [31352](#), [31354](#), [31356](#)
 - `\l_box_left_dim` ... [27026](#), [27056](#),
[27111](#), [27113](#), [27122](#), [27126](#), [27131](#),
[27137](#), [27142](#), [27146](#), [27185](#), [27300](#)
 - `\l_box_left_new_dim` [27030](#), [27071](#),
[27082](#), [27114](#), [27125](#), [27136](#), [27147](#)
 - `_box_log:nNnn` [26817](#)
 - `_box_resize:N`
... [27153](#), [27219](#), [27239](#), [27256](#), [27277](#)
 - `_box_resize:NNN` [27153](#)
 - `_box_resize_common:N`
..... [27195](#), [27307](#), [27339](#)

- _box_resize_set_corners:N 27153, 27212, 27232, 27252, 27269
 - _box_resize_to_ht:NnN 27202
 - _box_resize_to_ht_plus_dp:NnN 27202
 - _box_resize_to_wd:NnN 27202
 - _box_resize_to_wd_and_ht:NnnN 27260, 27263, 27265
 - _box_resize_to_wd_and_ht_plus_dp:NnnN 27153
 - _box_resize_to_wd_ht:NnnN 27202
 - \l_box_right_dim 27026, 27055, 27109, 27115, 27120, 27124, 27133, 27135, 27144, 27148, 27171, 27184, 27190, 27254, 27271, 27299, 27306
 - \l_box_right_new_dim 27030, 27082, 27116, 27127, 27138, 27149, 27189, 27305, 27359, 27361, 27367
 - _box_rotate:N 27035
 - _box_rotate:NnN 27035
 - _box_rotate_quadrant_four: 27035, 27140
 - _box_rotate_quadrant_one: 27035, 27107
 - _box_rotate_quadrant_three: 27035, 27129
 - _box_rotate_quadrant_two: 27035, 27118
 - _box_rotate_xdir:nnN 27035, 27085, 27113, 27115, 27124, 27126, 27135, 27137, 27146, 27148
 - _box_rotate_ydir:nnN 27035, 27096, 27109, 27111, 27120, 27122, 27131, 27133, 27142, 27144
 - _box_scale:N 27280, 27336
 - _box_scale:NnnN 27280
 - \l_box_scale_x_fp 27151, 27170, 27190, 27218, 27238, 27253, 27255, 27270, 27290, 27306, 27331, 27333, 27334, 27335, 27345, 27357
 - \l_box_scale_y_fp 27151, 27172, 27192, 27194, 27213, 27218, 27233, 27238, 27255, 27272, 27291, 27302, 27304, 27332, 27333, 27334, 27335, 27346, 27348
 - _box_set_trim:NnnnnN 31255
 - _box_set_viewport:NnnnnN 31307, 31310, 31312
 - _box_show:NnNn 26815, 26825, 26829
 - \l_box_sin_fp 27024, 27046, 27057, 27092, 27102
 - \l_box_top_dim 27026, 27053, 27109, 27113, 27122, 27124, 27133, 27137, 27142, 27148, 27175, 27182, 27194, 27216, 27236, 27275, 27297, 27302
 - \l_box_top_new_dim 27030, 27079, 27110, 27121, 27132, 27143, 27193, 27301, 27350, 27354
 - _box_viewport:NnnnnN 31306
 - \boxdir 794
 - \boxdirection 795
 - \boxmaxdepth 209
 - bp 219
 - \breakafterdirmode 796
 - \brokenpenalty 210
- C**
- \c 29167, 31010, 31031, 31057, 31114, 31115, 31134, 31135, 31138, 31139, 31146, 31147, 31158, 31159, 31166, 31167, 31170, 31171, 31226, 31227
 - \catcode 120, 121, 122, 123, 124, 125, 126, 127, 128, 132, 133, 134, 135, 136, 137, 138, 139, 140, 211
 - catcode commands:
 - \c_catcode_active_space_tl 273, 31711
 - \c_catcode_active_tl 135, 575, 10663, 10723
 - \c_catcode_letter_token 135, 575, 10596, 10645, 10713, 23294, 29135
 - \c_catcode_other_space_tl 131, 638, 10642, 12763, 12807, 12887, 12976, 13052
 - \c_catcode_other_token 135, 575, 10599, 10645, 10718, 23292, 29138
 - \catcodetable 797
 - cc 219
 - cctab commands:
 - \cctab_begin:N 223, 223, 926, 927, 930, 931, 932, 932, 933, 934, 22412
 - \cctab_const:Nn 223, 223, 22528, 22535, 22542, 22583
 - \cctab_end: 223, 223, 926, 927, 930, 931, 932, 933, 934, 22426
 - \cctab_gset:Nn 223, 223, 22342, 22531
 - \cctab_if_exist:N 22483
 - \cctab_if_exist:NTF 224, 22490
 - \cctab_if_exist_p:N 224
 - \cctab_new:N 223, 927, 927, 22277, 22530, 22534
 - \cctab_select:N 46, 46, 223, 223, 223, 22347, 22364, 22537, 22544, 22585
 - \c_code_cctab 224, 22551, 22581, 22585
 - \c_document_cctab 224, 929, 22547
 - \c_initex_cctab 224, 22347, 22534
 - \c_other_cctab 224, 22534, 22547
 - \c_str_cctab 224, 930, 22534

cctab internal commands:

\g__cctab_allocate_int
 [22273](#), [22405](#), [22407](#), [22409](#)
 __cctab_begin_aux:
 [931](#), [931](#), [22393](#), [22417](#)
 __cctab_chk_group_begin:n
 [932](#), [22418](#), [22437](#)
 __cctab_chk_group_end:n
 [932](#), [22431](#), [22437](#)
 __cctab_chk_if_valid:NTF
 [22344](#), [22365](#), [22414](#), [22487](#)
 __cctab_chk_if_valid_aux:NTF . [22487](#)
 \g__cctab_endlinechar_prop
 [928](#), [22276](#), [22323](#), [22325](#), [22372](#)
 \g__cctab_group_seq
 [22272](#), [22439](#), [22445](#)
 __cctab_gset:n
 [22315](#), [22349](#), [22421](#), [22581](#)
 __cctab_gset_aux:n [22315](#)
 __cctab_gstore:Nnn [22277](#)
 \l__cctab_internal_a_tl
 [931](#), [931](#), [932](#), [22274](#), [22372](#), [22373](#),
 [22398](#), [22408](#), [22416](#), [22419](#), [22420](#),
 [22421](#), [22428](#), [22430](#), [22432](#), [22433](#)
 \l__cctab_internal_b_tl
 [22274](#), [22445](#), [22449](#), [22456](#)
 \g__cctab_internal_cctab [22354](#)
 __cctab_internal_cctab_name: ...
 [22354](#), [22375](#), [22376](#), [22377](#), [22378](#)
 __cctab_nesting_number:N
 [22419](#), [22432](#), [22461](#)
 __cctab_nesting_number:w [22461](#)
 __cctab_new:N .. [927](#), [931](#), [22277](#),
 [22356](#), [22376](#), [22397](#), [22406](#), [22551](#)
 \g__cctab_next_cctab [22393](#)
 __cctab_select:N
 [930](#), [22364](#), [22422](#), [22433](#)
 \g__cctab_stack_seq
 [926](#), [927](#), [22270](#), [22420](#), [22428](#), [22479](#)
 \g__cctab_unused_seq
 [926](#), [927](#), [931](#), [932](#), [22270](#), [22416](#), [22430](#)

ceil [215](#)

\char [212](#), [10828](#)

char commands:

\l_char_active_seq .. [134](#), [159](#), [10261](#)
 \char_fold_case:N [32199](#)
 \char_foldcase:N
 [131](#), [10521](#), [32205](#), [32206](#)
 \char_generate:nn
 [45](#), [131](#), [383](#), [437](#), [1042](#),
 [1182](#), [3693](#), [3709](#), [5281](#), [5554](#), [5570](#),
 [10288](#), [10514](#), [10518](#), [10549](#), [10578](#),
 [10633](#), [10642](#), [12336](#), [23831](#), [24841](#),
 [29066](#), [29104](#), [29783](#), [29793](#), [29794](#),

..... [29900](#), [29986](#), [30063](#), [30064](#), [30065](#),
 [30076](#), [30101](#), [30129](#), [30156](#), [30179](#),
 [30196](#), [30208](#), [30220](#), [30225](#), [30250](#),
 [30268](#), [30297](#), [30299](#), [30303](#), [30339](#),
 [30340](#), [30345](#), [30347](#), [30355](#), [30356](#),
 [30361](#), [30363](#), [30556](#), [30557](#), [30896](#),
 [30917](#), [30919](#), [30978](#), [30992](#), [30994](#)
 \char_gset_active_eq:NN .. [130](#), [10267](#)
 \char_gset_active_eq:nN .. [130](#), [10267](#)
 \char_lower_case:N [32199](#)
 \char_lowercase:N
 [131](#), [10521](#), [32199](#), [32200](#)
 \char_mixed_case:N [32204](#)
 \char_mixed_case:Nn [32199](#)
 \char_set_active_eq:NN ... [130](#), [10267](#)
 \char_set_active_eq:nN ... [130](#), [10267](#)
 \char_set_catcode:nn .. [133](#), [149](#),
 [150](#), [151](#), [152](#), [153](#), [154](#), [155](#), [156](#),
 [157](#), [10167](#), [10174](#), [10176](#), [10178](#),
 [10180](#), [10182](#), [10184](#), [10186](#), [10188](#),
 [10190](#), [10192](#), [10194](#), [10196](#), [10198](#),
 [10200](#), [10202](#), [10204](#), [10206](#), [10208](#),
 [10210](#), [10212](#), [10214](#), [10216](#), [10218](#),
 [10220](#), [10222](#), [10224](#), [10226](#), [10228](#),
 [10230](#), [10232](#), [10234](#), [10236](#), [22386](#)
 \char_set_catcode_active:N
 [132](#), [10173](#), [10268](#), [10325](#), [10664](#),
 [11590](#), [23098](#), [26002](#), [30878](#), [31712](#)
 \char_set_catcode_active:n
 [132](#), [10205](#), [10387](#), [14524](#),
 [14525](#), [22558](#), [22565](#), [22591](#), [29034](#)
 \char_set_catcode_alignment:N ...
 [132](#), [5487](#), [10173](#), [10652](#), [26054](#)
 \char_set_catcode_alignment:n ...
 [132](#), [167](#), [10205](#), [10371](#), [22571](#)
 \char_set_catcode_comment:N
 [132](#), [5496](#), [10173](#)
 \char_set_catcode_comment:n
 [132](#), [10205](#), [22570](#)
 \char_set_catcode_end_line:N ...
 [132](#), [10173](#)
 \char_set_catcode_end_line:n ...
 [132](#), [10205](#), [22566](#)
 \char_set_catcode_escape:N
 [132](#), [5483](#), [10173](#)
 \char_set_catcode_escape:n
 [132](#), [10205](#), [22573](#)
 \char_set_catcode_group_begin:N .
 [132](#), [5484](#), [10173](#), [23171](#), [26005](#)
 \char_set_catcode_group_begin:n .
 [132](#), [10205](#), [10364](#), [22576](#)
 \char_set_catcode_group_end:N ...
 [132](#), [5485](#), [10173](#), [23174](#), [26022](#)

- \char_set_catcode_group_end:n . . .
..... [132](#), [10205](#), [10366](#), [22578](#)
- \char_set_catcode_ignore:N
..... [132](#), [5490](#), [10173](#)
- \char_set_catcode_ignore:n
.. [132](#), [164](#), [165](#), [10205](#), [22563](#), [22567](#)
- \char_set_catcode_invalid:N
..... [132](#), [10173](#)
- \char_set_catcode_invalid:n
.... [132](#), [10205](#), [22554](#), [22557](#), [22580](#)
- \char_set_catcode_letter:N
..... [132](#), [5493](#), [10173](#), [18530](#), [18531](#), [26031](#)
- \char_set_catcode_letter:n
..... [132](#), [168](#), [170](#), [10205](#),
[10383](#), [22560](#), [22562](#), [22572](#), [22575](#)
- \char_set_catcode_math_subscript:N
..... [132](#), [10173](#), [10656](#), [26019](#)
- \char_set_catcode_math_subscript:n
..... [132](#), [10205](#), [10378](#), [22590](#)
- \char_set_catcode_math_superscript:N
..... [132](#), [5489](#), [10173](#), [26057](#)
- \char_set_catcode_math_superscript:n
..... [132](#), [169](#), [10205](#), [10376](#), [22574](#)
- \char_set_catcode_math_toggle:N .
..... [132](#), [5486](#), [10173](#), [10650](#), [26034](#)
- \char_set_catcode_math_toggle:n .
..... [132](#), [10205](#), [10369](#), [22569](#)
- \char_set_catcode_other:N . . . [132](#),
[929](#), [5495](#), [5842](#), [5843](#), [6176](#), [6177](#),
[6358](#), [6359](#), [6360](#), [10173](#), [23331](#), [26037](#)
- \char_set_catcode_other:n
..... [132](#), [166](#),
[171](#), [10205](#), [10327](#), [10385](#), [22540](#),
[22559](#), [22561](#), [22564](#), [22577](#), [22589](#)
- \char_set_catcode_parameter:N . . .
..... [132](#), [5488](#), [10173](#), [26040](#)
- \char_set_catcode_parameter:n . . .
..... [132](#), [10205](#), [10374](#), [22568](#)
- \char_set_catcode_space:N
..... [132](#), [5491](#), [10173](#)
- \char_set_catcode_space:n . . . [132](#),
[172](#), [10205](#), [10381](#), [13811](#), [22545](#),
[22579](#), [22587](#), [22588](#), [28923](#), [32092](#)
- \char_set_lccode:nn
.. [133](#), [10237](#), [10274](#), [10391](#), [10392](#),
[11586](#), [11587](#), [11588](#), [11589](#), [31713](#)
- \char_set_mathcode:nn . . . [134](#), [10237](#)
- \char_set_sfcode:nn [134](#), [10237](#)
- \char_set_uccode:nn [133](#), [10237](#)
- \char_show_value_catcode:n [133](#), [10167](#)
- \char_show_value_lccode:n [133](#), [10237](#)
- \char_show_value_mathcode:n
..... [134](#), [10237](#)
- \char_show_value_sfcode:n [134](#), [10237](#)
- \char_show_value_uccode:n [134](#), [10237](#)
- \l_char_special_seq [134](#), [10261](#)
- \char_str_fold_case:N [32199](#)
- \char_str_foldcase:N
..... [131](#), [10521](#), [32213](#), [32214](#)
- \char_str_lower_case:N [32199](#)
- \char_str_lowercase:N
..... [131](#), [10521](#), [32207](#), [32208](#)
- \char_str_mixed_case:N [32212](#)
- \char_str_mixed_case:Nn [32199](#)
- \char_str_titlecase:N
..... [131](#), [10521](#), [32211](#), [32212](#)
- \char_str_upper_case:N [32199](#)
- \char_str_uppercase:N
..... [131](#), [10521](#), [32209](#), [32210](#)
- \char_titlecase:N
..... [131](#), [10521](#), [32203](#), [32204](#)
- \char_to_nfd:N [273](#), [10499](#), [29936](#)
- \char_to_utfviii_bytes:n
..... [273](#), [6507](#), [10421](#), [30308](#),
[30329](#), [30330](#), [30563](#), [30909](#), [30986](#)
- \char_upper_case:N [32199](#)
- \char_uppercase:N
..... [131](#), [10521](#), [32201](#), [32202](#)
- \char_value_catcode:n [133](#),
[149](#), [150](#), [151](#), [152](#), [153](#), [154](#), [155](#),
[156](#), [157](#), [3705](#), [3709](#), [10167](#), [10518](#),
[22336](#), [29067](#), [30251](#), [30897](#), [30979](#)
- \char_value_lccode:n . . . [133](#), [10237](#),
[10522](#), [10535](#), [10612](#), [10622](#), [28938](#)
- \char_value_mathcode:n . . . [134](#), [10237](#)
- \char_value_sfcode:n [134](#), [10237](#)
- \char_value_uccode:n
..... [134](#), [10237](#), [10524](#), [10614](#)
- char internal commands:
 __char_change_case:NN [10521](#)
- __char_change_case:nN [10521](#)
- __char_change_case:NNN [10521](#)
- __char_change_case:nNN [10521](#)
- __char_change_case:NNNN [10521](#)
- __char_change_case_catcode:N . . .
 [10514](#), [10521](#)
- __char_change_case_multi:nN . [10521](#)
- __char_change_case_multi:NNNNw .
 [10521](#)
- __char_data_auxi:w [28889](#),
[28929](#), [28934](#), [28962](#), [28967](#), [29000](#)
- __char_data_auxii:w
 [28895](#), [28899](#), [28945](#),
[28949](#), [28970](#), [28971](#), [28973](#), [28975](#)
- __char_data_auxiii:w . . . [28897](#), [28909](#)
- g__char_data_ior . . . [28860](#), [28888](#),
[28924](#), [28932](#), [28933](#), [28959](#), [28965](#),
[28966](#), [28989](#), [29002](#), [29018](#), [29019](#)

- _char_generate:n 28866,
28904, 28906, 28918, 28941, 28953,
28954, 28956, 28982, 28983, 28985
- _char_generate_aux:nn 10288
- _char_generate_aux:nnw 10288
- _char_generate_aux:w . 10290, 10294
- _char_generate_auxii:nnw ... 10288
- _char_generate_char:n .. 28864,
28902, 28917, 28940, 28951, 28980
- _char_generate_invalid_-
catcode: 10288
- _char_int_to_roman:w
..... 10287, 10396, 10415
- _char_quark_if_no_value:NTF ...
..... 10166, 10556, 10558
- _char_quark_if_no_value_p:N . 10166
- _char_str_change_case:nN ... 10521
- _char_str_change_case:nNN .. 10521
- _char_tmp:n
..... 10389, 10400, 10403, 10405
- _char_tmp:NN .. 29007, 29013, 29015
- _char_tmp:nN .. 10269, 10280, 10281
- \l_char_tmp_tl 10288
- \l_char_tmpa_tl 28912,
28913, 28915, 28924, 28926, 28929
- \l_char_tmptb_tl 28914, 28915
- _char_to_nfd:n 10499
- _char_to_nfd:Nw 10499
- _char_to_utfviii_bytes_auxi:n .
..... 10421
- _char_to_utfviii_bytes_-
auxii:Nnn 10421
- _char_to_utfviii_bytes_-
auxiii:n 10421
- _char_to_utfviii_bytes_end: . 10421
- _char_to_utfviii_bytes_-
output:nnn 10421
- _char_to_utfviii_bytes_-
outputi:nw 10421
- _char_to_utfviii_bytes_-
outputii:nw 10421
- _char_to_utfviii_bytes_-
outputiii:nw 10421
- _char_to_utfviii_bytes_-
outputiv:nw 10421
- \chardef 130, 142, 213, 934
- choice commands:
.choice: 188, 14966
- choices commands:
.choices:nn 188, 14968
- \cite 29184, 29191
- \cleaders 214
- \clearmarks 798
- clist commands:
\clist_clear:N 121,
121, 9626, 9643, 9806, 15149, 15191
- \clist_clear_new:N 121, 9630
- \clist_concat:NNN 121, 9669, 9695, 9708
- \clist_const:Nn 121, 9623
- \clist_count:N
126, 128, 10006, 10035, 10067, 10133
- \clist_count:n 126, 10006, 10098, 10124
- \clist_gclear:N 121, 9626, 9645
- \clist_gclear_new:N 121, 9630
- \clist_gconcat:NNN
..... 121, 9669, 9697, 9710
- \clist_get:NN 127, 9720
- \clist_get:NNTF 127, 9757
- \clist_gpop:NN 127, 9731
- \clist_gpop:NNTF 128, 9757
- \clist_gpush:Nn 128, 9782
- \clist_gput_left:Nn
..... 122, 9694, 9790, 9791,
9792, 9793, 9794, 9795, 9796, 9797
- \clist_gput_right:Nn 122, 9707
- \clist_gremove_all:Nn 123, 9816
- \clist_gremove_duplicates:N 123, 9800
- \clist_greverse:N 123, 9855
- .clist_gset:N 188, 14978
- \clist_gset:Nn 122, 5583, 9688
- \clist_gset_eq:NN ... 121, 9634, 9803
- \clist_gset_from_seq:NN
..... 121, 9642, 9819, 22781
- \clist_gsort:Nn 123, 9873, 22766
- \clist_if_empty:NTF 124, 9678, 9840,
9873, 9930, 9960, 9980, 10132, 14828
- \clist_if_empty:nTF 124, 9877
- \clist_if_empty_p:N 124, 9873
- \clist_if_empty_p:n 124, 9877
- \clist_if_exist:NTF
..... 121, 9684, 10033, 13679, 13779
- \clist_if_exist_p:N 121, 9684
- \clist_if_in:NnTF 120, 124, 9809, 9891
- \clist_if_in:nnTF ... 124, 9891, 16298
- \clist_item:Nn
..... 128, 560, 560, 10064, 10133
- \clist_item:nn 128, 560, 10095, 10128
- \clist_log:N 129, 10136
- \clist_log:n 129, 10150
- \clist_map_break: 125, 9934, 9939,
9948, 9952, 9968, 9986, 10002, 15373
- \clist_map_break:n 126, 9911,
10002, 15327, 15402, 22774, 22780
- \clist_map_function:NN .. 38, 124,
7415, 7425, 9914, 9928, 10011, 10146
- \clist_map_function:Nn 557

- \clist_map_function:nN [124, 270, 270, 557, 5586, 7420, 7430, 7441, 9944, 10155, 15504](#)
- \clist_map_inline:Nn [124, 125, 555, 9542, 9807, 9958, 15322, 15364, 15393, 22774, 22780](#)
- \clist_map_inline:nn [125, 3150, 9958, 13882, 14787, 14879, 15767, 28669, 28681](#)
- \clist_map_variable:NNn ... [125, 9978](#)
- \clist_map_variable:nNn ... [125, 9978](#)
- \clist_new:N [120, 121, 544, 9621, 9798, 10157, 10158, 10159, 10160, 14602](#)
- \clist_pop:NN [127, 9731](#)
- \clist_pop:NNTF [127, 9757](#)
- \clist_push:Nn [128, 9782](#)
- \clist_put_left:Nn [122, 9694, 9782, 9783, 9784, 9785, 9786, 9787, 9788, 9789](#)
- \clist_put_right:Nn [122, 9707, 9810, 15435, 15445, 15473](#)
- \clist_rand_item:N [128, 10123](#)
- \clist_rand_item:n .. [117, 128, 10123](#)
- \clist_remove_all:Nn [123, 9557, 9816](#)
- \clist_remove_duplicates:N [120, 123, 9800](#)
- \clist_reverse:N [123, 9855](#)
- \clist_reverse:n [123, 553, 9856, 9858, 9861](#)
- .clist_set:N [188, 14978](#)
- \clist_set:Nn [122, 9688, 9695, 9697, 9708, 9710, 9897, 9974, 9991, 14827](#)
- \clist_set_eq:NN [121, 9634, 9801, 15307](#)
- \clist_set_from_seq:NN [121, 9642, 9817, 22775](#)
- \clist_show:N [128, 129, 10136](#)
- \clist_show:n [129, 129, 10150](#)
- \clist_sort:Nn [123, 9873, 22766](#)
- \clist_use:Nn [127, 10031](#)
- \clist_use:Nnnn ... [126, 495, 10031](#)
- \c_empty_clist [129, 9565, 9722, 9737, 9759, 9773](#)
- \l_foo_clist [225](#)
- \g_tmpa_clist [129, 10157](#)
- \l_tmpa_clist [129, 10157](#)
- \g_tmpb_clist [129, 10157](#)
- \l_tmpb_clist [129, 10157](#)
- clist internal commands:
 - __clist_concat:NNNN [9669](#)
 - __clist_count:n [10006](#)
 - __clist_count:w [10006](#)
 - __clist_get:wN [9720, 9762](#)
 - __clist_if_empty_n:w [9877](#)
 - __clist_if_empty_n:wNw [9877](#)
 - __clist_if_in_return:nnN [9891](#)
 - __clist_if_recursion_tail-break:nN [9573, 9939, 9952](#)
 - __clist_if_recursion_tail-stop:n [9573, 9590, 9996, 10027](#)
 - __clist_if_wrap:nTF [545, 9595, 9620, 9661, 9822, 9903](#)
 - __clist_if_wrap:w [545, 9595](#)
 - \l_clist_internal_clist ... [548, 9566, 9700, 9701, 9713, 9714, 9897, 9898, 9899, 9974, 9975, 9991, 9992](#)
 - \l_clist_internal_remove_clist [9798, 9806, 9809, 9810, 9812](#)
 - \l_clist_internal_remove_seq ... [9798, 9824, 9825, 9826](#)
 - __clist_item:nnnN [10064, 10097](#)
 - __clist_item_n:nw [10095](#)
 - __clist_item_n_end:n [10095](#)
 - __clist_item_N_loop:nw [10064](#)
 - __clist_item_n_loop:nw [10095](#)
 - __clist_item_n_strip:n [10095](#)
 - __clist_item_n_strip:w [10095](#)
 - __clist_map_function:Nw [555, 9928, 9965](#)
 - __clist_map_function_n:Nn [556, 9944](#)
 - __clist_map_unbrace:Nw ... [556, 9944](#)
 - __clist_map_variable:Nnw [9978](#)
 - __clist_pop:NNN [9731](#)
 - __clist_pop:wN [9731](#)
 - __clist_pop:wwNNN .. [549, 9731, 9776](#)
 - __clist_pop_TF:NNN [9757](#)
 - __clist_put_left:NNNn [9694](#)
 - __clist_put_right:NNNn [9707](#)
 - __clist_rand_item:nn [10123](#)
 - __clist_remove_all: [9816](#)
 - __clist_remove_all:NNNn [9816](#)
 - __clist_remove_all:w [552, 9816](#)
 - __clist_remove_duplicates:NN . [9800](#)
 - __clist_reverse:wwNww [553, 9861](#)
 - __clist_reverse_end:ww ... [553, 9861](#)
 - __clist_sanitizate:n [9582, 9624, 9689, 9691](#)
 - __clist_sanitizate:Nn [545, 9582](#)
 - __clist_set_from_seq:n [9642](#)
 - __clist_set_from_seq:NNNN ... [9642](#)
 - __clist_show:NN [10136](#)
 - __clist_show:Nn [10150](#)
 - __clist_tmp:w [552, 9575, 9829, 9851, 9905, 9914, 9918, 9920](#)
 - __clist_trim_next:w [545, 556, 9576, 9585, 9593, 9947, 9955](#)
 - __clist_use:nwn [10031](#)
 - __clist_use:nwwwnwn ... [558, 10031](#)

- _clist_use:wnn [10031](#)
- _clist_use_i_delimit_by_s_
 - stop:nw [9569](#), [10091](#)
 - stop:w [552](#), [9569](#), [9832](#), [10077](#), [10082](#)
- _clist_wrap_item:w [545](#), [9591](#), [9619](#)
- \closein [215](#)
- \closeout [216](#)
- \clubpenalties [514](#)
- \clubpenalty [217](#)
- cm [219](#)
- code commands:
 - .code:n [188](#), [14976](#)
- coffin commands:
 - \coffin_attach:NnnNnnnn
 - [253](#), [1110](#), [28224](#)
 - \coffin_clear:N [251](#), [27434](#)
 - \coffin_display_handles:Nn [254](#), [28488](#)
 - \coffin_dp:N
 - [254](#), [27636](#), [28088](#), [28127](#), [28602](#)
 - \coffin_gattach:NnnNnnnn . [253](#), [28224](#)
 - \coffin_gclear:N [251](#), [27434](#)
 - \coffin_gjoin:NnnNnnnn ... [253](#), [28173](#)
 - \coffin_gresize:Nnn [253](#), [28067](#)
 - \coffin_grotate:Nn [253](#), [27906](#)
 - \coffin_gscale:Nnn [253](#), [28115](#)
 - \coffin_gset_eq:NN
 - [251](#), [27605](#), [28182](#), [28233](#)
 - \coffin_gset_horizontal_pole:Nnn
 - [252](#), [27666](#)
 - \coffin_gset_vertical_pole:Nnn ..
 - [252](#), [27666](#)
 - \coffin_ht:N
 - [254](#), [27636](#), [28088](#), [28127](#), [28601](#)
 - \coffin_if_exist:NTF [251](#), [27413](#), [27427](#)
 - \coffin_if_exist_p:N [251](#), [27413](#)
 - \coffin_join:NnnNnnnn ... [253](#), [28173](#)
 - \coffin_log_structure:N .. [255](#), [28588](#)
 - \coffin_mark_handle:Nnnn . [254](#), [28443](#)
 - \coffin_new:N
 - [251](#), [1085](#), [27452](#), [27629](#),
 - [27630](#), [27631](#), [27632](#), [27633](#), [27634](#),
 - [27635](#), [28357](#), [28367](#), [28368](#), [28369](#)
 - \coffin_resize:Nnn [253](#), [28067](#)
 - \coffin_rotate:Nn [253](#), [27906](#)
 - \coffin_scale:Nnn [253](#), [28115](#)
 - \coffin_scale:NnnNN [28115](#)
 - \coffin_set_eq:NN [251](#), [27605](#),
 - [28176](#), [28227](#), [28255](#), [28283](#), [28504](#)
 - \coffin_set_horizontal_pole:Nnn .
 - [252](#), [27666](#)
 - \coffin_set_vertical_pole:Nnn ...
 - [252](#), [27666](#)
- \coffin_show_structure:N
 - [254](#), [255](#), [28588](#)
- \coffin_typeset:Nnnnn ... [254](#), [28359](#)
- \coffin_wd:N
 - [254](#), [27636](#), [28084](#), [28131](#), [28603](#)
- \c_empty_coffin [255](#), [27629](#)
- \g_tmpa_coffin [255](#), [27632](#)
- \l_tmpa_coffin [255](#), [27632](#)
- \g_tmpb_coffin [255](#), [27632](#)
- \l_tmpb_coffin [255](#), [27632](#)
- coffin internal commands:
 - _coffin_align:NnnNnnnnN
 - .. [28187](#), [28238](#), [28259](#), [28266](#), [28362](#)
 - \l_coffin_aligned_coffin
 - [27629](#), [28188](#),
 - [28189](#), [28193](#), [28199](#), [28202](#), [28205](#),
 - [28221](#), [28222](#), [28239](#), [28240](#), [28241](#),
 - [28242](#), [28243](#), [28246](#), [28250](#), [28254](#),
 - [28255](#), [28260](#), [28261](#), [28262](#), [28263](#),
 - [28264](#), [28297](#), [28313](#), [28363](#), [28364](#),
 - [28573](#), [28580](#), [28582](#), [28584](#), [28586](#)
 - \l_coffin_aligned_internal_
 - coffin [27629](#), [28276](#), [28283](#)
 - _coffin_attach:NnnNnnnnN ... [28224](#)
 - _coffin_attach_mark:NnnNnnnn ..
 - [28224](#), [28450](#), [28466](#), [28482](#)
 - \l_coffin_bottom_corner_dim ...
 - [27902](#), [27936](#), [27940](#),
 - [28019](#), [28030](#), [28031](#), [28051](#), [28059](#)
 - \l_coffin_bounding_prop
 - [27898](#), [27925](#), [27956](#),
 - [27958](#), [27964](#), [27966](#), [27975](#), [28038](#)
 - \l_coffin_bounding_shift_dim ...
 - .. [27901](#), [27934](#), [28037](#), [28043](#), [28044](#)
 - _coffin_calculate_intersection:Nnn
 - [27795](#), [28268](#), [28271](#), [28566](#)
 - _coffin_calculate_intersection:nnnnnn
 - [27795](#)
 - _coffin_calculate_intersection:nnnnnnnn
 - [27795](#), [28517](#)
 - _coffin_color:n
 - .. [28423](#), [28447](#), [28454](#), [28492](#), [28527](#)
 - \c_coffin_corners_prop
 - [27382](#), [27459](#), [27655](#), [27662](#)
 - \l_coffin_corners_prop
 - [27899](#), [27916](#), [27920](#), [27945](#),
 - [27950](#), [27981](#), [28021](#), [28048](#), [28095](#),
 - [28099](#), [28105](#), [28111](#), [28146](#), [28160](#)
 - \l_coffin_cos_fp
 - [1093](#), [1095](#), [27896](#), [27915](#), [28002](#), [28011](#)
 - _coffin_display_attach:Nnnnn [28488](#)
 - \l_coffin_display_coffin
 - [28367](#), [28504](#), [28510](#), [28575](#),
 - [28576](#), [28581](#), [28583](#), [28585](#), [28586](#)

\l__coffin_display_coord_coffin .
 [28367](#), [28452](#),
 [28467](#), [28483](#), [28525](#), [28540](#), [28559](#)
 \l__coffin_display_font_tl
 [28412](#), [28455](#), [28528](#)
 __coffin_display_handles_-
 aux:nnnn [28488](#)
 __coffin_display_handles_-
 aux:nnnnnn [28488](#)
 \l__coffin_display_handles_prop .
 .. [28370](#), [28458](#), [28462](#), [28531](#), [28535](#)
 \l__coffin_display_offset_dim ...
 .. [28407](#), [28484](#), [28485](#), [28560](#), [28561](#)
 \l__coffin_display_pole_coffin ..
 .. [28367](#), [28445](#), [28451](#), [28490](#), [28523](#)
 \l__coffin_display_poles_prop ...
 [28411](#), [28495](#),
 [28500](#), [28503](#), [28505](#), [28507](#), [28514](#)
 \l__coffin_display_x_dim
 [28409](#), [28520](#), [28570](#)
 \l__coffin_display_y_dim
 [28409](#), [28521](#), [28572](#)
 \c__coffin_empty_coffin [28357](#), [28362](#)
 \l__coffin_error_bool
 [27403](#), [27799](#), [27803](#),
 [27817](#), [27839](#), [27870](#), [28516](#), [28518](#)
 __coffin_find_bounding_shift: ..
 [27928](#), [28035](#)
 __coffin_find_bounding_shift_-
 aux:nn [28035](#)
 __coffin_find_corner_maxima:N ..
 [27927](#), [28015](#)
 __coffin_find_corner_maxima_-
 aux:nn [28015](#)
 __coffin_get_pole:NnN
 [27642](#), [27797](#), [27798](#), [28324](#), [28325](#),
 [28328](#), [28329](#), [28497](#), [28498](#), [28501](#)
 __coffin_greset_structure:N ...
 [27448](#), [27652](#), [27716](#)
 __coffin_gupdate:N
 .. [27487](#), [27500](#), [27556](#), [27574](#), [27708](#)
 __coffin_gupdate_corners:N
 [27717](#), [27720](#)
 __coffin_gupdate_poles:N
 [27718](#), [27751](#)
 __coffin_if_exist:NTF
 [27425](#), [27436](#), [27445](#), [27467](#),
 [27480](#), [27505](#), [27537](#), [27550](#), [27579](#),
 [27607](#), [27619](#), [27674](#), [27692](#), [28596](#)
 \l__coffin_internal_box
 [27379](#), [27514](#),
 [27520](#), [27525](#), [27588](#), [27594](#), [27599](#),
 [27930](#), [27939](#), [27941](#), [27942](#), [27944](#)
 \l__coffin_internal_dim
 [27379](#), [27963](#), [27965](#), [27969](#),
 [28126](#), [28129](#), [28194](#), [28196](#), [28197](#)
 \l__coffin_internal_tl ... [27379](#),
 [28295](#), [28296](#), [28298](#), [28459](#), [28460](#),
 [28463](#), [28464](#), [28472](#), [28477](#), [28532](#),
 [28533](#), [28536](#), [28537](#), [28546](#), [28551](#)
 __coffin_join:NnnNnnnnN [28173](#)
 \l__coffin_left_corner_dim
 [27902](#), [27934](#), [27943](#),
 [28020](#), [28026](#), [28027](#), [28050](#), [28058](#)
 __coffin_mark_handle_aux:nnnnNnn
 [28443](#)
 __coffin_offset_corner:Nnnnn . [28304](#)
 __coffin_offset_corners:Nnn ...
 .. [28210](#), [28211](#), [28217](#), [28218](#), [28304](#)
 __coffin_offset_pole:Nnnnnnn . [28285](#)
 __coffin_offset_poles:Nnn
 [28208](#), [28209](#),
 [28214](#), [28215](#), [28251](#), [28252](#), [28285](#)
 \l__coffin_offset_x_dim
 [27404](#), [28191](#), [28192](#), [28195](#),
 [28206](#), [28208](#), [28210](#), [28216](#), [28219](#),
 [28253](#), [28272](#), [28280](#), [28569](#), [28577](#)
 \l__coffin_offset_y_dim
 [27404](#), [28209](#), [28211](#), [28216](#), [28219](#),
 [28253](#), [28274](#), [28281](#), [28571](#), [28578](#)
 \l__coffin_pole_a_tl
 [27406](#), [27797](#), [27802](#), [28324](#), [28327](#),
 [28328](#), [28331](#), [28497](#), [28499](#), [28502](#)
 \l__coffin_pole_b_tl [27406](#),
 [27798](#), [27802](#), [28325](#), [28327](#), [28329](#),
 [28331](#), [28498](#), [28499](#), [28501](#), [28502](#)
 \c__coffin_poles_prop
 [27389](#), [27461](#), [27657](#), [27664](#)
 \l__coffin_poles_prop
 [27899](#), [27918](#), [27922](#),
 [27947](#), [27952](#), [27989](#), [28056](#), [28097](#),
 [28101](#), [28107](#), [28113](#), [28152](#), [28167](#)
 __coffin_reset_structure:N
 .. [27439](#), [27652](#), [27710](#), [28199](#), [28243](#)
 __coffin_resize:NnnNN [28067](#)
 __coffin_resize_common:NnnN ...
 [28091](#), [28093](#), [28132](#)
 \l__coffin_right_corner_dim
 .. [27902](#), [27943](#), [28018](#), [28028](#), [28029](#)
 __coffin_rotate:NnNNN [27906](#)
 __coffin_rotate_bounding:nnn ...
 [27926](#), [27972](#)
 __coffin_rotate_corner:Nnnn ...
 [27921](#), [27972](#)
 __coffin_rotate_pole:Nnnnnn ...
 [27923](#), [27984](#)


```

\__coffin_rotate_vector:nnNN ...
    .. 27974, 27980, 27986, 27987, 27996
\__coffin_rule:nn 28437, 28448, 28493
\__coffin_scale:NnnNN .....
    ..... 28116, 28119, 28121
\__coffin_scale_corner:Nnnn ....
    ..... 28100, 28143
\__coffin_scale_pole:Nnnnnn ....
    ..... 28102, 28143
\__coffin_scale_vector:nnNN ....
    ..... 28136, 28145, 28151
\l__coffin_scale_x_fp 28063, 28083,
    28103, 28123, 28125, 28131, 28139
\l__coffin_scale_y_fp ... 28063,
    28085, 28124, 28125, 28129, 28141
\l__coffin_scaled_total_height_-
    dim ..... 28065, 28128, 28133
\l__coffin_scaled_width_dim ....
    ..... 28065, 28130, 28133
\__coffin_set_bounding:N 27924, 27954
\__coffin_set_horizontal_-
    pole:NnnN ..... 27666
\__coffin_set_pole:Nnn .....
    ..... 27515, 27589, 27666,
    28297, 28337, 28341, 28349, 28353
\__coffin_set_vertical:NnnNN . 27491
\__coffin_set_vertical:NnnNNw 27563
\__coffin_set_vertical_aux: ....
    ..... 27491, 27583
\__coffin_set_vertical_pole:NnnN
    ..... 27666
\__coffin_shift_corner:Nnnn ....
    ..... 27946, 28046
\__coffin_shift_pole:Nnnnnn ....
    ..... 27948, 28046
\__coffin_show_structure:NN .. 28588
\l__coffin_sin_fp .....
    1093, 1095, 27896, 27914, 28003, 28010
\l__coffin_slope_A_fp ..... 27401
\l__coffin_slope_B_fp ..... 27401
\__coffin_to_value:N .....
    27412, 27417, 27456, 27457, 27458,
    27460, 27610, 27611, 27612, 27613,
    27622, 27623, 27624, 27625, 27645,
    27654, 27656, 27661, 27663, 27676,
    27694, 27704, 27727, 27758, 27917,
    27919, 27949, 27951, 28096, 28098,
    28110, 28112, 28202, 28246, 28249,
    28287, 28306, 28313, 28496, 28607
\l__coffin_top_corner_dim .....
    .. 27902, 27940, 28017, 28032, 28033
\__coffin_update:N .....
    .. 27474, 27494, 27543, 27567, 27708
\__coffin_update_B:nnnnnnnnN . 28322

\__coffin_update_corners:N .....
    ..... 27711, 27720
\__coffin_update_corners:NN .. 27720
\__coffin_update_corners:NNN . 27720
\__coffin_update_poles:N .....
    ..... 27712, 27751, 28205, 28250
\__coffin_update_poles:NN .... 27751
\__coffin_update_poles:NNN ... 27751
\__coffin_update_T:nnnnnnnnN . 28322
\__coffin_update_vertical_-
    poles:NNN .... 28221, 28254, 28322
\l__coffin_x_dim ... 27408, 27806,
    27815, 27841, 27872, 27890, 27974,
    27976, 27980, 27982, 27986, 27991,
    28145, 28147, 28151, 28154, 28269,
    28273, 28292, 28300, 28520, 28567
\l__coffin_x_prime_dim .....
    ..... 27408, 27988,
    27992, 28269, 28273, 28567, 28570
\__coffin_x_shift_corner:Nnnn ...
    ..... 28106, 28158
\__coffin_x_shift_pole:Nnnnnn ...
    ..... 28108, 28158
\l__coffin_y_dim ..... 27408,
    27807, 27819, 27837, 27886, 27974,
    27976, 27980, 27982, 27986, 27991,
    28145, 28147, 28151, 28154, 28270,
    28275, 28293, 28300, 28521, 28568
\l__coffin_y_prime_dim .....
    ..... 27408, 27988,
    27993, 28270, 28275, 28568, 28572
\color ..... 28432, 28433
color commands:
\color_ensure_current: . 256, 1082,
    27471, 27484, 27539, 27552, 28639
\color_group_begin: .....
    ..... 256, 256, 26848,
    26852, 26857, 26864, 26869, 26877,
    26883, 26897, 26903, 26910, 26915,
    26926, 26928, 26932, 26937, 26942,
    26947, 26954, 26959, 26966, 26971,
    26979, 26985, 27000, 27006, 28637
\color_group_end: ..... 256, 256,
    26848, 26852, 26857, 26864, 26869,
    26889, 26910, 26915, 26926, 26928,
    26932, 26937, 26942, 26947, 26954,
    26959, 26966, 26971, 26992, 28637
\color_select:n ..... 28429, 28430
color internal commands:
\__color_backend_pickup:N .... 28641
\l__color_current_tl .....
    ... 1112, 28637, 28641, 28642, 28649
\__color_select:N ..... 28642, 28645
\__color_select:nn ..... 28645

```

- `\columnwidth` 27532, 27533
- `\copy` 218
- `\copyfont` 908
- `cos` 216
- `cosd` 216
- `cot` 216
- `cotd` 216
- `\count` 219, 10836
- `\countdef` 220
- `\cr` 221
- `\crampeddisplaystyle` 799
- `\crampedscriptscriptstyle` 800
- `\crampedscriptstyle` 802
- `\crampedtextstyle` 803
- `\crrcr` 222
- `\creationdate` 772
- `\cs` 18384, 29370
- cs commands:
 - `\cs:w` 17, 1040, 1040, 1374, 1396, 1398, 1451, 1758, 1786, 1979, 2043, 2192, 2241, 2250, 2252, 2256, 2257, 2258, 2320, 2326, 2332, 2338, 2358, 2360, 2365, 2372, 2373, 2438, 2442, 2481, 3099, 5292, 5298, 8151, 8237, 8874, 8926, 9173, 9175, 10505, 13899, 14196, 14243, 14309, 14335, 15427, 16060, 16079, 16146, 16969, 17158, 17190, 17608, 17634, 17647, 17683, 17727, 18229, 19934, 21026, 25939, 25942, 26709, 31363
 - `\cs_argument_spec:N` 18, 2129, 32195, 32196
 - `\cs_end:` 17, 364, 1374, 1396, 1398, 1402, 1451, 1752, 1758, 1780, 1786, 1906, 1979, 2043, 2192, 2241, 2250, 2252, 2256, 2257, 2258, 2320, 2326, 2332, 2338, 2358, 2360, 2365, 2372, 2373, 2438, 2442, 2481, 3099, 5298, 5301, 8151, 8237, 8874, 8879, 8907, 8916, 8926, 9170, 9176, 9178, 9180, 9182, 9184, 9186, 9188, 9190, 9192, 9194, 9196, 10505, 13899, 14196, 14243, 14309, 14335, 15427, 16063, 16079, 16154, 16972, 17162, 17194, 17614, 17640, 17653, 17686, 17730, 18235, 19934, 21029, 25806, 25956, 26709, 31361, 31363
 - `\cs_generate_from_arg_count:NNnn` 14, 1959, 2001
 - `\cs_generate_variant:Nn` .. 10, 25, 27, 28, 108, 266, 358, 2798, 3141, 3143, 3145, 3147, 3261, 3262, 3332, 3339, 3363, 3524, 3535, 3536, 3541, 3542, 3547, 3548, 3551, 3552, 3557, 3558, 3584, 3585, 3586, 3587, 3588, 3589, 3606, 3607, 3608, 3609, 3610, 3611, 3612, 3613, 3630, 3631, 3632, 3633, 3634, 3635, 3636, 3637, 3683, 3684, 3685, 3686, 3750, 3751, 3752, 3753, 3815, 3816, 3821, 3822, 3992, 4010, 4024, 4033, 4055, 4060, 4062, 4071, 4083, 4084, 4119, 4122, 4127, 4128, 4228, 4239, 4240, 4262, 4273, 4410, 4417, 4419, 4496, 4517, 4519, 4566, 4581, 4582, 4585, 4586, 4597, 4619, 4620, 4621, 4622, 4655, 4656, 4661, 4662, 4751, 4809, 4827, 4853, 4904, 4965, 5043, 5062, 5100, 5115, 5132, 5133, 5134, 5147, 5189, 5192, 7391, 7394, 7397, 7400, 7403, 7432, 7433, 7434, 7435, 7436, 7437, 7443, 7479, 7480, 7485, 7486, 7508, 7509, 7510, 7511, 7516, 7517, 7518, 7519, 7536, 7537, 7562, 7563, 7580, 7581, 7637, 7638, 7688, 7701, 7702, 7720, 7746, 7747, 7799, 7805, 7825, 7854, 7862, 7879, 7880, 7957, 7980, 7994, 8028, 8030, 8153, 8174, 8189, 8190, 8195, 8196, 8198, 8200, 8213, 8214, 8215, 8216, 8225, 8226, 8227, 8228, 8233, 8234, 8236, 8843, 8847, 8934, 8940, 8949, 8950, 8951, 8952, 8955, 8956, 8967, 8968, 8990, 8992, 9130, 9131, 9136, 9137, 9391, 9404, 9625, 9665, 9666, 9667, 9668, 9682, 9683, 9692, 9693, 9703, 9704, 9705, 9706, 9716, 9717, 9718, 9719, 9730, 9755, 9756, 9814, 9815, 9853, 9854, 9859, 9860, 9943, 9977, 10001, 10014, 10054, 10063, 10087, 10094, 10135, 10137, 10139, 10283, 10284, 10285, 10286, 10497, 10553, 11101, 11107, 11110, 11113, 11116, 11119, 11140, 11148, 11156, 11215, 11216, 11217, 11218, 11225, 11226, 11245, 11246, 11247, 11248, 11261, 11271, 11307, 11309, 11311, 11313, 11330, 11331, 11393, 11408, 11422, 11428, 11430, 11958, 12371, 12372, 12373, 12374, 12396, 12397, 12398, 12399, 12428, 12433, 12467, 12488, 12646, 12665, 12673, 12685, 12700, 12703, 12721, 12829, 13338, 13347, 13348, 13349, 13689, 13773, 13901, 13907, 13911, 13912, 13917, 13918, 13927, 13928, 13931, 13934, 13942, 13943, 13951, 13952, 14195, 14225, 14245, 14251, 14254, 14255, 14260, 14261, 14270,

- 14271, 14273, 14275, 14280, 14281,
 14286, 14287, 14308, 14316, 14317,
 14319, 14337, 14343, 14348, 14349,
 14354, 14355, 14364, 14365, 14367,
 14369, 14374, 14375, 14380, 14381,
 14385, 14387, 14635, 14731, 14747,
 14802, 14809, 14876, 14943, 14949,
 15126, 15140, 15146, 15156, 15182,
 15188, 15198, 15238, 15276, 15670,
 15672, 15702, 15744, 15753, 15762,
 15771, 15779, 15798, 15800, 15814,
 15835, 15878, 16415, 16418, 18118,
 18125, 18126, 18127, 18130, 18131,
 18134, 18135, 18140, 18141, 18148,
 18149, 18150, 18151, 18153, 18155,
 18379, 18441, 21524, 21578, 21656,
 21701, 21716, 21770, 22109, 22129,
 22158, 22212, 22220, 22228, 22314,
 22353, 22366, 22425, 22533, 22744,
 22746, 22768, 22771, 22777, 22783,
 23478, 24192, 26711, 26716, 26717,
 26722, 26723, 26728, 26729, 26734,
 26735, 26743, 26744, 26745, 26751,
 26754, 26760, 26763, 26769, 26772,
 26775, 26776, 26804, 26805, 26813,
 26816, 26819, 26828, 26846, 26859,
 26860, 26871, 26872, 26885, 26886,
 26905, 26906, 26923, 26924, 26949,
 26950, 26961, 26962, 26973, 26974,
 26987, 26988, 27008, 27009, 27012,
 27013, 27016, 27022, 27037, 27040,
 27158, 27164, 27204, 27207, 27224,
 27227, 27244, 27247, 27261, 27264,
 27282, 27285, 27311, 27314, 27320,
 27326, 27442, 27451, 27464, 27477,
 27490, 27496, 27502, 27547, 27560,
 27569, 27576, 27616, 27628, 27668,
 27671, 27686, 27689, 27707, 27908,
 27911, 28073, 28080, 28117, 28120,
 28178, 28184, 28229, 28235, 28366,
 28487, 28563, 28590, 28593, 29216,
 29507, 29548, 30816, 31251, 31254,
 31257, 31260, 31308, 31311, 31393,
 31396, 31445, 31475, 31636, 31639,
 31684, 31688, 32114, 32115, 32119,
 32123, 32144, 32153, 32164, 32174
 \cs_gset:Nn [14](#), [1974](#), [2038](#)
 .cs_gset:Np [188](#), [14986](#)
 \cs_gset:Npn [10](#), [12](#), [1435](#), [1854](#), [1868](#),
[1870](#), [3413](#), [3414](#), [3451](#), [3493](#), [7829](#),
[10350](#), [11483](#), [11485](#), [11523](#), [13241](#),
[13320](#), [13418](#), [13425](#), [13480](#), [13502](#),
[14995](#), [14997](#), [16849](#), [29849](#), [31907](#),
[32125](#), [32127](#), [32129](#), [32131](#), [32133](#),
[32135](#), [32137](#), [32139](#), [32176](#), [32179](#),
[32182](#), [32185](#), [32188](#), [32191](#), [32194](#),
[32196](#), [32198](#), [32200](#), [32202](#), [32204](#),
[32206](#), [32208](#), [32210](#), [32212](#), [32214](#)
 \cs_gset:Npx [12](#),
[1435](#), [1855](#), [1868](#), [1871](#), [7834](#), [25814](#)
 \cs_gset_eq:NN [15](#), [1886](#), [1903](#), [1911](#),
[3522](#), [3550](#), [5449](#), [5453](#), [7389](#), [7608](#),
[7839](#), [7844](#), [8946](#), [8948](#), [9498](#), [10281](#),
[11105](#), [11396](#), [11404](#), [12485](#), [12682](#)
 \cs_gset_nopar:Nn [14](#), [1974](#), [2038](#)
 \cs_gset_nopar:Npn
 [12](#), [1435](#), [1852](#), [1860](#), [1864](#), [3221](#)
 \cs_gset_nopar:Npx
 . [12](#), [1179](#), [1435](#), [1853](#), [1860](#), [1865](#),
[3507](#), [3528](#), [3533](#), [3579](#), [3581](#), [3583](#),
[3599](#), [3601](#), [3603](#), [3605](#), [3623](#), [3625](#),
[3627](#), [3629](#), [31580](#), [31606](#), [31611](#), [31638](#)
 \cs_gset_protected:Nn [14](#), [1974](#), [2038](#)
 .cs_gset_protected:Np [188](#), [14986](#)
 \cs_gset_protected:Npn [12](#), [1435](#),
[1858](#), [1880](#), [1882](#), [4014](#), [4813](#), [7889](#),
[8461](#), [9963](#), [11399](#), [12583](#), [14153](#),
[14999](#), [15001](#), [18446](#), [22514](#), [22695](#),
[22703](#), [22716](#), [23108](#), [23376](#), [23381](#),
[31823](#), [31845](#), [31854](#), [31864](#), [31902](#),
[32102](#), [32109](#), [32112](#), [32117](#), [32121](#),
[32142](#), [32146](#), [32155](#), [32166](#), [32216](#)
 \cs_gset_protected:Npx
 [12](#), [1435](#), [1859](#),
[1880](#), [1883](#), [8472](#), [14160](#), [18453](#), [31834](#)
 \cs_gset_protected_nopar:Nn
 [14](#), [1974](#), [2038](#)
 \cs_gset_protected_nopar:Npn
 [12](#), [1435](#), [1856](#), [1874](#), [1876](#)
 \cs_gset_protected_nopar:Npx
 [12](#), [1435](#), [1857](#), [1874](#), [1877](#)
 \cs_if_eq:NNTF [22](#),
[1187](#), [2070](#), [2077](#), [2078](#), [2081](#), [2082](#),
[2085](#), [2086](#), [9540](#), [11533](#), [14899](#),
[16614](#), [16624](#), [16650](#), [16652](#), [16654](#),
[16854](#), [24816](#), [29392](#), [29414](#), [29454](#),
[29457](#), [29709](#), [30763](#), [31854](#), [31864](#)
 \cs_if_eq_p:NN [22](#), [2070](#), [24833](#), [31823](#)
 \cs_if_exist [238](#)
 \cs_if_exist:N [22](#), [3559](#), [3560](#), [7487](#),
[7489](#), [8201](#), [8203](#), [9002](#), [9004](#), [9684](#),
[9686](#), [11332](#), [11334](#), [13919](#), [13921](#),
[14262](#), [14264](#), [14356](#), [14358](#), [18180](#),
[18181](#), [22483](#), [22485](#), [26736](#), [26738](#)
 \cs_if_exist:NTF [16](#), [22](#),
[303](#), [355](#), [434](#), [578](#), [609](#), [1738](#), [1795](#),
[1797](#), [1799](#), [1801](#), [1803](#), [1805](#), [1807](#),
[1809](#), [2090](#), [2195](#), [2265](#), [2301](#), [2391](#),

- 2415, 2483, 2514, 2765, 3031, 4672,
 5420, 5429, 5433, 5447, 7592, 8176,
 8177, 8178, 8179, 8855, 8856, 8902,
 9247, 9248, 9249, 9251, 9255, 9423,
 9538, 10347, 10501, 10801, 10820,
 11454, 12270, 12417, 12421, 12447,
 12504, 12631, 12635, 13077, 13260,
 13361, 13637, 13791, 13880, 14649,
 14754, 14845, 14850, 15305, 15347,
 15355, 15367, 15379, 15385, 15396,
 15412, 15499, 15560, 15568, 15712,
 16847, 17021, 18108, 22375, 22475,
 22512, 22547, 22693, 22711, 22714,
 24550, 25928, 27415, 27417, 27530,
 27532, 28413, 28419, 28420, 28423,
 28429, 28432, 28439, 29429, 29445,
 29731, 29739, 29872, 30598, 30608,
 30615, 30951, 31513, 31532, 32098
 \cs_if_exist_p:N ... [22](#), [303](#), [1738](#),
 9263, 10636, 10637, 13317, 13351,
 13397, 13533, 22451, 29469, 30775
 \cs_if_exist_use:N
 [16](#), [326](#), [1794](#), 11838,
 11856, 12798, 15381, 15400, 24998
 \cs_if_exist_use:NTF
 [16](#), [1794](#), 1796, 1798,
 1804, 1806, 3061, 3130, 3366, 9513,
 16296, 16979, 16981, 23784, 23791,
 24155, 24160, 24197, 24618, 24704,
 25863, 29751, 29763, 29826, 29828
 \cs_if_free:NTF [22](#),
[106](#), [609](#), [1766](#), 1835, 2973, 3000, 11828
 \cs_if_free_p:N ... [21](#), [22](#), [106](#), [1766](#)
 \cs_log:N [16](#), [335](#), [2115](#)
 \cs_meaning:N [15](#), [312](#),
[1383](#), [1399](#), 1407, 2127, 9420, 22524
 \cs_new:Nn [12](#), [107](#), [1974](#), [2038](#)
 \cs_new:Npn .. [10](#), [11](#), [14](#), [106](#), [106](#),
[362](#), [947](#), [1187](#), 1525, 1542, [1844](#),
[1868](#), 1872, 1945, 1947, 1949, 1957,
 2009, 2075, 2076, 2077, 2078, 2079,
 2080, 2081, 2082, 2083, 2084, 2085,
 2086, 2131, 2135, 2144, 2153, 2162,
 2165, 2174, 2175, 2185, 2186, 2187,
 2188, 2189, 2190, 2191, 2193, 2197,
 2201, 2204, 2217, 2223, 2229, 2240,
 2242, 2249, 2251, 2253, 2260, 2261,
 2263, 2267, 2271, 2277, 2279, 2284,
 2289, 2295, 2303, 2310, 2311, 2317,
 2323, 2329, 2335, 2341, 2348, 2355,
 2362, 2369, 2378, 2379, 2381, 2386,
 2393, 2397, 2400, 2410, 2411, 2413,
 2417, 2420, 2421, 2423, 2425, 2431,
 2437, 2439, 2445, 2447, 2454, 2461,
 2462, 2463, 2464, 2465, 2467, 2476,
 2478, 2481, 2482, 2485, 2489, 2492,
 2494, 2499, 2509, 2512, 2516, 2534,
 2535, 2537, 2543, 2548, 2550, 2556,
 2576, 2578, 2580, 2593, 2600, 2615,
 2621, 2627, 2632, 2633, 2656, 2686,
 2693, 2699, 2727, 2733, 2738, 2744,
 2793, 2794, 2795, 2796, 2865, 2886,
 2908, 2911, 2919, 2932, 2947, 2958,
 2990, 3095, 3097, 3231, 3237, 3245,
 3252, 3259, 3263, 3269, 3302, 3312,
 3315, 3419, 3428, 3461, 3466, 3468,
 3477, 3483, 3488, 3515, 3678, 3739,
 3854, 3932, 3935, 3936, 3937, 3938,
 3949, 3964, 3969, 3974, 3979, 3984,
 3986, 3995, 3997, 4003, 4005, 4025,
 4031, 4034, 4056, 4058, 4061, 4063,
 4072, 4077, 4082, 4085, 4097, 4098,
 4099, 4101, 4108, 4115, 4117, 4120,
 4131, 4143, 4151, 4157, 4163, 4169,
 4176, 4187, 4196, 4198, 4205, 4211,
 4213, 4215, 4229, 4231, 4233, 4241,
 4246, 4251, 4263, 4264, 4265, 4266,
 4274, 4320, 4329, 4360, 4381, 4388,
 4396, 4402, 4409, 4411, 4416, 4418,
 4420, 4421, 4429, 4441, 4450, 4459,
 4464, 4470, 4493, 4494, 4495, 4497,
 4539, 4554, 4555, 4671, 4687, 4729,
 4734, 4739, 4744, 4749, 4754, 4760,
 4765, 4770, 4775, 4780, 4782, 4788,
 4790, 4798, 4800, 4802, 4828, 4854,
 4856, 4858, 4869, 4878, 4881, 4892,
 4901, 4903, 4905, 4913, 4915, 4922,
 4943, 4953, 4958, 4963, 4964, 4966,
 4974, 4976, 4984, 4990, 4996, 5015,
 5017, 5026, 5032, 5039, 5041, 5044,
 5054, 5061, 5063, 5071, 5076, 5081,
 5092, 5099, 5101, 5107, 5109, 5114,
 5116, 5122, 5123, 5128, 5129, 5130,
 5131, 5135, 5140, 5145, 5148, 5150,
 5158, 5163, 5230, 5238, 5245, 5286,
 5288, 5294, 5300, 5302, 5307, 5312,
 5328, 5344, 5358, 5455, 5461, 5505,
 5511, 5543, 5553, 5564, 5590, 5597,
 5644, 5677, 5691, 5762, 5772, 5809,
 5871, 5912, 5914, 5931, 5937, 5960,
 5990, 6011, 6020, 6097, 6117, 6137,
 6160, 6167, 6194, 6297, 6311, 6338,
 6347, 6349, 6370, 6375, 6381, 6386,
 6454, 6477, 6489, 6498, 6504, 6509,
 7377, 7471, 7477, 7507, 7520, 7575,
 7656, 7686, 7714, 7719, 7741, 7776,
 7778, 7786, 7792, 7800, 7806, 7808,
 7810, 7819, 7855, 7863, 7881, 7896,

7906, 7934, 7952, 7956, 7958, 7981,
7982, 7983, 7990, 7992, 8055, 8060,
8062, 8063, 8069, 8077, 8083, 8101,
8109, 8117, 8130, 8132, 8139, 8141,
8237, 8244, 8258, 8263, 8269, 8280,
8285, 8292, 8294, 8296, 8298, 8300,
8302, 8304, 8314, 8319, 8324, 8329,
8334, 8336, 8342, 8360, 8368, 8376,
8382, 8388, 8396, 8404, 8410, 8416,
8423, 8439, 8449, 8451, 8487, 8501,
8507, 8539, 8571, 8573, 8575, 8581,
8587, 8599, 8607, 8619, 8627, 8660,
8693, 8695, 8697, 8699, 8701, 8706,
8711, 8716, 8721, 8722, 8723, 8724,
8725, 8726, 8727, 8728, 8729, 8730,
8731, 8732, 8733, 8734, 8735, 8736,
8737, 8746, 8747, 8756, 8762, 8764,
8773, 8780, 8786, 8788, 8790, 8806,
8817, 8840, 8873, 8913, 8914, 8923,
8924, 8971, 8987, 9014, 9015, 9024,
9025, 9035, 9040, 9045, 9047, 9055,
9056, 9057, 9058, 9059, 9060, 9061,
9062, 9063, 9064, 9074, 9090, 9100,
9116, 9126, 9128, 9132, 9134, 9138,
9146, 9151, 9159, 9165, 9172, 9174,
9176, 9177, 9179, 9181, 9183, 9185,
9187, 9189, 9191, 9193, 9195, 9197,
9202, 9203, 9204, 9205, 9206, 9207,
9208, 9209, 9210, 9211, 9221, 9223,
9450, 9452, 9569, 9570, 9576, 9582,
9588, 9618, 9619, 9658, 9754, 9850,
9852, 9861, 9868, 9871, 9884, 9890,
9928, 9937, 9944, 9950, 9957, 10002,
10004, 10006, 10024, 10031, 10055,
10056, 10059, 10061, 10064, 10072,
10088, 10095, 10103, 10105, 10119,
10121, 10122, 10123, 10125, 10130,
10169, 10239, 10245, 10251, 10257,
10288, 10294, 10333, 10341, 10406,
10421, 10426, 10470, 10472, 10474,
10477, 10480, 10486, 10492, 10498,
10499, 10511, 10521, 10523, 10525,
10534, 10536, 10545, 10551, 10554,
10564, 10570, 10577, 10579, 10611,
10613, 10615, 10621, 10623, 10629,
10750, 10780, 10855, 10867, 10868,
10876, 10885, 10894, 10907, 10908,
10909, 10910, 10913, 10969, 10977,
10979, 10981, 10991, 11001, 11092,
11202, 11249, 11255, 11262, 11270,
11350, 11356, 11378, 11387, 11409,
11416, 11423, 11425, 11449, 11522,
11557, 11619, 11624, 11629, 11631,
11633, 11635, 11641, 11650, 11656,
11662, 11807, 11809, 11826, 11959,
12324, 12333, 12339, 12351, 12356,
12361, 12366, 12375, 12388, 12390,
12392, 12394, 12400, 12567, 12569,
12643, 12726, 12734, 12772, 12789,
12844, 12895, 12904, 12923, 12924,
12932, 12938, 12946, 12956, 12961,
12967, 12973, 13046, 13048, 13050,
13098, 13109, 13120, 13155, 13160,
13166, 13175, 13177, 13186, 13188,
13189, 13191, 13247, 13252, 13270,
13272, 13283, 13284, 13285, 13287,
13305, 13400, 13402, 13404, 13406,
13411, 13427, 13430, 13437, 13451,
13461, 13471, 13493, 13495, 13568,
13582, 13596, 13630, 13717, 13722,
13727, 13732, 13738, 13752, 13790,
13895, 13953, 13958, 13960, 13968,
13976, 13984, 13986, 13998, 14004,
14017, 14019, 14021, 14023, 14025,
14033, 14038, 14043, 14048, 14053,
14055, 14061, 14063, 14071, 14079,
14085, 14091, 14099, 14107, 14113,
14119, 14126, 14140, 14174, 14176,
14182, 14196, 14197, 14204, 14212,
14214, 14216, 14302, 14305, 14309,
14311, 14314, 14382, 14410, 14412,
14419, 14421, 14423, 14434, 14441,
14445, 14453, 14462, 14466, 14474,
14476, 14484, 14488, 14495, 14502,
14509, 14518, 14528, 14534, 14540,
14541, 14542, 14543, 14544, 14556,
14568, 14576, 14581, 14587, 14714,
15421, 15423, 15488, 15497, 15503,
15505, 15510, 15514, 15518, 15522,
15528, 15540, 15544, 15548, 15671,
15673, 15675, 15686, 15745, 15747,
15754, 15760, 15778, 15780, 15788,
15885, 15886, 15887, 15888, 15889,
15890, 15891, 15892, 15893, 15894,
15904, 15928, 15930, 15932, 15941,
15943, 15950, 15962, 15963, 15965,
15975, 15985, 15995, 16005, 16013,
16015, 16022, 16024, 16025, 16030,
16037, 16051, 16053, 16069, 16070,
16078, 16080, 16089, 16091, 16103,
16108, 16112, 16117, 16119, 16121,
16123, 16125, 16132, 16134, 16142,
16144, 16156, 16158, 16160, 16162,
16186, 16188, 16190, 16191, 16192,
16194, 16196, 16198, 16200, 16218,
16233, 16234, 16240, 16257, 16270,
16276, 16402, 16403, 16404, 16405,
16406, 16407, 16408, 16413, 16416,

16462, 16464, 16466, 16468, 16474,
16478, 16480, 16489, 16490, 16499,
16512, 16525, 16532, 16546, 16562,
16574, 16585, 16595, 16601, 16612,
16622, 16648, 16659, 16676, 16687,
16692, 16712, 16714, 16725, 16730,
16743, 16766, 16767, 16771, 16788,
16789, 16813, 16821, 16839, 16868,
16894, 16898, 16901, 16903, 16909,
16921, 16933, 16940, 16946, 16954,
16977, 16992, 17011, 17019, 17034,
17049, 17060, 17070, 17080, 17085,
17094, 17111, 17124, 17129, 17135,
17137, 17144, 17174, 17202, 17218,
17229, 17234, 17252, 17270, 17281,
17296, 17301, 17312, 17322, 17332,
17348, 17396, 17401, 17408, 17416,
17422, 17427, 17431, 17448, 17456,
17488, 17505, 17519, 17538, 17546,
17555, 17564, 17575, 17577, 17591,
17601, 17602, 17619, 17626, 17631,
17644, 17657, 17662, 17692, 17706,
17736, 17737, 17741, 17758, 17780,
17782, 17793, 17825, 17829, 17844,
17861, 17885, 17887, 17889, 17891,
17901, 17906, 17917, 17929, 17940,
17953, 17973, 17991, 17993, 18005,
18011, 18019, 18033, 18040, 18051,
18058, 18072, 18176, 18178, 18195,
18217, 18222, 18239, 18266, 18267,
18268, 18269, 18285, 18296, 18304,
18316, 18322, 18328, 18336, 18344,
18350, 18356, 18364, 18372, 18390,
18403, 18425, 18473, 18479, 18490,
18514, 18516, 18518, 18520, 18528,
18532, 18539, 18546, 18547, 18548,
18549, 18550, 18551, 18554, 18556,
18585, 18593, 18604, 18606, 18608,
18610, 18617, 18641, 18643, 18653,
18668, 18677, 18691, 18699, 18707,
18714, 18721, 18729, 18739, 18753,
18764, 18765, 18771, 18788, 18795,
18797, 18804, 18809, 18826, 18827,
18828, 18847, 18853, 18863, 18875,
18882, 18896, 18904, 18942, 18951,
18972, 18974, 18976, 18985, 18996,
19008, 19023, 19036, 19049, 19057,
19075, 19093, 19100, 19108, 19118,
19119, 19128, 19129, 19138, 19148,
19162, 19172, 19183, 19191, 19193,
19204, 19210, 19245, 19266, 19268,
19270, 19272, 19279, 19288, 19293,
19300, 19307, 19327, 19332, 19349,
19360, 19365, 19375, 19377, 19387,
19394, 19396, 19402, 19404, 19406,
19410, 19429, 19430, 19435, 19443,
19444, 19467, 19480, 19487, 19495,
19496, 19497, 19498, 19499, 19500,
19508, 19514, 19516, 19518, 19540,
19545, 19555, 19565, 19576, 19589,
19600, 19605, 19612, 19621, 19623,
19632, 19641, 19655, 19657, 19659,
19672, 19682, 19687, 19696, 19704,
19711, 19717, 19726, 19728, 19740,
19745, 19753, 19758, 19768, 19774,
19780, 19787, 19794, 19796, 19801,
19803, 19808, 19810, 19824, 19834,
19846, 19851, 19858, 19868, 19870,
19872, 19883, 19897, 19911, 19931,
19944, 19946, 19951, 19964, 19969,
19977, 19982, 19992, 20004, 20034,
20035, 20036, 20038, 20040, 20042,
20056, 20062, 20071, 20090, 20096,
20106, 20125, 20133, 20166, 20172,
20181, 20183, 20197, 20256, 20264,
20282, 20299, 20300, 20305, 20330,
20353, 20382, 20398, 20408, 20419,
20440, 20455, 20460, 20465, 20467,
20481, 20487, 20502, 20510, 20520,
20530, 20543, 20561, 20567, 20581,
20596, 20634, 20636, 20638, 20640,
20642, 20657, 20672, 20687, 20702,
20717, 20732, 20740, 20754, 20756,
20762, 20774, 20782, 20789, 21015,
21022, 21059, 21067, 21068, 21079,
21086, 21088, 21094, 21105, 21115,
21122, 21129, 21144, 21183, 21196,
21227, 21233, 21240, 21260, 21262,
21279, 21294, 21307, 21314, 21319,
21321, 21330, 21343, 21346, 21367,
21380, 21395, 21413, 21428, 21438,
21447, 21460, 21476, 21493, 21506,
21512, 21514, 21519, 21520, 21521,
21522, 21525, 21530, 21536, 21541,
21543, 21566, 21574, 21576, 21579,
21584, 21590, 21595, 21597, 21620,
21645, 21654, 21655, 21657, 21662,
21664, 21669, 21671, 21681, 21689,
21697, 21699, 21702, 21707, 21712,
21714, 21715, 21717, 21722, 21727,
21729, 21734, 21741, 21755, 21760,
21762, 21772, 21774, 21776, 21778,
21780, 21791, 21801, 21803, 21809,
21816, 21822, 21832, 21837, 21839,
21847, 21848, 21862, 21869, 21875,
21876, 21889, 21904, 21910, 21932,
21947, 21957, 21978, 21987, 22010,
22028, 22039, 22044, 22055, 22072,

22077, 22124, 22130, 22144, 22213,
 22221, 22229, 22235, 22242, 22247,
 22253, 22265, 22357, 22462, 22464,
 22471, 22758, 22913, 22926, 22931,
 22937, 22938, 22945, 22952, 22959,
 22966, 22973, 22974, 22976, 22983,
 22989, 23066, 23071, 23072, 23080,
 23086, 23263, 23268, 23275, 23312,
 23317, 23332, 23355, 23360, 23408,
 23414, 23432, 23437, 23452, 23454,
 23456, 23463, 23494, 23523, 23528,
 23553, 23555, 23782, 23788, 23799,
 23804, 23817, 23822, 23834, 23846,
 23856, 23865, 23885, 23996, 24014,
 24022, 24034, 24046, 24538, 24814,
 24829, 24869, 24910, 25070, 25198,
 25590, 25716, 25718, 25724, 25725,
 25732, 25742, 25894, 26259, 26264,
 26663, 26704, 28660, 28661, 28665,
 28666, 29030, 29035, 29050, 29060,
 29071, 29085, 29102, 29115, 29117,
 29118, 29199, 29207, 29214, 29217,
 29219, 29225, 29236, 29248, 29272,
 29292, 29308, 29315, 29329, 29340,
 29349, 29354, 29360, 29373, 29383,
 29388, 29399, 29404, 29410, 29434,
 29439, 29441, 29452, 29465, 29466,
 29481, 29482, 29495, 29497, 29513,
 29515, 29517, 29519, 29521, 29523,
 29525, 29527, 29529, 29539, 29546,
 29549, 29551, 29557, 29568, 29573,
 29587, 29599, 29613, 29619, 29625,
 29630, 29636, 29650, 29661, 29670,
 29677, 29684, 29691, 29700, 29705,
 29716, 29721, 29725, 29727, 29729,
 29761, 29771, 29777, 29788, 29820,
 29822, 29824, 29833, 29864, 29866,
 29868, 29870, 29881, 29889, 29895,
 29917, 29931, 29940, 29942, 29947,
 29955, 30025, 30032, 30045, 30070,
 30080, 30087, 30110, 30123, 30133,
 30140, 30149, 30161, 30168, 30188,
 30202, 30213, 30231, 30244, 30635,
 30644, 30651, 30653, 30655, 30661,
 30672, 30673, 30678, 30683, 30690,
 30701, 30706, 30708, 30710, 30715,
 30720, 30731, 30742, 30747, 30754,
 30759, 30770, 30772, 30795, 30796,
 30806, 30949, 31359, 31399, 31401,
 31403, 31405, 31407, 31409, 31411,
 31413, 31415, 31420, 31426, 31432,
 31435, 31436, 31446, 31454, 31456,
 31462, 31468, 31655, 31663, 31683,
 31685, 31686, 31689, 31691, 31693,
 31695, 31703, 31771, 31773, 31803
 \cs_new:Npx . . . 11, 34, 34, 356, 357,
 1844, 1868, 1873, 2824, 6511, 10015,
 10025, 12771, 12783, 13142, 13150,
 13296, 16044, 16880, 17468, 18612,
 20621, 20627, 21239, 23288, 23805,
 23807, 23809, 23811, 23813, 23815,
 29253, 29424, 29749, 29799, 29908,
 30258, 31372, 31374, 31376, 31383
 \cs_new_eq:NN
 15, 107, 327, 329, 573, 1658,
 1886, 2164, 2173, 2210, 2480, 2774,
 2775, 2776, 2777, 2778, 2779, 2780,
 2781, 2782, 2783, 2784, 2785, 2786,
 2787, 2788, 2791, 2792, 3033, 3229,
 3503, 3506, 3809, 3810, 4565, 4579,
 4580, 4583, 4584, 4650, 5187, 5188,
 5190, 5191, 5522, 5538, 5540, 6525,
 7384, 7404, 7405, 7406, 7407, 7408,
 7409, 7410, 7411, 7635, 7995, 7996,
 7997, 7998, 7999, 8000, 8001, 8002,
 8003, 8004, 8005, 8006, 8007, 8008,
 8009, 8010, 8011, 8012, 8013, 8014,
 8015, 8016, 8017, 8018, 8019, 8020,
 8048, 8049, 8050, 8051, 8052, 8180,
 8181, 8184, 8235, 8486, 8842, 8846,
 8931, 8933, 8953, 8954, 9233, 9234,
 9235, 9236, 9239, 9240, 9241, 9242,
 9414, 9565, 9621, 9622, 9626, 9627,
 9628, 9629, 9630, 9631, 9632, 9633,
 9634, 9635, 9636, 9637, 9638, 9639,
 9640, 9641, 9782, 9783, 9784, 9785,
 9786, 9787, 9788, 9789, 9790, 9791,
 9792, 9793, 9794, 9795, 9796, 9797,
 10287, 10349, 10651, 10653, 10654,
 10655, 10657, 10660, 10661, 10903,
 10904, 10905, 11120, 11121, 11122,
 11123, 11124, 11125, 11126, 11127,
 12427, 12449, 12501, 12645, 12727,
 13238, 13636, 13704, 13710, 13890,
 13891, 13892, 14194, 14224, 14228,
 14229, 14307, 14310, 14313, 14318,
 14322, 14323, 14384, 14386, 14390,
 14391, 15642, 15643, 15882, 15883,
 15884, 16088, 16269, 16545, 16573,
 16581, 16582, 16583, 16592, 16594,
 17395, 17514, 17515, 17516, 18117,
 18128, 18129, 21769, 21771, 21815,
 23059, 23060, 23491, 23497, 23626,
 23627, 23727, 23735, 23756, 23795,
 23796, 23797, 23798, 23975, 25469,
 26067, 26703, 26740, 26741, 26742,
 26773, 26774, 26785, 26786, 26787,
 26892, 26921, 26922, 26995, 27010,

- 27011, 27412, 27636, 27637, 27638,
 27639, 27640, 27641, 28637, 28638,
 28655, 28656, 28657, 29585, 29723,
 29818, 29847, 29883, 29885, 29887,
 30280, 30282, 31592, 31593, 32080
 \cs_new_nopar:Nn [13](#), [1974](#), [2038](#)
 \cs_new_nopar:Npn
 [11](#), [327](#), [328](#), [1844](#), [1860](#), [1866](#)
 \cs_new_nopar:Npx [11](#), [1844](#), [1860](#), [1867](#)
 \cs_new_protected:Nn . [13](#), [1974](#), [2038](#)
 \cs_new_protected:Npn
 [11](#), [362](#), [1187](#), 1529, 1546,
[1844](#), [1880](#), 1884, 1886, 1887, 1888,
 1889, 1890, 1891, 1892, 1893, 1894,
 1899, 1900, 1901, 1902, 1904, 1959,
 1969, 1971, 1982, 1991, 2088, 2097,
 2099, 2101, 2103, 2105, 2113, 2115,
 2116, 2118, 2119, 2121, 2176, 2211,
 2376, 2405, 2475, 2506, 2798, 2811,
 2829, 2833, 2836, 2845, 2969, 2986,
 2996, 3005, 3029, 3037, 3049, 3057,
 3068, 3070, 3072, 3074, 3076, 3087,
 3089, 3102, 3110, 3121, 3140, 3142,
 3144, 3146, 3148, 3218, 3320, 3322,
 3333, 3340, 3346, 3364, 3373, 3378,
 3383, 3388, 3393, 3399, 3404, 3411,
 3417, 3426, 3436, 3438, 3440, 3442,
 3444, 3449, 3494, 3519, 3525, 3530,
 3537, 3539, 3543, 3545, 3549, 3550,
 3553, 3555, 3572, 3574, 3576, 3578,
 3580, 3582, 3590, 3592, 3594, 3596,
 3598, 3600, 3602, 3604, 3614, 3616,
 3618, 3620, 3622, 3624, 3626, 3628,
 3646, 3652, 3654, 3656, 3668, 3687,
 3702, 3720, 3742, 3744, 3746, 3748,
 3754, 3776, 3782, 3811, 3813, 3817,
 3819, 3905, 3906, 3907, 4011, 4022,
 4040, 4046, 4048, 4123, 4125, 4235,
 4237, 4516, 4518, 4520, 4525, 4527,
 4540, 4611, 4613, 4615, 4617, 4623,
 4638, 4651, 4653, 4657, 4659, 4810,
 4825, 4834, 4844, 4846, 5196, 5318,
 5334, 5350, 5356, 5360, 5362, 5382,
 5400, 5408, 5418, 5427, 5481, 5529,
 5537, 5539, 5541, 5551, 5557, 5581,
 5592, 5598, 5601, 5603, 5608, 5625,
 5634, 5654, 5667, 5745, 5793, 5846,
 5929, 5935, 5958, 5988, 6009, 6082,
 6178, 6183, 6185, 6187, 6263, 6265,
 6267, 6272, 6282, 6361, 6366, 6368,
 6421, 6423, 6425, 6430, 6440, 7386,
 7392, 7395, 7398, 7401, 7412, 7417,
 7422, 7427, 7438, 7444, 7446, 7448,
 7481, 7483, 7491, 7499, 7512, 7514,
 7522, 7524, 7526, 7538, 7540, 7542,
 7564, 7566, 7568, 7595, 7596, 7597,
 7619, 7630, 7660, 7668, 7678, 7689,
 7691, 7693, 7695, 7703, 7721, 7723,
 7725, 7826, 7831, 7836, 7842, 7848,
 7869, 7886, 7914, 7916, 7918, 7924,
 7926, 7928, 8027, 8029, 8031, 8148,
 8154, 8187, 8188, 8191, 8193, 8197,
 8199, 8205, 8207, 8209, 8211, 8217,
 8219, 8221, 8223, 8229, 8231, 8238,
 8453, 8455, 8457, 8464, 8466, 8468,
 8480, 8844, 8848, 8871, 8876, 8877,
 8888, 8890, 8891, 8892, 8933, 8935,
 8941, 8943, 8945, 8947, 8957, 8962,
 8983, 8985, 8989, 8991, 8993, 9229,
 9264, 9281, 9323, 9329, 9337, 9348,
 9371, 9381, 9388, 9394, 9401, 9405,
 9410, 9465, 9469, 9499, 9505, 9575,
 9623, 9642, 9644, 9646, 9669, 9671,
 9673, 9688, 9690, 9694, 9696, 9698,
 9707, 9709, 9711, 9720, 9728, 9731,
 9733, 9735, 9743, 9771, 9800, 9802,
 9804, 9816, 9818, 9820, 9855, 9857,
 9901, 9958, 9972, 9978, 9989, 9994,
 10136, 10138, 10140, 10150, 10151,
 10152, 10167, 10171, 10173, 10175,
 10177, 10179, 10181, 10183, 10185,
 10187, 10189, 10191, 10193, 10195,
 10197, 10199, 10201, 10203, 10205,
 10207, 10209, 10211, 10213, 10215,
 10217, 10219, 10221, 10223, 10225,
 10227, 10229, 10231, 10233, 10235,
 10237, 10241, 10243, 10247, 10249,
 10253, 10255, 10259, 10271, 10914,
 10916, 10918, 10923, 10930, 10939,
 10957, 10959, 10961, 10963, 10965,
 10967, 11050, 11067, 11072, 11079,
 11084, 11086, 11102, 11108, 11111,
 11114, 11117, 11133, 11141, 11149,
 11157, 11162, 11169, 11171, 11173,
 11178, 11180, 11192, 11194, 11203,
 11209, 11219, 11227, 11236, 11294,
 11295, 11296, 11315, 11317, 11319,
 11394, 11427, 11429, 11431, 11457,
 11465, 11470, 11472, 11479, 11481,
 11488, 11529, 11558, 11578, 11583,
 11594, 11671, 11673, 11714, 11797,
 11811, 11836, 11858, 11859, 11872,
 11877, 11903, 11912, 11914, 11916,
 11933, 11960, 11962, 11964, 11966,
 11973, 12427, 12431, 12445, 12456,
 12477, 12489, 12490, 12491, 12518,
 12520, 12531, 12533, 12552, 12554,
 12556, 12571, 12573, 12575, 12581,

12588, 12597, 12599, 12601, 12606,
12645, 12649, 12652, 12666, 12674,
12686, 12687, 12688, 12698, 12701,
12704, 12710, 12716, 12723, 12725,
12735, 12766, 12777, 12795, 12830,
12853, 12865, 12884, 12888, 12977,
12993, 13002, 13010, 13019, 13026,
13032, 13198, 13232, 13333, 13389,
13508, 13510, 13512, 13514, 13524,
13554, 13654, 13659, 13665, 13666,
13671, 13690, 13705, 13711, 13762,
13767, 13774, 13775, 13776, 13803,
13816, 13828, 13838, 13896, 13902,
13908, 13909, 13913, 13915, 13923,
13925, 13929, 13932, 13935, 13937,
13944, 13946, 14027, 14149, 14156,
14168, 14226, 14230, 14240, 14246,
14252, 14253, 14256, 14258, 14266,
14268, 14272, 14274, 14276, 14278,
14282, 14284, 14320, 14324, 14332,
14338, 14344, 14346, 14350, 14352,
14360, 14362, 14366, 14368, 14370,
14372, 14376, 14378, 14388, 14392,
14627, 14629, 14636, 14641, 14646,
14659, 14666, 14670, 14679, 14684,
14693, 14699, 14716, 14732, 14748,
14750, 14752, 14768, 14779, 14781,
14783, 14800, 14803, 14810, 14825,
14838, 14843, 14854, 14861, 14863,
14877, 14887, 14915, 14924, 14933,
14944, 14950, 14952, 14954, 14956,
14958, 14960, 14962, 14964, 14966,
14968, 14970, 14972, 14974, 14976,
14978, 14980, 14982, 14984, 14986,
14988, 14990, 14992, 14994, 14996,
14998, 15000, 15002, 15004, 15006,
15008, 15010, 15012, 15014, 15016,
15018, 15020, 15022, 15024, 15026,
15028, 15030, 15032, 15034, 15036,
15038, 15040, 15042, 15044, 15046,
15048, 15050, 15052, 15054, 15056,
15058, 15060, 15062, 15064, 15066,
15068, 15070, 15072, 15074, 15076,
15078, 15080, 15082, 15084, 15086,
15088, 15090, 15092, 15094, 15096,
15098, 15100, 15102, 15104, 15106,
15127, 15129, 15135, 15141, 15147,
15154, 15157, 15176, 15183, 15189,
15196, 15199, 15218, 15239, 15241,
15247, 15252, 15257, 15277, 15282,
15286, 15292, 15297, 15303, 15317,
15343, 15362, 15377, 15391, 15407,
15431, 15455, 15487, 15573, 15575,
15577, 15649, 15658, 15693, 15695,
15703, 15714, 15722, 15729, 15735,
15763, 15772, 15797, 15799, 15801,
15812, 15817, 15822, 15836, 15841,
15846, 15861, 15872, 15895, 15898,
16009, 16294, 16311, 16313, 16315,
16317, 16345, 16347, 16349, 16351,
16371, 16373, 16375, 16377, 16379,
16381, 16383, 16385, 16387, 18116,
18119, 18121, 18123, 18132, 18133,
18136, 18138, 18142, 18143, 18144,
18145, 18146, 18152, 18154, 18156,
18161, 18163, 18442, 18449, 18461,
21272, 22097, 22110, 22149, 22159,
22171, 22176, 22186, 22194, 22200,
22279, 22284, 22291, 22293, 22295,
22317, 22319, 22331, 22342, 22364,
22369, 22382, 22395, 22403, 22412,
22426, 22437, 22443, 22508, 22521,
22528, 22653, 22672, 22679, 22691,
22724, 22743, 22745, 22747, 22766,
22769, 22772, 22778, 22784, 22799,
22808, 22828, 22838, 22849, 22859,
22869, 22870, 22877, 22883, 22893,
22903, 22999, 23005, 23007, 23022,
23088, 23099, 23117, 23127, 23129,
23153, 23160, 23172, 23175, 23178,
23188, 23196, 23203, 23212, 23227,
23244, 23255, 23365, 23372, 23374,
23392, 23402, 23492, 23495, 23498,
23500, 23509, 23515, 23521, 23559,
23561, 23562, 23567, 23573, 23581,
23593, 23609, 23628, 23638, 23646,
23648, 23656, 23668, 23688, 23690,
23695, 23703, 23705, 23712, 23717,
23719, 23721, 23728, 23736, 23738,
23740, 23742, 23749, 23754, 23757,
23763, 23990, 24054, 24067, 24078,
24091, 24124, 24153, 24158, 24163,
24184, 24193, 24202, 24207, 24214,
24227, 24229, 24231, 24233, 24239,
24261, 24274, 24301, 24306, 24340,
24375, 24396, 24398, 24408, 24417,
24422, 24431, 24440, 24456, 24469,
24475, 24486, 24499, 24505, 24524,
24544, 24575, 24586, 24601, 24614,
24632, 24640, 24645, 24647, 24649,
24666, 24685, 24687, 24710, 24722,
24742, 24763, 24770, 24777, 24789,
24795, 24853, 24888, 24897, 24916,
24935, 24941, 25004, 25014, 25016,
25018, 25025, 25081, 25094, 25110,
25115, 25128, 25144, 25151, 25158,
25160, 25162, 25169, 25183, 25199,
25208, 25222, 25234, 25252, 25261,

25263, 25275, 25284, 25296, 25309,
 25316, 25336, 25367, 25401, 25419,
 25425, 25434, 25473, 25486, 25508,
 25525, 25547, 25556, 25567, 25596,
 25611, 25620, 25629, 25641, 25648,
 25650, 25652, 25672, 25677, 25684,
 25689, 25694, 25699, 25748, 25783,
 25825, 25841, 25861, 25873, 25887,
 25921, 25935, 25946, 25953, 25962,
 25997, 26003, 26006, 26014, 26020,
 26023, 26032, 26035, 26038, 26041,
 26046, 26055, 26058, 26061, 26066,
 26072, 26077, 26082, 26087, 26095,
 26115, 26117, 26121, 26122, 26146,
 26154, 26163, 26175, 26184, 26192,
 26232, 26276, 26303, 26308, 26310,
 26340, 26368, 26679, 26681, 26683,
 26690, 26706, 26712, 26714, 26718,
 26720, 26724, 26726, 26730, 26732,
 26746, 26752, 26755, 26761, 26764,
 26770, 26777, 26779, 26781, 26783,
 26800, 26802, 26811, 26814, 26817,
 26820, 26822, 26829, 26847, 26849,
 26854, 26861, 26866, 26873, 26879,
 26887, 26893, 26899, 26907, 26912,
 26917, 26919, 26925, 26927, 26929,
 26934, 26939, 26944, 26951, 26956,
 26963, 26968, 26975, 26981, 26989,
 26996, 27002, 27014, 27017, 27035,
 27038, 27041, 27051, 27085, 27096,
 27107, 27118, 27129, 27140, 27153,
 27159, 27165, 27180, 27187, 27197,
 27202, 27205, 27208, 27222, 27225,
 27228, 27242, 27245, 27248, 27259,
 27262, 27265, 27280, 27283, 27286,
 27295, 27309, 27312, 27315, 27321,
 27327, 27339, 27425, 27434, 27443,
 27452, 27465, 27478, 27491, 27497,
 27503, 27535, 27548, 27561, 27562,
 27563, 27570, 27577, 27603, 27604,
 27605, 27617, 27642, 27652, 27659,
 27666, 27669, 27672, 27684, 27687,
 27690, 27702, 27708, 27714, 27720,
 27722, 27724, 27730, 27751, 27753,
 27755, 27761, 27795, 27810, 27906,
 27909, 27912, 27954, 27972, 27978,
 27984, 27996, 28015, 28024, 28035,
 28041, 28046, 28054, 28067, 28074,
 28081, 28093, 28115, 28118, 28121,
 28136, 28143, 28149, 28158, 28165,
 28173, 28179, 28185, 28224, 28230,
 28236, 28257, 28266, 28285, 28290,
 28304, 28309, 28322, 28333, 28345,
 28359, 28443, 28480, 28488, 28512,
 28556, 28564, 28588, 28591, 28594,
 28639, 28645, 28647, 28662, 28663,
 29502, 30811, 31249, 31252, 31255,
 31258, 31261, 31306, 31309, 31312,
 31367, 31369, 31371, 31391, 31394,
 31476, 31478, 31480, 31486, 31488,
 31490, 31496, 31498, 31555, 31561,
 31577, 31579, 31581, 31583, 31594,
 31599, 31604, 31609, 31614, 31631,
 31632, 31634, 31637, 31640, 31651,
 31653, 31665, 31670, 31675, 31718,
 31720, 31722, 31724, 31741, 31754,
 31759, 31785, 31809, 31811, 31832,
 31851, 31858, 31875, 31899, 31904
 \cs_new_protected:Npx [11](#), [356](#), [357](#),
 [361](#), [1844](#), [1880](#), [1885](#), [1976](#), [2040](#),
 [2813](#), [2817](#), [2822](#), [2981](#), [2985](#), [4592](#),
 [10277](#), [11016](#), [11031](#), [11036](#), [11041](#),
 [11682](#), [11684](#), [11686](#), [11688](#), [11690](#),
 [11699](#), [11701](#), [11703](#), [11982](#), [11984](#),
 [11986](#), [11988](#), [11990](#), [11999](#), [12001](#),
 [12003](#), [12468](#), [13214](#), [13677](#), [24369](#),
 [24383](#), [24385](#), [27528](#), [28427](#), [28437](#)
 \cs_new_protected_nopar:Nn [13](#), [1974](#), [2038](#)
 \cs_new_protected_nopar:Npn [11](#), [1844](#), [1861](#), [1874](#), [1878](#)
 \cs_new_protected_nopar:Npx [11](#), [1844](#), [1874](#), [1879](#)
 \cs_prefix_spec:N [18](#), [2129](#), [32193](#), [32194](#)
 \cs_replacement_spec:N [18](#), [2129](#), [15586](#), [32197](#), [32198](#)
 \cs_set:Nn [13](#), [332](#), [1974](#), [2038](#)
 .cs_set:Np [188](#), [14986](#)
 \cs_set:Npn [10](#), [11](#), [106](#), [106](#), [319](#), [328](#),
 [332](#), [591](#), [1421](#), [1451](#), [1458](#), [1460](#),
 [1463](#), [1464](#), [1465](#), [1466](#), [1467](#), [1468](#),
 [1469](#), [1470](#), [1471](#), [1472](#), [1473](#), [1474](#),
 [1475](#), [1476](#), [1477](#), [1478](#), [1479](#), [1480](#),
 [1481](#), [1482](#), [1483](#), [1485](#), [1486](#), [1487](#),
 [1488](#), [1489](#), [1490](#), [1491](#), [1492](#), [1493](#),
 [1516](#), [1518](#), [1520](#), [1523](#), [1540](#), [1589](#),
 [1592](#), [1654](#), [1702](#), [1704](#), [1706](#), [1708](#),
 [1713](#), [1719](#), [1720](#), [1724](#), [1731](#), [1734](#),
 [1794](#), [1796](#), [1798](#), [1800](#), [1802](#), [1804](#),
 [1806](#), [1808](#), [1827](#), [1844](#), [1860](#), [1868](#),
 [1868](#), [1974](#), [2038](#), [3459](#), [3481](#), [3728](#),
 [3785](#), [3914](#), [4129](#), [4640](#), [4676](#), [6288](#),
 [6445](#), [8085](#), [8093](#), [9418](#), [9829](#), [9918](#),
 [10326](#), [10639](#), [10927](#), [11196](#), [11474](#),
 [11476](#), [12742](#), [13064](#), [13571](#), [14987](#),
 [14989](#), [15526](#), [16320](#), [16328](#), [16337](#),

- 16354, 16362, 16390, 22739, 23767,
23768, 23769, 24086, 24087, 24653,
24655, 24672, 24674, 24968, 24995,
25034, 25528, 25827, 28675, 28864,
30913, 30914, 30976, 30983, 30988,
30989, 31730, 31732, 31738, 31856
\cs_set:Npx [11](#), [338](#), [693](#), [1421](#), [1868](#),
1869, 3787, 9905, 10925, 10944,
10950, 12800, 12801, 12802, 12803,
12804, 23511, 23517, 25185, 28866
\cs_set_eq:NN [15](#), [107](#), [329](#),
[524](#), 1656, [1886](#), 2817, 2835, 2985,
3549, 4674, 4675, 5369, 5378, 6190,
7570, 7571, 7573, 7727, 7728, 7739,
8880, 8942, 8944, 10280, 10510,
10688, 10921, 10942, 10949, 12806,
12807, 12808, 12809, 12811, 12813,
12814, 14814, 14830, 14834, 14882,
14893, 14903, 22441, 23001, 23002,
23006, 23156, 23199, 23224, 24960,
24969, 24992, 25534, 31727, 31737
\cs_set_nopar:Nn [13](#), [1974](#), [2038](#)
\cs_set_nopar:Npn [10](#), [11](#), [135](#), [328](#),
[1421](#), 1450, 1508, 1509, [1860](#), 1862,
14770, 14841, 15301, 28912, 28914
\cs_set_nopar:Npx [11](#), [690](#), [705](#), [1179](#), [1421](#), 1454,
[1860](#), 1863, 2213, 2407, 3573, 3575,
3577, 3591, 3593, 3595, 3597, 3615,
3617, 3619, 3621, 14661, 14674,
14681, 14688, 14689, 14819, 15289,
15294, 31578, 31596, 31601, 31635
\cs_set_protected:Nn [13](#), [1974](#), [2038](#)
.cs_set_protected:Np [188](#), [14986](#)
\cs_set_protected:Npn [10](#), [11](#),
[161](#), [329](#), [1421](#), 1437, 1439, 1441,
1443, 1445, 1447, 1452, 1495, 1496,
1501, 1506, 1507, 1510, 1522, 1524,
1526, 1527, 1528, 1530, 1539, 1541,
1543, 1544, 1545, 1547, 1556, 1568,
1594, 1611, 1630, 1638, 1646, 1655,
1657, 1659, 1671, 1685, 1722, 1810,
1823, 1825, 1829, 1831, 1833, 1841,
1846, [1880](#), 1880, 1914, 1935, 3003,
3164, 3921, 4561, 4588, 5791, 5844,
7644, 10269, 10389, 10776, 10794,
11048, 11602, 11668, 11969, 11971,
12322, 12450, 12722, 12724, 12863,
12944, 13044, 13060, 13536, 14294,
14408, 14554, 14801, 14991, 14993,
15461, 16275, 16769, 16845, 17429,
17503, 17517, 17739, 17756, 17791,
17827, 17842, 17859, 19408, 23003,
24381, 24406, 24415, 24945, 24954,
24956, 24958, 24961, 24963, 24970,
24972, 24977, 24979, 24984, 24986,
24988, 24990, 24993, 25691, 25692,
26119, 27540, 27553, 27584, 27867,
28684, 28889, 28899, 28909, 28934,
28949, 28967, 28975, 29007, 30289,
30292, 30323, 30334, 30350, 30548,
30551, 30572, 30603, 30891, 30903,
30965, 31016, 31849, 31855, 32090
\cs_set_protected:Npx [11](#), 147,
[1421](#), [1880](#), 1881, 11844, 14805, 32095
\cs_set_protected_nopar:Nn [14](#), [1974](#), [2038](#)
\cs_set_protected_nopar:Npn [12](#), [328](#), [1421](#), [1874](#), 1874
\cs_set_protected_nopar:Npx [12](#), [1421](#), [1874](#), 1875
\cs_show:N [16](#), [16](#), [22](#), [335](#), [2115](#)
\cs_split_function:N [17](#), 1535, 1552, 1664, 1665, [1722](#),
1946, 1987, 2804, 3107, 3336, 3357
\cs_to_sr:N [1181](#)
\cs_to_str:N [4](#),
[17](#), [51](#), [60](#), [323](#), [323](#), [324](#), [355](#), [425](#),
[1713](#), 1728, 2602, 2768, 3464, 3486,
5171, 5172, 5173, 5174, 5175, 5176,
5177, 5178, 5179, 5180, 5181, 5182,
12727, 16273, 23525, 24926, 29437,
30875, 31582, 31668, 31673, 31681
\cs_undefine:N [15](#), [521](#), [695](#),
[1902](#), 12008, 12009, 12010, 12452,
22115, 22399, 22459, 28658, 28659
cs internal commands:
__cs_count_signature:N [322](#), [1945](#)
__cs_count_signature:n [1945](#)
__cs_count_signature:nnN [1945](#)
__cs_generate_from_signature:n [1996](#), 2009
__cs_generate_from_signature:NNn [1978](#), 1982
__cs_generate_from_signature:nnNNNn [1986](#), 1991
__cs_generate_internal_c:NN [3070](#)
__cs_generate_internal_end:w [3053](#), 3087
__cs_generate_internal_long:nnnNNn [3091](#), 3095
__cs_generate_internal_long:w [3054](#), 3089
__cs_generate_internal_loop:nwnnw [3051](#),
[3057](#), [3069](#), [3071](#), [3073](#), [3075](#), 3078
__cs_generate_internal_N:NN [3068](#)
__cs_generate_internal_n:NN [3072](#)

__cs_generate_internal_one_-
 go:NNn [362](#), [3026](#), [3049](#)
 __cs_generate_internal_other:NN
 [3062](#), [3076](#)
 __cs_generate_internal_test:Nw .
 [3011](#), [3033](#), [3037](#)
 __cs_generate_internal_test_-
 aux:w .. [3013](#), [3029](#), [3034](#), [3040](#), [3043](#)
 __cs_generate_internal_variant:n
 [366](#), [2976](#), [2981](#), [3161](#), [3167](#)
 __cs_generate_internal_variant:NNn
 [362](#), [3001](#), [3005](#)
 __cs_generate_internal_variant:wnNwn
 [2983](#), [2996](#)
 __cs_generate_internal_variant_-
 loop:n [2981](#)
 __cs_generate_internal_x:NN . [3074](#)
 __cs_generate_variant:N . [2800](#), [2813](#)
 __cs_generate_variant:n [3102](#)
 __cs_generate_variant:nnNN
 [2803](#), [2836](#)
 __cs_generate_variant:nnNnn . [3102](#)
 __cs_generate_variant:Nnnw
 [2843](#), [2845](#)
 __cs_generate_variant:w [3102](#)
 __cs_generate_variant:ww [2813](#)
 __cs_generate_variant:wwNN
 [358](#), [359](#), [2852](#), [2969](#)
 __cs_generate_variant:wwNw .. [2813](#)
 __cs_generate_variant_F_-
 form:nnn [3102](#)
 __cs_generate_variant_loop:nNwN
 [358](#), [359](#), [2853](#), [2865](#)
 __cs_generate_variant_loop_-
 base:N [2865](#)
 __cs_generate_variant_loop_-
 end:nwwwNNnn . [358](#), [359](#), [2855](#), [2865](#)
 __cs_generate_variant_loop_-
 invalid:NNwNNnn [359](#), [2865](#)
 __cs_generate_variant_loop_-
 long:wNNnn [359](#), [2858](#), [2865](#)
 __cs_generate_variant_loop_-
 same:w [359](#), [2865](#)
 __cs_generate_variant_loop_-
 special:NNwNNnn [2865](#), [2964](#)
 __cs_generate_variant_p_-
 form:nnn [3102](#)
 __cs_generate_variant_same:N ...
 [359](#), [2910](#), [2958](#)
 __cs_generate_variant_T_-
 form:nnn [3102](#)
 __cs_generate_variant_TF_-
 form:nnn [3102](#)
 __cs_get_function_name:N [322](#)
 __cs_get_function_signature:N . [322](#)
 __cs_parm_from_arg_count_-
 test:nnTF [1914](#)
 __cs_split_function_auxi:w .. [1722](#)
 __cs_split_function_auxii:w . [1722](#)
 __cs_tmp:w [323](#), [356](#),
 [361](#), [362](#), [366](#), [1722](#), [1737](#), [1844](#),
 [1860](#), [1862](#), [1863](#), [1864](#), [1865](#), [1866](#),
 [1867](#), [1868](#), [1869](#), [1870](#), [1871](#), [1872](#),
 [1873](#), [1874](#), [1875](#), [1876](#), [1877](#), [1878](#),
 [1879](#), [1880](#), [1881](#), [1882](#), [1883](#), [1884](#),
 [1885](#), [1974](#), [2014](#), [2015](#), [2016](#), [2017](#),
 [2018](#), [2019](#), [2020](#), [2021](#), [2022](#), [2023](#),
 [2024](#), [2025](#), [2026](#), [2027](#), [2028](#), [2029](#),
 [2030](#), [2031](#), [2032](#), [2033](#), [2034](#), [2035](#),
 [2036](#), [2037](#), [2038](#), [2046](#), [2047](#), [2048](#),
 [2049](#), [2050](#), [2051](#), [2052](#), [2053](#), [2054](#),
 [2055](#), [2056](#), [2057](#), [2058](#), [2059](#), [2060](#),
 [2061](#), [2062](#), [2063](#), [2064](#), [2065](#), [2066](#),
 [2067](#), [2068](#), [2069](#), [2817](#), [2835](#), [2977](#),
 [2985](#), [3003](#), [3048](#), [3164](#), [3171](#), [3172](#),
 [3173](#), [3174](#), [3175](#), [3176](#), [3177](#), [3178](#),
 [3179](#), [3180](#), [3181](#), [3182](#), [3183](#), [3184](#),
 [3185](#), [3186](#), [3187](#), [3188](#), [3189](#), [3190](#),
 [3191](#), [3192](#), [3193](#), [3194](#), [3195](#), [3196](#),
 [3197](#), [3198](#), [3199](#), [3200](#), [3201](#), [3202](#),
 [3203](#), [3204](#), [3205](#), [3206](#), [3207](#), [3208](#),
 [3209](#), [3210](#), [3211](#), [3212](#), [3213](#), [3214](#)
 __cs_to_str:N [323](#), [1713](#)
 __cs_to_str:w [323](#), [1713](#)
 __cs_use_i_delimit_by_s_stop:nw
 [2794](#), [3115](#)
 __cs_use_none_delimit_by_q_-
 recursion_stop:w
 [2794](#), [2841](#), [2848](#), [3128](#)
 __cs_use_none_delimit_by_s_-
 stop:w [2794](#), [3119](#)
 csc [216](#)
 cscd [216](#)
 \csname [4](#), [21](#), [25](#), [30](#), [42](#),
 [64](#), [66](#), [67](#), [68](#), [79](#), [86](#), [107](#), [111](#), [130](#), [223](#)
 \csstring [804](#)
 \currentcjktoken [1108](#), [1165](#)
 \currentgrouplevel [515](#)
 \currentgroupstype [516](#)
 \currentifbranch [517](#)
 \currentiflevel [518](#)
 \currentifttype [519](#)
 \currentspacingmode [1109](#)
 \currentxspacingmode [1110](#)

 D
 \d [29167](#), [31009](#)
 \day [224](#), [1308](#), [9442](#)

- dd 219
- \deadcycles 225
- debug commands:
 - \debug_off: 303
 - \debug_off:n 24, 1186, 1186, 1187, 1187, 1187, 1496
 - \debug_on: 303
 - \debug_on:n 24, 1186, 1186, 1496
 - \debug_resume: . 24, 1082, 1506, 27462
 - \debug_suspend: 24, 1082, 1506, 27455
- debug internal commands:
 - \g__debug_deprecation_off_tl . 1508
 - \g__debug_deprecation_on_tl . 1508
- \def 48, 49, 50, 85, 87, 88, 104, 105, 108, 119, 143, 182, 226
- default commands:
 - .default:n 189, 15002
- \defaultthyphenchar 227
- \defaultskewchar 228
- deg 218
- \delcode 229
- \delimiter 230
- \delimiterfactor 231
- \delimitershortfall 232
- deprecation internal commands:
 - __deprecation_date_compare:nNnTF 31771, 31788, 31791
 - __deprecation_date_compare_-aux:w 31771
 - \l_deprecation_grace_period_-bool 1185, 31768, 31787, 31797, 31871
 - __deprecation_just_error:nNn . . 1186, 31809
 - __deprecation_minus_six_-months:w 31785
 - __deprecation_not_yet_deprecated:nTF 1186, 31785, 31819
 - __deprecation_old:Nnn 31899
 - __deprecation_old_protected:Nnn 31899
 - __deprecation_patch_aux:Nn . . . 1187, 31809
 - __deprecation_patch_aux:nNnNnn . 31809
 - __deprecation_warn_once:nNnn 31809
- \detokenize 42, 130, 520
- \DH 29173, 30584, 30925
- \dh 29173, 30584, 30935
- dim commands:
 - \dim_abs:n 172, 671, 13953
 - \dim_add:Nn 172, 13935
 - \dim_case:nn 175, 14033
 - \dim_case:nnn 31963
 - \dim_case:nnTF 175, 14033, 14038, 14043, 31964
 - \dim_compare:nNnTF 173, 174, 175, 175, 175, 176, 205, 13988, 14057, 14093, 14101, 14110, 14116, 14128, 14131, 14142, 27813, 27816, 27821, 27835, 27838, 27843, 28191, 28196, 28206, 28335, 28347, 31269, 31286, 31320, 31334, 31344
 - \dim_compare:nTF 173, 174, 176, 176, 176, 176, 13993, 14065, 14073, 14082, 14088
 - \dim_compare_p:n 174, 13993
 - \dim_compare_p:nNn 173, 13988
 - \dim_const:Nn 171, 664, 674, 13902, 14232, 14233, 15645
 - \dim_do_until:nn 176, 14063
 - \dim_do_until:nNnn 175, 14091
 - \dim_do_while:nn 176, 14063
 - \dim_do_while:nNnn 175, 14091
 - \dim_eval:n 173, 174, 177, 177, 664, 1061, 13905, 14036, 14041, 14046, 14051, 14146, 14174, 14227, 14231, 27519, 27593, 27679, 27697, 27734, 27738, 27739, 27743, 27747, 27748, 27765, 27770, 27776, 27783, 27790, 27933, 27957, 27960, 27961, 27968, 28050, 28051, 28058, 28059, 28162, 28169, 28318, 28319, 28601, 28602, 28603
 - \dim_gadd:Nn 172, 13935
 - .dim_gset:N 189, 15010
 - \dim_gset:Nn 172, 664, 13923
 - \dim_gset_eq:NN 172, 13929
 - \dim_gsub:Nn 172, 13935
 - \dim_gzero:N 171, 13908, 13916
 - \dim_gzero_new:N 171, 13913
 - \dim_if_exist:NnTF 171, 13914, 13916, 13919
 - \dim_if_exist_p:N 171, 13919
 - \dim_log:N 179, 14228
 - \dim_log:n 179, 14228
 - \dim_max:nn . . 172, 13953, 28029, 28033
 - \dim_min:nn 172, 13953, 28027, 28031, 28044
 - \dim_new:N 171, 171, 13896, 13904, 13914, 13916, 14234, 14235, 14236, 14237, 27026, 27027, 27028, 27029, 27030, 27031, 27032, 27033, 27380, 27404, 27405, 27408, 27409, 27410, 27411, 27901, 27902, 27903, 27904, 27905, 28065, 28066, 28407, 28409, 28410
 - \dim_ratio:nn . 173, 673, 13984, 14221

- `.dim_set:N` [189](#), [15010](#)
- `\dim_set:Nn` [172](#), [13923](#),
[27053](#), [27054](#), [27055](#), [27087](#), [27098](#),
[27182](#), [27183](#), [27184](#), [27199](#), [27297](#),
[27298](#), [27299](#), [27301](#), [27303](#), [27305](#),
[27509](#), [27582](#), [27815](#), [27819](#), [27837](#),
[27841](#), [27872](#), [27886](#), [27963](#), [27998](#),
[28006](#), [28017](#), [28018](#), [28019](#), [28020](#),
[28026](#), [28028](#), [28030](#), [28032](#), [28037](#),
[28043](#), [28126](#), [28128](#), [28130](#), [28138](#),
[28140](#), [28194](#), [28269](#), [28270](#), [28272](#),
[28274](#), [28292](#), [28293](#), [28408](#), [28520](#),
[28521](#), [28567](#), [28568](#), [28569](#), [28571](#)
- `\dim_set_eq:NN` [172](#), [13929](#), [27531](#), [27533](#)
- `\dim_show:N` [178](#), [14224](#)
- `\dim_show:n` [179](#), [673](#), [14226](#)
- `\dim_sign:n` [177](#), [14176](#)
- `\dim_step_function:nnnN`
[176](#), [671](#), [14119](#), [14171](#)
- `\dim_step_inline:nnnn` ... [176](#), [14149](#)
- `\dim_step_variable:nnnNn` . [177](#), [14149](#)
- `\dim_sub:Nn` [172](#), [13935](#)
- `\dim_to_decimal:n`
[177](#), [14197](#), [14213](#), [14218](#)
- `\dim_to_decimal_in_bp:n`
[178](#), [178](#), [14212](#)
- `\dim_to_decimal_in_sp:n` [178](#),
[178](#), [762](#), [14214](#), [16907](#), [16944](#), [17542](#)
- `\dim_to_decimal_in_unit:nn` [178](#), [14216](#)
- `\dim_to_fp:n` . [178](#), [762](#), [781](#), [14224](#),
[21734](#), [27091](#), [27092](#), [27102](#), [27103](#),
[27171](#), [27174](#), [27175](#), [27200](#), [27215](#),
[27216](#), [27235](#), [27236](#), [27254](#), [27271](#),
[27274](#), [27275](#), [27826](#), [27827](#), [27828](#),
[27848](#), [27849](#), [27850](#), [27860](#), [27861](#),
[27877](#), [27878](#), [27879](#), [27880](#), [27890](#),
[27891](#), [28002](#), [28003](#), [28010](#), [28011](#),
[28084](#), [28087](#), [28088](#), [28139](#), [28141](#)
- `\dim_until_do:nn` [176](#), [14063](#)
- `\dim_until_do:nNnn` [175](#), [14091](#)
- `\dim_use:N` [177](#),
[177](#), [1061](#), [13956](#), [13962](#), [13963](#),
[13964](#), [13970](#), [13971](#), [13972](#), [13996](#),
[14015](#), [14175](#), [14179](#), [14194](#), [14200](#),
[27965](#), [27969](#), [27976](#), [27982](#), [27991](#),
[27992](#), [27993](#), [28147](#), [28154](#), [28300](#)
- `\dim_while_do:nn` [176](#), [14063](#)
- `\dim_while_do:nNnn` [176](#), [14091](#)
- `\dim_zero:N` [171](#), [171](#), [13908](#), [13914](#),
[27056](#), [27185](#), [27300](#), [27806](#), [27807](#)
- `\dim_zero_new:N` [171](#), [13913](#)
- `\c_max_dim` [179](#), [182](#), [715](#),
[14232](#), [14327](#), [15674](#), [15716](#), [15724](#),
[28017](#), [28018](#), [28019](#), [28020](#), [28037](#)
- `\g_tmpa_dim` [179](#), [14234](#)
- `\l_tmpa_dim` [179](#), [14234](#)
- `\g_tmpb_dim` [179](#), [14234](#)
- `\l_tmpb_dim` [179](#), [14234](#)
- `\c_zero_dim`
[179](#), [14128](#), [14131](#), [14184](#), [14232](#),
[14326](#), [15741](#), [26914](#), [26936](#), [27370](#),
[27813](#), [27816](#), [27821](#), [27835](#), [27838](#),
[27843](#), [28191](#), [28196](#), [28206](#), [31273](#),
[31284](#), [31290](#), [31302](#), [31320](#), [31324](#),
[31332](#), [31334](#), [31338](#), [31344](#), [31354](#)
- dim internal commands:
 - `__dim_abs:N` [13953](#)
 - `__dim_case:nnTF` [14033](#)
 - `__dim_case:nw` [14033](#)
 - `__dim_case_end:nw` [14033](#)
 - `__dim_compare:w` [13993](#)
 - `__dim_compare:wNN` [667](#), [13993](#)
 - `__dim_compare_!:w` [13993](#)
 - `__dim_compare_<:w` [13993](#)
 - `__dim_compare_=:w` [13993](#)
 - `__dim_compare_>:w` [13993](#)
 - `__dim_compare_end:w` .. [14001](#), [14025](#)
 - `__dim_compare_error:` ... [667](#), [13993](#)
 - `__dim_eval:w` [673](#), [13890](#),
[13924](#), [13926](#), [13936](#), [13940](#), [13945](#),
[13949](#), [13956](#), [13962](#), [13963](#), [13964](#),
[13970](#), [13971](#), [13972](#), [13987](#), [13990](#),
[13996](#), [14015](#), [14020](#), [14122](#), [14123](#),
[14124](#), [14175](#), [14179](#), [14200](#), [14215](#)
 - `__dim_eval_end:` [13890](#),
[13924](#), [13926](#), [13936](#), [13940](#), [13945](#),
[13949](#), [13956](#), [13966](#), [13974](#), [13987](#),
[13990](#), [14175](#), [14179](#), [14200](#), [14215](#)
 - `__dim_maxmin:wwN` [13953](#)
 - `__dim_ratio:n` [13984](#)
 - `__dim_sign:Nw` [14176](#)
 - `__dim_step:NnnN` [14119](#)
 - `__dim_step:NNnnnn` [14149](#)
 - `__dim_step:wwwN` [14119](#)
 - `__dim_tmp:w` [666](#)
 - `__dim_to_decimal:w` [14197](#)
 - `__dim_use_none_delimit_by_s_-
stop:w` [13895](#), [14011](#)
- `\dimen` [233](#), [10835](#)
- `\dimendef` [234](#)
- `\dimexpr` [521](#)
- `\directlua` [6](#), [33](#), [35](#), [805](#)
- `\disablecjktoken` [1166](#)
- `\discretionary` [235](#)
- `\disinhibitglue` [1111](#)
- `\displayindent` [236](#)
- `\displaylimits` [237](#)
- `\displaystyle` [238](#)

`\displaywidowpenalties` 522
`\displaywidowpenalty` 239
`\displaywidth` 240
`\divide` 241
`\DJ` 29174, 30585, 30926
`\dj` 29174, 30585, 30936
`\do` 1214
`\doublehyphendemerits` 242
`\dp` 243
`\draftmode` 909
`\dtou` 1112
`\dump` 244
`\dviextension` 806
`\dvifedback` 807
`\dvivariable` 808

E

`\edef` 94, 117, 245
`\efcode` 689
`\elapsedtime` 773
`\else` 5, 26, 28, 65, 69, 72, 75, 76, 80, 81, 246
 else commands:

`\else:`
 . 23, 102, 102, 103, 107, 114, 114,
 166, 185, 250, 250, 250, 317, 318,
 324, 356, 388, 402, 402, 529, 807,
 1359, 1404, 1582, 1590, 1616, 1742,
 1745, 1754, 1760, 1770, 1773, 1782,
 1788, 1908, 1930, 1939, 1953, 2011,
 2012, 2073, 2235, 2508, 2660, 2688,
 2703, 2711, 2748, 2818, 2869, 2870,
 2872, 2876, 2888, 2889, 2890, 2891,
 2892, 2893, 2894, 2895, 2896, 2960,
 2961, 2963, 3012, 3042, 3129, 3241,
 3279, 3287, 3298, 3308, 3327, 3351,
 3355, 3397, 3456, 3473, 3827, 3837,
 3848, 3863, 3871, 3887, 3901, 3945,
 3960, 4256, 4286, 4307, 4325, 4333,
 4343, 4356, 4372, 4443, 4455, 4504,
 4507, 4510, 4698, 4705, 4711, 4947,
 5003, 5006, 5009, 5021, 5036, 5251,
 5259, 5267, 5414, 5465, 5466, 5470,
 5475, 5516, 5569, 5681, 5695, 5917,
 5947, 5950, 5980, 5983, 6000, 6003,
 6104, 6109, 6127, 6146, 6149, 6198,
 6203, 6206, 6321, 6333, 6342, 6464,
 6469, 7586, 7664, 7673, 8073, 8084,
 8105, 8121, 8124, 8145, 8183, 8283,
 8310, 8348, 8356, 8657, 8690, 8741,
 8858, 8883, 8909, 8918, 8978, 9010,
 9030, 9052, 9070, 9086, 9096, 9112,
 9122, 9214, 9216, 9218, 9220, 9724,
 9739, 9761, 9775, 10299, 10302,
 10310, 10316, 10355, 10362, 10434,

10445, 10465, 10583, 10586, 10589,
 10592, 10595, 10598, 10601, 10670,
 10675, 10680, 10685, 10692, 10699,
 10704, 10709, 10714, 10719, 10724,
 10729, 10734, 10739, 10761, 10767,
 10770, 10805, 10808, 10872, 10881,
 10889, 10898, 10934, 10973, 10987,
 10996, 11006, 11063, 11360, 12510,
 13602, 13611, 13622, 13959, 13980,
 13991, 14001, 14026, 14186, 14189,
 15679, 15936, 15953, 15954, 15969,
 15979, 16074, 16150, 16212, 16215,
 16229, 16247, 16251, 16503, 16516,
 16536, 16564, 16565, 16587, 16608,
 16631, 16632, 16665, 16682, 16700,
 16735, 16739, 16775, 16792, 16798,
 16802, 16806, 16965, 16998, 17006,
 17039, 17043, 17055, 17065, 17075,
 17106, 17119, 17154, 17164, 17183,
 17196, 17209, 17213, 17224, 17247,
 17264, 17276, 17290, 17303, 17307,
 17315, 17317, 17327, 17338, 17354,
 17370, 17376, 17381, 17388, 17410,
 17440, 17463, 17491, 17494, 17670,
 17674, 17681, 17700, 17714, 17718,
 17725, 17747, 17764, 17770, 17802,
 17834, 17850, 17870, 17911, 17926,
 17959, 17961, 17967, 17982, 18035,
 18186, 18202, 18213, 18251, 18254,
 18257, 18260, 18291, 18300, 18309,
 18312, 18483, 18496, 18499, 18506,
 18524, 18548, 18549, 18564, 18574,
 18623, 18626, 18635, 18647, 18658,
 18672, 18685, 18725, 18759, 18779,
 18816, 18834, 18837, 18843, 18857,
 18892, 18910, 18913, 18916, 18919,
 18980, 19053, 19123, 19124, 19133,
 19168, 19251, 19255, 19259, 19321,
 19356, 19371, 19636, 19665, 19669,
 19829, 19838, 19892, 19903, 19919,
 19927, 19986, 20066, 20077, 20082,
 20116, 20129, 20141, 20147, 20268,
 20276, 20315, 20322, 20344, 20372,
 20387, 20391, 20413, 20444, 20447,
 20472, 20475, 20516, 20524, 20535,
 20538, 20653, 20668, 20683, 20698,
 20713, 20728, 20749, 20794, 21100,
 21138, 21139, 21148, 21192, 21247,
 21248, 21249, 21353, 21375, 21390,
 21408, 21456, 21472, 21678, 21745,
 21750, 21914, 21950, 21963, 21993,
 21997, 22005, 22032, 22058, 22066,
 22083, 22086, 22135, 22139, 22191,
 22250, 22262, 22684, 22685, 23134,

- 23137, 23140, 23150, 23165, 23192,
 23207, 23234, 23250, 23283, 23291,
 23293, 23295, 23297, 23299, 23301,
 23303, 23305, 23323, 23344, 23348,
 23420, 23424, 23613, 23614, 23619,
 23620, 23635, 23642, 23840, 23850,
 23894, 23903, 23915, 23916, 23918,
 23920, 23923, 23924, 23927, 23928,
 23937, 23939, 23941, 23944, 23945,
 23947, 23983, 23986, 24007, 24010,
 24018, 24026, 24029, 24038, 24041,
 24050, 24058, 24061, 24071, 24177,
 24284, 24328, 24332, 24335, 24346,
 24351, 24450, 24596, 24609, 24698,
 24727, 24766, 24784, 24893, 24927,
 25177, 25195, 25214, 25249, 25302,
 25349, 25353, 25360, 25381, 25392,
 25533, 25649, 25735, 25793, 25867,
 25902, 25914, 25940, 25958, 26150,
 26294, 26319, 26371, 26789, 26791,
 26797, 28943, 29040, 29044, 29055,
 29076, 29080, 29089, 29090, 29091,
 29092, 29093, 29094, 29095, 29096,
 29097, 29108, 29122, 29125, 29128,
 29131, 29134, 29137, 29140, 30004,
 30008, 30011, 30015, 31242, 31362
 \em 30857
 em 219
 \emergencystretch 247
 \emph 29369, 30830
 \enablecjktoken 1167
 \end .. 248, 302, 18380, 29184, 29191, 30871
 end internal commands:
 __regex_end 26138
 \endcsname 4, 21, 25, 30, 42,
 64, 66, 67, 68, 79, 86, 107, 111, 130, 249
 \endgroup .. 19, 20, 24, 29, 48, 98, 115, 250
 \endinput 99, 251
 \endL 523
 \endlinechar 129, 141, 252
 \endR 524
 \enquote 18382
 \ensuremath 29186
 \epTeXinputencoding 1113
 \epTeXversion 1114
 \eqno 253
 \errhelp 90, 254
 \errmessage 91, 255
 \ERROR 10402
 \errorcontextlines 256
 \errorstopmode 257
 \escapechar 258
 escapehex 28729
 \ETC 23480
 \eTeXglueshrinkorder 809
 \eTeXgluestretchorder 810
 \eTeXrevision 525
 \eTeXversion 526
 \etoksapp 811
 \etokspre 812
 \euc 1115
 \everycr 259
 \everydisplay 260
 \everyeof 527
 \everyhbox 261
 \everyjob 40, 41, 262
 \everymath 263
 \everypar 264
 \everyvbox 265
 ex 219
 \exceptionpenalty 813
 \exhyphenpenalty 266
 exp 214
 exp commands:
 \exp:w 36, 36, 37, 317,
 323, 339, 340, 341, 348, 349, 398,
 398, 404, 419, 532, 545, 621, 639,
 752, 753, 755, 756, 758, 759, 778,
 782, 783, 1179, 1381, 1517, 1519,
 2207, 2220, 2226, 2274, 2278, 2282,
 2287, 2293, 2299, 2315, 2327, 2333,
 2339, 2344, 2346, 2353, 2384, 2389,
 2398, 2403, 2412, 2414, 2422, 2429,
 2435, 2443, 2452, 2459, 2473, 2493,
 2497, 2502, 2504, 2541, 2723, 3082,
 3725, 3957, 3966, 3971, 3976, 3981,
 4219, 4378, 4448, 4731, 4736, 4741,
 4746, 4762, 4767, 4772, 4777, 4930,
 4939, 4994, 8316, 8321, 8326, 8331,
 9021, 9167, 9585, 9593, 9653, 9947,
 9955, 10290, 12326, 12909, 14000,
 14035, 14040, 14045, 14050, 15982,
 16097, 16101, 16479, 16605, 16606,
 16607, 16608, 16727, 16745, 16774,
 16818, 16830, 16835, 16843, 16851,
 16872, 16878, 16950, 16963, 16964,
 16973, 16986, 17004, 17005, 17025,
 17038, 17042, 17064, 17092, 17105,
 17118, 17142, 17153, 17163, 17182,
 17195, 17208, 17211, 17223, 17246,
 17275, 17289, 17306, 17326, 17337,
 17343, 17353, 17399, 17406, 17437,
 17452, 17460, 17477, 17493, 17497,
 17506, 17543, 17552, 17561, 17566,
 17568, 17579, 17581, 17596, 17599,
 17606, 17617, 17703, 17751, 17769,
 17772, 17786, 17799, 17849, 17867,
 17938, 17950, 17979, 17981, 17985,

17987, 18045, 18055, 18065, 18077,
 18193, 18210, 18220, 18375, 18376,
 18377, 18568, 18571, 18579, 18589,
 18597, 19609, 20140, 20162, 20317,
 20494, 20770, 21513, 21528, 21545,
 21582, 21599, 21641, 21660, 21673,
 21705, 21720, 21731, 21853, 21900,
 21940, 21976, 22156, 22256, 29203,
 29242, 29533, 29579, 29593, 29605,
 30639, 31400, 31402, 31404, 31406,
 31408, 31410, 31412, 31414, 31597,
 31602, 31607, 31612, 31628, 31646
 \exp_after:wN 33, 35, 36, 316, 320,
 338, 340, 403, 408, 514, 608, 730,
 752, 753, 755, 756, 820, 821, 882,
 883, 953, 975, 1042, 1132, 1179,
 1378, 1396, 1398, 1403, 1405, 1517,
 1519, 1573, 1597, 1615, 1617, 1636,
 1644, 1652, 1676, 1681, 1688, 1717,
 1721, 1726, 1737, 1753, 1755, 1758,
 1781, 1783, 1786, 1907, 1909, 1918,
 1938, 1940, 1979, 2043, 2139, 2148,
 2157, 2169, 2179, 2185, 2192, 2194,
 2206, 2207, 2219, 2220, 2225, 2226,
 2231, 2236, 2238, 2241, 2250, 2252,
 2255, 2256, 2257, 2260, 2262, 2264,
 2268, 2273, 2278, 2281, 2286, 2291,
 2292, 2293, 2297, 2298, 2299, 2305,
 2306, 2313, 2314, 2315, 2319, 2320,
 2321, 2325, 2326, 2327, 2331, 2332,
 2333, 2337, 2338, 2339, 2343, 2344,
 2345, 2346, 2350, 2351, 2352, 2353,
 2357, 2358, 2359, 2364, 2365, 2366,
 2367, 2371, 2372, 2373, 2374, 2380,
 2383, 2384, 2388, 2389, 2402, 2403,
 2410, 2412, 2414, 2418, 2422, 2424,
 2427, 2428, 2433, 2434, 2438, 2441,
 2442, 2446, 2449, 2450, 2451, 2456,
 2457, 2458, 2466, 2469, 2470, 2471,
 2472, 2477, 2479, 2481, 2482, 2493,
 2496, 2501, 2526, 2527, 2528, 2539,
 2540, 2552, 2553, 2554, 2559, 2567,
 2568, 2569, 2570, 2571, 2572, 2595,
 2596, 2597, 2598, 2658, 2659, 2661,
 2683, 2688, 2689, 2701, 2702, 2704,
 2708, 2709, 2710, 2713, 2715, 2718,
 2722, 2723, 2724, 2729, 2730, 2741,
 2747, 2749, 2753, 2754, 2757, 2758,
 2768, 2815, 2819, 2841, 2848, 2868,
 3011, 3013, 3040, 3041, 3043, 3080,
 3082, 3099, 3117, 3128, 3234, 3240,
 3242, 3266, 3317, 3318, 3397, 3431,
 3472, 3480, 3491, 3649, 3671, 3672,
 3673, 3674, 3733, 3734, 3735, 3796,
 3797, 3798, 3803, 3845, 3856, 3857,
 3883, 3897, 3941, 3943, 4061, 4190,
 4217, 4248, 4253, 4254, 4255, 4257,
 4271, 4281, 4298, 4322, 4332, 4335,
 4352, 4353, 4363, 4368, 4369, 4384,
 4385, 4386, 4444, 4446, 4447, 4448,
 4453, 4454, 4456, 4534, 4535, 4792,
 4793, 4805, 4860, 4883, 4917, 4918,
 4929, 4930, 4938, 4946, 4948, 4955,
 4960, 4978, 4979, 4980, 4992, 4993,
 5020, 5022, 5028, 5034, 5048, 5068,
 5079, 5095, 5103, 5111, 5118, 5125,
 5137, 5225, 5241, 5260, 5269, 5290,
 5291, 5296, 5297, 5322, 5323, 5338,
 5339, 5387, 5392, 5457, 5630, 5639,
 5685, 5801, 5837, 5839, 5854, 5860,
 6024, 6026, 6091, 6110, 6111, 6125,
 6126, 6153, 6154, 6269, 6291, 6304,
 6305, 6332, 6427, 6448, 6457, 7482,
 7484, 7496, 7504, 7648, 7682, 7694,
 7707, 7708, 7709, 7731, 7732, 7777,
 7812, 7813, 7814, 7898, 7899, 7901,
 7902, 7910, 7911, 7938, 7939, 7965,
 7966, 7969, 8065, 8079, 8084, 8087,
 8088, 8095, 8096, 8112, 8113, 8134,
 8135, 8144, 8255, 8260, 8265, 8288,
 8290, 8418, 8419, 8420, 8445, 8446,
 8629, 8657, 8662, 8690, 8703, 8713,
 8740, 8742, 8743, 8751, 8768, 8812,
 8874, 8881, 8917, 8919, 8926, 9019,
 9037, 9042, 9046, 9168, 9362, 9363,
 9364, 9429, 9584, 9592, 9653, 9725,
 9740, 9762, 9776, 9837, 9845, 9851,
 9946, 9954, 10038, 10039, 10042,
 10043, 10290, 10291, 10336, 10344,
 10410, 10411, 10412, 10503, 10504,
 10746, 10765, 10812, 10850, 10879,
 10880, 10882, 10888, 10891, 10933,
 10936, 10972, 10974, 10984, 10985,
 10986, 10988, 10994, 10995, 10997,
 11004, 11005, 11007, 11057, 11062,
 11064, 11070, 11199, 11380, 11381,
 11382, 11611, 11820, 11821, 12327,
 12328, 12329, 12330, 12402, 12403,
 12446, 12592, 12610, 12650, 12839,
 12842, 12892, 12901, 12904, 12907,
 12908, 12910, 12950, 13016, 13047,
 13068, 13080, 13136, 13222, 13223,
 13224, 13708, 13812, 13955, 13959,
 13962, 13963, 13970, 13971, 13995,
 14000, 14011, 14014, 14121, 14122,
 14123, 14178, 14199, 14298, 14662,
 14690, 14703, 14858, 14866, 15267,
 15427, 15428, 15507, 15530, 15697,

15698, 15699, 15717, 15725, 15749,
15750, 15794, 15935, 15937, 15938,
15956, 15957, 15958, 15968, 15970,
15978, 15980, 15987, 15988, 15989,
15990, 15991, 15992, 15997, 15998,
15999, 16000, 16001, 16002, 16003,
16046, 16059, 16062, 16073, 16075,
16090, 16094, 16095, 16096, 16099,
16100, 16164, 16166, 16193, 16197,
16222, 16226, 16243, 16250, 16252,
16334, 16342, 16359, 16368, 16414,
16479, 16548, 16549, 16550, 16610,
16620, 16639, 16645, 16664, 16666,
16668, 16679, 16680, 16683, 16694,
16698, 16705, 16706, 16717, 16718,
16727, 16734, 16736, 16737, 16745,
16774, 16792, 16793, 16796, 16797,
16799, 16800, 16804, 16805, 16807,
16808, 16817, 16818, 16823, 16829,
16835, 16843, 16851, 16870, 16871,
16874, 16875, 16877, 16884, 16885,
16887, 16905, 16906, 16934, 16937,
16942, 16943, 16948, 16949, 16951,
16960, 16961, 16962, 16963, 16966,
16967, 16968, 16971, 16986, 17003,
17004, 17014, 17015, 17025, 17037,
17041, 17054, 17056, 17064, 17074,
17076, 17082, 17087, 17089, 17091,
17097, 17098, 17102, 17104, 17116,
17117, 17139, 17141, 17147, 17150,
17152, 17156, 17161, 17166, 17167,
17177, 17178, 17180, 17181, 17184,
17188, 17193, 17207, 17210, 17222,
17231, 17238, 17239, 17240, 17241,
17243, 17245, 17256, 17257, 17258,
17259, 17261, 17263, 17265, 17266,
17267, 17273, 17274, 17284, 17288,
17289, 17291, 17292, 17293, 17298,
17304, 17305, 17316, 17318, 17325,
17326, 17328, 17329, 17336, 17342,
17352, 17420, 17433, 17434, 17435,
17436, 17450, 17451, 17453, 17458,
17459, 17474, 17476, 17493, 17497,
17506, 17540, 17541, 17542, 17548,
17549, 17550, 17551, 17557, 17558,
17559, 17560, 17567, 17580, 17588,
17594, 17595, 17597, 17598, 17604,
17605, 17607, 17633, 17646, 17668,
17669, 17671, 17672, 17679, 17680,
17682, 17685, 17697, 17698, 17699,
17701, 17702, 17703, 17712, 17713,
17715, 17716, 17723, 17724, 17726,
17729, 17744, 17745, 17746, 17749,
17750, 17751, 17761, 17762, 17763,
17766, 17767, 17768, 17771, 17775,
17784, 17785, 17796, 17797, 17798,
17801, 17803, 17804, 17805, 17832,
17833, 17835, 17836, 17837, 17847,
17848, 17849, 17851, 17852, 17853,
17865, 17866, 17869, 17871, 17872,
17873, 17893, 17894, 17895, 17896,
17897, 17898, 17899, 17909, 17910,
17912, 17913, 17914, 17920, 17931,
17932, 17933, 17934, 17935, 17936,
17937, 17938, 17943, 17944, 17945,
17946, 17947, 17948, 17949, 17965,
17966, 17968, 17969, 17976, 17977,
17978, 17983, 17984, 17986, 18002,
18017, 18026, 18036, 18042, 18043,
18044, 18049, 18062, 18063, 18064,
18070, 18192, 18209, 18219, 18244,
18245, 18292, 18374, 18482, 18484,
18523, 18525, 18528, 18563, 18565,
18567, 18570, 18577, 18578, 18581,
18582, 18587, 18588, 18595, 18596,
18631, 18632, 18633, 18635, 18646,
18671, 18673, 18679, 18680, 18684,
18687, 18709, 18711, 18724, 18726,
18732, 18734, 18737, 18743, 18745,
18747, 18748, 18749, 18751, 18756,
18758, 18760, 18764, 18767, 18773,
18774, 18778, 18780, 18781, 18782,
18790, 18792, 18793, 18800, 18806,
18813, 18814, 18819, 18820, 18821,
18822, 18841, 18842, 18843, 18849,
18850, 18851, 18856, 18858, 18866,
18868, 18870, 18871, 18873, 18884,
18886, 18888, 18889, 18894, 18945,
18946, 18953, 18954, 18956, 18958,
18960, 18963, 18966, 18968, 18970,
18979, 18981, 18987, 18989, 18991,
18992, 18993, 18999, 19001, 19003,
19004, 19005, 19026, 19027, 19030,
19038, 19040, 19044, 19045, 19046,
19047, 19052, 19054, 19061, 19064,
19067, 19070, 19079, 19082, 19085,
19088, 19095, 19097, 19103, 19111,
19113, 19115, 19132, 19134, 19141,
19143, 19146, 19152, 19154, 19156,
19157, 19158, 19160, 19174, 19175,
19178, 19196, 19198, 19200, 19212,
19215, 19218, 19221, 19224, 19227,
19230, 19233, 19237, 19249, 19253,
19257, 19260, 19275, 19281, 19283,
19285, 19295, 19319, 19322, 19334,
19336, 19340, 19341, 19342, 19344,
19345, 19347, 19354, 19362, 19363,
19369, 19370, 19376, 19379, 19380,

19381, 19382, 19390, 19432, 19437,
19439, 19446, 19449, 19452, 19455,
19458, 19461, 19469, 19470, 19482,
19490, 19492, 19502, 19504, 19511,
19520, 19522, 19525, 19528, 19531,
19534, 19547, 19549, 19557, 19559,
19567, 19569, 19579, 19582, 19585,
19592, 19607, 19608, 19625, 19627,
19628, 19685, 19698, 19700, 19706,
19719, 19721, 19723, 19747, 19761,
19763, 19770, 19772, 19813, 19814,
19815, 19817, 19818, 19819, 19821,
19822, 19828, 19830, 19831, 19837,
19839, 19840, 19841, 19842, 19854,
19860, 19862, 19899, 19906, 19913,
19933, 19934, 19936, 19938, 19940,
19953, 19958, 19959, 19960, 19961,
19962, 19966, 19971, 19973, 19979,
19985, 19987, 19988, 19994, 19995,
19996, 19997, 19998, 19999, 20000,
20001, 20006, 20008, 20010, 20012,
20014, 20019, 20021, 20023, 20025,
20027, 20029, 20047, 20051, 20059,
20060, 20065, 20067, 20076, 20079,
20080, 20081, 20083, 20084, 20085,
20093, 20099, 20111, 20114, 20115,
20117, 20118, 20142, 20143, 20146,
20148, 20164, 20168, 20169, 20170,
20186, 20192, 20258, 20259, 20260,
20267, 20269, 20270, 20275, 20277,
20278, 20287, 20288, 20290, 20293,
20296, 20312, 20316, 20317, 20321,
20323, 20359, 20365, 20366, 20368,
20370, 20371, 20373, 20374, 20384,
20385, 20388, 20389, 20390, 20392,
20393, 20394, 20411, 20412, 20414,
20415, 20421, 20423, 20426, 20429,
20432, 20435, 20443, 20446, 20448,
20451, 20458, 20462, 20470, 20471,
20474, 20476, 20478, 20483, 20484,
20490, 20495, 20496, 20504, 20505,
20506, 20507, 20550, 20572, 20573,
20576, 20577, 20586, 20587, 20588,
20592, 20599, 20600, 20601, 20742,
20743, 20744, 20746, 20764, 20765,
20766, 20767, 20768, 20769, 20776,
20785, 20792, 20793, 21017, 21018,
21024, 21025, 21028, 21033, 21036,
21039, 21042, 21045, 21048, 21051,
21054, 21070, 21071, 21081, 21090,
21098, 21099, 21101, 21102, 21107,
21108, 21117, 21124, 21133, 21134,
21147, 21149, 21177, 21178, 21187,
21190, 21215, 21221, 21222, 21264,
21265, 21267, 21281, 21282, 21290,
21301, 21335, 21338, 21348, 21349,
21352, 21354, 21360, 21374, 21376,
21417, 21420, 21440, 21513, 21523,
21527, 21545, 21548, 21569, 21570,
21577, 21581, 21599, 21602, 21631,
21632, 21638, 21639, 21640, 21647,
21655, 21659, 21673, 21676, 21692,
21693, 21700, 21704, 21715, 21719,
21731, 21736, 21737, 21738, 21744,
21746, 21749, 21751, 21813, 21842,
21852, 21859, 21864, 21865, 21875,
21902, 21913, 21915, 21917, 21919,
21924, 21925, 21927, 21938, 21939,
21959, 21965, 21966, 21968, 21971,
21976, 21982, 21983, 22013, 22015,
22018, 22021, 22023, 22031, 22033,
22037, 22041, 22046, 22051, 22062,
22126, 22127, 22151, 22152, 22153,
22154, 22155, 22163, 22174, 22180,
22206, 22207, 22208, 22215, 22216,
22223, 22224, 22232, 22233, 22237,
22238, 22239, 22244, 22249, 22255,
22258, 22259, 22260, 22261, 22262,
22466, 22467, 22676, 22762, 22763,
22805, 22824, 22825, 22840, 22841,
22842, 23068, 23074, 23076, 23087,
23147, 23148, 23149, 23150, 23156,
23157, 23173, 23191, 23193, 23199,
23200, 23231, 23233, 23235, 23265,
23280, 23282, 23284, 23309, 23319,
23327, 23337, 23347, 23349, 23351,
23386, 23410, 23419, 23422, 23423,
23425, 23426, 23434, 23435, 23449,
23499, 23512, 23513, 23518, 23519,
23522, 23525, 23570, 23577, 23584,
23590, 23597, 23605, 23641, 23643,
23652, 23775, 23778, 23819, 23839,
23841, 23842, 23849, 23852, 23853,
23877, 23896, 23905, 24017, 24019,
24025, 24028, 24030, 24037, 24040,
24042, 24049, 24051, 24057, 24060,
24063, 24378, 24451, 24463, 24595,
24598, 24608, 24610, 24793, 24801,
24876, 24926, 25140, 25422, 25563,
25586, 25637, 25663, 25727, 25728,
25736, 25739, 25900, 25901, 25904,
25905, 25913, 25915, 25916, 25925,
25939, 25942, 26012, 26261, 26293,
26295, 28646, 28868, 28927, 28929,
29043, 29045, 29052, 29053, 29056,
29079, 29081, 29087, 29099, 29107,
29109, 29201, 29240, 29310, 29311,
29354, 29355, 29406, 29455, 29459,

- 29488, 29531, 29577, 29591, 29603,
 30296, 30298, 30302, 30344, 30346,
 30360, 30362, 30625, 30637, 30685,
 30686, 30747, 30748, 30749, 30756,
 30802, 30916, 30918, 30991, 30993,
 31440, 31450, 31455, 31458, 31597,
 31602, 31607, 31612, 31626, 31629,
 31644, 31646, 31647, 31654, 31659,
 31661, 31664, 31678, 31698, 31699,
 31706, 31707, 31748, 31762, 31793
 \exp_args:cc 29, [1395](#), [2249](#)
 \exp_args:Nc 27, 29, [333](#), [1395](#), 1399,
 1407, 1621, 1634, 1642, 1650, 1842,
 1861, 1887, 1892, 1899, 1958, 1970,
 2042, 2075, 2076, 2077, 2078, 2100,
 2104, [2249](#), 2812, 3765, 4016, 4566,
 8894, 12343, 12379, 12578, 15586,
 16628, 16867, 17755, 17779, 17809,
 17811, 17813, 17815, 17817, 17819,
 17821, 17823, 17841, 17857, 17858,
 17877, 17879, 22376, 22377, 22378,
 22406, 23369, 24748, 27726, 27757,
 31240, 31582, 31668, 31673, 31681
 \exp_args:Ncc
 . 31, 1889, 1893, 1901, 2083, 2084,
 2085, 2086, [2249](#), 3406, 3446, 5405
 \exp_args:Nccc 32, [2249](#)
 \exp_args:Ncco 32, [2348](#)
 \exp_args:Nccx 32, [3190](#)
 \exp_args:Ncf 31, [2289](#)
 \exp_args:NcNc 32, [2348](#)
 \exp_args:NcNo 32, [2348](#)
 \exp_args:Ncno 32, [3190](#)
 \exp_args:NcnV 32, [3190](#)
 \exp_args:Ncnx 32, [3190](#)
 \exp_args:Nco 31, [343](#), [2289](#)
 \exp_args:Ncoo 32, [3190](#)
 \exp_args:NcV 31, [2289](#)
 \exp_args:Ncv 31, [2289](#)
 \exp_args:NcVV 32, [3190](#)
 \exp_args:Ncx 31, [3164](#), [12335](#)
 \exp_args:Ne 30,
 339, 1460, 2202, [2265](#), 2490, 6491,
 6506, 10513, 10517, 12341, 12377,
 13100, 13102, 13181, 13249, 13271,
 13408, 13426, 13446, 13456, 13466,
 13494, 13724, 29375, 29436, 29446,
 29534, 29693, 29935, 30034, 30049,
 30112, 30640, 30789, 30908, 30985
 \exp_args:Nee . 31, [3164](#), 13591, 13754
 \exp_args:Neee 32, [3190](#), [13432](#)
 \exp_args:Nf 30, 1946, [2277](#),
 2536, 2602, 2637, 2651, 3714, 4390,
 4391, 4407, 4425, 4433, 4437, 4461,
 4467, 4477, 4907, 4909, 4968, 4970,
 4986, 5304, 5309, 6207, 6208, 6372,
 6383, 7780, 7781, 7797, 8317, 8322,
 8327, 8332, 8503, 8572, 8574, 8592,
 8601, 8612, 8621, 8759, 8776, 9014,
 10027, 10078, 10092, 10113, 10124,
 10172, 10242, 10248, 10254, 10260,
 10423, 10543, 10627, 11637, 12942,
 13000, 14036, 14041, 14046, 14051,
 15768, 18435, 21509, 22075, 22318,
 22523, 25280, 31368, 31370, 31791
 \exp_args:Nff . . 31, [3164](#), 4431, 16277
 \exp_args:Nffo 32, [3190](#)
 \exp_args:Nfo 31, [3164](#)
 \exp_args:NNc
 31, [312](#), 1888, 1891, 1900,
 1972, 2079, 2080, 2081, 2082, 2117,
 2120, [2249](#), 3082, 7891, 8460, 8471,
 12446, 12561, 12562, 12650, 14152,
 14159, 18445, 18452, 22103, 22408
 \exp_args:Nnc 31, [3164](#)
 \exp_args:NNcf 32, [3190](#)
 \exp_args:NNe 31, [2289](#)
 \exp_args:Nne 31, [3164](#)
 \exp_args:NNf 31,
 2289, 5672, 12445, 12649, 12826,
 14145, 15843, 15848, 20736, 20737
 \exp_args:Nnf
 31, [3164](#), 3692, 10566, 15584
 \exp_args:Nnff 32, [3190](#), 10572
 \exp_args:Nnnc 32, [3190](#)
 \exp_args:Nnnf 32, [3190](#)
 \exp_args:NNNo . . 32, [2260](#), 24112,
 25000, 25057, 26171, 26255, 32161
 \exp_args:NNno 32, [3190](#)
 \exp_args:Nnno 32, [3190](#)
 \exp_args:NNNV 32, [2348](#)
 \exp_args:NNV 12564
 \exp_args:NNnV 32, [3190](#)
 \exp_args:NNNx
 32, [938](#), [3190](#), 24120, 24591
 \exp_args:NNnx 32, [3190](#)
 \exp_args:Nnnx 32, [3190](#)
 \exp_args:NNo 25, 31,
 2260, 2545, 8491, 11193, 15425, 25821
 \exp_args:Nno 31, [3164](#),
 3452, 3677, 3738, 9370, 9982, 12535,
 13231, 14003, 16319, 16327, 16336,
 16353, 16361, 16389, 16893, 16897
 \exp_args:NNoo 32, [3190](#)
 \exp_args:NNox 32, [3190](#)
 \exp_args:Nnox 32, [3190](#)
 \exp_args:NNV 31, [2289](#)
 \exp_args:NNv 31, [2289](#)

- \exp_args:NnV [31](#), [3164](#)
- \exp_args:Nnv [31](#), [3164](#)
- \exp_args:NNVV [32](#), [3190](#)
- \exp_args:NNx
 . [31](#), [2125](#), [3164](#), [13781](#), [13795](#), [31860](#)
- \exp_args:Nnx [31](#), [3164](#)
- \exp_args:No [27](#),
 [30](#), [1179](#), [2109](#), [2114](#), [2260](#), [2545](#),
 [2549](#), [2579](#), [2617](#), [2680](#), [2697](#), [2735](#),
 [3048](#), [3071](#), [3078](#), [3150](#), [3167](#), [3665](#),
 [3670](#), [3905](#), [3906](#), [3907](#), [3934](#), [3935](#),
 [3936](#), [3937](#), [3938](#), [4004](#), [4023](#), [4032](#),
 [4047](#), [4121](#), [4124](#), [4126](#), [4236](#), [4238](#),
 [4265](#), [4274](#), [4409](#), [4416](#), [4418](#), [4475](#),
 [4484](#), [4799](#), [4826](#), [4831](#), [4845](#), [4903](#),
 [4914](#), [4964](#), [4975](#), [5042](#), [5061](#), [5099](#),
 [5114](#), [5533](#), [5586](#), [5869](#), [7474](#), [8491](#),
 [8578](#), [8584](#), [9345](#), [9360](#), [9844](#), [9856](#),
 [9858](#), [9893](#), [9898](#), [10107](#), [10111](#),
 [11814](#), [12594](#), [12708](#), [12738](#), [12834](#),
 [13219](#), [13291](#), [14304](#), [14973](#), [15007](#),
 [15035](#), [15057](#), [15128](#), [15137](#), [15143](#),
 [15178](#), [15185](#), [15240](#), [15425](#), [15458](#),
 [23373](#), [23396](#), [23453](#), [23455](#), [24188](#),
 [24245](#), [24265](#), [24901](#), [25254](#), [25868](#),
 [25911](#), [26316](#), [26346](#), [26821](#), [29074](#),
 [31558](#), [31683](#), [31687](#), [31731](#), [31788](#)
- \exp_args:Noc [31](#), [3164](#)
- \exp_args:Nof [31](#), [3164](#), [3707](#)
- \exp_args:Noo . [31](#), [3164](#), [24287](#), [25267](#)
- \exp_args:Noof [32](#), [3190](#)
- \exp_args:Nooo [32](#), [3190](#)
- \exp_args:Noooo [12343](#), [12379](#)
- \exp_args:Noox [32](#), [3190](#)
- \exp_args:Nox [31](#), [3164](#)
- \exp_args:NV [30](#), [2277](#),
 [12976](#), [13052](#), [13207](#), [14971](#), [15005](#),
 [15033](#), [15055](#), [22421](#), [23679](#), [29627](#)
- \exp_args:Nv .. [30](#), [2277](#), [29476](#), [30782](#)
- \exp_args:NVo [31](#), [3164](#)
- \exp_args:NVV [31](#), [2289](#), [12887](#)
- \exp_args:Nx [31](#),
 [1916](#), [2376](#), [3091](#), [4523](#), [5422](#), [8896](#),
 [8996](#), [10393](#), [11584](#), [12744](#), [14975](#),
 [15009](#), [15037](#), [15059](#), [18159](#), [22418](#),
 [22431](#), [24420](#), [24421](#), [24804](#), [25674](#)
- \exp_args:Nxo [31](#), [3164](#)
- \exp_args:Nxx [31](#), [3164](#)
- \exp_args_generate:n [266](#), [3148](#), [12338](#)
- \exp_end: [36](#), [36](#), [317](#), [320](#), [323](#), [340](#),
 [341](#), [348](#), [348](#), [395](#), [398](#), [398](#), [404](#),
 [419](#), [533](#), [621](#), [753](#), [782](#), [1179](#), [1179](#),
 [1382](#), [1622](#), [1635](#), [1643](#), [1651](#), [2238](#),
 [2247](#), [2504](#), [2534](#), [2724](#), [3082](#), [3741](#),
 [3996](#), [4197](#), [4384](#), [4385](#), [4456](#), [4789](#),
 [4963](#), [8343](#), [9199](#), [9202](#), [9203](#), [9204](#),
 [9205](#), [9206](#), [9207](#), [9208](#), [9209](#), [9210](#),
 [9212](#), [9580](#), [10321](#), [10333](#), [10336](#),
 [10341](#), [10344](#), [10350](#), [10358](#), [10406](#),
 [10411](#), [12330](#), [14062](#), [16610](#), [17573](#),
 [20164](#), [21902](#), [21904](#), [21976](#), [29222](#),
 [29554](#), [30658](#), [31433](#), [31588](#), [31617](#)
- \exp_end_continue_f:nw [37](#), [2504](#)
- \exp_end_continue_f:w
 [36](#), [37](#), [339](#), [755](#),
 [756](#), [2207](#), [2278](#), [2315](#), [2339](#), [2403](#),
 [2422](#), [2435](#), [2459](#), [2473](#), [2493](#), [2504](#),
 [3957](#), [9021](#), [9653](#), [12255](#), [12909](#),
 [14000](#), [15982](#), [16097](#), [16101](#), [16727](#),
 [16745](#), [16766](#), [16830](#), [16835](#), [16843](#),
 [16851](#), [16872](#), [16950](#), [16986](#), [16994](#),
 [17025](#), [17399](#), [17406](#), [17452](#), [17499](#),
 [17506](#), [17543](#), [17587](#), [17593](#), [17596](#),
 [17606](#), [17617](#), [17786](#), [17979](#), [17981](#),
 [17985](#), [17987](#), [18045](#), [18055](#), [18065](#),
 [18077](#), [18193](#), [18210](#), [18220](#), [18375](#),
 [18376](#), [18377](#), [18568](#), [18579](#), [18589](#),
 [18597](#), [19609](#), [20317](#), [20494](#), [20770](#),
 [21513](#), [21528](#), [21545](#), [21582](#), [21599](#),
 [21641](#), [21660](#), [21673](#), [21705](#), [21720](#),
 [21731](#), [21853](#), [21940](#), [22156](#), [22256](#)
- \exp_last_two_unbraced:Nnn
 [33](#), [2476](#), [27800](#), [28326](#), [28330](#)
- \exp_last_unbraced:Nco [33](#), [2410](#), [9965](#)
- \exp_last_unbraced:NcV [33](#), [2410](#)
- \exp_last_unbraced:Ne [33](#), [2410](#), [21784](#)
- \exp_last_unbraced:Nf
 . [33](#), [2410](#), [3335](#), [3356](#), [3463](#), [3485](#),
 [5753](#), [6038](#), [6225](#), [6397](#), [8590](#), [8610](#),
 [15784](#), [16272](#), [18184](#), [18661](#), [23830](#)
- \exp_last_unbraced:Nfo [33](#), [2410](#), [22106](#)
- \exp_last_unbraced:NNf [33](#), [2410](#)
- \exp_last_unbraced:NNNf [33](#), [2410](#), [8959](#)
- \exp_last_unbraced:NNNNf
 [33](#), [2410](#), [8964](#)
- \exp_last_unbraced:NNNNNo
 [33](#), [2410](#), [2828](#), [2832](#), [2995](#),
 [5157](#), [5870](#), [14713](#), [16050](#), [16068](#), [29115](#)
- \exp_last_unbraced:NNNo [33](#), [2410](#)
- \exp_last_unbraced:NnNo [33](#), [2410](#)
- \exp_last_unbraced:NNNV [33](#), [2410](#)
- \exp_last_unbraced:NNo
 [33](#), [2410](#), [4204](#), [9932](#),
 [12741](#), [13149](#), [28297](#), [29248](#), [30673](#)
- \exp_last_unbraced:Nno
 [33](#), [2410](#), [7857](#), [11411](#)
- \exp_last_unbraced:NNV [33](#), [2410](#)

- \exp_last_unbraced:No
- . 33, [2410](#), 28471, 28476, 28544, 28550
- \exp_last_unbraced:Noo
- 33, [2410](#), 11251, 11345
- \exp_last_unbraced:NV 33, [2410](#)
- \exp_last_unbraced:Nv 33, [2410](#), 10414
- \exp_last_unbraced:Nx
- 33, [2410](#), 5612, 5616, 5658
- \exp_not:N 34,
- 34, 90, 165, 219, 340, 347, 352,
- 401, 402, 577, 584, 760, 953, 962,
- 1035, 1039, [1378](#), 1577, 1663, 1666,
- 1978, 1979, 2042, 2043, [2185](#), 2231,
- 2377, [2481](#), 2481, 2559, 2563, 2605,
- 2658, 2678, 2688, 2701, 2805, 2807,
- 2808, 2815, 2816, 2817, 2818, 2819,
- 2820, 2826, 2852, 2861, 2916, 2917,
- 2977, 2983, 2985, 2991, 3052, 3069,
- 3071, 3099, 3789, 4278, 4281, 4295,
- 4298, 4331, 4338, 4594, 4595, 4817,
- 4818, 6513, 6514, 6515, 6517, 6518,
- 6520, 7473, 7475, 7873, 7920, 8476,
- 8739, 9907, 10017, 10020, 10028,
- 10029, 10278, 10363, 10367, 10396,
- 10581, 10584, 10587, 10590, 10593,
- 10596, 10599, 10665, 10669, 10674,
- 10679, 10684, 10691, 10698, 10703,
- 10708, 10713, 10718, 10723, 10733,
- 10738, 10743, 10746, 10747, 10750,
- 10760, 10765, 10780, 10798, 10803,
- 10804, 10805, 10806, 10807, 10808,
- 10810, 10812, 10813, 10814, 10818,
- 10819, 10822, 10823, 10844, 10847,
- 10848, 10850, 10851, 10855, 10858,
- 10859, 10861, 10864, 10946, 10952,
- 10983, 10986, 10993, 10994, 11003,
- 11004, 11018, 11019, 11034, 11039,
- 11044, 11053, 11054, 11300, 11323,
- 11683, 11685, 11687, 11689, 11694,
- 11695, 11700, 11702, 11704, 11983,
- 11985, 11987, 11989, 11994, 11995,
- 12000, 12002, 12004, 12470, 12471,
- 12475, 13145, 13153, 13216, 13219,
- 13220, 13222, 13223, 13224, 13225,
- 13228, 13229, 13298, 13300, 13679,
- 13680, 13681, 13682, 13683, 13686,
- 13687, 14164, 14204, 14721, 14723,
- 14737, 14739, 14793, 14794, 14806,
- 14870, 14871, 14938, 14939, 15110,
- 15111, 15112, 15113, 15114, 15117,
- 15119, 15121, 15122, 15161, 15162,
- 15163, 15164, 15167, 15169, 15171,
- 15172, 15203, 15204, 15205, 15206,
- 15209, 15211, 15213, 15214, 15222,
- 15223, 15224, 15225, 15226, 15229,
- 15231, 15233, 15234, 16046, 16047,
- 16791, 16792, 16887, 16888, 16889,
- 16890, 16996, 17036, 17040, 17062,
- 17155, 17187, 17272, 17286, 17303,
- 17314, 17324, 17361, 17364, 17470,
- 17471, 17473, 17474, 17475, 17476,
- 17477, 17478, 17481, 17483, 17485,
- 17664, 17666, 17708, 17710, 17831,
- 17846, 18457, 18614, 20623, 20624,
- 20625, 20629, 20630, 20631, 22471,
- 23135, 23138, 23290, 23291, 23292,
- 23293, 23294, 23295, 23296, 23297,
- 23298, 23299, 23300, 23301, 23302,
- 23303, 23304, 23305, 23308, 23309,
- 23310, 23806, 23808, 23810, 23812,
- 23814, 23816, 24176, 24178, 24371,
- 24373, 24384, 24388, 24555, 25009,
- 25668, 25882, 26004, 26017, 26380,
- 28419, 28420, 28433, 28440, 28868,
- 28872, 28876, 28877, 28882, 29037,
- 29038, 29041, 29052, 29120, 29123,
- 29126, 29129, 29132, 29135, 29138,
- 29255, 29256, 29257, 29259, 29261,
- 29265, 29269, 29270, 29426, 29427,
- 29430, 29431, 29446, 29613, 29677,
- 29751, 29756, 29757, 29801, 29806,
- 29808, 29809, 29810, 29812, 29813,
- 29815, 29910, 29912, 29915, 30260,
- 30266, 30268, 30269, 30273, 30274,
- 30276, 30278, 30297, 30299, 30303,
- 30345, 30347, 30361, 30363, 30917,
- 30919, 30992, 30994, 31373, 31375,
- 31379, 31381, 31386, 31388, 31492,
- 31744, 31745, 31838, 31846, 31862,
- 31865, 31866, 31879, 31882, 32099
- \exp_not:n 16, 29, 30, 34, 34,
- 34, 34, 34, 35, 35, 35, 53, 54, 54,
- 56, 56, 57, 78, 79, 84, 85, 90, 126,
- 127, 128, 128, 147, 165, 198, 269,
- 272, 303, 379, 386, 396, 405, 485,
- 489, 546, 547, 550, 553, 593, 685,
- 948, 953, 958, 962, 970, 975, 1035,
- 1040, 1040, 1044, 1132, 1134, 1179,
- 1181, 1187, [1378](#), 1578, 1584, 1586,
- 1592, 1593, 1668, 1918, 2131, 2132,
- [2185](#), 2198, 2214, 2394, 2407, [2481](#),
- 2562, 2604, 2921, 2936, 2951, 3026,
- 3073, 3096, 3509, 3528, 3554, 3556,
- 3573, 3575, 3579, 3581, 3591, 3593,
- 3595, 3597, 3599, 3601, 3603, 3605,
- 3615, 3617, 3619, 3621, 3623, 3625,
- 3627, 3629, 3786, 3790, 3791, 3792,
- 4116, 4118, 4406, 4523, 4632, 4696,

4838, 7495, 7496, 7503, 7504, 7520,
 7552, 7572, 7575, 7578, 7687, 7719,
 7743, 7796, 7874, 7930, 7981, 7991,
 8477, 9620, 9662, 9663, 9677, 9679,
 9749, 9844, 9852, 9872, 9908, 10021,
 10027, 10055, 10060, 10091, 10122,
 10397, 10508, 10817, 10925, 10947,
 10953, 11153, 11188, 11258, 11301,
 11304, 11305, 11324, 11328, 11677,
 11694, 11846, 11977, 11994, 12702,
 12719, 14165, 14532, 14538, 14544,
 14796, 14806, 14821, 14873, 14940,
 15115, 15123, 15151, 15164, 15165,
 15173, 15193, 15206, 15207, 15215,
 15227, 15235, 15439, 15449, 15475,
 15477, 16882, 18120, 18122, 18124,
 18458, 18615, 21795, 22915, 23163,
 23277, 23307, 24176, 24178, 24807,
 25066, 25187, 25406, 25499, 25657,
 25669, 25724, 25939, 25942, 25950,
 26004, 26012, 26029, 26044, 26085,
 26268, 26273, 27648, 28661, 28664,
 28666, 29056, 29079, 29377, 29378,
 29379, 29465, 29486, 29695, 29696,
 30954, 30958, 30959, 31449, 31588,
 31589, 31597, 31602, 31607, 31612,
 31626, 31644, 31657, 31664, 31845
 \exp_stop_f:
 .. 35, 36, 102, 339, 399, 485, 499,
 639, 722, 734, 800, 801, 889, 917,
 948, 953, 2204, 2665, 2668, 2683,
 2691, 2746, 4443, 4452, 4501, 4502,
 4508, 4697, 4704, 4711, 4945, 4961,
 5000, 5001, 5007, 5019, 5035, 5036,
 5249, 5257, 5464, 5465, 5466, 5471,
 5472, 5514, 5648, 5683, 5684, 5882,
 5944, 5948, 5978, 5981, 5997, 6001,
 6022, 6100, 6102, 6122, 6123, 6140,
 6142, 6196, 6199, 6200, 6319, 6324,
 6460, 6465, 7507, 8067, 8081, 8091,
 8099, 8282, 8287, 8441, 9466, 10168,
 10170, 10238, 10240, 10244, 10246,
 10250, 10252, 10256, 10258, 10296,
 10297, 10304, 10305, 10306, 10307,
 10312, 10313, 10330, 10345, 10353,
 10414, 10428, 10429, 10435, 11053,
 11054, 11055, 11056, 12446, 12650,
 12897, 12911, 12923, 13620, 14009,
 14180, 15677, 15680, 15852, 15856,
 15885, 16090, 16205, 16220, 16245,
 16479, 16483, 16487, 16489, 16493,
 16497, 16505, 16510, 16523, 16530,
 16543, 16554, 16555, 16566, 16567,
 16576, 16579, 16590, 16632, 16696,
 16701, 16773, 16803, 16959, 17002,
 17053, 17073, 17100, 17114, 17149,
 17176, 17185, 17204, 17220, 17236,
 17254, 17315, 17334, 17350, 17365,
 17379, 17588, 17667, 17678, 17711,
 17722, 17960, 17964, 18258, 18264,
 18266, 18281, 18290, 18298, 18306,
 18307, 18504, 18624, 18630, 18645,
 18682, 18755, 18777, 18831, 18832,
 18840, 19177, 19195, 19248, 19252,
 19256, 19274, 19309, 19310, 19311,
 19312, 19313, 19339, 19351, 19367,
 19384, 19662, 19663, 19760, 19853,
 19888, 19901, 19906, 19915, 19917,
 20044, 20073, 20078, 20108, 20145,
 20185, 20261, 20311, 20357, 20358,
 20363, 20369, 20387, 20410, 20442,
 20445, 20492, 20512, 20518, 20533,
 20545, 20583, 20604, 20644, 20659,
 20674, 20689, 20704, 20719, 20747,
 20791, 21057, 21067, 21097, 21249,
 21251, 21288, 21300, 21332, 21373,
 21382, 21397, 21416, 21449, 21462,
 21546, 21600, 21648, 21651, 21674,
 21883, 21887, 21894, 21895, 21950,
 21951, 21952, 21961, 21971, 21989,
 22058, 22061, 22064, 22079, 22132,
 22136, 22178, 22250, 22257, 22675,
 22934, 23101, 23110, 23111, 23145,
 23215, 23223, 23246, 23248, 23249,
 23253, 23270, 23321, 23324, 23340,
 23346, 23418, 23421, 23612, 23613,
 23614, 23620, 23640, 23892, 23912,
 23913, 23917, 23921, 23922, 23925,
 23926, 23934, 23935, 23938, 23942,
 23943, 23946, 24005, 24100, 24344,
 24349, 24363, 24364, 24377, 24448,
 24449, 24488, 24588, 24765, 24924,
 25192, 25210, 25236, 25246, 25298,
 25311, 25322, 25338, 25389, 25606,
 25734, 25738, 25802, 25865, 25878,
 25898, 25903, 25909, 25955, 26008,
 26025, 26048, 26266, 26271, 26292,
 26370, 26382, 26387, 27935, 29106,
 30002, 30005, 30006, 30009, 31697,
 31705, 31744, 31745, 31746, 31747
 exp internal commands:
 __exp_arg_last_unbraced:nn .. 2378
 __exp_arg_next:Nnn 2185, 2192
 __exp_arg_next:nnn
 340, 2185, 2194, 2202, 2206, 2219, 2225
 __exp_e:N 2520, 2550
 __exp_e:nn 342, 348, 2274,
 2398, 2516, 2536, 2541, 2549, 2577,

- 2579, 2624, 2625, 2630, 2697, 2715
 __exp_e:Nnn 350, 2550
 __exp_e_end:nn 348, 2516, 2649
 __exp_e_expandable:Nnn ... 350, 2550
 __exp_e_group:n 2523, 2537
 __exp_e_if_toks_register:N .. 2761
 __exp_e_if_toks_register:NTF ...
 2712, 2761
 __exp_e_noexpand:Nnn 2570, 2605, 2627
 __exp_e_primitive:Nnn ... 2572, 2580
 __exp_e_primitive_aux:NNnn .. 2580
 __exp_e_primitive_aux:NNw ... 2580
 __exp_e_primitive_other:NNnn . 2580
 __exp_e_primitive_other_-
 aux:nNNnn 2580
 __exp_e_protected:Nnn 350, 2550
 __exp_e_put:nn
 349, 351, 353, 2537, 2630, 2642, 2729
 __exp_e_put:nnn 354, 2537, 2735
 __exp_e_space:nn 2527, 2535
 __exp_e_the:N 2693
 __exp_e_the:Nnn 2571, 2606, 2693
 __exp_e_the_errhelp: 2761
 __exp_e_the_everycr: 2761
 __exp_e_the_everydisplay: ... 2761
 __exp_e_the_everyeof: 2761
 __exp_e_the_everyhbox: 2761
 __exp_e_the_everyjob: 2761
 __exp_e_the_everymath: 2761
 __exp_e_the_everypar: 2761
 __exp_e_the_everyvbox: 2761
 __exp_e_the_output: 2761
 __exp_e_the_pdfpageattr: 2761
 __exp_e_the_pdfpagesresources: 2761
 __exp_e_the_pdfpagesattr: ... 2761
 __exp_e_the_pdfpkmode: 2761
 __exp_e_the_toks:N 354, 2733
 __exp_e_the_toks:n . 354, 2709, 2733
 __exp_e_the_toks:wnn 354, 2708, 2733
 __exp_e_the_toks_reg:N 2693
 __exp_e_the_XeTeXinterchartoks:
 2761
 __exp_e_unexpanded:N 2632
 __exp_e_unexpanded:nN 352, 2632
 __exp_e_unexpanded:nn 2632
 __exp_e_unexpanded:Nnn
 2569, 2604, 2632
 __exp_eval_error_msg:w 2229
 __exp_eval_register:N
 2220, 2226, 2229,
 2282, 2287, 2293, 2299, 2327, 2333,
 2345, 2346, 2353, 2384, 2389, 2412,
 2414, 2429, 2443, 2452, 2497, 2502
 \l__exp_internal_tl
 314, 1450, 1454, 1455,
 2185, 2185, 2213, 2215, 2407, 2408
 __exp_last_two_unbraced:nnN . 2476
 \expandafter 4, 20, 21, 24, 25, 29, 30, 40,
 41, 64, 66, 67, 68, 79, 86, 107, 115, 267
 \expanded 314, 815
 \expandglyphsinfont 910
 \ExplFileDate 7, 13806, 13821, 13835, 13839
 \ExplFileDescription 7, 13805, 13818
 \ExplFileExtension .. 13808, 13823, 13832
 \ExplFileName 7, 13807, 13822, 13831
 \ExplFileVersion . 7, 13809, 13824, 13833
 \explicitdiscretionary 816
 \explicithyphenpenalty 814
 \ExplLoaderFileDate 31788, 31794
 \ExplSyntaxOff 4,
 7, 7, 117, 120, 147, 161, 277, 278, 304
 \ExplSyntaxOn 4,
 7, 7, 120, 143, 277, 278, 304, 381, 564
- ## F
- fact 214
 false 219
 \fam 268
 \fi 18, 23, 32, 44, 45, 46, 71, 74,
 76, 77, 78, 81, 82, 93, 101, 113, 114, 269
 fi commands:
 \fi: 23, 102, 102,
 103, 107, 114, 114, 166, 185, 250,
 250, 250, 317, 318, 320, 323, 324,
 349, 381, 386, 388, 419, 421, 424,
 505, 514, 534, 568, 647, 734, 759,
 775, 806, 953, 1359, 1406, 1574,
 1582, 1590, 1598, 1618, 1623, 1636,
 1644, 1652, 1654, 1677, 1682, 1689,
 1716, 1721, 1747, 1748, 1756, 1762,
 1775, 1776, 1784, 1790, 1910, 1931,
 1941, 1955, 2012, 2073, 2170, 2180,
 2234, 2237, 2244, 2245, 2511, 2518,
 2528, 2541, 2554, 2559, 2562, 2563,
 2564, 2565, 2573, 2582, 2598, 2662,
 2690, 2696, 2705, 2710, 2716, 2719,
 2723, 2731, 2741, 2750, 2754, 2758,
 2826, 2842, 2849, 2858, 2872, 2873,
 2878, 2879, 2880, 2898, 2899, 2900,
 2901, 2902, 2903, 2904, 2905, 2906,
 2914, 2933, 2935, 2965, 2966, 2967,
 3014, 3044, 3116, 3127, 3137, 3235,
 3243, 3267, 3281, 3289, 3300, 3310,
 3330, 3360, 3361, 3433, 3456, 3475,
 3478, 3659, 3666, 3802, 3803, 3829,
 3839, 3850, 3865, 3873, 3889, 3903,
 3913, 3917, 3947, 3962, 4191, 4244,

4249, 4258, 4260, 4288, 4309, 4327,
4336, 4345, 4351, 4358, 4363, 4374,
4378, 4386, 4445, 4457, 4506, 4512,
4513, 4698, 4705, 4711, 4806, 4874,
4878, 4879, 4897, 4950, 4963, 5005,
5011, 5012, 5023, 5036, 5037, 5058,
5096, 5122, 5226, 5234, 5242, 5253,
5270, 5272, 5416, 5468, 5469, 5474,
5477, 5478, 5518, 5571, 5650, 5651,
5686, 5687, 5689, 5697, 5884, 5917,
5953, 5954, 5985, 5986, 6006, 6007,
6025, 6108, 6112, 6122, 6132, 6148,
6152, 6155, 6160, 6162, 6205, 6209,
6210, 6301, 6306, 6317, 6323, 6335,
6338, 6340, 6344, 6458, 6472, 6473,
7548, 7551, 7588, 7649, 7666, 7676,
7730, 7735, 8074, 8075, 8084, 8107,
8124, 8125, 8127, 8144, 8145, 8186,
8240, 8248, 8275, 8283, 8289, 8312,
8350, 8358, 8443, 8658, 8691, 8739,
8744, 8860, 8885, 8911, 8920, 8980,
9012, 9030, 9052, 9072, 9088, 9098,
9114, 9124, 9214, 9216, 9218, 9220,
9222, 9224, 9358, 9366, 9726, 9741,
9764, 9778, 10301, 10304, 10305,
10306, 10307, 10312, 10313, 10318,
10319, 10320, 10357, 10367, 10409,
10417, 10419, 10463, 10464, 10467,
10603, 10604, 10605, 10606, 10607,
10608, 10609, 10670, 10675, 10680,
10685, 10692, 10699, 10704, 10709,
10714, 10719, 10724, 10729, 10734,
10739, 10761, 10772, 10773, 10822,
10823, 10874, 10883, 10892, 10900,
10937, 10975, 10989, 10998, 11008,
11053, 11054, 11055, 11065, 11072,
11074, 11362, 12512, 12593, 12611,
12849, 12890, 12899, 12920, 12930,
12934, 12941, 12949, 13216, 13229,
13604, 13613, 13624, 13959, 13982,
13991, 14008, 14012, 14026, 14029,
14191, 14192, 15682, 15683, 15718,
15726, 15792, 15939, 15955, 15959,
15971, 15981, 16076, 16129, 16132,
16133, 16138, 16152, 16190, 16191,
16192, 16193, 16194, 16195, 16196,
16197, 16198, 16199, 16200, 16201,
16214, 16216, 16227, 16230, 16244,
16249, 16253, 16394, 16485, 16486,
16495, 16496, 16507, 16508, 16509,
16520, 16521, 16522, 16529, 16540,
16541, 16542, 16552, 16553, 16557,
16558, 16566, 16569, 16570, 16578,
16589, 16609, 16632, 16667, 16684,
16703, 16704, 16713, 16719, 16739,
16740, 16768, 16777, 16794, 16801,
16809, 16810, 16911, 16912, 16913,
16916, 16919, 16958, 16974, 17000,
17001, 17008, 17016, 17045, 17046,
17049, 17051, 17052, 17057, 17067,
17070, 17072, 17077, 17108, 17121,
17126, 17132, 17135, 17136, 17170,
17171, 17198, 17199, 17212, 17215,
17226, 17249, 17268, 17278, 17294,
17303, 17309, 17315, 17319, 17324,
17330, 17345, 17356, 17375, 17385,
17387, 17393, 17414, 17442, 17465,
17498, 17500, 17623, 17673, 17677,
17687, 17688, 17704, 17717, 17721,
17731, 17732, 17752, 17773, 17776,
17806, 17838, 17854, 17874, 17915,
17927, 17940, 17942, 17962, 17963,
17970, 17988, 18027, 18037, 18188,
18204, 18215, 18244, 18245, 18246,
18253, 18255, 18256, 18262, 18263,
18266, 18293, 18301, 18302, 18310,
18311, 18313, 18314, 18485, 18498,
18508, 18509, 18514, 18515, 18516,
18517, 18518, 18519, 18526, 18536,
18543, 18554, 18555, 18566, 18583,
18628, 18629, 18636, 18649, 18664,
18674, 18688, 18718, 18727, 18761,
18783, 18801, 18818, 18835, 18836,
18838, 18839, 18844, 18859, 18892,
18921, 18922, 18923, 18924, 18925,
18938, 18982, 19055, 19122, 19124,
19125, 19135, 19164, 19167, 19168,
19179, 19199, 19262, 19263, 19264,
19276, 19315, 19316, 19317, 19318,
19324, 19327, 19329, 19339, 19357,
19372, 19384, 19391, 19638, 19642,
19644, 19648, 19655, 19656, 19666,
19667, 19670, 19762, 19832, 19843,
19855, 19887, 19894, 19905, 19921,
19928, 19989, 20046, 20056, 20058,
20068, 20086, 20087, 20119, 20122,
20131, 20133, 20135, 20149, 20163,
20187, 20271, 20279, 20310, 20318,
20324, 20335, 20338, 20341, 20350,
20360, 20362, 20368, 20375, 20378,
20387, 20395, 20416, 20449, 20450,
20477, 20479, 20497, 20498, 20517,
20528, 20537, 20540, 20551, 20554,
20557, 20575, 20585, 20596, 20598,
20607, 20654, 20669, 20684, 20699,
20714, 20729, 20732, 20734, 20751,
20796, 21103, 21139, 21140, 21150,
21191, 21192, 21216, 21243, 21244,

21247, 21249, 21250, 21255, 21267,
 21286, 21291, 21299, 21302, 21334,
 21344, 21345, 21355, 21377, 21392,
 21410, 21418, 21421, 21449, 21457,
 21473, 21545, 21563, 21599, 21617,
 21650, 21673, 21679, 21752, 21753,
 21884, 21885, 21894, 21901, 21906,
 21916, 21926, 21951, 21954, 21967,
 21999, 22007, 22008, 22036, 22058,
 22059, 22060, 22063, 22068, 22088,
 22089, 22141, 22142, 22183, 22191,
 22244, 22250, 22263, 22677, 22688,
 22689, 22733, 22764, 22806, 22817,
 22826, 22835, 22890, 22900, 22910,
 23022, 23024, 23078, 23082, 23086,
 23113, 23124, 23142, 23143, 23144,
 23151, 23167, 23173, 23176, 23184,
 23194, 23209, 23217, 23225, 23236,
 23252, 23272, 23285, 23307, 23325,
 23333, 23335, 23338, 23345, 23350,
 23427, 23428, 23571, 23578, 23579,
 23585, 23588, 23591, 23598, 23599,
 23602, 23606, 23607, 23617, 23618,
 23623, 23624, 23636, 23644, 23653,
 23654, 23682, 23843, 23854, 23906,
 23908, 23915, 23918, 23919, 23923,
 23927, 23928, 23929, 23930, 23939,
 23940, 23944, 23947, 23948, 23949,
 23985, 23988, 24009, 24012, 24020,
 24031, 24032, 24043, 24044, 24052,
 24064, 24065, 24075, 24076, 24089,
 24108, 24109, 24117, 24118, 24169,
 24179, 24205, 24219, 24223, 24286,
 24334, 24337, 24338, 24353, 24356,
 24379, 24446, 24447, 24452, 24479,
 24480, 24491, 24495, 24529, 24534,
 24542, 24577, 24584, 24589, 24599,
 24611, 24637, 24700, 24729, 24768,
 24775, 24786, 24860, 24877, 24881,
 24895, 24929, 25141, 25180, 25196,
 25220, 25241, 25250, 25307, 25314,
 25334, 25352, 25363, 25365, 25395,
 25398, 25423, 25535, 25564, 25587,
 25588, 25609, 25638, 25664, 25737,
 25795, 25807, 25870, 25884, 25885,
 25906, 25917, 25943, 25960, 26010,
 26012, 26027, 26029, 26050, 26152,
 26211, 26228, 26229, 26268, 26269,
 26273, 26274, 26296, 26301, 26338,
 26376, 26384, 26385, 26389, 26390,
 26789, 26791, 26797, 28919, 28920,
 28928, 28942, 28946, 28947, 28963,
 29042, 29046, 29057, 29078, 29082,
 29098, 29110, 29142, 29143, 29144,

29145, 29146, 29147, 29148, 29296,
 29302, 30013, 30014, 30017, 30018,
 31244, 31364, 31616, 31623, 31628,
 31654, 31660, 31664, 31679, 31700,
 31708, 31744, 31745, 31746, 31752

file commands:

\file_add_path:nN 31965
 \file_compare_timestamp:nNn ... 170
 \file_compare_timestamp:nNnTF ...
 170, 13588
 \file_compare_timestamp_p:nNn ...
 170, 13588
 \g_file_curr_dir_str
 166, 13055, 13694, 13700, 13713
 \g_file_curr_ext_str
 166, 13055, 13696, 13702, 13715
 \g_file_curr_name_str
 166, 9497, 11498,
 13055, 13695, 13701, 13714, 31978
 \g_file_current_name_tl 31977
 \file_full_name:n
 167, 13247, 13342, 13409,
 13426, 13433, 13494, 13592, 13593
 \file_get:nnN
 . 167, 13198, 32141, 32143, 32145,
 32149, 32154, 32158, 32165, 32169
 \file_get:nnNTF ... 167, 13198, 13200
 \file_get_full_name:nN
 167, 304, 13333, 31966
 \file_get_full_name:nNTF ... 167,
 12437, 13205, 13333, 13335, 13347,
 13348, 13650, 13656, 13661, 13673
 \file_get_hex_dump:nN ... 168, 13508
 \file_get_hex_dump:nnnN .. 168, 13554
 \file_get_hex_dump:nnnNTF
 168, 13554, 13556
 \file_get_hex_dump:nNTF
 168, 13508, 13509
 \file_get_md5five_hash:nN
 169, 13510, 13518
 \file_get_md5five_hash:nN\file_-
 get_size:nN 13508
 \file_get_md5five_hash:nN\file_-
 get_size:nNTF 13508
 \file_get_md5five_hash:nNTF 169, 13511
 \file_get_size:nN 169
 \file_get_size:nNTF 169, 13513
 \file_get_timestamp:nN ... 169, 13508
 \file_get_timestamp:nNTF
 169, 13508, 13515
 \file_hex_dump:n 168, 168, 13430
 \file_hex_dump:nnn
 168, 168, 13430, 13563

- \file_if_exist:nTF
 - . [167](#), [167](#), [167](#), [170](#), [5435](#), [13648](#),
 - [13870](#), [13872](#), [13876](#), [31968](#), [31970](#)
- \file_if_exist_input:n ... [170](#), [13654](#)
- \file_if_exist_input:nTF
 - [170](#), [13654](#), [31967](#), [31969](#)
- \file_input:n [170](#), [170](#), [170](#),
- [170](#), [5439](#), [13671](#), [31968](#), [31970](#), [32103](#)
- \file_input_stop: [170](#), [13665](#)
- \file_list: [31971](#)
- \file_log_list: ... [170](#), [13774](#), [31972](#)
- \file_md5five_hash:n . [169](#), [169](#), [13400](#)
- \file_parse_full_name:n
 - [168](#), [659](#), [13717](#)
- \file_parse_full_name:nnNN
 - .. [168](#), [168](#), [168](#), [13374](#), [13698](#), [13762](#)
- \file_parse_full_name_apply:nN ..
 - [168](#), [659](#), [13717](#), [13764](#)
- \file_path_include:n [170](#), [31973](#)
- \file_path_remove:n [31975](#)
- \l_file_search_path_seq
 - [167](#), [167](#), [168](#), [169](#), [169](#),
 - [169](#), [13089](#), [13258](#), [13359](#), [31974](#), [31976](#)
- \file_show_list: [170](#), [13774](#)
- \file_size:n [169](#), [169](#), [13400](#)
- \file_timestamp:n ... [169](#), [169](#), [13400](#)
- file internal commands:
 - \l_file_base_name_tl
 - [13084](#), [13356](#), [13392](#)
 - __file_compare_timestamp:nnN . [13588](#)
 - __file_const:nn [13884](#)
 - __file_details:nn [13400](#)
 - __file_details_aux:nn . [13400](#), [13447](#)
 - \l_file_dir_str
 - [13086](#), [13375](#), [13699](#), [13700](#)
 - __file_ext_check:n
 - [13267](#), [13278](#), [13285](#)
 - __file_ext_check:nn .. [13300](#), [13305](#)
 - __file_ext_check:nnw . [13291](#), [13296](#)
 - __file_ext_check:nw
 - [13286](#), [13287](#), [13294](#)
 - \l_file_ext_str
 - .. [13086](#), [13375](#), [13376](#), [13699](#), [13702](#)
 - __file_full_name:n [13247](#)
 - __file_full_name_assign:nnnNNN .
 - [13765](#), [13767](#)
 - __file_full_name_aux:nN [13247](#)
 - __file_full_name_aux:Nnn [13247](#)
 - \l_file_full_name_tl
 - [13084](#), [13205](#), [13208](#),
 - [13366](#), [13368](#), [13374](#), [13379](#), [13381](#),
 - [13384](#), [13391](#), [13393](#), [13650](#), [13656](#),
 - [13657](#), [13661](#), [13662](#), [13673](#), [13674](#)
 - __file_get_aux:nnN [13198](#)
 - __file_get_details:nnN [13508](#)
 - __file_get_do:Nw [13198](#)
 - __file_get_full_name_search:nN .
 - [13333](#)
 - __file_hex_dump:n [13430](#)
 - __file_hex_dump_auxi:nnn [13430](#)
 - __file_hex_dump_auxii:nnnn . [13430](#)
 - __file_hex_dump_auxiii:nnnn . [13430](#)
 - __file_hex_dump_auxiiv:nnn . [13430](#)
 - __file_hex_dump_auxiv:nnn
 - [13464](#), [13466](#), [13471](#), [13480](#)
 - __file_id_info_auxi:w [13803](#)
 - __file_id_info_auxii:w .. [662](#), [13803](#)
 - __file_id_info_auxiii:w [13803](#)
 - __file_if_recursion_tail_-
 - break:NN [13096](#)
 - __file_if_recursion_tail_stop:N
 - [13096](#), [13122](#)
 - __file_if_recursion_tail_stop_-
 - do:Nn [13096](#)
 - __file_if_recursion_tail_stop_-
 - do:nn [13097](#), [13162](#)
 - __file_input:n . [13657](#), [13662](#), [13671](#)
 - __file_input_pop: [13671](#)
 - __file_input_pop:nnn [13671](#)
 - __file_input_push:n [13671](#)
 - \g_file_internal_ior [13370](#), [13378](#),
 - [13380](#), [13383](#), [13393](#), [13394](#), [13396](#)
 - \l_file_internal_tl
 - [13054](#), [13707](#), [13708](#)
 - __file_list:N [13774](#)
 - __file_list_aux:n [13774](#)
 - \c_file_marker_tl
 - [647](#), [13197](#), [13220](#), [13233](#)
 - __file_md5five_hash:n [13400](#)
 - __file_name_cleanup:w [13247](#)
 - __file_name_end: [13247](#)
 - __file_name_ext_check:n [13247](#)
 - __file_name_ext_check:nn [13247](#)
 - __file_name_ext_check:nnw ... [13247](#)
 - __file_name_ext_check:nw [13247](#)
 - \l_file_name_str
 - [13086](#), [13375](#), [13699](#), [13701](#)
 - __file_parse_full_name_area:nw .
 - [659](#), [13727](#)
 - __file_parse_full_name_auxi:nN .
 - [13724](#), [13727](#)
 - __file_parse_full_name_base:nw .
 - [659](#), [13735](#), [13738](#)
 - __file_parse_full_name_tidy:nnnN
 - [659](#), [13745](#), [13746](#), [13748](#), [13752](#)
 - __file_quark_if_nil:nTF
 - .. [13093](#), [13179](#), [13193](#), [13289](#), [13298](#)
 - __file_quark_if_nil_p:n [13093](#)

- \g_file_record_seq ... [658](#), [660](#),
[661](#), [13083](#), [13681](#), [13784](#), [13798](#), [13799](#)
- _file_size:n
.. [13238](#), [13256](#), [13274](#), [13307](#), [13311](#)
- \g_file_stack_seq
..... [658](#), [13058](#), [13692](#), [13707](#)
- _file_str_cmp:nn [13568](#), [13617](#)
- _file_str_escape:n [13568](#)
- _file_timestamp:n [13588](#)
- _file_tmp:w
.. [13060](#), [13064](#), [13068](#), [13074](#), [13080](#)
- \l_file_tmp_seq
..... [13090](#), [13778](#), [13781](#),
[13784](#), [13785](#), [13787](#), [13795](#), [13800](#)
- \filedump [774](#)
- \filemoddate [775](#)
- \filesize [776](#)
- \finalhyphendemerits [270](#)
- \firstmark [271](#)
- \firstmarks [528](#)
- \firstvalidlanguage [817](#)
- flag commands:
- \flag_clear:n
 [104](#), [104](#), [5531](#), [5559](#), [5620](#),
 [5662](#), [5748](#), [5796](#), [5797](#), [5849](#), [5850](#),
 [6084](#), [6085](#), [6086](#), [6087](#), [6088](#), [6189](#),
 [6284](#), [6285](#), [6286](#), [6287](#), [6442](#), [6443](#),
 [6444](#), [8876](#), [8889](#), [24798](#), [26234](#), [26235](#)
- \flag_clear_new:n
 . [104](#), [450](#), [6031](#), [6032](#), [6033](#), [6034](#),
 [6212](#), [6213](#), [6214](#), [6392](#), [6393](#), [8888](#)
- \flag_height:n .. [105](#), [5359](#), [8897](#),
 [8913](#), [8927](#), [26244](#), [26245](#), [26251](#), [26252](#)
- \flag_if_exist:n [105](#)
- \flag_if_exist:nTF .. [105](#), [8889](#), [8900](#)
- \flag_if_exist_p:n [105](#), [8900](#)
- \flag_if_raised:n [105](#)
- \flag_if_raised:nTF [105](#), [5352](#), [5357](#),
 [5359](#), [6058](#), [6064](#), [6069](#), [6076](#), [6246](#),
 [6251](#), [6256](#), [6407](#), [6414](#), [8905](#), [24806](#)
- \flag_if_raised_p:n [105](#), [8905](#)
- \flag_log:n [104](#), [8890](#)
- \flag_new:n [104](#), [104](#), [450](#), [521](#), [5221](#),
 [5222](#), [8871](#), [8889](#), [16290](#), [16291](#),
 [16292](#), [16293](#), [24788](#), [26138](#), [26139](#)
- \flag_raise:n [105](#), [5517](#),
 [5567](#), [5680](#), [5694](#), [5768](#), [5781](#), [5819](#),
 [5824](#), [5905](#), [6105](#), [6106](#), [6129](#), [6130](#),
 [6143](#), [6144](#), [6163](#), [6164](#), [6170](#), [6171](#),
 [6201](#), [6351](#), [6352](#), [6461](#), [6462](#), [6466](#),
 [6467](#), [6481](#), [6482](#), [8924](#), [26267](#), [26272](#)
- \flag_raise_if_clear:n
 . [267](#), [16324](#), [16333](#), [16341](#), [16358](#),
 [16367](#), [16398](#), [24825](#), [24847](#), [31359](#)
- \flag_show:n [104](#), [8890](#)
- flag fp commands:
- flag_fp_division_by_zero . [210](#), [16290](#)
- flag_fp_invalid_operation [210](#), [16290](#)
- flag_fp_overflow [210](#), [16290](#)
- flag_fp_underflow [210](#), [16290](#)
- flag internal commands:
- _flag_clear:wn [8876](#)
- _flag_height_end:wn [8913](#)
- _flag_height_loop:wn [8913](#)
- _flag_show:Nn [8890](#)
- \floatingpenalty [272](#)
- floor [215](#)
- \fmtname [106](#)
- \font [273](#)
- \fontcharhp [529](#)
- \fontcharht [530](#)
- \fontcharic [531](#)
- \fontcharwd [532](#)
- \fontdimen [274](#)
- \fontencoding [30819](#)
- \fontfamily [30820](#)
- \fontid [818](#)
- \fontname [275](#)
- \fontseries [30821](#)
- \fontshape [30822](#)
- \fontsize [30825](#)
- \footnotesize [30861](#)
- \forcecjktoken [1168](#)
- \formatname [819](#)
- fp commands:
- \c_e_fp [209](#), [211](#), [18165](#)
- \fp_abs:n [214](#), [219](#), [912](#), [21774](#), [27200](#),
 [27302](#), [27304](#), [27306](#), [28129](#), [28131](#)
- \fp_add:Nn [203](#), [912](#), [912](#), [18142](#)
- \fp_compare:nNnTF
 [205](#), [206](#), [206](#), [206](#), [207](#), [207](#),
 [18206](#), [18347](#), [18353](#), [18358](#), [18366](#),
 [18427](#), [18433](#), [27057](#), [27059](#), [27064](#),
 [27333](#), [27348](#), [27357](#), [27869](#), [28103](#)
- \fp_compare:nTF
 [205](#), [206](#), [207](#), [207](#), [207](#), [207](#),
 [213](#), [18190](#), [18319](#), [18325](#), [18330](#), [18338](#)
- \fp_compare_p:n [206](#), [18190](#)
- \fp_compare_p:nNn [205](#), [18206](#)
- \fp_const:Nn
 [202](#), [18119](#), [18165](#), [18166](#), [18167](#), [18168](#)
- \fp_do_until:nn [207](#), [18316](#)
- \fp_do_until:nNnn [206](#), [18344](#)
- \fp_do_while:nn [207](#), [18316](#)
- \fp_do_while:nNnn [206](#), [18344](#)
- \fp_eval:n
 [203](#), [204](#), [206](#), [213](#), [213](#), [213](#),
 [213](#), [213](#), [214](#), [214](#), [214](#), [214](#), [214](#), [214](#),

- 214, 214, 215, 215, 215, 215, 216,
 216, 216, 216, 217, 217, 217, 218,
 218, 219, 219, 796, 21769, 31507, 31526
 \fp_format:nn 220
 \fp_gadd:Nn 203, 18142
 .fp_gset:N 189, 15018
 \fp_gset:Nn .. 203, 18119, 18143, 18145
 \fp_gset_eq:NN 203, 18128, 18133
 \fp_gsub:Nn 203, 18142
 \fp_gzero:N 202, 18132, 18139
 \fp_gzero_new:N 203, 18136
 \fp_if_exist:NTF
 205, 18137, 18139, 18180
 \fp_if_exist_p:N 205, 18180
 \fp_if_nan:n 266
 \fp_if_nan:nTF 220, 266, 18182
 \fp_if_nan_p:n 266, 18182
 \fp_log:N 210, 18152
 \fp_log:n 210, 18161
 \fp_max:nn 219, 21776
 \fp_min:nn 219, 21776
 \fp_new:N
 .. 202, 203, 18116, 18137, 18139,
 18169, 18170, 18171, 18172, 27023,
 27024, 27025, 27151, 27152, 27401,
 27402, 27896, 27897, 28063, 28064
 .fp_set:N 189, 15018
 \fp_set:Nn 203, 18119, 18142, 18144,
 27045, 27046, 27047, 27170, 27172,
 27213, 27233, 27253, 27270, 27272,
 27290, 27291, 27331, 27332, 27914,
 27915, 28083, 28085, 28123, 28124
 \fp_set_eq:NN .. 203, 18128, 18132,
 27218, 27238, 27255, 27334, 27335
 \fp_show:N 210, 18152
 \fp_show:n 210, 18161
 \fp_sign:n 204, 21772
 \fp_step_function:nnnN
 208, 18372, 18464
 \fp_step_inline:nnnn 208, 18442
 \fp_step_variable:nnnNn .. 208, 18442
 \fp_sub:Nn 203, 18142
 \fp_to_decimal:N
 204, 205, 16283, 21576, 21607, 21769
 \fp_to_decimal:n
 .. 203, 204, 204, 205, 205, 21576,
 21771, 21773, 21775, 21777, 21779
 \fp_to_dim:N 204, 910, 21699
 \fp_to_dim:n 204, 209, 21699, 27089,
 27100, 27200, 27824, 27846, 27874,
 27888, 28000, 28008, 28139, 28141
 \fp_to_int:N 204, 21715
 \fp_to_int:n 204, 21715
 \fp_to_scientific:N
 204, 21522, 21553, 21560
 \fp_to_scientific:n . 204, 205, 21522
 \fp_to_tl:N
 205, 222, 16284, 18159, 21655
 \fp_to_tl:n 205,
 15896, 16323, 16332, 16357, 16366,
 16395, 18000, 18015, 18162, 18164,
 18399, 18400, 18419, 18430, 21655
 \fp_trap:nn 210, 210,
 738, 16294, 16409, 16410, 16411, 16412
 \fp_until_do:nn 207, 18316
 \fp_until_do:nnnn 207, 18344
 \fp_use:N 205, 222, 21769
 \fp_while_do:nn 207, 18316
 \fp_while_do:nnnn 207, 18344
 \fp_zero:N 202, 203, 18132, 18137
 \fp_zero_new:N 203, 18136
 \c_inf_fp 209,
 218, 15910, 17508, 18935, 19017,
 19355, 20115, 20138, 20340, 20343,
 20347, 20371, 20573, 20736, 22261
 \c_nan_fp 218, 741, 765, 15910,
 16334, 16342, 16414, 16620, 16639,
 16645, 16668, 16835, 16843, 16851,
 16929, 16986, 17025, 17420, 17497,
 17509, 18002, 18017, 18423, 20314,
 21813, 21859, 22174, 22233, 22259
 \c_one_fp 208, 793,
 896, 17512, 17945, 17966, 18165,
 18523, 19376, 20109, 20309, 20361,
 20546, 20660, 20690, 21239, 21875
 \c_pi_fp .. 209, 218, 775, 17510, 18167
 \g_tmpa_fp 209, 18169
 \l_tmpa_fp 209, 18169
 \g_tmpb_fp 209, 18169
 \l_tmpb_fp 209, 18169
 \c_zero_fp 208, 796, 811, 923, 15910,
 15964, 17513, 17957, 17969, 18117,
 18132, 18133, 18525, 18528, 18764,
 18931, 20118, 20139, 20337, 20374,
 21454, 21560, 21744, 22258, 27057,
 27059, 27064, 27348, 27357, 28103
 fp internal commands:
 __fp_&_o:ww 798, 807, 18529
 __fp_&_tuple_o:ww 18529
 __fp_*_o:ww 18896
 __fp_*_tuple_o:ww 19402
 __fp+_o:ww 809, 810, 838, 18617
 __fp-_o:ww 809, 810, 18612
 __fp/_o:ww 818, 860, 19008
 __fp^_o:ww 20305
 __fp_acos_o:w 901, 903, 21395
 __fp_acot_o:Nw . 20635, 20637, 21227

- __fp_acotii_o:Nww [21237](#), [21240](#)
- __fp_acotii_o:ww [896](#)
- __fp_acsc_normal_o:NnwNnw
 [903](#), [21453](#), [21468](#), [21476](#)
- __fp_acsc_o:w [21447](#)
- __fp_add:NNNn [18142](#)
- __fp_add_big_i:wNww [812](#)
- __fp_add_big_i_o:wNww
 [809](#), [812](#), [18684](#), [18691](#)
- __fp_add_big_ii:wNww [812](#)
- __fp_add_big_ii_o:wNww [18687](#), [18691](#)
- __fp_add_inf_o:Nww [18633](#), [18653](#)
- __fp_add_normal_o:Nww
 [811](#), [18632](#), [18668](#)
- __fp_add_npos_o:NnwNnw
 [812](#), [18671](#), [18677](#)
- __fp_add_return_ii_o:Nww
 [18635](#), [18641](#), [18646](#)
- __fp_add_significand_carry_-
 o:wwwNN [813](#), [18724](#), [18739](#)
- __fp_add_significand_no_carry_-
 o:wwwNN [813](#), [18726](#), [18729](#)
- __fp_add_significand_o:NnnwnnnN
 [812](#), [813](#), [18694](#), [18702](#), [18707](#)
- __fp_add_significand_pack:NNNNNN
 [18707](#)
- __fp_add_significand_test_o:N . [18707](#)
- __fp_add_zeros_o:Nww [18631](#), [18643](#)
- __fp_and_return:wNw [18529](#)
- __fp_array_bounds:NNnTF
 [22130](#), [22161](#), [22231](#)
- __fp_array_bounds_error:NNn . [22130](#)
- __fp_array_count:n [16013](#),
 [16604](#), [18273](#), [18274](#), [19415](#), [21495](#)
- __fp_array_gset:NNNNww [22149](#)
- __fp_array_gset:w [22149](#)
- __fp_array_gset_normal:w [22149](#)
- __fp_array_gset_recover:Nw . . [22149](#)
- __fp_array_gset_special:nnNNN . .
 [22149](#), [22206](#)
- __fp_array_gzero:N [922](#)
- __fp_array_if_all_fp:nTF
 [16025](#), [17995](#)
- __fp_array_if_all_fp_loop:w . [16025](#)
- \g__fp_array_int
 [22095](#), [22102](#), [22104](#), [22116](#)
- __fp_array_item:N [22213](#)
- __fp_array_item:NNNnN [22213](#)
- __fp_array_item:NwN [22213](#)
- __fp_array_item:w [22213](#)
- __fp_array_item_normal:w [22213](#)
- __fp_array_item_special:w . . . [22213](#)
- \l__fp_array_loop_int
 [22096](#), [22202](#), [22205](#), [22208](#)
- __fp_array_new:nNNN [22097](#)
- __fp_array_new:nNNNN [22106](#), [22110](#)
- __fp_array_to_clist:n
 [16672](#), [21780](#), [21899](#)
- __fp_array_to_clist_loop:Nw . [21780](#)
- __fp_asec_o:w [21460](#)
- __fp_asin_auxi_o:NnNww
 [901](#), [902](#), [903](#), [21425](#), [21428](#), [21487](#)
- __fp_asin_isqrt:wn [21428](#)
- __fp_asin_normal_o:NnwNnnnw . . .
 [21386](#), [21402](#), [21413](#)
- __fp_asin_o:w [21380](#)
- __fp_atan_auxi:ww [898](#), [21305](#), [21319](#)
- __fp_atan_auxii:w [21319](#)
- __fp_atan_combine_aux:ww [21346](#)
- __fp_atan_combine_o:NwwwwwN . . .
 [897](#), [898](#), [21264](#), [21281](#), [21346](#)
- __fp_atan_default:w [793](#), [896](#), [21227](#)
- __fp_atan_div:wnwnw
 [898](#), [21292](#), [21294](#)
- __fp_atan_inf_o:NNNw [896](#), [21252](#),
 [21253](#), [21254](#), [21262](#), [21398](#), [21471](#)
- __fp_atan_near:wwn [21294](#)
- __fp_atan_near_aux:wwn [21294](#)
- __fp_atan_normal_o:NnwNnw
 [896](#), [21256](#), [21272](#)
- __fp_atan_o:Nw [20639](#), [20641](#), [21227](#)
- __fp_atan_Taylor_break:w [21330](#)
- __fp_atan_Taylor_loop:www
 [899](#), [21325](#), [21330](#)
- __fp_atan_test_o:NwwNwN
 [902](#), [21275](#), [21279](#), [21435](#)
- __fp_atanii_o:Nww [21231](#), [21240](#)
- __fp_basics_pack_high:NNNNNw . . .
 [813](#), [830](#), [16123](#), [18732](#), [18884](#),
 [18987](#), [18999](#), [19141](#), [19334](#), [19860](#)
- __fp_basics_pack_high_carry:w . .
 [731](#), [16123](#)
- __fp_basics_pack_low:NNNNNw . . .
 [820](#), [830](#),
 [16123](#), [18734](#), [18886](#), [18989](#), [19001](#),
 [19143](#), [19283](#), [19285](#), [19336](#), [19862](#)
- __fp_basics_pack_weird_high:NNNNNNNw
 [221](#), [16134](#), [18743](#), [19152](#)
- __fp_basics_pack_weird_low:NNNNw
 [221](#), [16134](#), [18745](#), [19154](#)
- \c__fp_big_leading_shift_int . . .
 [16109](#), [19213](#), [19548](#), [19558](#), [19568](#)
- \c__fp_big_middle_shift_int
 [16109](#), [19216](#), [19219](#), [19222](#),
 [19225](#), [19228](#), [19231](#), [19235](#), [19550](#),
 [19560](#), [19570](#), [19580](#), [19583](#), [19586](#)
- \c__fp_big_trailing_shift_int . . .
 [16109](#), [19239](#), [19593](#)

- \c__fp_Bigg_leading_shift_int ... [16114](#), [19062](#), [19080](#)
- \c__fp_Bigg_middle_shift_int ... [16114](#), [19065](#), [19068](#), [19083](#), [19086](#)
- \c__fp_Bigg_trailing_shift_int ... [16114](#), [19071](#), [19089](#)
- __fp_binary_rev_type_o:Nww ... [17631](#), [19405](#), [19407](#)
- __fp_binary_type_o:Nww ... [17631](#), [19403](#), [19416](#)
- \c__fp_block_int ... [15915](#), [19812](#)
- __fp_case_return:nw ... [734](#), [16191](#), [16221](#), [16224](#), [16229](#), [16733](#), [20074](#), [20570](#), [21252](#), [21253](#), [21254](#), [21547](#), [21601](#), [21675](#), [21677](#), [21678](#), [21744](#), [22179](#), [22181](#), [22182](#)
- __fp_case_return_i_o:ww ... [16198](#), [18634](#), [18648](#), [18657](#), [18929](#), [21243](#)
- __fp_case_return_ii_o:ww ... [16198](#), [18930](#), [20359](#), [20377](#), [21244](#)
- __fp_case_return_o:Nw ... [734](#), [735](#), [16192](#), [19355](#), [20109](#), [20114](#), [20117](#), [20309](#), [20314](#), [20337](#), [20340](#), [20343](#), [20546](#), [20660](#), [20690](#), [21454](#), [21456](#)
- __fp_case_return_o:Nww ... [16196](#), [18931](#), [18932](#), [18935](#), [18936](#), [20361](#), [20370](#), [20373](#)
- __fp_case_return_same_o:w ... [734](#), [735](#), [16194](#), [19164](#), [19168](#), [19356](#), [19368](#), [19371](#), [19893](#), [20121](#), [20334](#), [20550](#), [20553](#), [20645](#), [20653](#), [20668](#), [20683](#), [20698](#), [20705](#), [20713](#), [20728](#), [21383](#), [21391](#), [21409](#), [21455](#), [21472](#)
- __fp_case_use:nw ... [734](#), [16190](#), [18659](#), [18927](#), [18928](#), [18933](#), [18934](#), [19016](#), [19019](#), [19166](#), [19352](#), [19886](#), [19889](#), [20345](#), [20556](#), [20646](#), [20651](#), [20661](#), [20666](#), [20676](#), [20681](#), [20691](#), [20696](#), [20706](#), [20711](#), [20721](#), [20726](#), [21385](#), [21388](#), [21398](#), [21400](#), [21406](#), [21450](#), [21452](#), [21463](#), [21466](#), [21471](#), [21550](#), [21557](#), [21604](#), [21611](#)
- __fp_change_func_type:NNN ... [16053](#), [17424](#), [19398](#), [21532](#), [21586](#), [21663](#), [21709](#), [21724](#), [22163](#)
- __fp_change_func_type_aux:w ... [16053](#)
- __fp_change_func_type_chk:NNN [16053](#)
- __fp_chk:w ... [720](#), [721](#), [723](#), [775](#), [810](#), [811](#), [812](#), [814](#), [820](#), [822](#), [15897](#), [15910](#), [15911](#), [15912](#), [15913](#), [15914](#), [15924](#), [15929](#), [15931](#), [15932](#), [15960](#), [15963](#), [15965](#), [15975](#), [15988](#), [16007](#), [16202](#), [16218](#), [16390](#), [16395](#), [16622](#), [16676](#), [16685](#), [16687](#), [17522](#), [18240](#), [18241](#), [18403](#), [18419](#), [18423](#), [18487](#), [18488](#), [18491](#), [18502](#), [18503](#), [18511](#), [18512](#), [18520](#), [18532](#), [18535](#), [18539](#), [18542](#), [18618](#), [18638](#), [18639](#), [18641](#), [18642](#), [18643](#), [18651](#), [18654](#), [18665](#), [18666](#), [18668](#), [18677](#), [18753](#), [18905](#), [18939](#), [18940](#), [18943](#), [19024](#), [19162](#), [19170](#), [19172](#), [19349](#), [19358](#), [19360](#), [19365](#), [19373](#), [19375](#), [19377](#), [19381](#), [19883](#), [19895](#), [19897](#), [20106](#), [20123](#), [20125](#), [20306](#), [20325](#), [20327](#), [20328](#), [20331](#), [20348](#), [20351](#), [20354](#), [20379](#), [20380](#), [20382](#), [20398](#), [20487](#), [20500](#), [20502](#), [20506](#), [20510](#), [20543](#), [20559](#), [20642](#), [20655](#), [20657](#), [20670](#), [20672](#), [20685](#), [20687](#), [20700](#), [20702](#), [20715](#), [20717](#), [20730](#), [20740](#), [21241](#), [21257](#), [21258](#), [21262](#), [21273](#), [21380](#), [21393](#), [21395](#), [21411](#), [21414](#), [21424](#), [21447](#), [21458](#), [21460](#), [21474](#), [21476](#), [21481](#), [21543](#), [21564](#), [21567](#), [21597](#), [21618](#), [21621](#), [21671](#), [21687](#), [21690](#), [21765](#), [21766](#), [21876](#), [21878](#), [21910](#), [22176](#), [22184](#), [22187](#), [22266](#)
- __fp_compare:wNNNNw ... [17885](#)
- __fp_compare_aux:wn ... [18206](#)
- __fp_compare_back:ww ... [917](#), [18222](#), [18501](#), [21894](#)
- __fp_compare_back_any:ww ... [799](#), [799](#), [800](#), [17960](#), [18219](#), [18222](#), [18290](#)
- __fp_compare_back_tuple:ww ... [18267](#)
- __fp_compare_nan:w ... [799](#), [18222](#)
- __fp_compare_npos:nwnw ... [798](#), [799](#), [801](#), [18250](#), [18296](#), [18755](#), [19662](#)
- __fp_compare_return:w ... [18190](#)
- __fp_compare_significand:nnnnnnnn ... [18296](#)
- __fp_cos_o:w ... [20657](#)
- __fp_cot_o:w ... [881](#), [20717](#)
- __fp_cot_zero_o:Nnw ... [880](#), [882](#), [20675](#), [20717](#)
- __fp_csc_o:w ... [20672](#)
- __fp_decimate:nNnnnn ... [732](#), [735](#), [876](#), [16144](#), [16209](#), [16236](#), [16689](#), [18693](#), [18701](#), [18780](#), [20152](#), [20156](#), [20525](#), [21627](#)
- __fp_decimate_:Nnnnn ... [16156](#)
- __fp_decimate_auxi:Nnnnn ... [733](#), [16160](#)
- __fp_decimate_auxii:Nnnnn ... [16160](#)
- __fp_decimate_auxiii:Nnnnn ... [16160](#)
- __fp_decimate_auxiv:Nnnnn ... [16160](#)
- __fp_decimate_auxix:Nnnnn ... [16160](#)
- __fp_decimate_auxv:Nnnnn ... [16160](#)
- __fp_decimate_auxvi:Nnnnn ... [16160](#)

- __fp_decimate_auxvii:Nnnnn . . . [16160](#)
- __fp_decimate_auxviii:Nnnnn . . . [16160](#)
- __fp_decimate_auxx:Nnnnn [16160](#)
- __fp_decimate_auxxi:Nnnnn [16160](#)
- __fp_decimate_auxxii:Nnnnn [16160](#)
- __fp_decimate_auxxiii:Nnnnn [16160](#)
- __fp_decimate_auxxiv:Nnnnn [16160](#)
- __fp_decimate_auxxv:Nnnnn [16160](#)
- __fp_decimate_auxxvi:Nnnnn [16160](#)
- __fp_decimate_pack:nnnnnnnnnw [733](#), [16167](#), [16186](#)
- __fp_decimate_pack:nnnnnnw [16187](#), [16188](#)
- __fp_decimate_tiny:Nnnnn [16156](#)
- __fp_div_npos_o:Nww [822](#), [822](#), [19013](#), [19023](#)
- __fp_div_significand_calc:wnnnnnnn [825](#), [826](#), [19040](#), [19049](#), [19097](#), [19966](#), [19973](#)
- __fp_div_significand_calc_-i:wnnnnnnn [19049](#)
- __fp_div_significand_calc_-ii:wnnnnnnn [19049](#)
- __fp_div_significand_i_o:wnnw [822](#), [825](#), [19030](#), [19036](#)
- __fp_div_significand_ii:wnw [827](#), [19044](#), [19045](#), [19046](#), [19093](#)
- __fp_div_significand_iii:wnnnnn [828](#), [19047](#), [19100](#)
- __fp_div_significand_iv:wnnnnnnn [828](#), [19103](#), [19108](#)
- __fp_div_significand_large_o:wwwNNNwN [830](#), [19134](#), [19148](#)
- __fp_div_significand_pack:NNN [829](#), [862](#), [19095](#), [19128](#), [19953](#), [19971](#), [19979](#)
- __fp_div_significand_small_o:wwwNNNwN [830](#), [19132](#), [19138](#)
- __fp_div_significand_test_o:w [829](#), [829](#), [19038](#), [19129](#)
- __fp_div_significand_v:NN [19113](#), [19115](#), [19118](#)
- __fp_div_significand_v:NNw [19108](#)
- __fp_div_significand_vi:Nw [828](#), [19108](#)
- __fp_division_by_zero_o:Nnw [738](#), [16354](#), [16402](#), [19353](#), [19890](#), [20736](#), [20737](#)
- __fp_division_by_zero_o:NNww [738](#), [16362](#), [16402](#), [19017](#), [19020](#), [20347](#)
- \c__fp_empty_tuple_fp [16008](#), [16829](#), [17483](#), [17493](#)
- __fp_ep_compare:www [19657](#), [21288](#)
- __fp_ep_compare_aux:www [19657](#)
- __fp_ep_div:wwwwn [894](#), [19687](#), [19798](#), [21217](#), [21304](#), [21308](#), [21317](#), [21484](#)
- __fp_ep_div_eps_pack:NNNNw [19717](#)
- __fp_ep_div_epsilon:wnNNNN [852](#)
- __fp_ep_div_epsilon:wnNNNNN [19714](#), [19717](#)
- __fp_ep_div_epsilonii:wnNNNNN [19717](#)
- __fp_ep_div_esti:wwwwn [851](#), [19693](#), [19696](#)
- __fp_ep_div_estii:wnnnwn [19696](#)
- __fp_ep_div_estiii:NNNNwwwwn [19696](#)
- __fp_ep_inv_to_float_o:wN [882](#)
- __fp_ep_inv_to_float_o:wwN [892](#), [19794](#), [19802](#), [20679](#), [20694](#)
- __fp_ep_isqrt:wnw [19740](#), [21445](#)
- __fp_ep_isqrt_aux:wnw [19740](#)
- __fp_ep_isqrt_auxi:wnw [19743](#), [19745](#)
- __fp_ep_isqrt_auxii:wnnnwn [19740](#)
- __fp_ep_isqrt_epsilon:wN [854](#), [19777](#), [19780](#)
- __fp_ep_isqrt_epsilonii:wwN [19780](#)
- __fp_ep_isqrt_esti:wwwnnwn [19755](#), [19758](#)
- __fp_ep_isqrt_estii:wwwnnwn [19758](#)
- __fp_ep_isqrt_estiii:NNNNwwwwn [19758](#)
- __fp_ep_mul:wwwwn [878](#), [19672](#), [20586](#), [20599](#), [21174](#), [21204](#), [21432](#), [21443](#)
- __fp_ep_mul_raw:wwwwn [19672](#), [20758](#), [21124](#)
- __fp_ep_to_ep:wnw [19623](#), [19674](#), [19677](#), [19689](#), [19692](#), [19742](#), [21433](#)
- __fp_ep_to_ep_end:www [19623](#)
- __fp_ep_to_ep_loop:N [891](#), [19623](#), [21125](#)
- __fp_ep_to_ep_zero:ww [19623](#)
- __fp_ep_to_fixed:wnw [19605](#), [20755](#), [21311](#), [21320](#), [21430](#), [21919](#)
- __fp_ep_to_fixed_auxi:www [19605](#)
- __fp_ep_to_fixed_auxii:nnnnnnwn [19605](#)
- __fp_ep_to_float_o:wN [882](#)
- __fp_ep_to_float_o:wwN [880](#), [892](#), [19794](#), [19806](#), [20610](#), [20649](#), [20664](#), [21223](#)
- __fp_error:nnnn [16323](#), [16331](#), [16340](#), [16357](#), [16365](#), [16393](#), [16416](#), [16615](#), [16617](#), [16638](#), [16643](#), [17419](#), [17998](#), [18013](#), [18399](#), [18418](#), [18429](#), [21538](#), [21592](#), [21666](#), [22173](#)
- __fp_exp_after_?_f:nw [729](#), [761](#), [16813](#)
- __fp_exp_after_any_f:Nnw [16078](#)

- __fp_exp_after_any_f:nw
 729, 16078, 16104, 16815, 17588
- __fp_exp_after_array_f:w
 729, 16089,
 17473, 18569, 18580, 18590, 18598
- __fp_exp_after_expr_mark_f:nw ..
 761, 16813
- __fp_exp_after_expr_stop_f:nw 16078
- __fp_exp_after_f:nw
 725, 761, 15965, 16083, 17521, 17659
- __fp_exp_after_normal:nNNw
 15968, 15978, 15995
- __fp_exp_after_normal:Nwwwww ...
 15997, 16005
- __fp_exp_after_o:w .. 725, 15965,
 16195, 16199, 16201, 16683, 16727,
 16745, 17980, 18519, 18537, 18546,
 18555, 18642, 19379, 20499, 20504
- __fp_exp_after_special:nNNw ...
 726, 15970, 15980, 15985
- __fp_exp_after_tuple_f:nw
 16089, 17787
- __fp_exp_after_tuple_o:w
 16089, 18544, 18547, 18550, 18552
- \c__fp_exp_intarray
 20199, 20285, 20292, 20295, 20297
- __fp_exp_intarray:w 20256
- __fp_exp_intarray_aux:w 20256
- __fp_exp_large:NwN 869, 20256, 20483
- __fp_exp_large_after:wwn 869, 20256
- __fp_exp_normal_o:w .. 20111, 20125
- __fp_exp_o:w 19869, 20106
- __fp_exp_overflow:NN 20125
- __fp_exp_pos_large:NnnNwn
 20157, 20256
- __fp_exp_pos_o:NNwnw
 20128, 20130, 20133
- __fp_exp_pos_o:Nwnw 20125
- __fp_exp_Taylor:Nnnwn
 20153, 20172, 20302
- __fp_exp_Taylor_break:Nww ... 20172
- __fp_exp_Taylor_ii:ww . 20178, 20181
- __fp_exp_Taylor_loop:www 20172
- __fp_expand:n 912
- __fp_exponent:w 15932
- __fp_facorial_int_o:n 877
- __fp_fact_int_o:n 20564, 20567
- __fp_fact_int_o:w 20561
- __fp_fact_loop_o:w ... 20579, 20581
- \c__fp_fact_max_arg_int 20542, 20569
- __fp_fact_o:w 19873, 20543
- __fp_fact_pos_o:w 20558, 20561
- __fp_fact_small_o:w .. 20584, 20596
- \c__fp_five_int 16477,
 16501, 16514, 16527, 16534, 16587
- __fp_fixed_<calculation>:wnn .. 840
- __fp_fixed_add:nnNnnwn 19498
- __fp_fixed_add:Nnnnnwnn 19498
- __fp_fixed_add:wnn 840,
 843, 19498, 19738, 20048, 20056,
 20067, 20085, 21316, 21376, 21934
- __fp_fixed_add_after:NNNNwn . 19498
- __fp_fixed_add_one:wN 841, 19430,
 19731, 20189, 20198, 21442, 21925
- __fp_fixed_add_pack:NNNNwn . 19498
- __fp_fixed_continue:wn
 19429, 19675,
 19680, 19690, 20267, 20458, 20793,
 21162, 21434, 21443, 21917, 21929
- __fp_fixed_div_int:wN 19467
- __fp_fixed_div_int:wwN
 842, 19467, 20047, 20188, 21335
- __fp_fixed_div_int_after:Nw ...
 842, 19467
- __fp_fixed_div_int_auxi:wnn . 19467
- __fp_fixed_div_int_auxii:wnn ...
 842, 19467
- __fp_fixed_div_int_pack:Nw ...
 842, 19467
- __fp_fixed_div_myriad:wn
 19435, 19735
- __fp_fixed_inv_to_float_o:wN ...
 19801, 20130, 20394
- __fp_fixed_mul:nnnnnnnw 19518
- __fp_fixed_mul:wnn
 840, 841, 844, 890, 892, 19518,
 19684, 19715, 19730, 19732, 19736,
 19789, 19792, 19805, 20049, 20059,
 20099, 20190, 20288, 20303, 20404,
 21131, 21185, 21323, 21356, 21358
- __fp_fixed_mul_add:nnnnwnnnn ...
 847, 19587, 19589
- __fp_fixed_mul_add:nnnnwnnnN ...
 847, 19594, 19600
- __fp_fixed_mul_add:Nwnnnwnnn ...
 846, 19551, 19561, 19572, 19576
- __fp_fixed_mul_add:www
 845, 19545, 21939
- __fp_fixed_mul_after:wnn
 844, 19437, 19443, 19446,
 19520, 19547, 19557, 19567, 20421
- __fp_fixed_mul_one_minus_-
 mul:wnn 19545
- __fp_fixed_mul_short:wnn
 841, 19444,
 19713, 19734, 19776, 19778, 21369

__fp_fixed_mul_sub_back:wwn ...
 [845](#), [19545](#),
 19790, 21152, 21154, 21155, 21156,
 21157, 21158, 21159, 21160, 21161,
 21165, 21167, 21168, 21169, 21170,
 21171, 21172, 21173, 21198, 21200,
 21201, 21202, 21203, 21206, 21208,
 21209, 21210, 21211, 21336, 21344
 __fp_fixed_one_minus_mul:wn ...
 [845](#), [846](#), 19565
 __fp_fixed_sub:wn [19498](#), 19782,
 20065, 20081, 20093, 20797, 21317,
 21374, 21440, 21927, 21936, 21968
 __fp_fixed_to_float_o:Nw
 [19808](#), [20074](#)
 __fp_fixed_to_float_o:wN
 [840](#), [856](#),
 900, 19795, [19808](#), 20094, 20104,
 20128, 20390, 21364, 21867, 21973
 __fp_fixed_to_float_pack:ww ...
 [19841](#), [19851](#)
 __fp_fixed_to_float_rad_o:wN ...
 [19803](#), [21364](#)
 __fp_fixed_to_float_round_
 up:wnnnnw [19854](#), [19858](#)
 __fp_fixed_to_float_zero:w
 [19837](#), [19846](#)
 __fp_fixed_to_loop:N
 [19814](#), [19824](#), [19828](#)
 __fp_fixed_to_loop_end:w
 [19830](#), [19834](#)
 __fp_from_dim:wNNnnnnnn [21734](#)
 __fp_from_dim:wnnnnwNn [21761](#), [21762](#)
 __fp_from_dim:wnnnnwNw [21734](#)
 __fp_from_dim:wNw [21734](#)
 __fp_from_dim_test:ww
 [911](#), [16905](#), [16942](#), [17540](#), [21734](#)
 __fp_func_to_name:N
 [16270](#), [17419](#), [17428](#)
 __fp_func_to_name_aux:w [16270](#)
 \c__fp_half_prec_int
 [15915](#), [17146](#), [17178](#)
 __fp_if_type_fp:NTwFw . [727](#), [793](#),
 15945, [16024](#), [16032](#), [16039](#), [16055](#),
 16082, 18007, 18021, 18198, 18224,
 18225, 18392, 18393, 18394, 18560
 __fp_inf_fp:N [15928](#), [16378](#)
 __fp_int:wIF [16202](#), [21878](#)
 __fp_int_eval:w
 [730](#), [744](#), [746](#), [746](#), [759](#), [775](#),
 [812](#), [820](#), [820](#), [823](#), [827](#), [856](#), [15882](#),
 15942, 16017, 16148, 16151, 16551,
 16555, 16567, 16568, 16604, 16695,
 16699, 16738, 16952, 16957, 16999,
 17088, 17099, 17148, 17179, 17185,
 17186, 17232, 17242, 17244, 17260,
 17262, 17285, 17287, 17454, 17676,
 17720, 17920, 18211, 18681, 18689,
 18710, 18712, 18733, 18735, 18744,
 18746, 18775, 18781, 18791, 18793,
 18867, 18869, 18885, 18887, 18891,
 18907, 18947, 18955, 18957, 18959,
 18961, 18964, 18967, 18969, 18988,
 18990, 19000, 19002, 19028, 19031,
 19039, 19041, 19062, 19065, 19068,
 19071, 19080, 19083, 19086, 19089,
 19096, 19098, 19104, 19112, 19114,
 19116, 19122, 19142, 19144, 19153,
 19155, 19176, 19197, 19201, 19213,
 19216, 19219, 19222, 19225, 19228,
 19231, 19234, 19238, 19250, 19254,
 19258, 19261, 19282, 19284, 19286,
 19296, 19335, 19337, 19346, 19433,
 19438, 19440, 19447, 19450, 19453,
 19456, 19459, 19462, 19471, 19483,
 19491, 19493, 19503, 19505, 19512,
 19521, 19523, 19526, 19529, 19532,
 19535, 19548, 19550, 19558, 19560,
 19568, 19570, 19580, 19583, 19586,
 19593, 19608, 19626, 19629, 19685,
 19699, 19701, 19707, 19720, 19722,
 19724, 19748, 19764, 19771, 19772,
 19795, 19812, 19816, 19861, 19863,
 19907, 19918, 19937, 19939, 19941,
 19954, 19967, 19972, 19974, 19980,
 19997, 19998, 19999, 20000, 20001,
 20002, 20007, 20009, 20011, 20013,
 20015, 20020, 20022, 20024, 20026,
 20028, 20030, 20052, 20060, 20144,
 20193, 20270, 20278, 20286, 20292,
 20295, 20401, 20422, 20424, 20427,
 20430, 20433, 20436, 20452, 20478,
 20492, 20508, 20578, 20588, 20593,
 20745, 20777, 20786, 21018, 21032,
 21035, 21038, 21041, 21044, 21047,
 21050, 21053, 21056, 21072, 21082,
 21091, 21109, 21118, 21125, 21136,
 21146, 21179, 21189, 21214, 21223,
 21266, 21283, 21285, 21297, 21298,
 21339, 21350, 21361, 21419, 21571,
 21694, 21747, 21843, 21866, 21920,
 21972, 21994, 21996, 21998, 22003,
 22022, 22034, 22042, 22047, 22052
 __fp_int_eval_end:
 [15882](#), 15942, 16020, 16139, 16604,
 16709, 16713, 17921, 18211, 18891,
 18926, 19118, 19493, 19629, 20452,
 20508, 20778, 20787, 21136, 21146,

- 21189, 21214, 21298, 22001, 22003
- __fp_int_p:w [16202](#)
- __fp_int_to_roman:w [15882](#),
[16151](#), [17160](#), [17192](#), [19934](#), [22104](#)
- __fp_invalid_operation:nw
. [738](#), [16320](#), [16402](#), [16414](#), [21552](#),
[21559](#), [21606](#), [21613](#), [21713](#), [21728](#)
- __fp_invalid_operation_o:nw ...
. [738](#), [16413](#), [17428](#), [19166](#), [19392](#),
[19886](#), [20556](#), [20565](#), [20652](#), [20667](#),
[20682](#), [20697](#), [20712](#), [20727](#), [21389](#),
[21407](#), [21423](#), [21451](#), [21464](#), [21480](#)
- __fp_invalid_operation_o:Nww ...
..... [738](#), [16328](#), [16402](#),
[17629](#), [18661](#), [18933](#), [18934](#), [20493](#)
- __fp_invalid_operation_o:nww . [19417](#)
- __fp_invalid_operation_tl_o:nn .
.... [738](#), [16337](#), [16402](#), [16670](#), [21898](#)
- __fp_kind:w [15943](#), [16663](#), [18184](#)
- \c__fp_leading_shift_int
..... [16105](#), [19438](#),
[19447](#), [19521](#), [20422](#), [21072](#), [21109](#)
- __fp_ln_c:NwNw [863](#), [864](#), [20031](#), [20062](#)
- __fp_ln_div_after:Nw
..... [862](#), [19933](#), [19982](#)
- __fp_ln_div_i:w [19955](#), [19964](#)
- __fp_ln_div_ii:wwn
.. [19958](#), [19959](#), [19960](#), [19961](#), [19969](#)
- __fp_ln_div_vi:wwn ... [19962](#), [19977](#)
- __fp_ln_exponent:wn [865](#), [19909](#), [20071](#)
- __fp_ln_exponent_one:ww [20076](#), [20090](#)
- __fp_ln_exponent_small:NNww ...
..... [20079](#), [20083](#), [20096](#)
- \c__fp_ln_i_fixed_tl [19874](#)
- \c__fp_ln_ii_fixed_tl [19874](#)
- \c__fp_ln_iii_fixed_tl [19874](#)
- \c__fp_ln_iv_fixed_tl [19874](#)
- \c__fp_ln_ix_fixed_tl [19874](#)
- __fp_ln_npos_o:w
..... [857](#), [858](#), [19895](#), [19897](#)
- __fp_ln_o:w .. [857](#), [873](#), [19871](#), [19883](#)
- __fp_ln_significand:NNNNnnnN ...
..... [859](#), [19908](#), [19911](#), [20402](#)
- __fp_ln_square_t_after:w
..... [20006](#), [20038](#)
- __fp_ln_square_t_pack:NNNNNw ...
.. [20008](#), [20010](#), [20012](#), [20014](#), [20036](#)
- __fp_ln_t_large:NNw
..... [862](#), [19987](#), [19994](#), [20004](#)
- __fp_ln_t_small:Nw ... [19985](#), [19992](#)
- __fp_ln_t_small:w [862](#)
- __fp_ln_Taylor:wwNw [863](#), [20039](#), [20040](#)
- __fp_ln_Taylor_break:w [20045](#), [20056](#)
- __fp_ln_Taylor_loop:www
..... [20041](#), [20042](#), [20051](#)
- __fp_ln_twice_t_after:w [20019](#), [20035](#)
- __fp_ln_twice_t_pack:Nw . [20021](#),
[20023](#), [20025](#), [20027](#), [20029](#), [20034](#)
- \c__fp_ln_vi_fixed_tl [19874](#)
- \c__fp_ln_vii_fixed_tl [19874](#)
- \c__fp_ln_viii_fixed_tl [19874](#)
- \c__fp_ln_x_fixed_tl
..... [19874](#), [20093](#), [20100](#)
- __fp_ln_x_ii:wnnnn ... [19913](#), [19931](#)
- __fp_ln_x_iii:NNNNNNw . [19940](#), [19944](#)
- __fp_ln_x_iii_var:NNNNNw
..... [19938](#), [19946](#)
- __fp_ln_x_iv:wnnnnnnnn
..... [861](#), [19936](#), [19951](#)
- __fp_logb_aux_o:w [19349](#)
- __fp_logb_o:w [18607](#), [19349](#)
- \c__fp_max_exp_exponent_int
..... [15921](#), [20136](#)
- \c__fp_max_exponent_int .. [15919](#),
[15925](#), [15953](#), [19646](#), [19848](#), [20457](#)
- \c__fp_middle_shift_int
..... [16105](#), [19450](#),
[19453](#), [19456](#), [19459](#), [19523](#), [19526](#),
[19529](#), [19532](#), [20424](#), [20427](#), [20430](#),
[20433](#), [21075](#), [21082](#), [21112](#), [21118](#)
- __fp_minmax_aux_o:Nw [18473](#)
- __fp_minmax_auxi:ww
..... [18495](#), [18507](#), [18514](#)
- __fp_minmax_auxii:ww
..... [18497](#), [18505](#), [18514](#)
- __fp_minmax_break_o:w . [18488](#), [18518](#)
- __fp_minmax_loop:Nww
..... [805](#), [18482](#), [18484](#), [18490](#)
- __fp_minmax_o:Nw
..... [798](#), [18177](#), [18179](#), [18473](#)
- \c__fp_minus_min_exponent_int ...
..... [15919](#), [15954](#)
- __fp_misused:n . [15895](#), [15899](#), [16010](#)
- __fp_mul_cases_o:NnNnw
..... [822](#), [18898](#), [18904](#), [19010](#)
- __fp_mul_cases_o:nNnnw [18904](#)
- __fp_mul_npos_o:Nww
[818](#), [820](#), [822](#), [911](#), [18901](#), [18942](#), [21764](#)
- __fp_mul_significand_drop:NNNNNw
..... [820](#), [18951](#)
- __fp_mul_significand_keep:NNNNNw
..... [18951](#)
- __fp_mul_significand_large_-
f:NwwNNNN [18981](#), [18985](#)
- __fp_mul_significand_o:nnnnNnnnn
..... [820](#), [820](#), [18949](#), [18951](#)

- __fp_mul_significand_small_-
f:NNwwwN [18979](#), [18996](#)
- __fp_mul_significand_test_f:NNN
..... [821](#), [18953](#), [18976](#)
- \c_fp_myriad_int [15918](#),
[19433](#), [19464](#), [19465](#), [19542](#), [19603](#)
- __fp_neg_sign:N
..... [810](#), [15941](#), [18615](#), [18768](#)
- __fp_not_o:w [798](#), [17447](#), [18520](#)
- \c_fp_one_fixed_tl [19427](#),
[20047](#), [20260](#), [20458](#), [20485](#), [21268](#),
[21335](#), [21440](#), [21917](#), [21927](#), [21968](#)
- __fp_overflow:w [725](#),
[738](#), [740](#), [15956](#), [16402](#), [20138](#), [20572](#)
- \c__fp_overflowing_fp
..... [15922](#), [21553](#), [21607](#)
- __fp_pack:NNNNNw .. [16105](#), [19439](#),
[19449](#), [19452](#), [19455](#), [19458](#), [19461](#),
[19522](#), [19525](#), [19528](#), [19531](#), [19534](#),
[20423](#), [20426](#), [20429](#), [20432](#), [20435](#)
- __fp_pack_big:NNNNNNw ... [16109](#),
[19215](#), [19218](#), [19221](#), [19224](#), [19227](#),
[19230](#), [19233](#), [19237](#), [19549](#), [19559](#),
[19569](#), [19579](#), [19582](#), [19585](#), [19592](#)
- __fp_pack_Bigg:NNNNNNw
..... [16114](#), [19064](#),
[19067](#), [19070](#), [19082](#), [19085](#), [19088](#)
- __fp_pack_eight:wNNNNNNNN
..... [731](#), [816](#), [16121](#),
[18877](#), [19186](#), [19614](#), [20764](#), [20765](#)
- __fp_pack_twice_four:wNNNNNNNN .
..... [731](#), [16119](#), [16720](#), [16721](#),
[18819](#), [18820](#), [19615](#), [19616](#), [19617](#),
[19649](#), [19650](#), [19651](#), [19839](#), [19840](#),
[20175](#), [20176](#), [20177](#), [20766](#), [20767](#),
[21061](#), [21062](#), [21063](#), [21064](#), [21757](#)
- __fp_parse:n [751](#), [762](#),
[774](#), [782](#), [795](#), [796](#), [803](#), [912](#), [912](#),
[922](#), [16751](#), [16902](#), [17564](#), [18120](#),
[18122](#), [18124](#), [18147](#), [18184](#), [18193](#),
[18210](#), [18220](#), [18377](#), [18437](#), [19362](#),
[21528](#), [21582](#), [21660](#), [21705](#), [21720](#),
[21773](#), [21775](#), [21777](#), [21779](#), [22156](#)
- __fp_parse_after:ww [17564](#)
- __fp_parse_apply_binary:NwNwN ..
..... [755](#), [759](#), [759](#), [787](#), [17602](#), [17797](#)
- __fp_parse_apply_binary_chk:NN .
..... [17602](#), [17633](#), [17646](#)
- __fp_parse_apply_binary_-
error:NNN [17602](#)
- __fp_parse_apply_comma:NwNwN ...
..... [787](#), [17756](#)
- __fp_parse_apply_compare:NwNNNNNwN
..... [17944](#), [17953](#)
- __fp_parse_apply_compare_-
aux:NNwN [17965](#), [17968](#), [17973](#)
- __fp_parse_apply_function:NNNwN
..... [778](#), [17396](#), [17557](#)
- __fp_parse_apply_unary:NNNwN ...
..... [17401](#), [17433](#), [17548](#)
- __fp_parse_apply_unary_chk:nNNNNw
..... [17412](#), [17413](#), [17416](#)
- __fp_parse_apply_unary_chk:nNNNw
..... [17401](#)
- __fp_parse_apply_unary_chk:NwNw
..... [17401](#)
- __fp_parse_apply_unary_error:NNw
..... [17401](#), [19399](#)
- __fp_parse_apply_unary_type:NNN
..... [17401](#)
- __fp_parse_caseless_inf:N ... [17514](#)
- __fp_parse_caseless_infinity:N .
..... [17514](#)
- __fp_parse_caseless_nan:N ... [17514](#)
- __fp_parse_compare:NNNNNNN .. [17885](#)
- __fp_parse_compare_auxi:NNNNNNN
..... [17885](#)
- __fp_parse_compare_auxii:NNNNN .
..... [17885](#)
- __fp_parse_compare_end:NNNNw . [17885](#)
- __fp_parse_continue:NwN
..... [755](#), [756](#), [783](#), [17591](#), [17604](#),
[17784](#), [17983](#), [18577](#), [18587](#), [18595](#)
- __fp_parse_continue_compare:NNwNN
..... [17976](#), [17991](#)
- __fp_parse_digits_:N [16769](#)
- __fp_parse_digits_i:N [16769](#)
- __fp_parse_digits_ii:N [16769](#)
- __fp_parse_digits_iii:N [16769](#)
- __fp_parse_digits_iv:N [16769](#)
- __fp_parse_digits_v:N [16769](#)
- __fp_parse_digits_vi:N
..... [16769](#), [17104](#), [17152](#)
- __fp_parse_digits_vii:N
..... [768](#), [16769](#), [17091](#), [17141](#)
- __fp_parse_excl_error: [17885](#)
- __fp_parse_expand:w
..... [758](#), [758](#), [759](#), [759](#), [16766](#), [16768](#),
[16778](#), [16818](#), [16878](#), [16922](#), [16931](#),
[16934](#), [16938](#), [16975](#), [17009](#), [17047](#),
[17049](#), [17068](#), [17070](#), [17092](#), [17109](#),
[17122](#), [17142](#), [17172](#), [17200](#), [17216](#),
[17227](#), [17250](#), [17279](#), [17289](#), [17296](#),
[17310](#), [17326](#), [17346](#), [17357](#), [17443](#),
[17466](#), [17478](#), [17553](#), [17562](#), [17570](#),
[17583](#), [17703](#), [17751](#), [17775](#), [17801](#),
[17849](#), [17869](#), [17938](#), [17951](#), [18573](#)

- __fp_parse_exponent:N
772, 16877, 17083, 17232, 17299, 17301
- __fp_parse_exponent:Nw
17107, 17120,
17169, 17197, 17248, 17277, 17296
- __fp_parse_exponent_aux:NN .. 17301
- __fp_parse_exponent_body:N
17328, 17332
- __fp_parse_exponent_digits:N ...
17336, 17348
- __fp_parse_exponent_keep:N .. 17359
- __fp_parse_exponent_keep:NTF ...
17339, 17359
- __fp_parse_exponent_sign:N
17318, 17322
- __fp_parse_function:NNN
16463, 16465, 16467,
16470, 17546, 18177, 18179, 20635,
20637, 20639, 20641, 21802, 21804
- __fp_parse_function_all_fp_-
o:nnw 16597, 17993, 18475
- __fp_parse_function_one_two:nnw
.... 896, 18005, 21229, 21235, 21871
- __fp_parse_function_one_two_-
aux:nnw 18005
- __fp_parse_function_one_two_-
auxii:nnw 18005
- __fp_parse_function_one_two_-
error_o:w 18005
- __fp_parse_infix:NN
761, 764, 781, 785,
786, 16817, 16987, 17026, 17506,
17521, 17543, 17659, 17662, 17749
- __fp_parse_infix_!:N 17885
- __fp_parse_infix_&:Nw 17842
- __fp_parse_infix(:N 17825
- __fp_parse_infix):N 17739
- __fp_parse_infix*:N 17827
- __fp_parse_infix+:N
759, 16766, 17791
- __fp_parse_infix_,:N 17756
- __fp_parse_infix -:N 17791
- __fp_parse_infix/:N 17791
- __fp_parse_infix::N . 17859, 18558
- __fp_parse_infix<:N 17885
- __fp_parse_infix=:N 17885
- __fp_parse_infix>:N 17885
- __fp_parse_infix?:N 17859
- __fp_parse_infix{operation}_:N 759
- __fp_parse_infix^:N 17791
- __fp_parse_infix_after_operand:NwN
.... 764, 16870, 16948, 17450, 17657
- __fp_parse_infix_after_paren:NN
..... 17475, 17501, 17706
- __fp_parse_infix_and:N 17791, 17858
- __fp_parse_infix_check:NNN
17682, 17692, 17726
- __fp_parse_infix_comma:w 787, 17756
- __fp_parse_infix_end:N
782, 786, 17571, 17576, 17584, 17737
- __fp_parse_infix_juxt:N
785, 17672, 17680, 17791
- __fp_parse_infix_mark:NNN
17669, 17713, 17736
- __fp_parse_infix_mul:N
785, 789, 17697,
17716, 17724, 17791, 17826, 17835
- __fp_parse_infix_or:N . 17791, 17857
- __fp_parse_infix_|:Nw 17842
- __fp_parse_large:N 767, 17054, 17137
- __fp_parse_large_leading:wwNN ..
770, 17139, 17144
- __fp_parse_large_round:NN
771, 17180, 17252
- __fp_parse_large_round_aux:wNN .
17252
- __fp_parse_large_round_test:NN .
17252
- __fp_parse_large_trailing:wwNN .
771, 17150, 17174
- __fp_parse_letters:N
764, 765, 16963, 16977
- __fp_parse_lparen_after:NwN . 17456
- __fp_parse_o:n
751, 17564, 18375, 18376
- __fp_parse_one:Nw
754-757, 759, 766, 781,
783, 16766, 16789, 17031, 17395, 17597
- __fp_parse_one_digit:NN
779, 16805, 16946
- __fp_parse_one_fp:NN
760, 16797, 16813
- __fp_parse_one_other:NN 16808, 16954
- __fp_parse_one_register:NN
16800, 16868
- __fp_parse_one_register_aux:Nw .
16868
- __fp_parse_one_register_-
auxii:wwwNw 16868
- __fp_parse_one_register_dim:ww .
16868
- __fp_parse_one_register_int:www
..... 16868
- __fp_parse_one_register_-
math:NNw 16909
- __fp_parse_one_register_mu:www .
16868

- __fp_parse_one_register_-special:N [16873](#), [16909](#)
- __fp_parse_one_register_wd:Nw [16909](#)
- __fp_parse_one_register_wd:w . [16909](#)
- __fp_parse_operand:Nw
 [754–757](#), [758](#), [782](#), [787](#),
 [16766](#), [17439](#), [17441](#), [17462](#), [17464](#),
 [17553](#), [17562](#), [17569](#), [17582](#), [17591](#),
 [17774](#), [17800](#), [17868](#), [17951](#), [18572](#)
- __fp_parse_pack_carry:w . [769](#), [17124](#)
- __fp_parse_pack_leading:NNNNnw
 [17087](#), [17124](#), [17147](#)
- __fp_parse_pack_trailing:NNNNnw
 .. [17097](#), [17124](#), [17166](#), [17177](#), [17184](#)
- __fp_parse_prefix:NNN . [16966](#), [17011](#)
- __fp_parse_prefix_!:Nw [17429](#)
- __fp_parse_prefix(:Nw [17456](#)
- __fp_parse_prefix):Nw [17488](#)
- __fp_parse_prefix+:Nw [17395](#)
- __fp_parse_prefix-:Nw [17429](#)
- __fp_parse_prefix.:Nw [17448](#)
- __fp_parse_prefix_unknown:NNN [17011](#)
- __fp_parse_return_semicolon:w ..
 [16767](#), [16776](#), [17007](#),
 [17214](#), [17225](#), [17308](#), [17340](#), [17355](#)
- __fp_parse_round:Nw [16468](#)
- __fp_parse_round_after:wN
 [773](#), [17229](#), [17234](#), [17284](#)
- __fp_parse_round_loop:N ... [772](#),
 [773](#), [773](#), [17202](#), [17245](#), [17263](#), [17288](#)
- __fp_parse_round_up:N [17202](#)
- __fp_parse_small:N [767](#), [17074](#), [17085](#)
- __fp_parse_small_leading:wwNN ..
 [768](#), [17089](#), [17094](#), [17156](#)
- __fp_parse_small_round:NN
 [17116](#), [17234](#), [17273](#)
- __fp_parse_small_trailing:wwNN .
 [769](#), [17102](#), [17111](#), [17188](#)
- __fp_parse_strim_end:w [17060](#)
- __fp_parse_strim_zeros:N
 [767](#), [779](#), [17041](#), [17060](#), [17454](#)
- __fp_parse_trim_end:w [17034](#)
- __fp_parse_trim_zeros:N [16952](#), [17034](#)
- __fp_parse_unary_function:NNN ..
 [17546](#), [18605](#), [18607](#), [18609](#), [18611](#),
 [19869](#), [19871](#), [19873](#), [20623](#), [20629](#)
- __fp_parse_word:Nw [764](#), [16960](#), [16977](#)
- __fp_parse_word_abs:N [18604](#)
- __fp_parse_word_acos:N [20615](#)
- __fp_parse_word_acosd:N [20615](#)
- __fp_parse_word_acot:N [20634](#)
- __fp_parse_word_acotd:N [20634](#)
- __fp_parse_word_acsc:N [20615](#)
- __fp_parse_word_acscd:N [20615](#)
- __fp_parse_word_asec:N [20615](#)
- __fp_parse_word_asecd:N [20615](#)
- __fp_parse_word_asin:N [20615](#)
- __fp_parse_word_asind:N [20615](#)
- __fp_parse_word_atan:N [20634](#)
- __fp_parse_word_atand:N [20634](#)
- __fp_parse_word_bp:N [17517](#)
- __fp_parse_word_cc:N [17517](#)
- __fp_parse_word_ceil:N [16462](#)
- __fp_parse_word_cm:N [17517](#)
- __fp_parse_word_cos:N [20615](#)
- __fp_parse_word_cosd:N [20615](#)
- __fp_parse_word_cot:N [20615](#)
- __fp_parse_word_cotd:N [20615](#)
- __fp_parse_word_csc:N [20615](#)
- __fp_parse_word_cscd:N [20615](#)
- __fp_parse_word_dd:N [17517](#)
- __fp_parse_word_deg:N [17503](#)
- __fp_parse_word_em:N [17536](#)
- __fp_parse_word_ex:N [17536](#)
- __fp_parse_word_exp:N [19868](#)
- __fp_parse_word_fact:N [19868](#)
- __fp_parse_word_false:N [17503](#)
- __fp_parse_word_floor:N [16462](#)
- __fp_parse_word_in:N [17517](#)
- __fp_parse_word_inf:N
 [17503](#), [17514](#), [17515](#)
- __fp_parse_word_ln:N [19868](#)
- __fp_parse_word_logb:N [18604](#)
- __fp_parse_word_max:N [18176](#)
- __fp_parse_word_min:N [18176](#)
- __fp_parse_word_mm:N [17517](#)
- __fp_parse_word_nan:N . [17503](#), [17516](#)
- __fp_parse_word_nc:N [17517](#)
- __fp_parse_word_nd:N [17517](#)
- __fp_parse_word_pc:N [17517](#)
- __fp_parse_word_pi:N [17503](#)
- __fp_parse_word_pt:N [17517](#)
- __fp_parse_word_rand:N [21801](#)
- __fp_parse_word_randint:N ... [21801](#)
- __fp_parse_word_round:N [16468](#)
- __fp_parse_word_sec:N [20615](#)
- __fp_parse_word_secd:N [20615](#)
- __fp_parse_word_sign:N [18604](#)
- __fp_parse_word_sin:N [20615](#)
- __fp_parse_word_sind:N [20615](#)
- __fp_parse_word_sp:N [17517](#)
- __fp_parse_word_sqrt:N [18604](#)
- __fp_parse_word_tan:N [20615](#)
- __fp_parse_word_tand:N [20615](#)
- __fp_parse_word_true:N [17503](#)
- __fp_parse_word_trunc:N [16462](#)
- __fp_parse_zero:
 [767](#), [17056](#), [17076](#), [17080](#)

- __fp_pow_B:wwN 20405, 20440
- __fp_pow_C_neg:w 20443, 20460
- __fp_pow_C_overflow:w
..... 20448, 20455, 20476
- __fp_pow_C_pack:w 20462, 20470, 20481
- __fp_pow_C_pos:w 20446, 20465
- __fp_pow_C_pos_loop:wN
..... 20466, 20467, 20474
- __fp_pow_exponent:Nwnnnnw
..... 20411, 20414, 20419
- __fp_pow_exponent:wnN . 20403, 20408
- __fp_pow_neg:www .. 875, 20316, 20487
- __fp_pow_neg_aux:wNN ... 875, 20487
- __fp_pow_neg_case:w .. 20489, 20510
- __fp_pow_neg_case_aux:nnnnn . 20510
- __fp_pow_neg_case_aux:Nnnw
..... 876, 20510
- __fp_pow_normal_o:ww
..... 871, 20321, 20353
- __fp_pow_npos_aux:NNnw
..... 20388, 20392, 20398
- __fp_pow_npos_o:Nww 872, 20365, 20382
- __fp_pow_zero_or_inf:ww
..... 871, 20323, 20330
- \c__fp_prec_and_int ... 16751, 17822
- \c__fp_prec_colon_int
..... 16751, 17880, 18572
- \c__fp_prec_comma_int
..... 779, 16751, 16825,
17462, 17490, 17760, 17765, 17774
- \c__fp_prec_comp_int
..... 16751, 17908, 17951
- \c__fp_prec_end_int 782,
786, 16751, 16827, 17569, 17582, 17743
- \c__fp_prec_func_int
.... 779, 16751, 17461, 17553, 17562
- \c__fp_prec_hat_int ... 16751, 17810
- \c__fp_prec_hatii_int . 16751, 17810
- \c__fp_prec_int
15915, 16148, 16209, 16236, 16689,
20156, 20522, 20525, 21625, 21627,
21633, 21684, 21882, 21921, 21972
- \c__fp_prec_juxt_int .. 16751, 17812
- \c__fp_prec_not_int
..... 779, 16751, 17446, 17447
- \c__fp_prec_or_int 16751, 17824
- \c__fp_prec_plus_int
..... 753, 16751, 17818, 17820
- \c__fp_prec_quest_int
..... 16751, 17863, 17878
- \c__fp_prec_times_int
..... 16751, 17814, 17816
- \c__fp_prec_tuple_int
.... 779, 16751, 16826, 17464, 17492
- __fp_rand_myriads:n
..... 916, 917, 21837, 21854, 21940
- __fp_rand_myriads_get:w 21837
- __fp_rand_myriads_loop:w 21837
- __fp_rand_o:Nw
..... 21802, 21809, 21815, 21848
- __fp_rand_o:w 21848
- __fp_randinat_wide_aux:w 22010
- __fp_randinat_wide_auxii:w .. 22010
- __fp_randint:n 22072
- __fp_randint:ww 21978, 22082
- __fp_randint_auxi_o:ww 21869
- __fp_randint_auxii:wn 21869
- __fp_randint_auxiii_o:ww 21869
- __fp_randint_auxiv_o:ww 21869
- __fp_randint_auxv_o:w 21869
- __fp_randint_badarg:w ... 917, 21869
- __fp_randint_default:w 21869
- __fp_randint_o:Nw 21804, 21815, 21869
- __fp_randint_o:w 21869
- __fp_randint_split_aux:w 22010
- __fp_randint_split_o:Nw . 920, 22010
- __fp_randint_wide_aux:w
..... 920, 22013, 22044
- __fp_randint_wide_auxii:w
..... 22046, 22055
- __fp_reverse_args:Nww
..... 902, 903, 15891,
21215, 21290, 21403, 21469, 21966
- __fp_round:NNN 744, 744, 746, 821,
837, 16478, 16548, 18736, 18747,
18991, 19003, 19145, 19156, 19340
- __fp_round:Nwn . 16606, 16659, 21732
- __fp_round:Nww . 16607, 16628, 16659
- __fp_round:Nwww 16608, 16622
- __fp_round_aux_o:Nw 16595
- __fp_round_digit:Nw .. 733, 746,
820, 821, 837, 16166, 16562, 18750,
18893, 18994, 19006, 19159, 19345
- __fp_round_name_from_cs:N
.. 16598, 16618, 16644, 16648, 16671
- __fp_round_neg:NNN 744,
746, 817, 16573, 18855, 18870, 18888
- __fp_round_no_arg_o:Nw 16605, 16612
- __fp_round_normal:NnnwNNnn .. 16659
- __fp_round_normal:NNwNnn 16659
- __fp_round_normal:NwNNnw 16659
- __fp_round_normal_end:wwNnn . 16659
- __fp_round_o:Nw
.. 16463, 16465, 16467, 16471, 16595
- __fp_round_pack:Nw 16659
- __fp_round_return_one:
..... 744, 16478, 16484,

- 16494, 16502, 16506, 16515, 16519,
 16528, 16535, 16539, 16577, 16588
 __fp_round_s:NNWw
 .. 744, 746, 773, 16546, 17238, 17256
 __fp_round_special:NwNnn 16659
 __fp_round_special_aux:Nw 16659
 __fp_round_to_nearest:NNN
 747, 748, 16471, 16474,
 16478, 16582, 16614, 16624, 21732
 __fp_round_to_nearest_neg:NNN 16573
 __fp_round_to_nearest_ninf:NNN .
 748, 16478, 16593
 __fp_round_to_nearest_ninf_-
 neg:NNN 16573
 __fp_round_to_nearest_pinf:NNN .
 748, 16478, 16584
 __fp_round_to_nearest_pinf_-
 neg:NNN 16573
 __fp_round_to_nearest_zero:NNN .
 748, 16478
 __fp_round_to_nearest_zero_-
 neg:NNN 16573
 __fp_round_to_ninf:NNN
 16465, 16478, 16581, 16652
 __fp_round_to_ninf_neg:NNN .. 16573
 __fp_round_to_pinf:NNN
 16467, 16478, 16573, 16654
 __fp_round_to_pinf_neg:NNN .. 16573
 __fp_round_to_zero:NNN
 16463, 16478, 16650
 __fp_round_to_zero_neg:NNN .. 16573
 __fp_rrrot:www 15892, 21336
 __fp_sanitize:Nw
 812, 814, 820, 822, 831,
 877, 893, 900, 917, 15950, 16728,
 16746, 18679, 18773, 18945, 19026,
 19174, 19899, 20142, 20384, 20576,
 21177, 21221, 21348, 21864, 21959
 __fp_sanitize:wN
 764, 768, 15950, 16951, 17453
 __fp_sanitize_zero:w 15950
 __fp_sec_o:w 20687
 __fp_set_sign_o:w
 .. 17446, 18605, 19376, 19377, 19398
 __fp_show:NN 18152
 __fp_sign_aux_o:w 19365
 __fp_sign_o:w 18609, 19365
 __fp_sin_o:w 737, 778, 778, 901, 20642
 __fp_sin_series_aux_o:NNwww . 21129
 __fp_sin_series_o:NNwww .. 880,
 894, 20648, 20663, 20678, 20693, 21129
 __fp_small_int:wTF
 876, 16218, 16661, 20563
 __fp_small_int_normal:NwTF . 16218
 __fp_small_int_test:NnnwNTF . 16218
 __fp_small_int_test:NnnwNw
 16237, 16240
 __fp_small_int_true:wTF 16218
 __fp_sqrt_auxi_o:NNNNwnnN
 19196, 19204
 __fp_sqrt_auxii_o:NnnnnnnnN ...
 832, 834, 19206, 19210, 19290, 19302
 __fp_sqrt_auxiii_o:wnnnnnnnn ...
 19207, 19245, 19291
 __fp_sqrt_auxiv_o:NNNNNw 19245
 __fp_sqrt_auxix_o:wnwnw 19279
 __fp_sqrt_auxv_o:NNNNNw 19245
 __fp_sqrt_auxvi_o:NNNNNw 19245
 __fp_sqrt_auxvii_o:NNNNNw ... 19245
 __fp_sqrt_auxviii_o:nnnnnnnn ...
 .. 19267, 19269, 19271, 19277, 19279
 __fp_sqrt_auxx_o:Nnnnnnnn
 19275, 19293
 __fp_sqrt_auxxi_o:wnnnN 19293
 __fp_sqrt_auxxii_o:nnnnnnnnw ...
 19303, 19307
 __fp_sqrt_auxxiii_o:w 19307
 __fp_sqrt_auxxiv_o:wnnnnnnnN ...
 19319, 19322, 19330, 19332
 __fp_sqrt_Newton_o:wnw
 832, 19181, 19192, 19193
 __fp_sqrt_npos_auxi_o:wnnnN . 19172
 __fp_sqrt_npos_auxii_o:wnnnnnnnN
 19172
 __fp_sqrt_npos_o:w ... 19169, 19172
 __fp_sqrt_o:w 18611, 19162
 __fp_step:NNnnnn 18442
 __fp_step:NnnnnN 18372
 __fp_step:wwwN 18372
 __fp_step_fp:wwwN 18372
 __fp_str_if_eq:nn 16255,
 17364, 17378, 17666, 17710, 20356
 __fp_sub_back_far_o:NnnwnnnnN ..
 816, 18782, 18828
 __fp_sub_back_near_after:wnnnNw
 18788, 18866
 __fp_sub_back_near_o:nnnnnnnnN .
 815, 18778, 18788
 __fp_sub_back_near_pack:NNNNNw
 18788, 18868
 __fp_sub_back_not_far_o:wwwNN .
 18843, 18863
 __fp_sub_back_quite_far_ii:NN 18847
 __fp_sub_back_quite_far_o:wwNN .
 18841, 18847
 __fp_sub_back_shift:wnnnn
 815, 18800, 18804
 __fp_sub_back_shift_ii:ww ... 18804

- __fp_sub_back_shift_iii:NNNNNNNNw [18804](#)
- __fp_sub_back_shift_iv:nnnnw . [18804](#)
- __fp_sub_back_very_far_ii-
o:nnNwwNN [18875](#)
- __fp_sub_back_very_far_o:wwwNN
..... [18842](#), [18875](#)
- __fp_sub_eq_o:Nnnnw [18753](#)
- __fp_sub_npos_i_o:Nnnnw
..... [814](#), [18758](#), [18767](#), [18771](#)
- __fp_sub_npos_ii_o:Nnnnw [18753](#)
- __fp_sub_npos_o:NnnNnw
..... [814](#), [18673](#), [18753](#)
- __fp_tan_o:w [20702](#)
- __fp_tan_series_aux_o:Nnnww . [21183](#)
- __fp_tan_series_o:NNwww
..... [881](#), [882](#), [20709](#), [20724](#), [21183](#)
- __fp_ternary:NwnN . [798](#), [17878](#), [18556](#)
- __fp_ternary_auxi:NwnN
..... [798](#), [808](#), [18556](#)
- __fp_ternary_auxii:NwnN
..... [798](#), [808](#), [17880](#), [18556](#)
- __fp_tmp:w [733](#), [788](#),
[16160](#), [16170](#), [16171](#), [16172](#), [16173](#),
[16174](#), [16175](#), [16176](#), [16177](#), [16178](#),
[16179](#), [16180](#), [16181](#), [16182](#), [16183](#),
[16184](#), [16185](#), [16275](#), [16277](#), [16769](#),
[16781](#), [16782](#), [16783](#), [16784](#), [16785](#),
[16786](#), [16787](#), [16845](#), [16867](#), [17429](#),
[17446](#), [17447](#), [17503](#), [17508](#), [17509](#),
[17510](#), [17511](#), [17512](#), [17513](#), [17517](#),
[17525](#), [17526](#), [17527](#), [17528](#), [17529](#),
[17530](#), [17531](#), [17532](#), [17533](#), [17534](#),
[17535](#), [17739](#), [17755](#), [17756](#), [17779](#),
[17791](#), [17809](#), [17811](#), [17813](#), [17815](#),
[17817](#), [17819](#), [17821](#), [17823](#), [17827](#),
[17841](#), [17842](#), [17857](#), [17858](#), [17859](#),
[17877](#), [17879](#), [19408](#), [19422](#), [19423](#)
- __fp_to_decimal:w
.. [21587](#), [21597](#), [21714](#), [21731](#), [22218](#)
- __fp_to_decimal_dispatch:w [906](#),
[909](#), [910](#), [18435](#), [21577](#), [21581](#), [21584](#)
- __fp_to_decimal_huge:wnnnn .. [21597](#)
- __fp_to_decimal_large:Nnnw .. [21597](#)
- __fp_to_decimal_normal:wnnnnn ..
..... [21597](#), [21685](#)
- __fp_to_decimal_recover:w ... [21584](#)
- __fp_to_dim:w [21699](#)
- __fp_to_dim_dispatch:w .. [910](#), [21699](#)
- __fp_to_dim_recover:w [21699](#)
- __fp_to_int:w [910](#), [21724](#), [21729](#)
- __fp_to_int_dispatch:w [21715](#)
- __fp_to_int_recover:w [21715](#)
- __fp_to_scientific:w
..... [907](#), [21533](#), [21543](#)
- __fp_to_scientific_dispatch:w ..
..... [905](#), [909](#), [21523](#), [21527](#), [21530](#)
- __fp_to_scientific_normal:wnnnnn
..... [21543](#)
- __fp_to_scientific_normal:wNw [21543](#)
- __fp_to_scientific_recover:w . [21530](#)
- __fp_to_tl:w ... [21663](#), [21671](#), [22226](#)
- __fp_to_tl_dispatch:w
[904](#), [908](#), [21655](#), [21659](#), [21662](#), [21795](#)
- __fp_to_tl_normal:nnnnn [21671](#)
- __fp_to_tl_recover:w [21662](#)
- __fp_to_tl_scientific:wnnnnn . [21671](#)
- __fp_to_tl_scientific:wNw ... [21671](#)
- \c__fp_trailing_shift_int
..... [16105](#), [19440](#),
[19462](#), [19535](#), [20436](#), [21075](#), [21112](#)
- __fp_trap_division_by_zero_-
set:N [16345](#)
- __fp_trap_division_by_zero_set_-
error: [16345](#)
- __fp_trap_division_by_zero_set_-
flag: [16345](#)
- __fp_trap_division_by_zero_set_-
none: [16345](#)
- __fp_trap_invalid_operation_-
set:N [16311](#)
- __fp_trap_invalid_operation_-
set_error: [16311](#)
- __fp_trap_invalid_operation_-
set_flag: [16311](#)
- __fp_trap_invalid_operation_-
set_none: [16311](#)
- __fp_trap_overflow_set:N [16371](#)
- __fp_trap_overflow_set:NnNn . [16371](#)
- __fp_trap_overflow_set_error: [16371](#)
- __fp_trap_overflow_set_flag: . [16371](#)
- __fp_trap_overflow_set_none: . [16371](#)
- __fp_trap_underflow_set:N ... [16371](#)
- __fp_trap_underflow_set_error: .
..... [16371](#)
- __fp_trap_underflow_set_flag: [16371](#)
- __fp_trap_underflow_set_none: [16371](#)
- __fp_trig:NNNNNwn . [20648](#), [20663](#),
[20678](#), [20693](#), [20708](#), [20723](#), [20740](#)
- \c__fp_trig_intarray [889](#),
[20801](#), [21031](#), [21034](#), [21037](#), [21040](#),
[21043](#), [21046](#), [21049](#), [21052](#), [21055](#)
- __fp_trig_large:ww ... [20748](#), [21015](#)
- __fp_trig_large_auxi:w [21015](#)
- __fp_trig_large_auxii:w . [889](#), [21015](#)
- __fp_trig_large_auxiii:w [889](#), [21015](#)
- __fp_trig_large_auxix:Nw [21088](#)

- __fp_trig_large_auxv:ww [21065](#), [21068](#)
 - __fp_trig_large_auxvi:wnnnnnnnn [21068](#)
 - __fp_trig_large_auxvii:w [21071](#), [21088](#)
 - __fp_trig_large_auxviii:w ... [21088](#)
 - __fp_trig_large_auxviii:ww [21090](#), [21094](#)
 - __fp_trig_large_auxx:wNNNNN . [21088](#)
 - __fp_trig_large_auxxi:w [21088](#)
 - __fp_trig_large_pack:NNNNNw ... [21068](#), [21117](#)
 - __fp_trig_small:ww [883](#), [891](#), [20750](#), [20754](#), [20760](#), [21127](#)
 - __fp_trigd_large:ww .. [20748](#), [20762](#)
 - __fp_trigd_large_auxi:nnnnwNNNN [20762](#)
 - __fp_trigd_large_auxii:wNw .. [20762](#)
 - __fp_trigd_large_auxiii:www . [20762](#)
 - __fp_trigd_small:ww [884](#), [20750](#), [20756](#), [20799](#)
 - __fp_trim_zeros:w [21514](#), [21638](#), [21647](#), [21698](#)
 - __fp_trim_zeros_dot:w [21514](#)
 - __fp_trim_zeros_end:w [21514](#)
 - __fp_trim_zeros_loop:w [21514](#)
 - __fp_tuple_ [18546](#), [18547](#), [18550](#), [18551](#)
 - __fp_tuple_&o:ww [18529](#)
 - __fp_tuple_&tuple_o:ww [18529](#)
 - __fp_tuple_*o:ww [19402](#)
 - __fp_tuple_+tuple_o:ww [19408](#)
 - __fp_tuple_-tuple_o:ww [19408](#)
 - __fp_tuple_/o:ww [19402](#)
 - __fp_tuple_chk:w [726](#), [727](#), [16008](#), [16014](#), [16015](#), [16092](#), [16095](#), [17788](#), [18000](#), [18015](#), [18040](#), [18043](#), [18059](#), [18060](#), [18063](#), [18270](#), [18271](#), [19411](#), [19412](#), [19418](#), [19419](#), [21493](#)
 - __fp_tuple_compare_back:ww .. [18267](#)
 - __fp_tuple_compare_back_loop:w . [18267](#)
 - __fp_tuple_compare_back_-tuple:ww [18267](#)
 - __fp_tuple_convert:Nw [21493](#), [21542](#), [21596](#), [21670](#)
 - __fp_tuple_convert_end:w [21493](#)
 - __fp_tuple_convert_loop:nNw . [21493](#)
 - __fp_tuple_count:w [16013](#)
 - __fp_tuple_count_loop:Nw [16013](#)
 - __fp_tuple_map_loop_o:nw [18040](#)
 - __fp_tuple_map_o:nw [18040](#), [19395](#), [19403](#), [19405](#), [19407](#)
 - __fp_tuple_mapthread_loop_o:nw . [18058](#), [19416](#)
 - __fp_tuple_mapthread_o:nww [18058](#), [19416](#)
 - __fp_tuple_not_o:w [18520](#)
 - __fp_tuple_set_sign_aux_o:Nnw [19387](#)
 - __fp_tuple_set_sign_aux_o:w . [19387](#)
 - __fp_tuple_set_sign_o:w [19387](#)
 - __fp_tuple_to_decimal:w [21584](#)
 - __fp_tuple_to_scientific:w .. [21530](#)
 - __fp_tuple_to_tl:w [21662](#)
 - __fp_tuple_|o:ww [18529](#)
 - __fp_tuple_|tuple_o:ww [18529](#)
 - __fp_type_from_scan:N [728](#), [16037](#), [17610](#), [17612](#), [17636](#), [17638](#), [17649](#), [17651](#), [18231](#), [18233](#)
 - __fp_type_from_scan:w [16037](#)
 - __fp_type_from_scan_other:N ... [16037](#), [16061](#), [16079](#)
 - __fp_underflow:w [725](#), [738](#), [740](#), [15957](#), [16402](#), [20139](#)
 - __fp_use_i:ww [848](#), [902](#), [15893](#), [19652](#), [21422](#)
 - __fp_use_i:www [15893](#)
 - __fp_use_i_delimit_by_s_stop:nw [15904](#), [18199](#), [18561](#)
 - __fp_use_i_until_s:nw [891](#), [15888](#), [15937](#), [15947](#), [16210](#), [20792](#), [21070](#), [21076](#), [21107](#), [21882](#), [21953](#), [22164](#)
 - __fp_use_ii_until_s:nnw [15888](#), [15935](#), [15946](#)
 - __fp_use_none_stop_f:n [15885](#), [19817](#), [19818](#), [19819](#)
 - __fp_use_none_until_s:w [15888](#), [19198](#), [20496](#), [21417](#), [21420](#)
 - __fp_use_s:n [15886](#)
 - __fp_use_s:nn [15886](#)
 - __fp_zero_fp:N . [15928](#), [16386](#), [16734](#)
 - __fp_|o:ww [798](#), [18529](#)
 - __fp_|tuple_o:ww [18529](#)
 - __fp_ [18532](#), [18539](#), [18548](#), [18549](#)
 - fpararray commands:
 - \fpararray_count:N [222](#), [222](#), [222](#), [22124](#), [22136](#), [22147](#), [22203](#)
 - \fpararray_gset:Nnn ... [222](#), [923](#), [22149](#)
 - \fpararray_gzero:N [222](#), [22200](#)
 - \fpararray_item:Nn [222](#), [923](#), [22213](#)
 - \fpararray_item_to_tl:Nn ... [222](#), [22213](#)
 - \fpararray_new:Nn [222](#), [22097](#)
 - \futurelet [276](#)
- G**
- \gdef [277](#)
 - \GetIdInfo [7](#), [13803](#)

- \gleaders 825
- \global 179, 278
- \globaldefs 279
- \glueexpr 533
- \glueshrink 534
- \glueshrinkorder 535
- \gluestretch 536
- \gluestretchorder 537
- \gluetomu 538
- group commands:
 - \group_align_safe_begin/end: 534, 956
 - \group_align_safe_begin:
 - 115, 386, 390, 526, 3784,
 - 4171, 9018, 9221, 10926, 10941,
 - 23091, 29209, 29541, 30646, 31726
 - \group_align_safe_end:
 - 115, 386, 390, 3807, 4197,
 - 9020, 9221, 10935, 10946, 10952,
 - 23094, 29221, 29553, 30657, 31734
 - \group_begin: 9, 381,
 - 1112, 1388, 2117, 2120, 2123, 2504,
 - 2975, 3166, 3562, 3658, 3881, 3894,
 - 4560, 4587, 4863, 4886, 5274, 5384,
 - 5437, 5610, 5656, 5747, 5795, 5841,
 - 5848, 6175, 6357, 7605, 7642, 9357,
 - 9417, 10267, 10273, 10324, 10401,
 - 10645, 10663, 10687, 10775, 10793,
 - 11047, 11561, 11585, 11601, 11667,
 - 11968, 12321, 12558, 12751, 12797,
 - 13059, 13217, 13810, 14407, 14553,
 - 15525, 18529, 22346, 22552, 22749,
 - 22786, 23090, 23097, 23170, 23330,
 - 23672, 23765, 24080, 24579, 24943,
 - 25036, 25427, 25785, 25996, 26156,
 - 26165, 26177, 26186, 26194, 26312,
 - 26342, 26833, 28637, 28863, 28922,
 - 29006, 29033, 30284, 30291, 30318,
 - 30550, 30571, 30602, 30877, 30964,
 - 31711, 32089, 32148, 32157, 32168
 - \c_group_begin_token
 - 55, 135, 274, 402, 574, 4303,
 - 4340, 10645, 10669, 23135, 26876,
 - 26882, 26896, 26902, 26978, 26984,
 - 26999, 27005, 29089, 29294, 31749
 - \group_end: ... 9, 9, 487, 930, 933,
 - 1112, 1388, 2117, 2120, 2126, 2513,
 - 2978, 3169, 3568, 3680, 3730, 3884,
 - 3898, 4578, 4610, 4868, 4891, 5284,
 - 5397, 5440, 5623, 5665, 5760, 5807,
 - 5866, 5928, 6356, 6488, 7614, 7652,
 - 7657, 9374, 9445, 10275, 10282,
 - 10404, 10420, 10662, 10666, 10694,
 - 10792, 10840, 11071, 11580, 11593,
 - 11612, 11825, 12014, 12337, 12564,
 - 12755, 12826, 13082, 13235, 13813,
 - 14527, 14591, 15539, 18553, 22350,
 - 22582, 22753, 22794, 23003, 23095,
 - 23116, 23177, 23354, 23685, 23779,
 - 24113, 24121, 24592, 25001, 25043,
 - 25050, 25058, 25431, 25432, 25822,
 - 26060, 26161, 26172, 26256, 26318,
 - 26379, 26839, 28638, 28931, 29003,
 - 29020, 29049, 30309, 30317, 30564,
 - 30570, 30597, 30628, 30881, 31234,
 - 31715, 32097, 32151, 32161, 32172
- \c_group_end_token
 - 135, 274, 574, 10645, 10674, 23138,
 - 26890, 26993, 29090, 29300, 31750
- \group_insert_after:N 9, 1394, 23681
- groups commands:
 - .groups:n 189, 15026
- H
- \H 29167, 31007, 31154, 31155, 31182, 31183
- \halign 280
- \hangafter 281
- \hangindent 282
- \hbadness 283
- \hbox 284
- hbox commands:
 - \hbox:n 243, 26847, 27072, 27368, 28441
 - \hbox_gset:Nn
 - 244, 26849, 27039, 27162,
 - 27206, 27226, 27246, 27263, 27284,
 - 27313, 27324, 27482, 27910, 31253
 - \hbox_gset:Nw 244, 26873, 27552
 - \hbox_gset_end: ... 244, 26873, 27555
 - \hbox_gset_to_wd:Nnn 244, 26861
 - \hbox_gset_to_wd:Nnw 244, 26893
 - \hbox_overlap_left:n 244, 26917
 - \hbox_overlap_right:n ... 244, 26917
 - \hbox_set:Nn 244, 244, 26849,
 - 27036, 27068, 27069, 27156, 27203,
 - 27223, 27230, 27243, 27260, 27281,
 - 27310, 27318, 27341, 27469, 27907,
 - 27930, 28189, 28276, 28573, 31250,
 - 31263, 31271, 31279, 31288, 31297,
 - 31314, 31322, 31330, 31336, 31349
 - \hbox_set:Nw 244, 26873, 27539
 - \hbox_set_end: 244, 244, 26873, 27542
 - \hbox_set_to_wd:Nnn . 244, 244, 26861
 - \hbox_set_to_wd:Nnw 244, 26893
 - \hbox_to_wd:nn 244, 26907, 27359
 - \hbox_to_zero:n
 - 244, 26907, 26918, 26920
 - \hbox_unpack:N 244, 26921, 28193
 - \hbox_unpack_clear:N 32116

- \hbox_unpack_drop:N 247, 26921, 32116, 32118
 - hcoffin commands:
 - \hcoffin_gset:Nn 251, 27465
 - \hcoffin_gset:Nw 252, 27535
 - \hcoffin_gset_end: 252, 27535
 - \hcoffin_set:Nn 251, 27465, 28445, 28452, 28490, 28525
 - \hcoffin_set:Nw 252, 27535
 - \hcoffin_set_end: 252, 27535
 - \hfi 1116
 - \hfil 285
 - \hfill 286
 - \hfilneg 287
 - \hfuzz 288
 - \hjcode 820
 - \hoffset 289
 - \holdinginserts 290
 - hook commands:
 - \hook_gput_code:nnn ... 22475, 22477
 - \hpack 821
 - \hrule 291
 - \hsize 292
 - \hskip 293
 - \hss 294
 - \ht 295
 - \Huge 30858
 - \huge 30862
 - hundred commands:
 - \c_one_hundred 31953
 - \hyphenation 296
 - \hyphenationbounds 822
 - \hyphenationmin 823
 - \hyphenchar 297
 - \hyphenpenalty 298
 - \hyphenpenaltymode 824
- I
- \i 30595, 30937, 31063, 31065, 31067, 31069, 31120, 31123, 31126, 31129, 31200
 - \if 299
 - if commands:
 - \if:w 23, 130, 322, 323, 359, 985, 1359, 1716, 2011, 2012, 2867, 2870, 2871, 2872, 2873, 2888, 2889, 2890, 2891, 2892, 2893, 2894, 2895, 2896, 2960, 2961, 2963, 8739, 10887, 16663, 17036, 17040, 17062, 17155, 17187, 17206, 17272, 17286, 17303, 17324, 17831, 17846, 18184, 20387, 21894, 23981, 28936, 28944, 28961, 29073
 - \if_bool:N 114, 114, 523, 1369, 8931, 8976
 - \if_box_empty:N ... 250, 26785, 26797
 - \if_case:w 102, 419, 421, 457, 514, 734, 822, 875, 917, 1919, 2558, 4961, 5035, 5260, 6302, 8048, 8630, 8663, 10413, 12911, 15952, 16205, 16220, 16603, 16632, 17919, 17960, 18620, 18755, 18830, 18855, 18907, 19351, 19367, 19384, 19661, 19888, 19915, 20073, 20108, 20266, 20311, 20363, 20489, 20512, 20545, 20604, 20644, 20659, 20674, 20689, 20704, 20719, 21245, 21298, 21382, 21397, 21449, 21462, 21546, 21600, 21674, 21891, 22178, 22257, 23146, 23340, 23631, 23895, 24696, 24725, 24782, 25192, 25246, 25606, 25937, 31743
 - \if_catcode:w 23, 391, 402, 403, 584, 1359, 3007, 3941, 4294, 4338, 4350, 4367, 10581, 10584, 10587, 10590, 10593, 10596, 10599, 10669, 10674, 10679, 10684, 10691, 10698, 10703, 10708, 10713, 10718, 10723, 10733, 10760, 10978, 10983, 11053, 11054, 16791, 16996, 17314, 17361, 17664, 17708, 23135, 23138, 23292, 23294, 23296, 23298, 23300, 23302, 23304, 29037, 29038, 29074, 29089, 29090, 29091, 29092, 29093, 29094, 29095, 29096, 29097, 29120, 29123, 29126, 29129, 29132, 29135, 29138, 31744, 31745
 - \if_charcode:w 23, 130, 401, 402, 424, 584, 961, 1359, 4277, 4331, 5119, 5240, 5917, 10738, 10980, 13600, 13609, 16208, 18197, 18559, 23205, 23229, 23278, 23838, 23848, 24330, 25791
 - \if_cs_exist:N 23, 1374, 1743, 1771, 2507, 10768, 10896
 - \if_cs_exist:w 23, 1374, 1402, 1752, 1780, 1906, 8879, 8907, 8916, 31361
 - \if_dim:w 185, 13890, 13978, 13990, 14013, 14184
 - \if_eof:w 166, 628, 12501, 12508, 12591, 12609
 - \if_false: 23, 108, 349, 381, 386, 390, 400, 489, 505, 534, 568, 647, 953, 1042, 1359, 2518, 2528, 2541, 2554, 2582, 2598, 2696, 2710, 2716, 2723, 2731, 2741, 2754, 2758, 3659, 3666, 3802, 3803, 3913, 3917, 3956, 4244,

4249, 4260, 4351, 4363, 4378, 4386,
 7548, 7551, 7730, 7735, 8260, 9222,
 9358, 9366, 10365, 10409, 12890,
 12930, 12934, 12941, 12949, 13216,
 13229, 14000, 23082, 23124, 23173,
 23176, 24089, 24108, 24109, 24118,
 24169, 24205, 24219, 24223, 24446,
 24479, 24491, 24495, 24529, 24534,
 24542, 24577, 24584, 24589, 24637,
 24860, 24877, 24881, 26012, 26029,
 26268, 26273, 26384, 26389, 29296,
 29302, 31616, 31628, 31654, 31664
 \if_hbox:N 250, 26785, 26789
 \if_int_compare:w
 22, 102, 505, 506, 1392,
 2746, 4443, 4452, 4501, 4502, 4508,
 4695, 4704, 4709, 4945, 5000, 5001,
 5007, 5019, 5035, 5249, 5257, 5464,
 5465, 5466, 5471, 5472, 5514, 5566,
 5647, 5648, 5679, 5682, 5683, 5693,
 5882, 5944, 5948, 5978, 5981, 5997,
 6001, 6022, 6100, 6102, 6121, 6122,
 6140, 6142, 6196, 6199, 6200, 6318,
 6319, 6460, 6465, 8048, 8103, 8144,
 8145, 8240, 8293, 8295, 8297, 8299,
 8301, 8303, 8305, 8308, 8441, 9222,
 9224, 10296, 10297, 10304, 10305,
 10306, 10307, 10312, 10313, 10353,
 10428, 10429, 10435, 10878, 12897,
 13616, 14029, 15677, 15680, 15724,
 15790, 15953, 15954, 16148, 16245,
 16483, 16493, 16501, 16514, 16527,
 16534, 16555, 16567, 16576, 16587,
 16696, 16701, 16773, 16803, 16956,
 16958, 16995, 17000, 17053, 17073,
 17100, 17114, 17149, 17176, 17204,
 17220, 17236, 17254, 17314, 17334,
 17350, 17363, 17377, 17438, 17461,
 17490, 17492, 17665, 17675, 17677,
 17709, 17719, 17721, 17743, 17760,
 17765, 17795, 17863, 17908, 18208,
 18255, 18258, 18289, 18298, 18301,
 18306, 18307, 18310, 18313, 18500,
 18624, 18645, 18682, 18777, 18831,
 18832, 18835, 18838, 18908, 18917,
 19122, 19195, 19248, 19252, 19256,
 19274, 19309, 19310, 19311, 19312,
 19313, 19339, 19663, 19666, 19760,
 19853, 19901, 19917, 20044, 20078,
 20136, 20145, 20185, 20356, 20358,
 20369, 20387, 20410, 20442, 20445,
 20492, 20522, 20569, 20583, 20747,
 20791, 21249, 21287, 21296, 21332,
 21416, 21419, 21648, 21881, 21949,
 21950, 21951, 21961, 21989, 21994,
 21995, 22058, 22059, 22060, 22064,
 22079, 22084, 22132, 22136, 22674,
 22731, 22760, 22801, 22812, 22815,
 22833, 22888, 22898, 22908, 23110,
 23182, 23215, 23223, 23246, 23270,
 23321, 23336, 23418, 23421, 23569,
 23575, 23576, 23583, 23586, 23589,
 23595, 23596, 23600, 23603, 23604,
 23612, 23613, 23614, 23620, 23650,
 23651, 23892, 23912, 23913, 23914,
 23917, 23921, 23922, 23925, 23926,
 23934, 23935, 23938, 23942, 23943,
 23946, 24005, 24027, 24039, 24048,
 24056, 24059, 24069, 24072, 24100,
 24173, 24278, 24344, 24349, 24377,
 24444, 24477, 24588, 24605, 24891,
 24924, 25139, 25210, 25236, 25298,
 25311, 25322, 25338, 25389, 25421,
 25584, 25585, 25632, 25659, 25734,
 25802, 25865, 25875, 25878, 25898,
 25955, 26008, 26025, 26048, 26197,
 26226, 26266, 26271, 26292, 26370,
 26382, 26387, 28937, 29106, 30002,
 30005, 30006, 30009, 31697, 31705
 \if_int_odd:w 103,
 894, 8048, 8175, 8346, 8354, 8854,
 10303, 10311, 10328, 11052, 16505,
 16552, 16564, 17956, 18891, 19177,
 20533, 21097, 21136, 21146, 21189,
 21213, 21373, 22057, 23346, 23640,
 24016, 24024, 24036, 24450, 24765
 \if_meaning:w 23,
 371, 387, 388, 402, 806, 1123, 1359,
 1570, 1596, 1614, 1673, 1678, 1687,
 1740, 1758, 1768, 1786, 1937, 1951,
 2072, 2168, 2231, 2232, 2559, 2562,
 2563, 2564, 2565, 2658, 2688, 2701,
 2707, 2815, 2838, 2847, 3039, 3112,
 3124, 3125, 3233, 3239, 3265, 3277,
 3285, 3317, 3324, 3348, 3352, 3430,
 3455, 3470, 3835, 3845, 3856, 3869,
 3885, 3899, 4189, 4253, 4322, 4804,
 4872, 4895, 5056, 5094, 5410, 6150,
 6299, 6314, 6341, 6456, 7584, 7647,
 7662, 7670, 8071, 8074, 8084, 8119,
 8124, 8125, 8275, 9030, 9052, 9722,
 9737, 9759, 9773, 10728, 10765,
 10803, 10806, 10870, 10932, 10971,
 11055, 11358, 12847, 13959, 14006,
 14187, 15934, 15955, 15967, 15977,
 16072, 16127, 16136, 16227, 16242,
 16244, 16394, 16482, 16492, 16504,
 16517, 16518, 16537, 16538, 16552,

- 16553, 16564, 16565, 16631, 16678,
 16713, 16716, 16732, 16739, 16792,
 16795, 16911, 16912, 16913, 16914,
 16917, 17013, 17126, 17132, 17362,
 17410, 17621, 17694, 17957, 17975,
 18025, 18035, 18244, 18245, 18246,
 18247, 18248, 18249, 18481, 18493,
 18494, 18522, 18534, 18541, 18558,
 18621, 18656, 18670, 18716, 18723,
 18799, 18811, 18911, 18914, 18925,
 18978, 19051, 19121, 19124, 19131,
 19164, 19165, 19168, 19389, 19634,
 19645, 19826, 19836, 19885, 19984,
 20064, 20113, 20127, 20274, 20308,
 20320, 20333, 20336, 20339, 20342,
 20368, 20469, 20473, 20532, 20549,
 20555, 21139, 21192, 21243, 21244,
 21246, 21247, 21267, 21284, 21351,
 21449, 21545, 21599, 21673, 21743,
 21748, 21880, 21912, 21923, 22030,
 22191, 22244, 22250, 22684, 22685,
 23132, 23162, 23190, 23290, 23680,
 23980, 24004, 24326, 24329, 24772,
 25174, 25346, 25357, 25372, 25527,
 25560, 25910, 26148, 26225, 26278,
 26317, 28913, 28915, 28926, 29052,
 31240, 31621, 31658, 31677, 31746
 \if_mode_horizontal: . 23, 1370, 9216
 \if_mode_inner: 23, 1370, 9218
 \if_mode_math: 23, 1370, 9220
 \if_mode_vertical: 23, 1370, 2178, 9214
 \if_predicate:w 106, 108, 114, 8931,
 9008, 9068, 9083, 9094, 9109, 9120
 \if_true: 23, 108, 388, 1359
 \if_vbox:N 250, 26785, 26791
 \ifabsdim 911
 \ifabsnum 912
 \ifcase 300
 \ifcat 301
 \ifcondition 826
 \ifcsname 539
 \ifdbbox 1117
 \ifddir 1118
 \ifdefined 540
 \ifdim 302
 \ifeof 303
 \iffalse 304
 \iffontchar 541
 \ifhbox 305
 \ifhmode 306
 \ifincsname 690
 \ifinner 307
 \ifjfont 1119
 \ifmbox 1120
 \ifmdir 1121
 \ifmmode 308
 \ifnum 27, 34, 63, 76, 81, 309
 \ifodd 310
 \ifpdfabsdim 646
 \ifpdfabsnum 647
 \ifpdfprimitive 648
 \ifprimitive 778
 \iftbox 1122
 \iftdir 1124
 \iftfont 1123
 \iftrue 311, 31240
 \ifvbox 312
 \ifvmode 313
 \ifvoid 314
 \ifx 4, 21, 25,
 30, 64, 66, 67, 68, 79, 86, 106, 107, 315
 \ifybox 1125
 \ifydir 1126
 \ignoreligaturesinfont 913
 \ignorespaces 316
 \IJ 29175, 30586, 30927
 \ij 29175, 30586, 30939
 \immediate 317
 \immediateassigned 827
 \immediateassignment 828
 in 219
 \indent 318
 inf 218
 \infty 16914, 16915
 inherit commands:
 .inherit:n 189, 15028
 \inhibitglue 1127
 \inhibitxspcode 1128
 \initcatcodetable 829
 initial commands:
 .initial:n 190, 15030
 \input 31, 319
 \inputlineno 320
 \insert 321
 \inserttht 914
 \insertpenalties 322
 int commands:
 \c_eight 31933
 \c_eleven 31939
 \c_fifteen 31947
 \c_five 31927
 \l_foo_int 235
 \c_four 31925
 \c_fourteen 31945
 \int_abs:n 91, 499, 8077, 15724
 \int_add:Nn 92, 8205, 13006, 23621,
 24767, 25328, 25329, 25569, 25656

- \int_case:nn [95](#), [514](#), [8314](#),
8493, 8499, 22622, 30036, 30051, 30114
- \int_case:nnn [31979](#)
- \int_case:nnTF
..... [95](#), 7962, [8314](#), 8319, 8324,
10035, 16823, 21495, 26668, 31980
- \int_compare:nNnTF
..... [93](#), [93](#), [94](#), [95](#), [95](#),
[96](#), [96](#), [205](#), 3661, 3689, 3704, 3712,
4398, 4405, 4472, 4924, 4926, 4935,
6500, 7599, 7788, 7795, 8156, 8162,
[8306](#), 8338, 8390, 8398, 8407, 8413,
8425, 8428, 8489, 8577, 8583, 8589,
8609, 8763, 8782, 8784, 8826, 9512,
10074, 10076, 10081, 10090, 10110,
10127, 10547, 10631, 12706, 12819,
13310, 13453, 13463, 14208, 15662,
15667, 15674, 15782, 15824, 15850,
18273, 19414, 21478, 21623, 21625,
22112, 22322, 22510, 22661, 23465,
23658, 23670, 23824, 24142, 24144,
24937, 25530, 25752, 25767, 25967,
26242, 26685, 29773, 29851, 29853,
29856, 29897, 29910, 29919, 29960,
29982, 29992, 30190, 30193, 30233,
30239, 30246, 30260, 31448, 31776,
31778, 31779, 31780, 31782, 31805
- \int_compare:nTF
.. [93](#), [94](#), [96](#), [96](#), [96](#), [96](#), [206](#), [667](#),
[8253](#), 8362, 8370, 8379, 8385, 12479,
12506, 12676, 21683, 22516, 25041,
26425, 26647, 26648, 26653, 26655
- \int_compare_p:n [94](#), [8253](#), 25048
- \int_compare_p:nN [22](#), [93](#),
[8306](#), 9428, 9491, 9493, 9495, 12621,
13442, 13443, 22448, 24836, 24837,
29968, 29969, 29970, 29971, 29972,
29973, 29974, 29975, 29976, 29977,
29978, 30094, 30095, 30096, 30144,
30152, 30153, 30174, 30175, 30217
- \int_const:Nn
..... [92](#), 5200, 5201, [8154](#), 8792,
8793, 8794, 8795, 8796, 8797, 8798,
8799, 8800, 8801, 8802, 8803, 8804,
8805, 8850, 8851, 8852, 9378, 9438,
9440, 9442, 9443, 9444, 9478, 12410,
12551, 12616, 12617, 15915, 15916,
15917, 15918, 15919, 15920, 15921,
16105, 16106, 16107, 16109, 16110,
16111, 16114, 16115, 16116, 16477,
16751, 16752, 16753, 16754, 16755,
16756, 16757, 16758, 16759, 16760,
16761, 16762, 16763, 16764, 16765,
20542, 21831, 23548, 23549, 23550,
23551, 23953, 23954, 23955, 23956,
23957, 23958, 23962, 23963, 23964,
23965, 23966, 23967, 23968, 23969,
23970, 23971, 23972, 23973, 23974
- \int_decr:N [92](#),
[8217](#), 22821, 22822, 22823, 22886,
22887, 22896, 22897, 22906, 22907,
23125, 25608, 25957, 26026, 26227
- \int_div_round:nn [91](#), [8109](#)
- \int_div_truncate:nn [91](#),
[91](#), 5305, 5310, 5971, 5972, 6027,
6207, 6373, 6384, [8109](#), 8504, 8602,
8622, 9441, 10441, 10454, 10459, 10471
- \int_do_until:nn [96](#), [8360](#)
- \int_do_until:nNnn [95](#), [8388](#)
- \int_do_while:nn [96](#), [8360](#)
- \int_do_while:nNnn [95](#), [8388](#)
- \int_eval:n [14](#), [28](#), [90](#),
[91](#), [91](#), [91](#), [93](#), [93](#), [94](#), [95](#), [102](#), [102](#),
[268](#), [303](#), [330](#), [395](#), [501](#), [517](#), [715](#),
[716](#), [720](#), [752](#), [799](#), [823–825](#), [969](#),
[1173](#), 1919, 1948, 1964, 3715, 4074,
4079, 4087, 4391, 4399, 4407, 4434,
4438, 4447, 4454, 4489, 4499, 4918,
4931, 4956, 4980, 4981, 4993, 4998,
5029, 5046, 5083, 5260, 5280, 5298,
5591, 6111, 6126, 6154, 6303, 6308,
6326, 6470, 7781, 7789, 7797, 7936,
[8060](#), 8317, 8322, 8327, 8332, 8486,
8572, 8574, 8704, 8714, 8749, 8760,
8766, 8777, 8808, 8845, 8849, 9169,
9466, 10008, 10017, 10068, 10078,
10092, 10099, 10114, 10168, 10170,
10238, 10240, 10244, 10246, 10250,
10252, 10256, 10258, 10291, 10292,
10424, 10476, 10479, 10484, 10490,
11264, 12198, 12463, 12661, 12942,
13000, 13434, 13435, 13457, 13467,
13474, 13475, 13487, 13488, 15661,
15699, 15700, 15751, 15768, 15820,
15844, 15853, 15857, 15925, 21820,
21980, 21983, 21984, 22074, 22075,
22107, 22155, 22217, 22225, 22294,
22318, 22763, 23027, 23028, 23248,
23324, 23328, 23351, 23640, 23896,
24765, 24971, 24975, 24978, 24982,
25159, 25161, 25175, 25176, 25178,
25179, 25322, 25412, 25443, 25627,
25675, 25750, 25877, 25882, 26429,
26474, 26475, 26674, 26815, 26825,
31451, 31699, 31707, 31806, 31807
- \int_eval:w [91](#), [304](#), [308](#),
[308](#), 4949, 5292, 7912, [8060](#), 8882,
8917, 12893, 12902, 12927, 12939,

- 15795, 19363, 23075, 23310, 23320
 \int_from_alph:n 99, 8747
 \int_from_base:nn
 100, 8764, 8787, 8789, 8791
 \int_from_bin:n 100, 8786, 31982
 \int_from_binary:n 31981
 \int_from_hex:n 100, 8786, 31984
 \int_from_hexadecimal:n 31983
 \int_from_oct:n 100, 8786, 31986
 \int_from_octal:n 31985
 \int_from_roman:n 100, 8806
 \int_gadd:Nn 92, 8205
 \int_gdecr:N 92, 4020,
 4823, 7846, 7894, 8217, 8484, 9969,
 11403, 12586, 14172, 18465, 23390
 \int_gincr:N 92, 4013, 4812,
 7838, 7888, 8217, 8459, 8470, 9962,
 11398, 12577, 14151, 14158, 15652,
 18444, 18451, 22102, 22405, 23368
 .int_gset:N 190, 15038
 \int_gset:Nn 93, 501, 8229, 11618
 \int_gset_eq:NN 92, 8197
 \int_gsub:Nn 93, 8205, 22116
 \int_gzero:N 92, 8187, 8194
 \int_gzero_new:N 92, 8191
 \int_if_even:nTF 95, 8344, 13168
 \int_if_even_p:n 95, 8344
 \int_if_exist:NTF 92, 8192,
 8194, 8201, 8820, 8824, 24691, 24746
 \int_if_exist_p:N 92, 8201
 \int_if_odd:nTF 95, 8344, 19749
 \int_if_odd_p:n 95, 8344, 25085
 \int_incr:N 92, 5631, 5641,
 5674, 7621, 8217, 14789, 15740,
 15774, 15866, 22205, 22735, 22831,
 22832, 23166, 23208, 23221, 23239,
 23505, 23506, 24582, 25006, 25167,
 25257, 25570, 25605, 25607, 25682,
 25944, 26009, 26168, 26224, 26288
 \int_log:N 101, 8846
 \int_log:n 101, 8848
 \int_max:nn .. 91, 912, 8077, 19610,
 20771, 25281, 25491, 26374, 26375
 \int_min:nn 91, 915, 8077
 \int_mod:nn .. 91, 5972, 5973, 6208,
8109, 8494, 8593, 8613, 9439, 10473
 \int_new:N 91,
 92, 5198, 8148, 8158, 8164, 8192,
 8194, 8862, 8863, 8864, 8865, 8866,
 8867, 9225, 12728, 12731, 12733,
 12746, 14600, 15644, 15646, 22095,
 22096, 22273, 22640, 22641, 22642,
 22643, 22644, 22645, 22646, 22647,
 22648, 22649, 22650, 23061, 23062,
 23063, 23064, 23531, 23532, 23533,
 23546, 23951, 23952, 23959, 23960,
 23977, 25102, 25104, 25105, 25106,
 25109, 25449, 25450, 25451, 25452,
 25453, 25454, 25455, 25456, 25457,
 25458, 25461, 25462, 25463, 25712,
 26137, 26140, 26141, 26142, 26689
 \int_rand:n
 100, 267, 15761, 21822, 21825, 22072
 \int_rand:nn 100, 117, 267, 914, 915,
 922, 1175, 4414, 7803, 8850, 10128,
 10133, 21816, 21819, 21978, 31441
 \int_range:nn 916
 .int_set:N 190, 15038
 \int_set:Nn 93, 303, 2124, 3662, 3697,
 5497, 5749, 5798, 5851, 7622, 8229,
 12537, 12539, 12712, 12714, 12729,
 12739, 12752, 12799, 12805, 12817,
 12822, 14794, 22373, 22374, 22389,
 22538, 22553, 22586, 22655, 22657,
 22659, 22681, 22682, 22697, 22705,
 22706, 22718, 22719, 22737, 22740,
 23120, 23183, 23493, 24773, 25103,
 25154, 25156, 25225, 25277, 25278,
 25288, 25299, 25323, 25341, 25390,
 25476, 25489, 25520, 25541, 25631,
 25666, 26173, 26321, 26347, 26824,
 26826, 26834, 26835, 26836, 26837
 \int_set_eq:NN ... 92, 3660, 3663,
 8197, 9359, 13218, 22698, 22728,
 23611, 24070, 24074, 24083, 24085,
 24128, 24181, 24481, 24581, 24594,
 24693, 25119, 25130, 25131, 25165,
 25166, 25216, 25320, 25321, 25373,
 25428, 25478, 25481, 25496, 25503,
 25517, 25519, 25522, 25536, 25539,
 25571, 25572, 25576, 25706, 26279
 \int_show:N 101, 8842
 \int_show:n 101, 519, 1173, 8844
 \int_sign:n 91, 671, 8063, 22455
 \int_step.... 237
 \int_step_function:nN
 97, 7612, 8416, 22557, 22558
 \int_step_function:nnN 97,
8416, 10400, 10405, 22559, 22560,
 22561, 22562, 22790, 25577, 26238
 \int_step_function:nnnN . 97, 270,
 270, 510, 803, 8416, 8483, 26350, 26358
 \int_step_inline:nn 97,
 716, 8453, 15655, 22299, 22333, 22384
 \int_step_inline:nnn
 97, 8453, 12414, 12628,
 22306, 22309, 22539, 25511, 26692
 \int_step_inline:nnnn . 97, 804, 8453

- \int_step_variable:nNn [97](#), [8453](#)
- \int_step_variable:nnNn [97](#), [8453](#)
- \int_step_variable:nnnNn [97](#), [8453](#)
- \int_sub:Nn [93](#), [8205](#), [11562](#), [13014](#),
[23615](#), [24285](#), [25376](#), [25384](#), [25393](#)
- \int_to_Alph:n [98](#), [99](#), [8507](#)
- \int_to_alph:n [98](#), [98](#), [99](#), [8507](#)
- \int_to_arabic:n [98](#), [8486](#)
- \int_to_Base:n [99](#)
- \int_to_base:n [99](#)
- \int_to_Base:nn [99](#), [100](#), [8571](#), [8698](#)
- \int_to_base:nn
. [99](#), [100](#), [8571](#), [8694](#), [8696](#), [8700](#)
- \int_to_bin:n [99](#), [99](#), [100](#), [8693](#), [31988](#)
- \int_to_binary:n [31987](#)
- \int_to_Hex:n [99](#), [100](#), [8693](#), [23827](#)
- \int_to_hex:n [99](#), [100](#), [8693](#), [31990](#)
- \int_to_hexadecimal:n [31989](#)
- \int_to_oct:n [99](#), [100](#), [8693](#), [31992](#)
- \int_to_octal:n [31991](#)
- \int_to_Roman:n [99](#), [100](#), [8701](#)
- \int_to_roman:n [99](#), [100](#), [8701](#)
- \int_to_symbols:nnn
. [98](#), [98](#), [8487](#), [8509](#), [8541](#)
- \int_until_do:nn [96](#), [8360](#)
- \int_until_do:nNnn [96](#), [8388](#)
- \int_use:N
[90](#), [93](#), [746](#), [751](#), [4015](#), [4017](#), [4814](#),
[4818](#), [5640](#), [6107](#), [6131](#), [6145](#), [6165](#),
[6172](#), [6354](#), [6463](#), [6468](#), [6484](#), [7839](#),
[7845](#), [7890](#), [7892](#), [8235](#), [8462](#), [8473](#),
[9964](#), [9966](#), [11397](#), [11405](#), [11522](#),
[12093](#), [12200](#), [12540](#), [12579](#), [12708](#),
[14154](#), [14161](#), [14795](#), [18447](#), [18454](#),
[22407](#), [22409](#), [22440](#), [23370](#), [24102](#),
[24175](#), [24246](#), [24257](#), [24266](#), [24270](#),
[24281](#), [24282](#), [24288](#), [24289](#), [24295](#),
[24296](#), [24464](#), [25085](#), [25147](#), [25149](#),
[25255](#), [25268](#), [25269](#), [25667](#), [25697](#),
[25804](#), [25816](#), [25912](#), [26173](#), [26686](#),
[31507](#), [31516](#), [31518](#), [31521](#), [31526](#),
[31535](#), [31537](#), [31541](#), [31544](#), [31549](#)
- \int_value:w [102](#), [308](#), [308](#), [354](#), [499](#),
[505](#), [528](#), [667](#), [715](#), [716](#), [724](#), [730](#),
[734](#), [746](#), [752](#), [753](#), [759](#), [762](#), [768](#),
[775](#), [800](#), [801](#), [810](#), [818](#), [826](#), [889](#),
[894](#), [907](#), [953](#), [1175](#), [1721](#), [2708](#),
[2710](#), [4447](#), [4454](#), [4918](#), [4919](#), [4931](#),
[4949](#), [4956](#), [4979](#), [4980](#), [4981](#), [4993](#),
[5029](#), [5544](#), [5630](#), [5652](#), [6027](#), [6103](#),
[6111](#), [6126](#), [6154](#), [7900](#), [7912](#), [8048](#),
[8061](#), [8062](#), [8065](#), [8066](#), [8079](#), [8080](#),
[8087](#), [8088](#), [8089](#), [8095](#), [8096](#), [8097](#),
[8111](#), [8113](#), [8114](#), [8131](#), [8134](#), [8135](#),
[8136](#), [8143](#), [8256](#), [8260](#), [8290](#), [8419](#),
[8420](#), [8421](#), [8447](#), [8657](#), [8690](#), [8882](#),
[8917](#), [8927](#), [9043](#), [9046](#), [9169](#), [9454](#),
[10291](#), [10292](#), [12893](#), [12902](#), [13987](#),
[14178](#), [14215](#), [15671](#), [15674](#), [15699](#),
[15700](#), [15746](#), [15751](#), [15795](#), [15999](#),
[16000](#), [16001](#), [16002](#), [16003](#), [16017](#),
[16165](#), [16226](#), [16244](#), [16551](#), [16681](#),
[16695](#), [16697](#), [16699](#), [16702](#), [16738](#),
[16876](#), [16906](#), [16907](#), [16944](#), [16952](#),
[17083](#), [17088](#), [17090](#), [17099](#), [17103](#),
[17140](#), [17148](#), [17151](#), [17157](#), [17168](#),
[17179](#), [17185](#), [17186](#), [17189](#), [17232](#),
[17242](#), [17244](#), [17260](#), [17262](#), [17285](#),
[17299](#), [17378](#), [17380](#), [17454](#), [17542](#),
[18243](#), [18276](#), [18631](#), [18632](#), [18633](#),
[18635](#), [18681](#), [18684](#), [18687](#), [18710](#),
[18712](#), [18733](#), [18735](#), [18744](#), [18746](#),
[18750](#), [18768](#), [18775](#), [18781](#), [18791](#),
[18793](#), [18807](#), [18815](#), [18823](#), [18867](#),
[18869](#), [18885](#), [18887](#), [18890](#), [18893](#),
[18947](#), [18955](#), [18957](#), [18959](#), [18961](#),
[18964](#), [18967](#), [18969](#), [18988](#), [18990](#),
[18994](#), [19000](#), [19002](#), [19006](#), [19028](#),
[19031](#), [19039](#), [19041](#), [19044](#), [19045](#),
[19046](#), [19047](#), [19062](#), [19065](#), [19068](#),
[19071](#), [19080](#), [19083](#), [19086](#), [19089](#),
[19096](#), [19098](#), [19104](#), [19112](#), [19114](#),
[19116](#), [19142](#), [19144](#), [19153](#), [19155](#),
[19159](#), [19176](#), [19197](#), [19201](#), [19213](#),
[19216](#), [19219](#), [19222](#), [19225](#), [19228](#),
[19231](#), [19234](#), [19238](#), [19250](#), [19254](#),
[19258](#), [19261](#), [19282](#), [19284](#), [19286](#),
[19296](#), [19320](#), [19323](#), [19335](#), [19337](#),
[19343](#), [19346](#), [19363](#), [19383](#), [19433](#),
[19438](#), [19440](#), [19447](#), [19450](#), [19453](#),
[19456](#), [19459](#), [19462](#), [19471](#), [19483](#),
[19491](#), [19493](#), [19503](#), [19505](#), [19512](#),
[19521](#), [19523](#), [19526](#), [19529](#), [19532](#),
[19535](#), [19548](#), [19550](#), [19558](#), [19560](#),
[19568](#), [19570](#), [19580](#), [19583](#), [19586](#),
[19593](#), [19608](#), [19626](#), [19629](#), [19685](#),
[19699](#), [19701](#), [19707](#), [19720](#), [19722](#),
[19724](#), [19748](#), [19764](#), [19771](#), [19772](#),
[19816](#), [19818](#), [19819](#), [19820](#), [19861](#),
[19863](#), [19900](#), [19907](#), [19914](#), [19935](#),
[19937](#), [19939](#), [19941](#), [19954](#), [19958](#),
[19959](#), [19960](#), [19961](#), [19962](#), [19967](#),
[19972](#), [19974](#), [19980](#), [19997](#), [19998](#),
[19999](#), [20000](#), [20001](#), [20002](#), [20007](#),
[20009](#), [20011](#), [20013](#), [20015](#), [20020](#),
[20022](#), [20024](#), [20026](#), [20028](#), [20030](#),
[20052](#), [20060](#), [20076](#), [20081](#), [20085](#),
[20144](#), [20193](#), [20261](#), [20270](#), [20278](#),

- 20289, 20291, 20294, 20297, 20386,
 20422, 20424, 20427, 20430, 20433,
 20436, 20443, 20446, 20448, 20452,
 20474, 20476, 20508, 20578, 20588,
 20593, 20603, 20745, 20777, 20786,
 21018, 21019, 21030, 21033, 21036,
 21039, 21042, 21045, 21048, 21051,
 21054, 21072, 21082, 21091, 21109,
 21118, 21125, 21135, 21179, 21188,
 21223, 21266, 21283, 21339, 21350,
 21361, 21571, 21647, 21694, 21739,
 21747, 21749, 21751, 21843, 21866,
 21920, 21960, 21972, 21983, 21984,
 22014, 22017, 22020, 22022, 22024,
 22031, 22034, 22042, 22047, 22052,
 22155, 22217, 22225, 22238, 22239,
 22240, 22250, 22763, 23075, 23087,
 23266, 23308, 23310, 23320, 23328,
 23347, 23349, 23357, 23820, 24314,
 24320, 24352, 24354, 24363, 24364,
 24488, 24913, 24928, 25728, 25729,
 25740, 26262, 26293, 26295, 28956,
 29088, 31441, 31451, 31699, 31707
 \int_while_do:nn 96, 8360
 \int_while_do:nNnn 96, 8388
 \int_zero:N 92,
 92, 5611, 5657, 7606, 8187, 8192,
 12859, 14786, 15737, 15766, 15863,
 22202, 23121, 23122, 23123, 23222,
 24082, 24283, 24730, 25038, 25118,
 25475, 25488, 25518, 25787, 26167
 \int_zero_new:N 92, 8191
 \c_max_int 101, 199,
 719, 915, 974, 1025, 8851, 22025,
 23586, 23600, 25572, 26812, 26818
 \c_nine 31935
 \c_one 31919
 \c_one_int 101,
 8218, 8220, 8222, 8224, 8850, 15795
 \c_seven 31931
 \c_six 31929
 \c_sixteen 31949
 \c_ten 31937
 \c_thirteen 31943
 \c_three 31923
 \g_tmpa_int 101, 8862
 \l_tmpa_int 2, 101, 230, 8862
 \g_tmpb_int 101, 8862
 \l_tmpb_int 2, 101, 8862
 \c_twelve 31941
 \c_two 31921
 \c_zero 31917
 \c_zero_int 101, 312,
 323, 323, 395, 1411, 1719, 1721,
 3660, 8144, 8145, 8156, 8187, 8188,
 8240, 8248, 8425, 8428, 8850, 9222,
 9224, 9359, 9457, 13218, 14029,
 15656, 15782, 21866, 21995, 22059
 int internal commands:
 __int_abs:N 8077
 __int_case:nnTF 8314
 __int_case:nw 8314
 __int_case_end:nw 8314
 __int_compare:nnN 506, 8253
 __int_compare:NNw ... 505, 506, 8253
 __int_compare:Nw . 504, 505, 506, 8253
 __int_compare:w 505, 8253
 __int_compare_!=:NNw 8253
 __int_compare_<:NNw 8253
 __int_compare_<=:NNw 8253
 __int_compare_=:NNw 8253
 __int_compare_==:NNw 8253
 __int_compare_>:NNw 8253
 __int_compare_>=:NNw 8253
 __int_compare_end=:NNw .. 506, 8253
 __int_compare_error:
 504, 505, 8238, 8256, 8258
 __int_compare_error:Nw
 504, 505, 506, 8238, 8278
 __int_constdef:Nw 8154
 __int_div_truncate:NwNw 8109
 __int_eval:w
 303, 499, 500, 505, 8048,
 8061, 8062, 8066, 8080, 8088, 8089,
 8096, 8097, 8111, 8113, 8114, 8131,
 8134, 8135, 8136, 8143, 8172, 8206,
 8208, 8210, 8212, 8230, 8232, 8256,
 8290, 8308, 8346, 8354, 8419, 8420,
 8421, 8447, 8630, 8657, 8663, 8690
 __int_eval_end:
 8048, 8061, 8066, 8080,
 8115, 8131, 8137, 8146, 8172, 8206,
 8208, 8210, 8212, 8230, 8232, 8308,
 8346, 8354, 8630, 8657, 8663, 8690
 __int_from_alph:N 517, 8747
 __int_from_alph:nN 517, 8747
 __int_from_base:N 517, 8764
 __int_from_base:nnN 517, 8764
 __int_from_roman:NN 8806
 \c_int_from_roman_C_int 8792
 \c_int_from_roman_c_int 8792
 \c_int_from_roman_D_int 8792
 \c_int_from_roman_d_int 8792
 __int_from_roman_error:w 8806
 \c_int_from_roman_I_int 8792
 \c_int_from_roman_i_int 8792
 \c_int_from_roman_L_int 8792
 \c_int_from_roman_l_int 8792

<code>\c__int_from_roman_M_int</code>	8792	<code>intarray</code> commands:
<code>\c__int_from_roman_m_int</code>	8792	<code>\intarray_const_from_clist:Nn</code> . . .
<code>\c__int_from_roman_V_int</code>	8792 199 , 15763 , 20199 , 20801
<code>\c__int_from_roman_v_int</code>	8792	<code>\intarray_count:N</code> 199 , 199 ,
<code>\c__int_from_roman_X_int</code>	8792	200 , 304 , 304 , 15662 , 15665 , 15667 ,
<code>\c__int_from_roman_x_int</code>	8792	15668 , 15671 , 15680 , 15690 , 15738 ,
<code>__int_if_recursion_tail_stop:N</code> .		15761 , 15782 , 15807 , 15864 , 22127
. 8058 , 8819		<code>\intarray_gset:Nnn</code>
<code>__int_if_recursion_tail_stop-</code>	 199 , 304 , 715 , 717 , 15693 , 22294
<code>do:Nn</code> 8058 , 8758 , 8775 , 8822		<code>\intarray_gset_rand:Nn</code> 267 , 15812
<code>\l__int_internal_a_int</code>	8866	<code>\intarray_gset_rand:Nnn</code> 267 , 15812
<code>\l__int_internal_b_int</code>	8866	<code>\intarray_gzero:N</code> 199 , 15735
<code>\c__int_max_constdef_int</code>	8154	<code>\intarray_item:Nn</code>
<code>__int_maxmin:wwN</code>	8077 200 , 304 , 715 , 717 , 15745 , 15761
<code>__int_mod:ww</code>	8109	<code>\intarray_log:N</code> 200 , 15797
<code>__int_pass_signs:wn</code>		<code>\intarray_new:Nn</code>
. 516 , 8737 , 8751 , 8768	 199 , 714 , 717 , 15649 , 22119 , 22120 ,
<code>__int_pass_signs_end:wn</code>	8737	22121 , 22292 , 23543 , 23544 , 23545 ,
<code>__int_show:nN</code>	8842	25464 , 25465 , 26143 , 26144 , 26145
<code>__int_sign:Nw</code>	8063	<code>\intarray_rand_item:N</code> 200 , 15760
<code>__int_step:NNnnnn</code>	8453	<code>\intarray_show:N</code> 200 , 718 , 15797
<code>__int_step:NwnnN</code>	8416	<code>\intarray_to_clist:N</code> 267 , 15778
<code>__int_step:wwN</code>	8416	intarray internal commands:
<code>__int_to_Base:nn</code>	8571	<code>__intarray_bounds:NNnTF</code>
<code>__int_to_base:nn</code>	8571 15675 , 15705 , 15756
<code>__int_to_Base:nnN</code>	8571	<code>__intarray_bounds_error:NNnw</code> 15675
<code>__int_to_base:nnN</code>	8571	<code>__intarray_const_from_clist:nN</code> .
<code>__int_to_Base:nnnN</code>	8571 15763
<code>__int_to_base:nnnN</code>	8571	<code>__intarray_count:w</code>
<code>__int_to_Letter:n</code>	8571 15642 , 15661 , 15671 , 15769 , 15790
<code>__int_to_letter:n</code>	8571	<code>__intarray_entry:w</code>
<code>__int_to_roman:N</code>	8701 15642 , 15694 , 15741 , 15746
<code>__int_to_roman:w</code>		<code>\g__intarray_font_int</code>
505 , 515 , 1392 , 8048 , 8266 , 8704 , 8714	 15646 , 15652 , 15654
<code>__int_to_Roman_aux:N</code>	8713 , 8716 , 8719	<code>__intarray_gset:Nnn</code> 15693
<code>__int_to_Roman_c:w</code>	8701	<code>__intarray_gset:Nww</code> 15697 , 15703
<code>__int_to_roman_c:w</code>	8701	<code>__intarray_gset_all_same:Nn</code> 15812
<code>__int_to_Roman_d:w</code>	8701	<code>__intarray_gset_overflow:Nnn</code> 15693
<code>__int_to_roman_d:w</code>	8701	<code>__intarray_gset_overflow:NNnn</code> . .
<code>__int_to_Roman_i:w</code>	8701 15717 , 15725 , 15729
<code>__int_to_roman_i:w</code>	8701	<code>__intarray_gset_overflow-</code>
<code>__int_to_Roman_l:w</code>	8701	<code>test:nw</code> 717 , 718 , 15707 ,
<code>__int_to_roman_l:w</code>	8701	15714 , 15722 , 15775 , 15831 , 15838
<code>__int_to_Roman_m:w</code>	8701	<code>__intarray_gset_rand:Nnn</code> 15812
<code>__int_to_roman_m:w</code>	8701	<code>__intarray_gset_rand_auxi:Nnnn</code> .
<code>__int_to_Roman_Q:w</code>	8701 15812
<code>__int_to_roman_Q:w</code>	8701	<code>__intarray_gset_rand_auxii:Nnnn</code>
<code>__int_to_Roman_v:w</code>	8701 15812
<code>__int_to_roman_v:w</code>	8701	<code>__intarray_gset_rand_auxiii:Nnnn</code>
<code>__int_to_Roman_x:w</code>	8701 15812
<code>__int_to_roman_x:w</code>	8701	<code>__intarray_item:Nn</code> 15745
<code>__int_to_symbols:nnnn</code>	8487	<code>__intarray_item:Nw</code> 15749 , 15754
<code>__int_use_none_delimit_by_s-</code>		
<code>stop:w</code> 8055 , 8288		

- \l__intarray_loop_int ... [15644](#),
[15737](#), [15740](#), [15741](#), [15766](#), [15769](#),
[15774](#), [15776](#), [15863](#), [15866](#), [15867](#)
- __intarray_new:N [15649](#), [15765](#)
- __intarray_show:NN
..... [15797](#), [15799](#), [15801](#)
- __intarray_signed_max_dim:n ...
..... [15673](#), [15732](#), [15733](#)
- \c__intarray_sp_dim
..... [15645](#), [15654](#), [15694](#)
- __intarray_to_clist:Nn [15778](#), [15808](#)
- __intarray_to_clist:w [15778](#)
- \interactionmode [542](#)
- \interlinepenalties [543](#)
- \interlinepenalty [323](#)
- ior commands:
 - \ior_close:N [159](#),
[160](#), [160](#), [267](#), [12458](#), [12477](#), [13370](#),
[13383](#), [28932](#), [28965](#), [29002](#), [29019](#)
 - \ior_get:NN [160](#),
[161](#), [161](#), [162](#), [162](#), [267](#), [12518](#), [12598](#)
 - \ior_get:NNTF [161](#), [12518](#), [12519](#)
 - \ior_get_str:NN [31993](#)
 - \ior_get_term:nN [267](#), [12552](#)
 - \ior_if_eof:N [628](#)
 - \ior_if_eof:NNTF [163](#), [12502](#), [12524](#),
[12544](#), [12584](#), [12603](#), [13380](#), [13394](#)
 - \ior_if_eof_p:N [163](#), [12502](#)
 - \ior_list_streams: [31995](#)
 - \ior_log_list: ... [160](#), [12489](#), [31998](#)
 - \ior_log_streams: [31997](#)
 - \ior_map_break: [162](#), [12567](#), [12585](#),
[12592](#), [12604](#), [12610](#), [28927](#), [28998](#)
 - \ior_map_break:n [163](#), [12567](#)
 - \ior_map_inline:Nn .. [162](#), [162](#), [12571](#)
 - \ior_map_variable:NNn
..... [162](#), [12597](#), [28924](#)
 - \ior_new:N
[159](#), [12427](#), [12429](#), [12430](#), [13399](#), [28860](#)
 - \ior_open:Nn [159](#),
[658](#), [12431](#), [28888](#), [28933](#), [28966](#), [29018](#)
 - \ior_open:NnTF ... [160](#), [12432](#), [12435](#)
 - \ior_shell_open:Nn [267](#), [31555](#)
 - \ior_shell_open:nN [267](#)
 - \ior_show_list: ... [160](#), [12489](#), [31996](#)
 - \ior_str_get:NN
.. [160](#), [161](#), [267](#), [12531](#), [12600](#), [31994](#)
 - \ior_str_get:NNTF .. [161](#), [12531](#), [12532](#)
 - \ior_str_get_term:nN ... [267](#), [12552](#)
 - \ior_str_map_inline:Nn
..... [162](#), [162](#), [12571](#), [28959](#), [28989](#)
 - \ior_str_map_variable:NNn [162](#), [12597](#)
 - \c_term_ior [32215](#)
 - \g_tmpa_ior [166](#), [12429](#)
 - \g_tmpb_ior [166](#), [12429](#)
- ior internal commands:
 - \l__ior_file_name_tl
..... [12434](#), [12437](#), [12439](#)
 - __ior_get:NN ... [12518](#), [12553](#), [12572](#)
 - __ior_get_term:NnN [12552](#)
 - \l__ior_internal_tl
..... [12409](#), [12590](#), [12594](#)
 - __ior_list:N [12489](#)
 - __ior_map_inline:NNn [12571](#)
 - __ior_map_inline:NNNn [12571](#)
 - __ior_map_inline_loop:NNN ... [12571](#)
 - __ior_map_variable:NNNn [12597](#)
 - __ior_map_variable_loop:NNNn . [12597](#)
 - __ior_new:N [624](#), [12445](#), [12462](#)
 - __ior_new_aux:N [12449](#), [12453](#)
 - __ior_open_stream:Nn [12456](#)
 - __ior_shell_open:nN [31555](#)
 - __ior_str_get:NN [12531](#), [12555](#), [12574](#)
 - \l__ior_stream_tl
..... [12412](#), [12459](#), [12463](#), [12470](#)
 - \g__ior_streams_prop
..... [12413](#), [12471](#), [12482](#), [12496](#)
 - \g__ior_streams_seq
..... [12411](#), [12459](#), [12483](#), [12484](#)
 - \c__ior_term_ior [12410](#),
[12427](#), [12479](#), [12485](#), [12506](#), [12562](#)
 - \c__ior_term_noprompt_ior
..... [12551](#), [12561](#)
- iow commands:
 - \iow_allow_break:
..... [165](#), [266](#), [12766](#), [12808](#), [12813](#)
 - \iow_allow_break:n [635](#)
 - \iow_char:N [164](#),
[5727](#), [12101](#), [12103](#), [12104](#), [12136](#),
[12230](#), [12263](#), [12727](#), [20305](#), [22601](#),
[22603](#), [22604](#), [22607](#), [22609](#), [22610](#),
[22613](#), [22615](#), [22616](#), [22617](#), [22621](#),
[22628](#), [23015](#), [23018](#), [23019](#), [23043](#),
[23044](#), [23051](#), [23052](#), [23806](#), [23808](#),
[23810](#), [23812](#), [23814](#), [23816](#), [24420](#),
[24421](#), [24962](#), [25075](#), [25076](#), [25077](#),
[25098](#), [26397](#), [26400](#), [26401](#), [26406](#),
[26440](#), [26449](#), [26453](#), [26458](#), [26478](#),
[26480](#), [26481](#), [26483](#), [26486](#), [26488](#),
[26495](#), [26499](#), [26502](#), [26503](#), [26506](#),
[26508](#), [26512](#), [26514](#), [26520](#), [26522](#),
[26526](#), [26528](#), [26532](#), [26537](#), [26539](#),
[26581](#), [26583](#), [26588](#), [26590](#), [26596](#),
[26601](#), [26606](#), [26610](#), [26620](#), [26623](#),
[26627](#), [26628](#), [26632](#), [26640](#), [26697](#)
 - \iow_close:N .. [160](#), [160](#), [12656](#), [12674](#)

- \iow_indent:n . [165](#), [165](#), [635](#), [636](#),
[5725](#), [6050](#), [6236](#), [12047](#), [12150](#),
[12777](#), [12809](#), [12814](#), [16426](#), [16438](#)
- \l_iow_line_count_int
. [165](#), [165](#), [636](#), [967](#), [11562](#), [12728](#),
[12818](#), [12823](#), [12861](#), [23467](#), [23471](#)
- \iow_list_streams: [31999](#)
- \iow_log:n
... [163](#), [1829](#), [4541](#), [11760](#), [11775](#),
[11776](#), [11782](#), [12722](#), [32020](#), [32100](#)
- \iow_log_list: [160](#), [12686](#), [32002](#)
- \iow_log_streams: [32001](#)
- \iow_new:N ... [159](#), [12645](#), [12647](#), [12648](#)
- \iow_newline: [164](#),
[164](#), [164](#), [165](#), [305](#), [408](#), [603](#), [633](#),
[11584](#), [12726](#), [12806](#), [12815](#), [12821](#),
[13790](#), [23417](#), [25009](#), [28601](#), [28602](#),
[28603](#), [31373](#), [31375](#), [31378](#), [31385](#)
- \iow_now:Nn
... [163](#), [163](#), [163](#), [164](#), [164](#), [9389](#),
[12716](#), [12722](#), [12723](#), [12724](#), [12725](#)
- \iow_open:Nn [160](#), [12652](#)
- \iow_shipout:Nn
..... [164](#), [164](#), [164](#), [633](#), [9402](#), [12701](#)
- \iow_shipout_x:Nn
..... [164](#), [164](#), [164](#), [633](#), [12698](#)
- \iow_show_list: ... [160](#), [12686](#), [32000](#)
- \iow_term:n [163](#), [267](#),
[1829](#), [11596](#), [11738](#), [11753](#), [11754](#),
[11788](#), [11814](#), [12722](#), [26687](#), [32022](#)
- \iow_wrap:nnnN [164](#), [164](#),
[165](#), [165](#), [165](#), [266](#), [408](#), [636](#), [1173](#),
[4526](#), [4541](#), [11560](#), [11563](#), [11575](#),
[11739](#), [11761](#), [11780](#), [11786](#), [11793](#),
[12769](#), [12775](#), [12780](#), [12792](#), [12795](#)
- \c_log_iow
[166](#), [629](#), [12616](#), [12676](#), [12722](#), [12723](#)
- \c_term_iow [166](#), [629](#), [12616](#),
[12645](#), [12676](#), [12682](#), [12724](#), [12725](#)
- \g_tmpa_iow [166](#), [12647](#)
- \g_tmpb_iow [166](#), [12647](#)
- iow internal commands:
- __iow_allow_break: [635](#), [12766](#), [12808](#)
- __iow_allow_break_error:
..... [635](#), [12766](#), [12813](#)
- \l_iow_file_name_tl
..... [12651](#), [12654](#), [12658](#), [12662](#)
- __iow_indent:n ... [635](#), [12777](#), [12809](#)
- __iow_indent_error:n
..... [635](#), [12777](#), [12814](#)
- \l_iow_indent_int [12745](#),
[12859](#), [12877](#), [12989](#), [13006](#), [13014](#)
- \l_iow_indent_tl .. [12745](#), [12860](#),
[12876](#), [12988](#), [13007](#), [13015](#), [13016](#)
- \l_iow_line_break_bool
[12749](#), [12855](#), [12983](#), [12997](#), [13005](#),
[13013](#), [13021](#), [13023](#), [13028](#), [13030](#)
- \l_iow_line_part_tl
.... [638](#), [640](#), [641](#), [12747](#), [12857](#),
[12869](#), [12890](#), [12948](#), [12951](#), [12982](#),
[12996](#), [12998](#), [13004](#), [13012](#), [13035](#)
- \l_iow_line_target_int
..... [642](#), [12731](#), [12817](#),
[12819](#), [12822](#), [12984](#), [12989](#), [13024](#)
- \l_iow_line_tl [12747](#), [12856](#), [12873](#),
[12963](#), [12979](#), [12995](#), [12996](#), [13004](#),
[13012](#), [13034](#), [13035](#), [13040](#), [13042](#)
- __iow_list:N [12686](#)
- __iow_new:N [12649](#), [12660](#)
- \l_iow_newline_tl [12730](#),
[12815](#), [12816](#), [12818](#), [12821](#), [13039](#)
- \l_iow_one_indent_int
..... [12732](#), [13006](#), [13014](#)
- \l_iow_one_indent_tl
..... [634](#), [12732](#), [13007](#)
- __iow_open_stream:Nn [12652](#)
- __iow_set_indent:n [634](#), [12732](#)
- \l_iow_stream_tl
..... [12626](#), [12657](#), [12661](#), [12668](#)
- \g_iow_streams_prop
..... [12627](#), [12669](#), [12679](#), [12693](#)
- \g_iow_streams_seq
..... [12625](#), [12657](#), [12680](#), [12681](#)
- __iow_tmp:w [640](#), [12863](#),
[12887](#), [12944](#), [12976](#), [13044](#), [13052](#)
- __iow_unindent:w .. [634](#), [12732](#), [13016](#)
- __iow_use_i_delimit_by_s_-
stop:nw [12643](#), [12848](#)
- __iow_with:nNnn [12704](#)
- __iow_wrap_allow_break:n [12993](#)
- \c_iow_wrap_allow_break_marker_-
tl [12751](#), [12771](#)
- __iow_wrap_break:w ... [12930](#), [12944](#)
- __iow_wrap_break_end:w .. [640](#), [12944](#)
- __iow_wrap_break_first:w [12944](#)
- __iow_wrap_break_loop:w [12944](#)
- __iow_wrap_break_none:w [12944](#)
- __iow_wrap_chunk:nw [12861](#), [12863](#),
[12999](#), [13000](#), [13008](#), [13017](#), [13024](#)
- __iow_wrap_do: [12825](#), [12830](#)
- __iow_wrap_end:n [13019](#)
- __iow_wrap_end_chunk:w
..... [638](#), [12881](#), [12888](#), [12980](#)
- \c_iow_wrap_end_marker_tl
..... [12751](#), [12835](#)
- __iow_wrap_fix_newline:w [12830](#)
- __iow_wrap_indent:n [13002](#)

- \c__iow_wrap_indent_marker_tl . . .
 [12751](#), [12785](#)
- __iow_wrap_line:nw
 [638](#), [641](#), [12875](#), [12879](#), [12888](#), [12987](#)
- __iow_wrap_line_aux:Nw [12888](#)
- __iow_wrap_line_end:NnnnnnnN [12888](#)
- __iow_wrap_line_end:nw
 [640](#), [12888](#), [12964](#), [12965](#), [12974](#)
- __iow_wrap_line_loop:w [12888](#)
- __iow_wrap_line_seven:nnnnnn [12888](#)
- \c__iow_wrap_marker_tl
 [635](#), [638](#), [12751](#), [12887](#)
- __iow_wrap_newline:n [13019](#)
- \c__iow_wrap_newline_marker_tl . .
 [637](#), [12751](#), [12850](#)
- __iow_wrap_next:nw
 [12863](#), [12942](#), [12984](#)
- __iow_wrap_next_line:w [12936](#), [12977](#)
- __iow_wrap_start:w [12830](#)
- __iow_wrap_store_do:n
 [12935](#), [13022](#), [13029](#), [13032](#)
- \l__iow_wrap_tl
 [637](#), [637](#), [642](#), [642](#), [12750](#),
 [12812](#), [12827](#), [12832](#), [12834](#), [12837](#),
 [12839](#), [12842](#), [12858](#), [13036](#), [13038](#)
- __iow_wrap_trim:N
 [642](#), [12965](#), [12996](#), [13022](#), [13029](#), [13044](#)
- __iow_wrap_trim:w [13044](#)
- __iow_wrap_trim_aux:w [13044](#)
- __iow_wrap_unindent:n [13002](#)
- \c__iow_wrap_unindent_marker_tl .
 [12751](#), [12787](#)
- \itshape [30853](#)
- J**
- \j [30596](#), [30938](#), [31133](#), [31212](#)
- \jcharwidowpenalty [1129](#)
- \jfam [1130](#)
- \jfont [1131](#)
- \jis [1132](#)
- job commands:
- \c_job_name_tl [31913](#)
- \jobname [324](#)
- K**
- \k [29167](#), [31011](#), [31086](#),
 [31087](#), [31104](#), [31105](#), [31127](#), [31128](#),
 [31129](#), [31184](#), [31185](#), [31210](#), [31211](#)
- \kanjiskip [1133](#)
- \kansuji [1134](#)
- \kansujichar [1135](#)
- \kcatcode [1136](#)
- \kchar [1169](#)
- \kchardef [1170](#)
- \kern [325](#)
- kernel internal commands:
- __kernel_backend_align_begin: . [310](#)
- __kernel_backend_align_end: . . [310](#)
- \g__kernel_backend_header_bool . [310](#)
- __kernel_backend_literal:n . . . [309](#)
- __kernel_backend_literal_pdf:n [309](#)
- __kernel_backend_literal_-
 postscript:n [309](#)
- __kernel_backend_literal_svg:n [309](#)
- __kernel_backend_matrix:n [310](#)
- __kernel_backend_postscript:n . [309](#)
- __kernel_backend_scope_begin: . [310](#)
- __kernel_backend_scope_end: . . [310](#)
- __kernel_chk_cs_exist:N [303](#)
- __kernel_chk_defined:NTF
 [303](#), [561](#), [598](#),
 [2088](#), [2107](#), [4522](#), [8033](#), [8894](#), [8995](#),
 [10142](#), [11433](#), [15803](#), [18158](#), [26097](#)
- __kernel_chk_expr:nNnN [303](#)
- __kernel_chk_if_free_cs:N
 [573](#), [600](#),
 [1833](#), [1848](#), [1896](#), [3220](#), [3521](#), [3527](#),
 [3532](#), [7388](#), [8150](#), [8168](#), [8937](#), [10646](#),
 [10648](#), [10658](#), [11104](#), [13898](#), [14242](#),
 [14334](#), [15651](#), [22281](#), [22297](#), [26708](#)
- \l__kernel_color_stack_int [310](#)
- __kernel_cs_parm_from_arg_-
 count:nnTF . . . [303](#), [1558](#), [1914](#), [1961](#)
- __kernel_deprecation_code:nn . . .
 [303](#),
 [1186](#), [1508](#), [31813](#), [31848](#), [31855](#), [31856](#)
- __kernel_deprecation_error:Nnn .
 [1186](#), [31816](#), [31875](#)
- \g__kernel_deprecation_undo_-
 recent_bool . . . [31784](#), [31798](#), [31825](#)
- __kernel_exp_not:w
 [303](#), [347](#), [2480](#), [2482](#),
 [2486](#), [2490](#), [2493](#), [2496](#), [2501](#), [4217](#),
 [4243](#), [4268](#), [8271](#), [29201](#), [29531](#), [30637](#)
- \l__kernel_expl_bool
 [142](#), [145](#), [160](#), [174](#), [1358](#)
- __kernel_file_input_pop: [304](#), [13671](#)
- __kernel_file_input_push:n
 [304](#), [13671](#)
- __kernel_file_missing:n
 [304](#), [12432](#), [13666](#), [13675](#)
- __kernel_file_name_expand_-
 group:nw [13098](#)
- __kernel_file_name_expand_-
 loop:w [13098](#)
- __kernel_file_name_expand_N_-
 type:Nw [13098](#)

- _kernel_file_name_expand_-
space:w [13098](#)
- _kernel_file_name_quote:n [1178](#),
[12475](#), [12671](#), [13189](#), [13228](#), [13686](#)
- _kernel_file_name_quote:nw . [13189](#)
- _kernel_file_name_sanitizet:n ..
..... [659](#), [12655](#),
[13098](#), [13250](#), [13357](#), [13669](#), [13725](#)
- _kernel_file_name_sanitizet:nN .
..... [304](#), [304](#)
- _kernel_file_name_strip_-
quotes:n [13098](#)
- _kernel_file_name_strip_-
quotes:nnn [13098](#)
- _kernel_file_name_strip_-
quotes:nnnw [13098](#)
- _kernel_file_name_trim_-
spaces:n [13098](#)
- _kernel_file_name_trim_-
spaces:nw [13098](#)
- _kernel_file_name_trim_spaces_-
aux:n [13098](#)
- _kernel_file_name_trim_spaces_-
aux:w [13098](#)
- _kernel_if_debug:TF
..... [1495](#), [31824](#), [31836](#)
- _kernel_int_add:nnn [304](#), [8141](#), [22025](#)
- _kernel_intarray_gset:Nnn [304](#),
[715](#), [15656](#), [15668](#), [15693](#), [15776](#),
[15867](#), [22189](#), [22190](#), [22192](#), [22196](#),
[22197](#), [22198](#), [22300](#), [22301](#), [22335](#),
[22338](#), [25514](#), [25598](#), [25600](#), [25602](#),
[25622](#), [25625](#), [25680](#), [26200](#), [26202](#),
[26208](#), [26216](#), [26218](#), [26221](#), [26282](#),
[26284](#), [26286](#), [26299](#), [26305](#), [26309](#)
- _kernel_intarray_item:Nn
..... [304](#), [716](#),
[889](#), [15745](#), [15793](#), [20285](#), [20291](#),
[20294](#), [20297](#), [21031](#), [21034](#), [21037](#),
[21040](#), [21043](#), [21046](#), [21049](#), [21052](#),
[21055](#), [22238](#), [22239](#), [22240](#), [22387](#),
[22390](#), [25593](#), [25614](#), [25617](#), [25633](#),
[25660](#), [25721](#), [25722](#), [25745](#), [25746](#),
[25754](#), [25760](#), [25762](#), [25769](#), [25775](#),
[25777](#), [25831](#), [25835](#), [26205](#), [26332](#)
- _kernel_ior_open:Nn . [304](#), [1178](#),
[12439](#), [12456](#), [13378](#), [13393](#), [31568](#)
- _kernel_iow_with:Nnn [305](#),
[408](#), [603](#), [633](#), [4530](#), [4532](#), [11597](#),
[11599](#), [11816](#), [11818](#), [12704](#), [12718](#)
- _kernel_msg_critical:nn [305](#), [12006](#)
- _kernel_msg_critical:nnn [305](#), [12006](#)
- _kernel_msg_critical:nnnn [305](#), [12006](#)
- _kernel_msg_critical:nnnnn [305](#), [12006](#)
- _kernel_msg_error:nn [306](#),
[1810](#), [9270](#), [12006](#), [22435](#), [22480](#),
[24062](#), [24095](#), [24143](#), [24146](#), [24607](#),
[24864](#), [25965](#), [26049](#), [27805](#), [31559](#)
- _kernel_msg_error:nnn
..... [306](#), [1498](#), [1503](#), [1571](#),
[1626](#), [1674](#), [1679](#), [1810](#), [1998](#), [2005](#),
[2093](#), [2839](#), [3113](#), [3325](#), [3349](#), [3353](#),
[3498](#), [3758](#), [4627](#), [5353](#), [5412](#), [7601](#),
[7632](#), [9352](#), [9471](#), [11187](#), [11829](#),
[12006](#), [13539](#), [13668](#), [14697](#), [14726](#),
[14742](#), [14910](#), [14928](#), [15664](#), [15874](#),
[15896](#), [16306](#), [22114](#), [22453](#), [22495](#),
[22501](#), [23009](#), [24101](#), [24303](#), [24706](#),
[24719](#), [24758](#), [24882](#), [25803](#), [25810](#),
[26063](#), [26842](#), [27430](#), [28686](#), [31565](#)
- _kernel_msg_error:nnnn
..... [306](#), [1562](#), [1602](#),
[1693](#), [1810](#), [1837](#), [1963](#), [2924](#), [3133](#),
[3156](#), [3368](#), [5444](#), [9291](#), [9301](#), [9314](#),
[11461](#), [11855](#), [12006](#), [12768](#), [14654](#),
[14707](#), [14761](#), [14775](#), [14919](#), [15416](#),
[15469](#), [16302](#), [22872](#), [22879](#), [24280](#),
[24345](#), [24569](#), [25969](#), [25985](#), [27647](#)
- _kernel_msg_error:nnnnn
..... [306](#), [12006](#), [12779](#), [15705](#),
[22161](#), [23025](#), [26249](#), [26372](#), [31882](#)
- _kernel_msg_error:nnnnnn
..... [306](#), [2939](#), [2953](#), [12006](#), [15731](#), [31865](#)
- _kernel_msg_expandable_-
error:nn [306](#),
[2510](#), [7379](#), [9200](#), [10298](#), [10300](#),
[10308](#), [10314](#), [10354](#), [11093](#), [12338](#),
[14499](#), [14506](#), [14546](#), [16834](#), [23801](#)
- _kernel_msg_expandable_-
error:nnn [306](#),
[2246](#), [2586](#), [2646](#), [2670](#), [4067](#), [7976](#),
[8249](#), [8430](#), [9455](#), [10050](#), [12338](#),
[13170](#), [13322](#), [13642](#), [14133](#), [16841](#),
[16856](#), [16861](#), [16927](#), [16984](#), [17023](#),
[17029](#), [17367](#), [17372](#), [17383](#), [17390](#),
[17481](#), [17495](#), [17695](#), [17748](#), [18414](#),
[21811](#), [21818](#), [21824](#), [23887](#), [28677](#)
- _kernel_msg_expandable_-
error:nnnn
..... [306](#), [12338](#), [12774](#), [15826](#), [17882](#),
[17903](#), [18575](#), [21990](#), [22080](#), [23826](#)
- _kernel_msg_expandable_-
error:nnnnn . [306](#), [12338](#), [12791](#),
[15756](#), [16417](#), [21857](#), [22231](#), [31879](#)


```

\__kernel_msg_expandable_-
  error:nnnnnn ... 306, 12338, 31866
\__kernel_msg_fatal:nn ... 305, 12006
\__kernel_msg_fatal:nnn ... 305, 12006
\__kernel_msg_fatal:nnnn ... 305, 12006
\__kernel_msg_fatal:nnnnn 305, 12006
\__kernel_msg_fatal:nnnnnn 305, 12006
\__kernel_msg_info:nn ... 306, 12012
\__kernel_msg_info:nnn ... 306, 12012
\__kernel_msg_info:nnnn ... 306, 12012
\__kernel_msg_info:nnnnn ... 306, 12012
\__kernel_msg_info:nnnnnn 306, 12012
\__kernel_msg_new:nnn .....
... 305, 5699, 5701, 5710, 11960,
12057, 12059, 12061, 12063, 12065,
12135, 12186, 12234, 12254, 12256,
12258, 12260, 12262, 12264, 12266,
12268, 12272, 12275, 12282, 12284,
12291, 12298, 13846, 14549, 14551,
15632, 15647, 16445, 16447, 16449,
16451, 16453, 16455, 16457, 18080,
18082, 18084, 18086, 18088, 18090,
18092, 18094, 18096, 18098, 18100,
18102, 18104, 18106, 18110, 18467,
18469, 18471, 21807, 23481, 26396,
26398, 26403, 26657, 28629, 31887
\__kernel_msg_new:nnnn ..... 305,
5573, 5703, 5718, 5732, 5738, 5785,
5830, 5920, 6035, 6215, 6222, 6394,
11960, 12015, 12023, 12031, 12038,
12049, 12067, 12076, 12083, 12090,
12097, 12106, 12115, 12122, 12128,
12137, 12144, 12153, 12159, 12166,
12173, 12176, 12188, 12195, 12203,
12210, 12218, 12226, 12245, 12309,
12315, 13327, 13840, 13852, 13857,
15596, 15599, 15602, 15608, 15614,
15620, 15626, 16279, 16419, 16434,
22594, 22600, 22606, 22612, 22619,
23014, 23031, 23038, 23047, 26409,
26416, 26422, 26432, 26438, 26462,
26469, 26477, 26485, 26492, 26498,
26505, 26511, 26519, 26525, 26531,
26541, 26548, 26557, 26560, 26568,
26574, 26580, 26587, 26594, 26604,
26615, 26625, 26635, 26644, 26650,
28613, 28620, 28623, 28691, 31570
\__kernel_msg_set:nnn ... 305, 11960
\__kernel_msg_set:nnnn ... 305, 11960
\__kernel_msg_warning:nn .....
..... 306, 12012, 24597
\__kernel_msg_warning:nnn .....
..... 306, 12012, 24513,
24517, 24559, 24621, 24659, 24678

\__kernel_msg_warning:nnnn .....
..... 306, 12012, 24210, 24359
\__kernel_msg_warning:nnnnn .....
..... 306, 12012, 31838
\__kernel_msg_warning:nnnnnn ...
..... 306, 11943, 12012
\__kernel_patch_deprecation:nnNNpn
..... 1185, 31809, 31901,
31906, 32108, 32111, 32116, 32120,
32124, 32126, 32128, 32130, 32132,
32134, 32136, 32138, 32141, 32145,
32154, 32165, 32175, 32178, 32181,
32184, 32187, 32190, 32193, 32195,
32197, 32199, 32201, 32203, 32205,
32207, 32209, 32211, 32213, 32215
\__kernel_prefix_arg_replacement:wN
..... 2129
\g__kernel_prng_map_int . 306, 393,
510, 671, 964, 1358, 4013, 4015,
4017, 4020, 4812, 4814, 4818, 4823,
7838, 7839, 7845, 7846, 7888, 7890,
7892, 7894, 8459, 8462, 8470, 8473,
8484, 9225, 9962, 9964, 9966, 9969,
11397, 11398, 11403, 11405, 12577,
12579, 12586, 14151, 14154, 14158,
14161, 14172, 18444, 18447, 18451,
18454, 18465, 23368, 23370, 23390
\__kernel_primitive:NN .....
..... 182, 186, 187, 188, 189, 190,
191, 192, 193, 194, 195, 196, 197,
198, 199, 200, 201, 202, 203, 204,
205, 206, 207, 208, 209, 210, 211,
212, 213, 214, 215, 216, 217, 218,
219, 220, 221, 222, 223, 224, 225,
226, 227, 228, 229, 230, 231, 232,
233, 234, 235, 236, 237, 238, 239,
240, 241, 242, 243, 244, 245, 246,
247, 248, 249, 250, 251, 252, 253,
254, 255, 256, 257, 258, 259, 260,
261, 262, 263, 264, 265, 266, 267,
268, 269, 270, 271, 272, 273, 274,
275, 276, 277, 278, 279, 280,
281, 282, 283, 284, 285, 286, 287,
288, 289, 290, 291, 292, 293, 294,
295, 296, 297, 298, 299, 300, 301,
302, 303, 304, 305, 306, 307, 308,
309, 310, 311, 312, 313, 314, 315,
316, 317, 318, 319, 320, 321, 322,
323, 324, 325, 326, 327, 328, 329,
330, 331, 332, 333, 334, 335, 336,
337, 338, 339, 340, 341, 342, 343,
344, 345, 346, 347, 348, 349, 350,
351, 352, 353, 354, 355, 356, 357,
358, 359, 360, 361, 362, 363, 364,

```


365, 366, 367, 368, 369, 370, 371,
372, 373, 374, 375, 376, 377, 378,
379, 380, 381, 382, 383, 384, 385,
386, 387, 388, 389, 390, 391, 392,
393, 394, 395, 396, 397, 398, 399,
400, 401, 402, 403, 404, 405, 406,
407, 408, 409, 410, 411, 412, 413,
414, 415, 416, 417, 418, 419, 420,
421, 422, 423, 424, 425, 426, 427,
428, 429, 430, 431, 432, 433, 434,
435, 436, 437, 438, 439, 440, 441,
442, 443, 444, 445, 446, 447, 448,
449, 450, 451, 452, 453, 454, 455,
456, 457, 458, 459, 460, 461, 462,
463, 464, 465, 466, 467, 468, 469,
470, 471, 472, 473, 474, 475, 476,
477, 478, 479, 480, 481, 482, 483,
484, 485, 486, 487, 488, 489, 490,
491, 492, 493, 494, 495, 496, 497,
498, 499, 500, 501, 502, 503, 504,
505, 506, 507, 508, 509, 510, 511,
512, 513, 514, 515, 516, 517, 518,
519, 520, 521, 522, 523, 524, 525,
526, 527, 528, 529, 530, 531, 532,
533, 534, 535, 536, 537, 538, 539,
540, 541, 542, 543, 544, 545, 546,
547, 548, 549, 550, 551, 552, 553,
554, 555, 556, 557, 558, 559, 560,
561, 562, 563, 564, 565, 566, 567,
568, 569, 570, 571, 572, 573, 574,
575, 576, 577, 578, 579, 580, 581,
582, 583, 584, 585, 586, 587, 588,
589, 590, 591, 592, 593, 594, 595,
596, 597, 598, 599, 600, 601, 603,
604, 605, 606, 607, 608, 609, 611,
612, 613, 614, 615, 616, 617, 618,
619, 620, 621, 622, 623, 624, 625,
626, 627, 628, 629, 630, 631, 632,
633, 634, 635, 636, 637, 638, 639,
640, 641, 642, 643, 644, 645, 646,
647, 648, 649, 650, 651, 652, 653,
654, 655, 656, 657, 658, 659, 660,
661, 662, 663, 664, 665, 666, 667,
668, 669, 670, 671, 672, 673, 674,
675, 676, 677, 678, 679, 680, 681,
682, 683, 684, 685, 686, 687, 688,
689, 690, 691, 692, 693, 694, 695,
696, 697, 698, 703, 712, 714, 715,
716, 717, 718, 719, 720, 721, 722,
723, 724, 726, 728, 730, 731, 732,
734, 735, 736, 737, 738, 739, 741,
743, 744, 746, 748, 749, 750, 751,
752, 753, 754, 755, 756, 757, 758,
759, 760, 761, 762, 763, 764, 765,
766, 767, 768, 769, 770, 771, 772,
773, 774, 775, 776, 777, 778, 779,
780, 781, 782, 783, 784, 785, 786,
788, 789, 791, 792, 793, 794, 795,
796, 797, 798, 799, 800, 802, 803,
804, 805, 806, 807, 808, 809, 810,
811, 812, 813, 814, 815, 816, 817,
818, 819, 820, 821, 822, 823, 824,
825, 826, 827, 828, 829, 830, 831,
832, 833, 834, 835, 836, 837, 838,
839, 840, 841, 842, 843, 844, 845,
846, 847, 848, 849, 850, 851, 852,
853, 854, 855, 856, 857, 858, 859,
860, 861, 862, 863, 864, 865, 866,
867, 868, 869, 870, 871, 872, 873,
874, 875, 876, 877, 878, 879, 880,
881, 882, 883, 884, 885, 886, 887,
888, 889, 890, 891, 892, 893, 894,
895, 896, 897, 898, 899, 901, 902,
903, 904, 905, 906, 907, 908, 909,
910, 911, 912, 913, 914, 915, 917,
919, 921, 922, 923, 924, 925, 926,
927, 928, 929, 930, 931, 932, 933,
934, 935, 936, 937, 938, 939, 940,
941, 942, 943, 944, 945, 946, 947,
948, 949, 950, 951, 952, 953, 954,
955, 956, 957, 958, 959, 960, 961,
962, 963, 964, 966, 968, 969, 970,
971, 973, 974, 975, 976, 978, 979,
981, 983, 984, 985, 986, 987, 989,
991, 992, 993, 994, 996, 997, 998,
999, 1000, 1001, 1002, 1003, 1004,
1005, 1006, 1007, 1008, 1009, 1010,
1011, 1012, 1013, 1014, 1015, 1016,
1017, 1018, 1019, 1020, 1021, 1022,
1023, 1024, 1025, 1026, 1027, 1028,
1029, 1030, 1031, 1032, 1033, 1035,
1037, 1038, 1040, 1042, 1043, 1044,
1045, 1047, 1048, 1049, 1051, 1053,
1055, 1056, 1057, 1058, 1059, 1060,
1061, 1062, 1063, 1064, 1065, 1066,
1068, 1070, 1071, 1072, 1073, 1074,
1075, 1076, 1077, 1078, 1079, 1080,
1081, 1082, 1083, 1084, 1085, 1086,
1088, 1090, 1091, 1092, 1093, 1094,
1095, 1096, 1097, 1098, 1099, 1100,
1101, 1102, 1103, 1104, 1105, 1106,
1107, 1108, 1109, 1110, 1111, 1112,
1113, 1114, 1115, 1116, 1117, 1118,
1119, 1120, 1121, 1122, 1123, 1124,
1125, 1126, 1127, 1128, 1129, 1130,
1131, 1132, 1133, 1134, 1135, 1136,
1137, 1138, 1139, 1140, 1141, 1142,
1143, 1144, 1145, 1146, 1147, 1148,

- 1149, 1151, 1153, 1154, 1155, 1156,
- 1157, 1159, 1160, 1161, 1162, 1163,
- 1164, 1165, 1166, 1167, 1168, 1169,
- 1170, 1171, 1172, 1173, 1174, 1175,
- 1176, 1177, 1178, 1179, 1180, 1181
- _kernel_quark_new_conditional:Nn
..... 308, 3320, 3644,
10166, 13093, 14626, 23558, 29027
- _kernel_quark_new_test:N
..... 307, 370,
371, 373, 375, 3320, 3643, 4558,
4559, 8058, 8059, 8973, 9573, 9574,
11100, 13096, 13097, 29032, 30634
- _kernel_randint:n
..... 308, 308, 308, 718, 916,
919, 15857, 21832, 21844, 22002, 22087
- _kernel_randint:nn
308, 719, 15853, 22006, 22010, 22085
- _c_kernel_randint_max_int
919, 1358, 15850, 21831, 22000, 22084
- _kernel_register_log:N
..... 308, 2097, 8846, 14228,
14229, 14322, 14323, 14390, 14391
- _kernel_register_show:N
..... 308, 308,
407, 2097, 8842, 14224, 14318, 14386
- _kernel_register_show_aux:NN 2097
- _kernel_register_show_aux:nNN 2097
- _kernel_show:NN 2115
- _kernel_str_to_other:n ... 309,
309, 415, 417, 422, 4858, 4910, 4971
- _kernel_str_to_other_fast:n ...
..... 309, 4819, 4839,
4881, 5386, 12738, 12834, 23772, 24901
- _kernel_str_to_other_fast_-
loop:w 4881
- _kernel_sys_configuration_-
load:n 9277, 9326, 9332, 32102, 32104
- _kernel_tl_to_str:w
..... 309, 391, 1385, 3318, 3857,
3942, 4061, 4254, 4690, 4794, 13585
- keys commands:
- \l_keys_choice_int
..... 188, 190, 192, 192,
194, 14600, 14786, 14789, 14794, 14795
- \l_keys_choice_tl
.... 188, 190, 192, 194, 14600, 14793
- \keys_define:nn ... 187, 12148, 14627
- \keys_if_choice_exist:nnnTF
..... 197, 15565
- \keys_if_choice_exist_p:nnn
..... 197, 15565
- \keys_if_exist:nnTF
..... 196, 712, 15558, 15582
- \keys_if_exist_p:nn 196, 15558
- \l_keys_key_str 194, 14603, 14727,
14743, 15268, 15269, 15368, 15372,
15397, 15400, 15401, 15437, 15494
- \l_keys_key_tl 14603, 15269
- \keys_log:nn 197, 15573
- \l_keys_path_str 194, 14608,
14655, 14674, 14681, 14689, 14690,
14691, 14708, 14720, 14722, 14724,
14736, 14738, 14740, 14755, 14758,
14762, 14770, 14772, 14773, 14776,
14791, 14805, 14815, 14820, 14830,
14834, 14841, 14846, 14850, 14851,
14856, 14862, 14866, 14868, 14883,
14894, 14900, 14904, 14920, 14929,
14936, 14977, 15259, 15267, 15305,
15308, 15347, 15351, 15356, 15365,
15379, 15381, 15382, 15386, 15394,
15417, 15447, 15470, 15482, 15491
- \l_keys_path_tl . 14608, 14691, 14762
- \keys_set:nn 186,
190, 194, 194, 195, 14858, 14862, 15106
- \keys_set_filter:nnn 196, 15176
- \keys_set_filter:nnnN ... 196, 15176
- \keys_set_filter:nnnnN ... 196, 15176
- \keys_set_groups:nnn 196, 15176
- \keys_set_known:nn 195, 15135
- \keys_set_known:nnN . 195, 703, 15135
- \keys_set_known:nnnN 195, 15135
- \keys_show:nn 197, 197, 15573
- \l_keys_value_tl 194, 14618, 14920,
15350, 15354, 15360, 15371, 15382,
15401, 15414, 15439, 15449, 15477
- keys internal commands:
- _keys_bool_set:Nn
.. 14716, 14951, 14953, 14955, 14957
- _keys_bool_set_inverse:Nn
.. 14732, 14959, 14961, 14963, 14965
- _keys_check_groups: . 15309, 15317
- _keys_choice_find:n . 14749, 15488
- _keys_choice_find:nn 15488
- _keys_choice_make:
.. 14719, 14735, 14748, 14780, 14967
- _keys_choice_make:N 14748
- _keys_choice_make_aux:N 14748
- _keys_choices_make:nn
.. 14779, 14969, 14971, 14973, 14975
- _keys_choices_make:Nnn 14779
- _keys_cmd_set:nn
14720, 14722, 14724, 14736, 14738,
14740, 14772, 14773, 14790, 14800,
14856, 14862, 14868, 14936, 14977
- \c_keys_code_root_str
..... 708, 14593, 14801,

- 14805, 14850, 15379, 15397, 15413,
15427, 15499, 15561, 15569, 15588
- __keys_cs_set:NNpn
.... 14803, 14987, 14989, 14991,
14993, 14995, 14997, 14999, 15001
- __keys_default_inherit: 15343
- \c__keys_default_root_str
..... 14593, 14815,
14820, 15347, 15351, 15368, 15372
- __keys_default_set:n 14729, 14745,
14810, 15003, 15005, 15007, 15009
- __keys_define:n 14632, 14636
- __keys_define:nn 14632, 14636
- __keys_define:nnn 14627
- __keys_define_aux:nn 14636
- __keys_define_code:n . 14650, 14699
- __keys_define_code:w 14699
- __keys_execute:
.. 15273, 15313, 15335, 15339, 15377
- __keys_execute:nn
..... 14851, 15377, 15500, 15501
- __keys_execute_inherit: 14847, 15377
- __keys_execute_unknown: . 708, 15377
- \l__keys_filtered_bool ... 14614,
15111, 15118, 15119, 15162, 15168,
15169, 15204, 15210, 15211, 15223,
15230, 15231, 15312, 15333, 15338
- __keys_find_key_module:wNN
..... 14866, 15247
- __keys_find_key_module_auxi:Nw .
..... 15247
- __keys_find_key_module_auxii:Nw
..... 15247
- __keys_find_key_module_auxiii:Nn
..... 15247
- __keys_find_key_module_auxiii:Nw
..... 15290, 15292
- __keys_find_key_module_auxiv:Nw
..... 15247
- \l__keys_groups_clist ... 14602,
14827, 14828, 14835, 15307, 15322
- \c__keys_groups_root_str
.. 14593, 14830, 14834, 15305, 15308
- __keys_groups_set:n .. 14825, 15027
- __keys_inherit:n 14838, 15029
- \c__keys_inherit_root_str
..... 14593, 14841,
14846, 15356, 15365, 15386, 15394
- \l__keys_inherit_str 14610,
14849, 15266, 15399, 15490, 15494
- __keys_initialise:n
.. 14843, 15031, 15033, 15035, 15037
- __keys_meta_make:n ... 14854, 15047
- __keys_meta_make:nn .. 14854, 15049
- \l__keys_module_str 14605, 14628,
14631, 14633, 14675, 14858, 15128,
15131, 15133, 15250, 15255, 15265,
15268, 15274, 15413, 15414, 15417
- __keys_multichoice_find:n
..... 14751, 15488
- __keys_multichoice_make:
..... 14748, 14782, 15051
- __keys_multichoices_make:nn ...
.. 14779, 15053, 15055, 15057, 15059
- \l__keys_no_value_bool
..... 14606, 14638,
14643, 14701, 14917, 14926, 15249,
15254, 15345, 15438, 15448, 15476
- \l__keys_only_known_bool
.... 14607, 15110, 15116, 15117,
15161, 15166, 15167, 15203, 15208,
15209, 15222, 15228, 15229, 15409
- __keys_parent:n
.... 14755, 14758, 14762, 14846,
15356, 15365, 15386, 15394, 15505
- __keys_parent_auxi:w 15505
- __keys_parent_auxii:w 15505
- __keys_parent_auxiii:n 15505
- __keys_parent_auxiv:w 15505
- __keys_prop_put:Nn
.. 14863, 15069, 15071, 15073, 15075
- __keys_property_find:n 14648, 14659
- __keys_property_find_auxi:w . 14659
- __keys_property_find_auxii:w . 14659
- __keys_property_find_auxiii:w 14659
- __keys_property_find_auxiv:w . 14659
- __keys_property_find_err:w
..... 14664, 14672, 14693, 14694
- \l__keys_property_str 14613, 14649,
14652, 14655, 14661, 14662, 14688,
14696, 14704, 14705, 14708, 14711
- \c__keys_props_root_str
..... 14599, 14649,
14705, 14711, 14950, 14952, 14954,
14956, 14958, 14960, 14962, 14964,
14966, 14968, 14970, 14972, 14974,
14976, 14978, 14980, 14982, 14984,
14986, 14988, 14990, 14992, 14994,
14996, 14998, 15000, 15002, 15004,
15006, 15008, 15010, 15012, 15014,
15016, 15018, 15020, 15022, 15024,
15026, 15028, 15030, 15032, 15034,
15036, 15038, 15040, 15042, 15044,
15046, 15048, 15050, 15052, 15054,
15056, 15058, 15060, 15062, 15064,
15066, 15068, 15070, 15072, 15074,
15076, 15078, 15080, 15082, 15084,

- 15086, 15088, 15090, 15092, 15094,
15096, 15098, 15100, 15102, 15104
- _keys_quark_if_no_value:NTF ...
..... 14626, 15433
- _keys_quark_if_no_value_p:N . 14626
- \l_keys_relative_tl 14611,
15113, 15122, 15123, 15164, 15172,
15173, 15206, 15214, 15215, 15225,
15234, 15235, 15433, 15443, 15457,
15458, 15462, 15463, 15471, 15483
- \l_keys_selective_bool
.... 14614, 15112, 15120, 15121,
15163, 15170, 15171, 15205, 15212,
15213, 15224, 15232, 15233, 15271
- \l_keys_selective_seq
.. 14616, 15240, 15243, 15245, 15320
- _keys_set:nn .. 15106, 15165, 15244
- _keys_set:nnn 15106
- _keys_set_filter:nnnn 15176
- _keys_set_filter:nnnnnN 15176
- _keys_set_keyval:n .. 15132, 15247
- _keys_set_keyval:nn . 15132, 15247
- _keys_set_keyval:nnn 15247
- _keys_set_known:nnn 15135
- _keys_set_known:nnnnN 15135
- _keys_set_selective: 15247
- _keys_set_selective:nnn 15176
- _keys_set_selective:nnnn 15176
- _keys_show:Nnn 15573
- _keys_store_unused:
..... 15314, 15334, 15340, 15377
- _keys_store_unused:w
..... 15461, 15482, 15487
- _keys_store_unused_aux: 15377
- _keys_tmp:n 15526, 15538
- \l_keys_tmp_bool
..... 14619, 15319, 15326, 15331
- \l_keys_tmpa_tl 14619, 14867
- \l_keys_tmpb_tl . 14619, 14867, 14872
- _keys_trim_spaces:n
.... 14631, 14661, 14690, 14791,
15131, 15263, 15458, 15499, 15500,
15525, 15561, 15569, 15580, 15589
- _keys_trim_spaces_auxi:w ... 15525
- _keys_trim_spaces_auxii:w .. 15525
- _keys_trim_spaces_auxiii:w . 15525
- \c_keys_type_root_str
..... 14593, 14755, 14758, 14770
- _keys_undefine: 14840, 14877, 15101
- \l_keys_unused_clist
.. 702, 14617, 15138, 15144, 15149,
15151, 15152, 15179, 15186, 15191,
15193, 15194, 15435, 15445, 15473
- _keys_validate_forbidden: .. 14887
- _keys_validate_required: ... 14887
- \c_keys_validate_root_str 14593,
14894, 14900, 14904, 15381, 15400
- _keys_value_or_default:n
..... 15270, 15343
- _keys_value_requirement:nn ...
.. 14822, 14887, 14947, 15103, 15105
- _keys_variable_set:NnnN
..... 14933, 14979, 14981,
14983, 14985, 15085, 15087, 15089,
15091, 15093, 15095, 15097, 15099
- _keys_variable_set_required:NnnN
..... 14933,
15011, 15013, 15015, 15017, 15019,
15021, 15023, 15025, 15039, 15041,
15043, 15045, 15061, 15063, 15065,
15067, 15077, 15079, 15081, 15083
- keyval commands:
 _keyval_parse:Nnn
 .. 198, 685, 686, 14410, 14632, 15132
- keyval internal commands:
 _keyval_blank_key_error:w
 14530, 14536, 14543
- _keyval_blank_true:w . 14491, 14543
- _keyval_clean_up_active:w
 683, 14429, 14466, 14513
- _keyval_clean_up_other:w
 684, 14471, 14488
- _keyval_end_loop_active:w
 14415, 14509
- _keyval_end_loop_other:w
 14426, 14509
- _keyval_if_blank:w
 14491, 14530, 14536, 14540
- _keyval_if_empty:w 14540
- _keyval_if_recursion_tail:w ...
 14414, 14425, 14540
- _keyval_key:nnN .. 684, 14493, 14528
- _keyval_loop_active:NNw
 14411, 14412, 14521
- _keyval_loop_other:NNw
 14416, 14423, 14516, 14520
- _keyval_misplaced_equal_after_
 active_error:w
 682, 14437, 14442, 14495
- _keyval_misplaced_equal_in_
 split_error:w ... 14448, 14454,
 14458, 14463, 14479, 14485, 14495
- _keyval_pair:nnNN
 683, 683, 14465, 14487, 14528
- _keyval_split_active:w
 681, 682, 14419, 14427, 14447, 14511
- _keyval_split_active_auxi:w ...
 14428, 14434, 14467, 14512

- `__keyval_split_active_auxii:w` .. [682](#), [682](#), [14434](#)
 - `__keyval_split_active_auxiii:w` . [682](#), [14434](#)
 - `__keyval_split_active_auxiv:w` .. [682](#), [14434](#)
 - `__keyval_split_active_auxv:w` . [14434](#)
 - `__keyval_split_other:w` [14419](#), [14436](#), [14457](#), [14469](#), [14478](#)
 - `__keyval_split_other_auxi:w` ... [683](#), [14470](#), [14474](#), [14489](#)
 - `__keyval_split_other_auxii:w` . [14474](#)
 - `__keyval_split_other_auxiii:w` .. [683](#), [14474](#)
 - `__keyval_tmp:n` [14554](#), [14590](#)
 - `__keyval_tmp:NN` ... [684](#), [14408](#), [14526](#)
 - `__keyval_trim:nN` [14444](#), [14465](#), [14475](#), [14487](#), [14493](#), [14553](#)
 - `__keyval_trim_auxi:w` [14553](#)
 - `__keyval_trim_auxii:w` [14553](#)
 - `__keyval_trim_auxiii:w` [14553](#)
 - `__keyval_trim_auxiv:w` [14553](#)
 - `\kuten` [1137](#), [1171](#)
- L**
- `\L` [29176](#), [30587](#), [30928](#)
 - `\l` [29176](#), [30587](#), [30940](#)
 - `l3kernel` [258](#), [28699](#)
 - `l3kernel.charcat` [258](#), [28733](#)
 - `l3kernel.elapsedtime` [258](#), [28738](#)
 - `l3kernel.filedump` [258](#), [28751](#)
 - `l3kernel.filemdfivesum` [258](#), [28768](#)
 - `l3kernel.filemoddate` [258](#), [28780](#)
 - `l3kernel.filesize` [258](#), [28825](#)
 - `l3kernel.resettimer` [258](#), [28738](#)
 - `l3kernel.shellescape` [258](#), [28845](#)
 - `l3kernel.strcmp` [258](#), [28835](#)
 - `\label` [29184](#), [29191](#), [30876](#)
 - `\language` [326](#)
 - `\LARGE` [30859](#)
 - `\Large` [30860](#)
 - `\large` [30863](#)
 - `\lastallocatedtoks` [22712](#)
 - `\lastbox` [327](#)
 - `\lastkern` [328](#)
 - `\lastlinefit` [544](#)
 - `\lastnamedcs` [830](#)
 - `\lastnodechar` [1138](#)
 - `\lastnodesubtype` [1139](#)
 - `\lastnodetype` [545](#)
 - `\lastpenalty` [329](#)
 - `\lastsavedboxresourceindex` [915](#)
 - `\lastsavedimageresourceindex` [917](#)
 - `\lastsavedimageresourcepages` [919](#)
 - `\lastskip` [330](#)
 - `\lastxpos` [921](#)
 - `\lastypos` [922](#)
 - `\latelua` [831](#)
 - `\lateluafunction` [832](#)
 - LaTeX3 error commands:
 - `\LaTeX3_error:` [620](#)
 - `\lccode` [331](#)
 - `\leaders` [332](#)
 - `\left` [333](#)
 - left commands:
 - `\c_left_brace_str` [71](#), [985](#), [5171](#), [13144](#), [23848](#), [24233](#), [24237](#), [24257](#), [24270](#), [24294](#), [24777](#), [24858](#), [25890](#), [25925](#), [25949](#), [29278](#)
 - `\leftghost` [833](#)
 - `\lefthyphenmin` [334](#)
 - `\leftmarginkern` [691](#)
 - `\leftskip` [335](#)
 - legacy commands:
 - `\legacy_if:nTF` [264](#), [31238](#)
 - `\legacy_if_p:n` [264](#), [31238](#)
 - `\legno` [336](#)
 - `\let` [2](#), [22](#), [179](#), [180](#), [337](#)
 - `\latcharcode` [834](#)
 - `\letterspacefont` [692](#)
 - `\limits` [338](#)
 - `\LineBreak` [54](#), [55](#), [56](#), [57](#), [58](#), [59](#), [60](#), [61](#), [85](#), [87](#)
 - `\linedir` [835](#)
 - `\linedirection` [836](#)
 - `\linepenalty` [339](#)
 - `\lineskip` [340](#)
 - `\lineskiplimit` [341](#)
 - `\linewidth` [27530](#), [27531](#)
 - `\ln` [20411](#), [20414](#)
 - `ln` [214](#)
 - `\localbrokenpenalty` [837](#)
 - `\localinterlinepenalty` [838](#)
 - `\localleftbox` [843](#)
 - `\localrightbox` [844](#)
 - `\loccount` [12421](#), [12635](#)
 - `\loctoks` [22684](#), [22685](#), [22711](#)
 - `logb` [215](#)
 - `\long` [182](#), [342](#), [10830](#), [10834](#)
 - `\LongText` [50](#), [97](#)
 - `\looseness` [343](#)
 - `\lower` [344](#)
 - `\lowercase` [345](#)
 - `\lpcode` [693](#)
 - lua commands:
 - `\lua_escape:n` [257](#), [4674](#), [4689](#), [9384](#), [9397](#), [13244](#), [13421](#), [13486](#), [13505](#), [13584](#), [13633](#), [28658](#), [28660](#), [32004](#)

\lua_escape_x:n	32003	\luaescapebottomoffset	1284
\lua_now:n ..	257, 4675, 4678, 9383, 10337, 13243, 13420, 13482, 13504, 13573, 13632, 28659, 28660, 32006	\luaexpagedir	1285
\lua_now_x:n	32005	\luaexpageheight	1286
\lua_shipout:n	257, 28660	\luaexpageleftoffset	1258
\lua_shipout_e:n		\luaexpagerightoffset	1287
	257, 9396, 28660, 32008	\luaexpagetopoffset	1259
\lua_shipout_x:n	32007	\luaexpagewidth	1288
lua internal commands:		\luaexpardir	1289
_lua_escape:n	28655, 28665	\luaexpostexhyphenchar	1260
_lua_now:n	28655, 28660	\luaexpostthyphenchar	1261
_lua_shipout:n	28655, 28662	\luaexpreexhyphenchar	1262
\luabytecode	839	\luaexprehyphenchar	1263
\luabytecodecall	840	\luaexrevision	849
\luacopyinputnodes	841	\luaextrightghost	1290
\luadef	842	\lualatexsavecatcodetable	1264
\luaescapestring	845	\lualatexscantexttokens	1265
\luafunction	846	\lualatexsuppressfontnotfounderror ...	1235, 1274
\luafunctioncall	847	\lualatexsuppressifcsnameerror	1267
lualatex commands:		\lualatexsuppresslongererror	1268
\lualatex_if_engine:TF		\lualatexsuppressmathparerror	1270
	32011, 32013, 32015	\lualatexsuppressoutererror	1271
\lualatex_if_engine_p:	32009	\lualatextextdir	1291
\lualatexalignmark	1236	\lualatexUchar	1272
\lualatexaligntab	1237	\lualatexversion	27, 81, 850
\lualatexattribute	1238		
\lualatexattributedef	1239	M	
\lualatexbanner	848	\mag	346
\lualatexbodydir	1275	\mark	347
\lualatexboxdir	1276	\marks	546
\lualatexcatcodetable	1240	math commands:	
\lualatexclearmarks	1241	_c_math_subscript_token	
\lualatexcrampeddisplaystyle	1242		135, 575, 10590,
\lualatexcrampedscriptsriptstyle	1244		10645, 10703, 23302, 29094, 29129
\lualatexcrampedscriptstyle	1245	_c_math_superscript_token	135,
\lualatexcrampedtextstyle	1246		575, 10587, 10645, 10698, 23300, 29126
\lualatexfontid	1247	_c_math_toggle_token	
\lualatexformatname	1248		135, 574, 10581,
\lualatexgladers	1249		10645, 10679, 23296, 29091, 29120
\lualatexinitcatcodetable	1250	\mathaccent	348
\lualatexlatelua	1251	\mathbin	349
\lualatexleftghost	1277	\mathchar	350, 10829
\lualatexlocalbrokenpenalty	1278	\mathchardef	351
\lualatexlocalinterlinepenalty	1280	\mathchoice	352
\lualatexlocalleftbox	1281	\mathclose	353
\lualatexlocalrightbox	1282	\mathcode	354
\lualatexluaescapestring	1252	\mathdelimitersmode	851
\lualatexluafunction	1253	\mathdir	852
\lualatexmathdir	1283	\mathdirection	853
\lualatexmathstyle	1254	\mathdisplayskipmode	854
\lualatexnokerns	1255	\matheqnogapstep	855
\lualatexnoligs	1256	\mathinner	355
\lualatexoutputbox	1257	\mathnolimitsmode	856
		\mathop	356

- `\mathopen` 357
- `\mathoption` 857
- `\mathord` 358
- `\mathpenaltiesmode` 858
- `\mathpunct` 359
- `\mathrel` 360
- `\mathrulesfam` 859
- `\mathscriptboxmode` 861
- `\mathscriptcharmode` 862
- `\mathscriptsmode` 860
- `\mathstyle` 863
- `\mathsurround` 361
- `\mathsurroundmode` 864
- `\mathsurroundskip` 865
- `max` 215
- max commands:
 - `\c_max_char_int` 101, 8852, 10313, 23824
 - `\c_max_register_int` 101, 237, 939, 940, 1412, 5679, 7599, 8048, 12093, 12198, 12200, 22657, 22682, 22719
 - `\maxdeadcycles` 362
 - `\maxdepth` 363
 - `\mdfivesum` 777
 - `\mdseries` 30852
 - `\meaning` 364
 - `\medmuskip` 365
 - `\message` 366
 - `\MessageBreak` 85
- meta commands:
 - `.meta:n` 190, 15046
 - `.meta:nn` 190, 15048
- `\middle` 547
- `min` 215
- minus commands:
 - `\c_minus_inf_fp` 209, 218, 15910, 18936, 19020, 19353, 19890, 20737, 22262
 - `\c_minus_zero_fp` 208, 15910, 18932, 21456, 22260
- `\mkern` 367
- `mm` 219
- mode commands:
 - `\mode_if_horizontal:TF` 113, 9215
 - `\mode_if_horizontal_p:` 113, 9215
 - `\mode_if_inner:TF` 113, 9217
 - `\mode_if_inner_p:` 113, 9217
 - `\mode_if_math:TF` 113, 9219
 - `\mode_if_math_p:` 113, 9219
 - `\mode_if_vertical:TF` 114, 9213
 - `\mode_if_vertical_p:` 114, 9213
 - `\mode_leave_vertical:` 24, 2176, 28361
- `\month` 368, 1314, 9443
- `\moveleft` 369
- `\moveright` 370
- msg commands:
 - `\msg_critical:nn` 155, 170, 11719
 - `\msg_critical:nnn` 155, 11719
 - `\msg_critical:nnnn` 155, 11719
 - `\msg_critical:nnnnn` 155, 11719
 - `\msg_critical:nnnnnn` 155, 11719
 - `\msg_critical_text:n` 153, 11619, 11722
 - `\msg_error:nn` 155, 11727
 - `\msg_error:nnn` 155, 11727
 - `\msg_error:nnnn` 155, 11727
 - `\msg_error:nnnnn` 155, 11727
 - `\msg_error:nnnnnn` ... 155, 156, 11727
 - `\msg_error_text:n` .. 153, 11619, 11730
 - `\msg_expandable_error:nn` . 157, 12375
 - `\msg_expandable_error:nnn` 157, 12375
 - `\msg_expandable_error:nnnn` 157, 12375
 - `\msg_expandable_error:nnnnn` ... 157, 12375
 - `\msg_expandable_error:nnnnnn` ... 157, 12375
 - `\msg_fatal:nn` 155, 11706
 - `\msg_fatal:nnn` 155, 11706
 - `\msg_fatal:nnnn` 155, 11706
 - `\msg_fatal:nnnnn` 155, 11706
 - `\msg_fatal:nnnnnn` 155, 11706
 - `\msg_fatal_text:n` .. 153, 11619, 11709
 - `\msg_gset:nnn` 152, 11465
 - `\msg_gset:nnnn` 152, 11465
 - `\msg_if_exist:nnTF` 153, 11452, 11459, 11839
 - `\msg_if_exist_p:nn` 153, 11452
 - `\msg_info:nn` 156, 11756
 - `\msg_info:nnn` 156, 11756
 - `\msg_info:nnnn` 156, 11756
 - `\msg_info:nnnnn` 156, 11756
 - `\msg_info:nnnnnn` 156, 156, 11756, 12013
 - `\msg_info_text:n` ... 154, 11619, 11758
 - `\msg_interrupt:nnn` 32017
 - `\msg_line_context:` 153, 601, 1827, 11522, 14550, 14552, 22601
 - `\msg_line_number:` 153, 11522
 - `\msg_log:n` 32019
 - `\msg_log:nn` 156, 11778
 - `\msg_log:nnn` 156, 11778
 - `\msg_log:nnnn` 156, 11778
 - `\msg_log:nnnnn` 156, 11778
 - `\msg_log:nnnnnn` .. 156, 156, 8029, 10138, 10151, 11429, 11778, 12490, 12687, 13775, 15576, 15799, 28592
 - `\msg_log_eval:Nn` . 268, 8849, 8986, 14231, 14325, 14393, 18164, 31367
 - `\g_msg_module_documentation_prop` 154
 - `\msg_module_name:n` 154, 11532, 11638, 11656, 11737, 11759

- \g_msg_module_name_prop
..... [154](#), [154](#), [11646](#), [11658](#), [11659](#)
 - \msg_module_type:n
..... [153](#), [154](#), [154](#), [11637](#), [11650](#)
 - \g_msg_module_type_prop
..... [154](#), [154](#), [11646](#), [11652](#), [11653](#)
 - \msg_new:nnn [152](#), [11465](#), [11963](#)
 - \msg_new:nnnn . [152](#), [600](#), [11465](#), [11961](#)
 - \msg_none:nn [156](#), [11790](#)
 - \msg_none:nnn [156](#), [11790](#)
 - \msg_none:nnnn [156](#), [11790](#)
 - \msg_none:nnnnn [156](#), [11790](#)
 - \msg_redirect_class:nn ... [158](#), [11912](#)
 - \msg_redirect_module:nnn . [158](#), [11912](#)
 - \msg_redirect_name:nnn ... [158](#), [11903](#)
 - \msg_see_documentation_text:n ...
..... [154](#), [11656](#)
 - \msg_set:nnn [152](#), [11465](#), [11967](#)
 - \msg_set:nnnn [152](#), [11465](#), [11965](#)
 - \msg_show:nn [268](#), [11791](#)
 - \msg_show:nnn [268](#), [11791](#)
 - \msg_show:nnnn [268](#), [11791](#)
 - \msg_show:nnnnn [268](#), [11791](#)
 - \msg_show:nnnnnn
..... [268](#), [268](#), [497](#), [561](#), [598](#),
[8027](#), [10136](#), [10150](#), [11427](#), [11791](#),
[12489](#), [12686](#), [13774](#), [15574](#), [15797](#),
[23397](#), [23405](#), [26091](#), [26100](#), [28589](#)
 - \msg_show_eval:Nn [268](#), [8845](#), [8984](#),
[14227](#), [14321](#), [14389](#), [18162](#), [31367](#)
 - \msg_show_item:n
.. [268](#), [268](#), [8037](#), [10146](#), [10155](#), [31372](#)
 - \msg_show_item:nn
..... [268](#), [598](#), [11437](#), [31372](#)
 - \msg_show_item_unbraced:n [268](#), [31372](#)
 - \msg_show_item_unbraced:nn . [268](#),
[626](#), [12497](#), [12694](#), [15584](#), [28608](#), [31372](#)
 - \msg_term:n [32021](#)
 - \msg_term:nn [156](#), [11784](#)
 - \msg_term:nnn [156](#), [11784](#)
 - \msg_term:nnnn [156](#), [11784](#)
 - \msg_term:nnnnn [156](#), [11784](#)
 - \msg_term:nnnnnn [156](#), [11784](#)
 - \msg_warning:nn [155](#), [11734](#)
 - \msg_warning:nnn [155](#), [11734](#)
 - \msg_warning:nnnn [155](#), [11734](#)
 - \msg_warning:nnnnn [155](#), [11734](#)
 - \msg_warning:nnnnnn [155](#), [11734](#), [12012](#)
 - \msg_warning_text:n [153](#), [11619](#), [11736](#)
- msg internal commands:
- __msg_chk_free:nn [11457](#), [11467](#)
 - __msg_chk_if_free:nn [11457](#)
 - __msg_class_chk_exist:nTF
.. [11826](#), [11841](#), [11908](#), [11918](#), [11923](#)
 - \l__msg_class_loop_seq
..... [612](#), [11835](#), [11927](#),
[11935](#), [11945](#), [11946](#), [11949](#), [11951](#)
 - __msg_class_new:nn ... [609](#), [613](#),
[11667](#), [11706](#), [11719](#), [11727](#), [11734](#),
[11756](#), [11778](#), [11784](#), [11790](#), [11791](#)
 - \l__msg_class_tl . [610](#), [612](#), [11831](#),
[11848](#), [11861](#), [11882](#), [11886](#), [11889](#),
[11897](#), [11936](#), [11938](#), [11940](#), [11954](#)
 - \c__msg_coding_error_text_tl ...
..... [11490](#),
[12018](#), [12026](#), [12052](#), [12070](#), [12079](#),
[12086](#), [12100](#), [12109](#), [12131](#), [12140](#),
[12147](#), [12156](#), [12162](#), [12169](#), [12179](#),
[12191](#), [12206](#), [12213](#), [12221](#), [12229](#)
 - \c__msg_continue_text_tl [11490](#), [11539](#)
 - __msg_critical_code:nnnnnn .. [12007](#)
 - \c__msg_critical_text_tl [11490](#), [11724](#)
 - \l__msg_current_class_tl
..... [612](#), [11831](#), [11843](#), [11881](#),
[11886](#), [11889](#), [11897](#), [11926](#), [11940](#)
 - __msg_error_code:nnnnnn [12011](#)
 - __msg_expandable_error:n
..... [621](#), [12321](#), [12341](#)
 - __msg_expandable_error:w [620](#), [12321](#)
 - __msg_expandable_error_module:nn
..... [12375](#)
 - __msg_fatal_code:nnnnnn [12006](#)
 - __msg_fatal_exit: [11706](#)
 - \c__msg_fatal_text_tl . [11490](#), [11711](#)
 - \c__msg_help_text_tl .. [11490](#), [11549](#)
 - \l__msg_hierarchy_seq
[610](#), [611](#), [11834](#), [11864](#), [11874](#), [11879](#)
 - \l__msg_internal_tl [11444](#),
[11575](#), [11581](#), [11717](#), [11815](#), [11821](#)
 - __msg_interrupt:n [11576](#), [11585](#)
 - __msg_interrupt:Nnnn [11529](#)
 - __msg_interrupt:NnnnN
..... [11529](#), [11708](#), [11721](#), [11729](#)
 - __msg_interrupt_more_text:n ...
..... [602](#), [11558](#)
 - __msg_interrupt_text:n [11558](#)
 - __msg_interrupt_wrap:nn
..... [11537](#), [11547](#), [11558](#)
 - __msg_kernel_class_new:nN
..... [614](#), [11968](#),
[12006](#), [12007](#), [12011](#), [12012](#), [12013](#)
 - __msg_kernel_class_new_aux:nN [11968](#)
 - \c__msg_more_text_prefix_tl
.. [11450](#), [11476](#), [11485](#), [11534](#), [11551](#)

- \l_msg_name_str
 11445, 11532, 11565, 11569, 11737,
 11745, 11749, 11759, 11767, 11771
 - \c_msg_no_info_text_tl 11490, 11541
 - _msg_no_more_text:nnnn 11529
 - \c_msg_on_line_text_tl 11490, 11525
 - _msg_redirect:nnn 11912
 - _msg_redirect_loop_chk:nnn ...
 11912, 11954
 - _msg_redirect_loop_list:n .. 11912
 - \l_msg_redirect_prop
 11833, 11861, 11906, 11909
 - \c_msg_return_text_tl
 11490, 12021, 12029, 12036
 - _msg_show:n 608, 11791
 - _msg_show:nn 11791
 - _msg_show:w 11791
 - _msg_show_dot:w 11791
 - _msg_show_eval:nnN 31367
 - _msg_text:n 11619
 - _msg_text:nn 11619
 - \c_msg_text_prefix_tl
 621, 11450, 11454, 11474,
 11483, 11538, 11548, 11742, 11764,
 11781, 11787, 11794, 12344, 12380
 - \l_msg_text_str
 11445, 11531, 11563, 11568, 11736,
 11741, 11748, 11758, 11763, 11770
 - _msg_tmp:w 12322, 12335
 - \c_msg_trouble_text_tl 11490
 - _msg_use:nnnnnn 11677, 11836
 - _msg_use_code: 610, 11836
 - _msg_use_hierarchy:nwN 11836
 - _msg_use_none_delimit_by_s_-
 stop:w 11449, 11867, 12403
 - _msg_use_redirect_module:n ...
 611, 11836
 - _msg_use_redirect_name:n ... 11836
 - \mskip 371
 - \muexpr 548
 - multichoice commands:
 .multichoice: 190, 15050
 - multichoices commands:
 .multichoices:nn 190, 15050
 - \multiply 372
 - \muskip 373, 10837
 - muskip commands:
 \c_max_muskip 185, 14394
 - \muskip_add:Nn 183, 14370
 - \muskip_const:Nn
 183, 14338, 14394, 14395
 - \muskip_eval:n
 184, 184, 14341, 14382, 14389, 14393
 - \muskip_gadd:Nn 183, 14370
 - .muskip_gset:N 190, 15060
 - \muskip_gset:Nn 184, 14360
 - \muskip_gset_eq:NN 184, 14366
 - \muskip_gsub:Nn 184, 14370
 - \muskip_gzero:N ... 183, 14344, 14353
 - \muskip_gzero_new:N 183, 14350
 - \muskip_if_exist:NTF
 183, 14351, 14353, 14356
 - \muskip_if_exist_p:N 183, 14356
 - \muskip_log:N 185, 14390
 - \muskip_log:n 185, 14390
 - \muskip_new:N
 ... 183, 183, 14332, 14340, 14351,
 14353, 14396, 14397, 14398, 14399
 - .muskip_set:N 190, 15060
 - \muskip_set:Nn 184, 14360
 - \muskip_set_eq:NN 184, 14366
 - \muskip_show:N 184, 14386
 - \muskip_show:n 185, 680, 14388
 - \muskip_sub:Nn 184, 14370
 - \muskip_use:N . 184, 184, 14383, 14384
 - \muskip_zero:N 183, 183, 14344, 14351
 - \muskip_zero_new:N 183, 14350
 - \g_tmpa_muskip 185, 14396
 - \l_tmpa_muskip 185, 14396
 - \g_tmpb_muskip 185, 14396
 - \l_tmpb_muskip 185, 14396
 - \c_zero_muskip 185, 14345, 14347, 14394
 - \muskipdef 374
 - \mutoglu 549
- N
- \n 28848, 28850, 28852
 - nan 218
 - nc 219
 - nd 219
 - \newbox 501
 - \newcatcodetable 22286
 - \newcount 501
 - \newdimen 501
 - \newlinechar 84, 375
 - \next 48, 94, 102, 105, 108, 116
 - \NG 29177, 30588, 30929
 - \ng 29177, 30588, 30941
 - \noalign 376
 - \noautospacing 1140
 - \noautoxspacing 1141
 - \noboundary 377
 - \nobreakspace 30884
 - \noexpand 85, 96, 99, 119, 378
 - \nohrule 866
 - \noindent 379
 - \nokerns 867
 - \noligs 868

- | | | |
|--|-------------------|--|
| <code>\nolimits</code> | 380 | or commands: |
| <code>\nonscript</code> | 381 | <code>\or:</code> <i>102, 419, 421, 734, 1359,</i> |
| <code>\nonstopmode</code> | 382 | 1921, 1922, 1923, 1924, 1925, 1926, |
| <code>\normaldeviate</code> | 923 | 1927, 1928, 1929, 2568, 2569, 2570, |
| <code>\normalend</code> <i>1331, 1332, 12417, 12447,</i> | <i>12631</i> | 2571, 2572, 4961, 5037, 5262, 5263, |
| <code>\normaleveryjob</code> | 1333 | 5264, 5265, 5266, 6304, 6305, <u>8048</u> , |
| <code>\normalexpanded</code> | 1342 | 8632, 8633, 8634, 8635, 8636, 8637, |
| <code>\normalfont</code> | 30847 | 8638, 8639, 8640, 8641, 8642, 8643, |
| <code>\normalhoffset</code> | 1345 | 8644, 8645, 8646, 8647, 8648, 8649, |
| <code>\normalinput</code> | 1334 | 8650, 8651, 8652, 8653, 8654, 8655, |
| <code>\normalitaliccorrection</code> | <i>1344, 1346</i> | 8656, 8665, 8666, 8667, 8668, 8669, |
| <code>\normallanguage</code> | 1335 | 8670, 8671, 8672, 8673, 8674, 8675, |
| <code>\normalleft</code> | <i>1352, 1353</i> | 8676, 8677, 8678, 8679, 8680, 8681, |
| <code>\normalmathop</code> | 1336 | 8682, 8683, 8684, 8685, 8686, 8687, |
| <code>\normalmiddle</code> | 1354 | 8688, 8689, 10363, 10367, 10370, |
| <code>\normalmonth</code> | 1337 | 10372, 10373, 10375, 10377, 10379, |
| <code>\normalouter</code> | 1338 | 10380, 10382, 10384, 10386, 10388, |
| <code>\normalover</code> | 1339 | 10416, 12913, 12914, 12915, 12916, |
| <code>\normalright</code> | 1355 | 12917, 12918, 12919, 15956, 15957, |
| <code>\normalshowtokens</code> | 1348 | 15958, 16207, 16222, 16223, 16606, |
| <code>\normalsize</code> | 30864 | 16607, 16632, 17923, 17924, 17925, |
| <code>\normalunexpanded</code> | 1341 | 17961, 18632, 18633, 18634, 18757, |
| <code>\normalvcenter</code> | 1340 | 18842, 18928, 18929, 18930, 18931, |
| <code>\normalvoffset</code> | 1347 | 18932, 18933, 18934, 18935, 18936, |
| <code>\nospaces</code> | 869 | 19015, 19018, 19354, 19355, 19369, |
| notexpanded commands: | | 19370, 19384, 19668, 19891, 19916, |
| <code>\notexpanded: <token></code> | <i>142</i> | 19922, 19923, 19924, 19925, 19926, |
| <code>\novrule</code> | 870 | 20075, 20110, 20112, 20120, 20313, |
| <code>\nulldelimiterspace</code> | 383 | 20364, 20367, 20376, 20491, 20514, |
| <code>\nullfont</code> | 384 | 20515, 20547, 20548, 20552, 20605, |
| <code>\num</code> | <i>202</i> | 20606, 20646, 20651, 20661, 20666, |
| <code>\number</code> | 385 | 20676, 20681, 20691, 20696, 20706, |
| <code>\numexpr</code> | 550 | 20711, 20721, 20726, 21253, 21254, |
| 21299, 21384, 21387, 21399, 21405, | | |
| 21452, 21454, 21455, 21465, 21471, | | |
| 21548, 21549, 21556, 21602, 21603, | | |
| 21610, 21676, 21677, 21897, 22180, | | |
| 22181, 22182, 22259, 22260, 22261, | | |
| 23148, 23149, 23342, 23343, 23632, | | |
| 23633, 23634, 23635, 23898, 23899, | | |
| 23900, 23901, 23902, 25194, 25248, | | |
| 25607, 25608, 31749, 31750, 31751 | | |
| <code>\oradical</code> | | |
| <code>\outer</code> | | |
| <code>\output</code> | | |
| <code>\outputbox</code> | | |
| <code>\outputmode</code> | | |
| <code>\outputpenalty</code> | | |
| <code>\over</code> | | |
| <code>\overfullrule</code> | | |
| <code>\overline</code> | | |
| <code>\overwithdelims</code> | | |
- O**
- | | |
|---|----------------------------|
| <code>\O</code> | 29178, 30589, 30930, 31220 |
| <code>\o</code> | 29178, 30589, 30942, 31221 |
| <code>\odelcode</code> | 1175 |
| <code>\odelimiter</code> | 1176 |
| <code>\OE</code> | 29179, 30590, 30931 |
| <code>\oe</code> | 29179, 30590, 30943 |
| <code>\omathaccent</code> | 1177 |
| <code>\omathchar</code> | 1178 |
| <code>\omathchardef</code> | 1179 |
| <code>\omathcode</code> | 1180 |
| <code>\omit</code> | 386 |
| one commands: | |
| <code>\c_minus_one</code> | 31915 |
| <code>\c_one_degree_fp</code> <i>209, 218, 17511, 18167</i> | |
| <code>\openin</code> | 387 |
| <code>\openout</code> | 388 |
| <code>\or</code> | 389 |

P

\PackageError	88, 96	\pdfextension	880
\pagebottomoffset	872	\pdffeedback	881
\pagedepth	397	\pdffiledump	655
\pagedir	873	\pdffilemoddate	656
\pagedirection	874	\pdffilesizes	657
\pagediscards	551	\pdffirstlineheight	658
\pagefilllstretch	398	\pdffontattr	588
\pagefillstretch	399	\pdffontexpand	659
\pagefilstretch	400	\pdffontname	589
\pagefistretch	1142	\pdffontobjnum	590
\pagegoal	401	\pdffontsize	660
\pageheight	925	\pdfgamma	591
\pageleftoffset	875	\pdfgentounicode	594
\pagerightoffset	876	\pdfglyptounicode	595
\pageshrink	402	\pdfhorigin	596
\pagestretch	403	\pdfignoreddimen	661
\pagetopoffset	877	\pdfimageapplygamma	592
\pagetotal	404	\pdfimagegamma	593
\pagewidth	926	\pdfimagehicolor	597
\par	10, 11, 11, 11, 12, 12, 12, 13, 13, 13, 14, 14, 14, 161, 329, 405, 1067, 26926, 26928, 26932, 26937, 26942, 26947, 26954, 26959, 26966, 26971, 26991	\pdfimageresolution	598
\pardir	878	\pdfincludechars	599
\pardirection	879	\pdfinclusioncopyfonts	600
\parfillskip	406	\pdfinclusionerrorlevel	601
\parindent	407	\pdfinfo	603
\parshape	408	\pdfinserttht	662
\parshapedimen	552	\pdflastannot	604
\parshapeindent	553	\pdflastlinedepth	663
\parshapelength	554	\pdflastlink	605
\parskip	409	\pdflastobj	606
\patterns	410	\pdflastxform	607
\pausing	411	\pdflastximage	608
pc	219	\pdflastximagecolordepth	609
\pdfadjustspacing	649	\pdflastximagepages	611
\pdfannot	577	\pdflastxpos	664
\pdfcatalog	578	\pdflastypos	665
\pdfcolorstack	580	\pdflinkmargin	612
\pdfcolorstackinit	581	\pdfliteral	613
\pdfcompresslevel	579	\pdfmajorversion	614
\pdfcopyfont	650	\pdfmapfile	666
\pdfcreationdate	582	\pdfmapline	667
\pdfdecimaldigits	583	\pdfmdfivesum	668
\pdfdest	584	\pdfminorversion	615
\pdfdestmargin	585	\pdfnames	616
\pdfdraftmode	651	\pdfnoligatures	669
\pdfeachlinedepth	652	\pdfnormaldeviate	670
\pdfeachlineheight	653	\pdfobj	617
\pdfelapsedtime	654	\pdfobjcompresslevel	618
\pdfendlink	586	\pdfoutline	619
\pdfendthread	587	\pdfoutput	620
		\pdfpageattr	621
		\pdfpagebox	623
		\pdfpageheight	671
		\pdfpageref	624
		\pdfpageresources	625

- \pdfpagesattr 622, 626
- \pdfpagewidth 672
- \pdfpkmode 673
- \pdfpkresolution 674
- \pdfprimitive 675
- \pdfprotrudechars 676
- \pdfpxdimen 677
- \pdfrandomseed 678
- \pdfrefobj 627
- \pdfrefxform 628
- \pdfrefximage 629
- \pdfresettimer 679
- \pdfrestore 630
- \pdfretval 631
- \pdfsave 632
- \pdfsavepos 680
- \pdfsetmatrix 633
- \pdfsetrandomseed 682
- \pdfshellescape 683
- \pdfstartlink 634
- \pdfstartthread 635
- \pdfstrcmp 22, 412, 681
- \pdfsuppressptexinfo 636
- pdftex commands:
 - \pdftex_if_engine:TF 32025, 32027, 32029
 - \pdftex_if_engine_p: 32023
 - \pdftexbanner 686
 - \pdftexrevision 687
 - \pdftexversion 76, 688
 - \pdfthread 637
 - \pdfthreadmargin 638
 - \pdftracingfonts 684, 1226, 1227
 - \pdftrailer 639
 - \pdfuniformdeviate 685
 - \pdfuniquerename 640
 - \pdfvariable 882
 - \pdfvorigin 641
 - \pdfxform 642
 - \pdfxformname 643
 - \pdfximage 644
 - \pdfximagebbox 645
- peek commands:
 - \peek_after:Nw 115, 139, 139, 139, 10914, 10927, 10955, 31738
 - \peek_catcode:NTF 139, 11010
 - \peek_catcode_collect_inline:Nn 274, 31718
 - \peek_catcode_ignore_spaces:NTF 140, 11024
 - \peek_catcode_remove:NTF . 140, 11010
 - \peek_catcode_remove_ignore_spaces:NTF 140, 11024
 - \peek_charcode:NTF 140, 11010
 - \peek_charcode_collect_inline:Nn 274, 31718
 - \peek_charcode_ignore_spaces:NTF 140, 11024
 - \peek_charcode_remove:NTF 140, 11010
 - \peek_charcode_remove_ignore_spaces:NTF 141, 11024
 - \peek_gafter:Nw 139, 139, 10914
 - \peek_meaning:NTF 141, 11010
 - \peek_meaning_collect_inline:Nn 274, 31718
 - \peek_meaning_ignore_spaces:NTF 141, 11024
 - \peek_meaning_remove:NTF . 141, 11010
 - \peek_meaning_remove_ignore_spaces:NTF 141, 11024
 - \peek_N_type:TF 142, 11047, 11084, 11086
 - \peek_remove_spaces:n 274, 585, 10923, 11033, 11038, 11043
- peek internal commands:
 - __peek_collect:N 1183, 31718
 - __peek_collect:NNn 31718
 - __peek_collect_remove:nw 31718
 - \l__peek_collect_tl 1183, 31717, 31729, 31731, 31756, 31761
 - __peek_collect_true:w .. 1183, 31718
 - __peek_execute_branches_...: . 1183
 - __peek_execute_branches_catcode: 585, 10977, 31719
 - __peek_execute_branches_catcode_aux: 10977
 - __peek_execute_branches_catcode_auxii:N 10977
 - __peek_execute_branches_catcode_auxiii: 10977
 - __peek_execute_branches_charcode: 585, 10977, 31721
 - __peek_execute_branches_meaning: 585, 10969, 31723
 - __peek_execute_branches_N_type: 11047
 - __peek_false:w 586, 1183, 10907, 10925, 10936, 10950, 10974, 10997, 11007, 11064, 11077, 31730
 - __peek_false_aux:n 1183, 31731, 31732
 - __peek_N_type:w 11047
 - __peek_N_type_aux:nw 11047
 - __peek_remove_spaces: 10923
 - \l__peek_search_tl 581, 584, 1183, 10906, 10943, 10994, 11004, 31728
 - \l__peek_search_token 581, 1183, 10905, 10942, 10971, 31727

- __peek_tmp:w 10907, 10921, 11048, 11070
- __peek_token_generic:NNTF 585, 586, 10957, 10959, 10961, 11081, 11085, 11087
- __peek_token_generic_aux:NNTF . 10939, 10958, 10964
- __peek_token_remove_generic:NNTF 585, 10957, 10965, 10967
- __peek_true:w 586, 1183, 10907, 10949, 10972, 10995, 11005, 11062, 11076, 11077, 31737
- __peek_true_aux:w 582, 583, 10907, 10920, 10927, 10928, 10944, 10958, 31738, 31739, 31757
- __peek_true_remove:w 582, 583, 10918, 10933, 10964, 31762
- __peek_use_none_delimit_by_s_-stop:w 586, 10913, 11060
- \penalty 412
- \pi 16917, 16918
- pi 218
- \pm 18382, 18383
- \postbreakpenalty 1143
- \postdisplaypenalty 413
- \postexhyphenchar 883
- \posthyphenchar 884
- \prebinoppenalty 885
- \prebreakpenalty 1144
- \predisplaydirection 555
- \predisplaygapfactor 886
- \predisplaypenalty 414
- \predisplaysize 415
- \preexhyphenchar 887
- \prehyphenchar 888
- \prerelpenalty 889
- \pretolerance 416
- \prevdepth 417
- \prevgraf 418
- prg commands:
 - \prg_break: 115, 452, 491, 492, 597, 598, 1033, 2173, 4404, 5324, 5340, 5458, 5508, 5614, 5618, 5660, 5755, 5802, 5855, 5861, 6092, 6173, 6345, 6486, 7783, 7816, 7859, 7904, 8432, 9226, 11363, 11384, 11413, 13394, 16018, 16027, 18047, 18067, 18068, 18278, 18279, 18292, 18392, 18393, 18394, 21786, 21838, 22062, 22928, 23003, 23337, 23411, 23441, 23442, 23443, 23444, 23445, 23446, 23798, 23802, 25709, 25736, 31459, 31465
 - \prg_break:n 115, 115, 2173, 4406, 5226, 5234, 5246, 7657, 7796, 8442, 9226, 11258, 15791, 16034, 25271
 - \prg_break_point: 115, 115, 651, 941, 942, 948, 1176, 2173, 4394, 5227, 5235, 5325, 5341, 5459, 5509, 5615, 5619, 5661, 5756, 5803, 5856, 5862, 6093, 6293, 6450, 7654, 7784, 7816, 7859, 7904, 7949, 7956, 8437, 9226, 11253, 11348, 11384, 11413, 13367, 15785, 16019, 16028, 18048, 18069, 18280, 18396, 21787, 21838, 22070, 22756, 22797, 22921, 22928, 23260, 23412, 23448, 23776, 25272, 25582, 25730, 31460
 - \prg_break_point:Nn 40, 114, 114, 337, 491, 510, 671, 2164, 4001, 4019, 4029, 4044, 4796, 4822, 4842, 7817, 7852, 7860, 7877, 7884, 7893, 8484, 9226, 9934, 9948, 9968, 9986, 11385, 11401, 11414, 12585, 12604, 14172, 18465, 23389
 - \prg_do_nothing: 9, 115, 381, 433, 481, 545, 560, 590, 659, 740, 911, 989, 1036, 2162, 2173, 2490, 2882, 2909, 3008, 3009, 3010, 3470, 3471, 3674, 4612, 4614, 5390, 6343, 7458, 7465, 7749, 7751, 9364, 9579, 9585, 9593, 9748, 9947, 9955, 10104, 10108, 10115, 11159, 11167, 11176, 12912, 13224, 13720, 13757, 13759, 15422, 16312, 16346, 16372, 16380, 17932, 21767, 22441, 22843, 23001, 23002, 23231, 23280, 24097, 24140, 24141, 24148, 24149, 25801, 25964
 - \prg_generate_conditional-variant:Nnn 108, 3102, 3291, 3313, 3831, 3841, 3852, 3875, 3891, 3908, 3919, 3993, 4290, 4311, 4700, 4713, 4721, 4752, 7590, 7658, 7752, 7754, 7768, 7770, 7772, 7774, 8982, 9766, 9780, 9781, 9924, 9926, 11292, 11293, 11341, 11365, 11376, 12444, 26792, 26794, 26798, 27423
 - \prg_map_break:Nn .. 114, 114, 337, 394, 557, 598, 2164, 4057, 4059, 4855, 4857, 7807, 7809, 9226, 10003, 10005, 11424, 11426, 12568, 12570
 - \prg_new_conditional:Nnn 106, 1539, 8932
 - \prg_new_conditional:Npnn .. 106, 106, 108, 308, 574, 585, 1522, 2070, 2761, 3275, 3283, 3293, 3303, 3452, 3823, 3833, 3843, 3859, 3867, 3923,

3939, 3950, 4275, 4292, 4313, 4348,
 4365, 4376, 4693, 4702, 4707, 5223,
 5232, 5247, 5255, 5942, 5976, 5995,
 7582, 8253, 8306, 8344, 8352, 8900,
 8905, 8932, 8974, 9006, 9066, 9081,
 9092, 9107, 9117, 9213, 9215, 9217,
 9219, 9595, 9877, 10667, 10672,
 10677, 10682, 10689, 10695, 10701,
 10706, 10711, 10716, 10721, 10726,
 10731, 10736, 10743, 10758, 10763,
 10798, 10844, 11336, 11343, 11452,
 12502, 13588, 13988, 13993, 14288,
 14296, 15558, 15565, 16202, 17359,
 18182, 18190, 18206, 23890, 23910,
 23932, 23978, 24002, 26788, 26790,
 26796, 27413, 29150, 30000, 31238
 \prg_new_eq_conditional:Nnn
 107, 1655, 3559, 3560, 4663, 4665,
 4667, 4669, 7487, 7489, 8021, 8022,
 8023, 8024, 8025, 8026, 8201, 8203,
 8932, 9002, 9004, 9684, 9686, 9873,
 9875, 11332, 11334, 13919, 13921,
 14262, 14264, 14356, 14358, 18180,
 18181, 22483, 22485, 26736, 26738
 \prg_new_protected_conditional:Nnn
 106, 1539, 8932
 \prg_new_protected_conditional:Npnn
 106, 1522, 3879, 3892, 3910,
 4715, 4723, 5364, 5373, 7639, 7748,
 7750, 7756, 7759, 7762, 7765, 8932,
 9342, 9757, 9767, 9769, 9891, 9895,
 11272, 11282, 11367, 12435, 12522,
 12542, 13203, 13339, 13516, 13518,
 13520, 13522, 13559, 13648, 22487,
 24324, 26105, 26110, 26123, 26125
 \prg_replicate:nn
 49, 82, 113, 532, 716,
 9165, 11566, 11746, 11768, 12744,
 15738, 15864, 19610, 20463, 20771,
 21027, 21073, 21110, 21633, 21641,
 22100, 22203, 23502, 24103, 24840,
 25201, 25227, 25374, 25382, 25806,
 26268, 26273, 26280, 26383, 26388
 \prg_return_false:
 107, 108, 318, 401, 487, 505,
 554, 554, 1048, 1516, 1582, 1590,
 1741, 1746, 1759, 1764, 1772, 1789,
 2073, 2771, 3280, 3288, 3299, 3309,
 3456, 3828, 3838, 3849, 3864, 3872,
 3888, 3902, 3916, 3930, 3946, 3961,
 4287, 4308, 4326, 4334, 4344, 4357,
 4371, 4385, 4698, 4705, 4711, 4719,
 4727, 5228, 5236, 5252, 5268, 5371,
 5380, 5946, 5949, 5952, 5979, 5982,
 5999, 6002, 6005, 7587, 7653, 7672,
 8251, 8283, 8288, 8311, 8349, 8357,
 8903, 8910, 8932, 8979, 9011, 9071,
 9087, 9097, 9113, 9123, 9214, 9216,
 9218, 9220, 9346, 9354, 9611, 9614,
 9760, 9774, 9880, 9915, 9921, 10670,
 10675, 10680, 10685, 10692, 10699,
 10704, 10709, 10714, 10719, 10724,
 10729, 10734, 10739, 10756, 10761,
 10766, 10771, 10804, 10807, 10819,
 10848, 10873, 10890, 10899, 11280,
 11290, 11339, 11359, 11374, 11455,
 12442, 12511, 12525, 12545, 13212,
 13344, 13371, 13529, 13551, 13565,
 13603, 13612, 13623, 13645, 13652,
 13991, 14010, 14025, 14026, 14292,
 14299, 15563, 15571, 16213, 16215,
 17374, 17386, 18187, 18201, 18214,
 22497, 22503, 23904, 23915, 23918,
 23923, 23927, 23928, 23936, 23939,
 23944, 23947, 23984, 23987, 24008,
 24011, 24331, 24336, 26151, 26789,
 26791, 26797, 27419, 27421, 29160,
 29163, 30003, 30007, 30010, 31243
 \prg_return_true:
 107, 108, 318, 388,
 402, 402, 487, 594, 647, 1046, 1048,
 1516, 1582, 1590, 1744, 1761, 1769,
 1774, 1787, 1792, 2073, 2763, 2771,
 3278, 3286, 3297, 3307, 3456, 3826,
 3836, 3847, 3862, 3870, 3886, 3900,
 3916, 3929, 3944, 3959, 4285, 4306,
 4324, 4342, 4355, 4373, 4384, 4698,
 4705, 4711, 4719, 4727, 5246, 5250,
 5258, 5271, 5371, 5380, 5946, 5952,
 5984, 5999, 6005, 7585, 7657, 7675,
 8283, 8309, 8347, 8355, 8903, 8908,
 8932, 8977, 9009, 9069, 9085, 9095,
 9111, 9121, 9214, 9216, 9218, 9220,
 9367, 9607, 9610, 9616, 9763, 9777,
 9881, 9911, 9921, 10670, 10675,
 10680, 10685, 10692, 10699, 10704,
 10709, 10714, 10719, 10724, 10729,
 10734, 10739, 10755, 10761, 10769,
 10818, 10871, 10897, 11278, 11288,
 11339, 11361, 11372, 11455, 12440,
 12509, 12514, 12516, 12528, 12548,
 13210, 13345, 13385, 13530, 13566,
 13601, 13610, 13621, 13651, 13991,
 14026, 14291, 14300, 15562, 15570,
 16206, 16211, 17369, 17392, 18185,
 18203, 18212, 22493, 23893, 23907,
 23915, 23918, 23923, 23927, 23939,
 23944, 23947, 23982, 24006, 24327,

- 24333, 26149, 26789, 26791, 26797,
27418, 29161, 30012, 30016, 31241
- \prg_set_conditional:Nnn
..... 106, 1539, 8932
- \prg_set_conditional:Npnn
..... 106, 107, 108, 1522,
1738, 1750, 1766, 1778, 8932, 13639
- \prg_set_eq_conditional:NNn
..... 107, 1655, 8932
- \prg_set_protected_conditional:Nnn
..... 106, 1539, 8932
- \prg_set_protected_conditional:Npnn
..... 106, 1522, 8932, 13354
- prg internal commands:
- _prg_break_point:Nn 337
- _prg_generate_conditional:nnNNNnnn
 1534, 1559, 1568
- _prg_generate_conditional:NNnnnnNw
 1568
- _prg_generate_conditional_-
 count:NNNnn 1539
- _prg_generate_conditional_-
 count:nnNNNnn 1539
- _prg_generate_conditional_-
 fast:nw 318, 319, 1568
- _prg_generate_conditional_-
 parm:NNNpnn 1522
- _prg_generate_conditional_-
 test:w 1568
- _prg_generate_F_form:wNNnnnnN 1611
- _prg_generate_p_form:wNNnnnnN .
 318, 1611
- _prg_generate_T_form:wNNnnnnN 1611
- _prg_generate_TF_form:wNNnnnnN
 1611
- _prg_p_true:w 320, 1611
- _prg_replicate:N 9165
- _prg_replicate 9165
- _prg_replicate_0:n 9165
- _prg_replicate_1:n 9165
- _prg_replicate_2:n 9165
- _prg_replicate_3:n 9165
- _prg_replicate_4:n 9165
- _prg_replicate_5:n 9165
- _prg_replicate_6:n 9165
- _prg_replicate_7:n 9165
- _prg_replicate_8:n 9165
- _prg_replicate_9:n 9165
- _prg_replicate_first:N 9165
- _prg_replicate_first -:n ... 9165
- _prg_replicate_first_0:n ... 9165
- _prg_replicate_first_1:n ... 9165
- _prg_replicate_first_2:n ... 9165
- _prg_replicate_first_3:n ... 9165
- _prg_replicate_first_4:n ... 9165
- _prg_replicate_first_5:n ... 9165
- _prg_replicate_first_6:n ... 9165
- _prg_replicate_first_7:n ... 9165
- _prg_replicate_first_8:n ... 9165
- _prg_replicate_first_9:n ... 9165
- _prg_set_eq_conditional:NNNn 1655
- _prg_set_eq_conditional:nnNnnNNw
 1663, 1671
- _prg_set_eq_conditional_F_-
 form:nnn 1671
- _prg_set_eq_conditional_F_-
 form:wNnnnn 1708
- _prg_set_eq_conditional_-
 loop:nnnnNw 1671
- _prg_set_eq_conditional_p_-
 form:nnn 1671
- _prg_set_eq_conditional_p_-
 form:wNnnnn 1702
- _prg_set_eq_conditional_T_-
 form:nnn 1671
- _prg_set_eq_conditional_T_-
 form:wNnnnn 1706
- _prg_set_eq_conditional_TF_-
 form:nnn 1671
- _prg_set_eq_conditional_TF_-
 form:wNnnnn 1704
- _prg_use_none_delimit_by_q_-
 recursion_stop:w
 1520, 1597, 1676, 1681, 1688
- \primitive 779
- prop commands:
- _c_empty_prop 151,
 589, 11095, 11105, 11109, 11112, 11338
- _prop_clear:N 145, 145,
 11108, 11115, 11135, 11138, 11143,
 11146, 11151, 11154, 25538, 28200
- _prop_clear_new:N 145, 11114
- _prop_const_from_keyval:Nn
 146, 11133, 27382, 27389
- _prop_count:N 147, 11262, 31441
- _prop_gclear:N 145, 11108, 11118
- _prop_gclear_new:N
 145, 1082, 11114, 27456, 27457
- _prop_get:Nn 115, 32031, 32033
- _prop_get:NnN 38, 39, 146,
 147, 11219, 28458, 28462, 28531, 28535
- _prop_get:NnNTF
 146, 148, 148, 5431, 11367,
 11861, 11881, 11936, 22372, 27644
- _prop_gpop:NnN 147, 11227
- _prop_gpop:NnNTF 147, 148, 11272
- .prop_gput:N 190, 15068

- \prop_gput:Nnn [146](#),
[5205](#), [5206](#), [5207](#), [5208](#), [5209](#), [5210](#),
[5211](#), [5212](#), [5213](#), [5214](#), [5215](#), [5216](#),
[5217](#), [5218](#), [5219](#), [11294](#), [11647](#),
[11649](#), [12425](#), [12471](#), [12639](#), [12669](#),
[22325](#), [27670](#), [27688](#), [27723](#), [27754](#)
- \prop_gput_if_new:Nnn . . . [146](#), [11315](#)
- \prop_gremove:Nn
- [147](#), [11203](#), [12482](#), [12679](#), [22323](#)
- \prop_gset_eq:NN [145](#), [11112](#), [11120](#),
[11145](#), [27458](#), [27460](#), [27622](#), [27624](#),
[27661](#), [27663](#), [27910](#), [28078](#), [28119](#)
- \prop_gset_from_keyval:Nn [145](#), [11133](#)
- \prop_hput:Nnn [11294](#)
- \prop_if_empty:NTF . [147](#), [11336](#), [31438](#)
- \prop_if_empty_p:N [147](#), [11336](#)
- \prop_if_exist:NTF
- [147](#), [11115](#), [11118](#), [11332](#), [14865](#)
- \prop_if_exist_p:N [147](#), [11332](#)
- \prop_if_in:NnTF
- [148](#), [11343](#), [11652](#), [11658](#)
- \prop_if_in_p:Nn [148](#), [11343](#)
- \prop_item:Nn [147](#),
[149](#), [11249](#), [11653](#), [11659](#), [32032](#), [32034](#)
- \prop_log:N [150](#), [11427](#)
- \prop_map_break:
- [149](#), [598](#), [11385](#), [11401](#), [11414](#), [11423](#)
- \prop_map_break:n [150](#), [11423](#)
- \prop_map_function:NN
- [149](#), [149](#), [268](#), [596](#), [598](#), [11267](#),
[11378](#), [11437](#), [12496](#), [12693](#), [28606](#)
- \prop_map_inline:Nn
- [149](#), [11394](#), [26290](#),
[27920](#), [27922](#), [27925](#), [27945](#), [27947](#),
[28021](#), [28038](#), [28099](#), [28101](#), [28105](#),
[28107](#), [28287](#), [28306](#), [28505](#), [28514](#)
- \prop_map_tokens:Nn
- [149](#), [149](#), [493](#), [11409](#)
- \prop_new:N [145](#),
[145](#), [5204](#), [11102](#), [11115](#), [11118](#),
[11128](#), [11129](#), [11130](#), [11131](#), [11132](#),
[11646](#), [11648](#), [11670](#), [11833](#), [12413](#),
[12627](#), [14865](#), [22276](#), [25459](#), [25460](#),
[27898](#), [27899](#), [27900](#), [28370](#), [28411](#)
- \prop_pop:NnN [146](#), [11227](#)
- \prop_pop:NnNTF [146](#), [148](#), [11272](#)
- .prop_put:N [190](#), [15068](#)
- \prop_put:Nnn [146](#),
[358](#), [587](#), [588](#), [11185](#), [11294](#), [11909](#),
[11925](#), [11942](#), [25696](#), [27667](#), [27685](#),
[27704](#), [27721](#), [27752](#), [27956](#), [27958](#),
[27964](#), [27966](#), [27975](#), [27981](#), [27989](#),
[28048](#), [28056](#), [28146](#), [28152](#), [28160](#),
[28167](#), [28311](#), [28371](#), [28373](#), [28375](#),
[28377](#), [28379](#), [28381](#), [28383](#), [28385](#),
[28387](#), [28389](#), [28391](#), [28393](#), [28395](#),
[28397](#), [28399](#), [28401](#), [28403](#), [28405](#)
- \prop_put_if_new:Nnn [146](#), [11315](#)
- \prop_rand_key_value:N . . . [269](#), [31436](#)
- \prop_remove:Nn [147](#), [11203](#),
[11906](#), [11921](#), [28500](#), [28503](#), [28507](#)
- \prop_set_eq:NN [145](#), [11109](#), [11120](#),
[11137](#), [25707](#), [27610](#), [27612](#), [27654](#),
[27656](#), [27907](#), [27916](#), [27918](#), [28071](#),
[28095](#), [28097](#), [28116](#), [28244](#), [28495](#)
- \prop_set_from_keyval:Nn
- [145](#), [589](#), [11133](#)
- \prop_show:N [150](#), [11427](#)
- \g_tmpa_prop [150](#), [11128](#)
- \l_tmpa_prop [150](#), [11128](#)
- \g_tmpb_prop [150](#), [11128](#)
- \l_tmpb_prop [150](#), [11128](#)
- prop internal commands:
- __prop_count:nn [11262](#)
- __prop_from_keyval:n [11133](#)
- __prop_from_keyval_key:n [11133](#)
- __prop_from_keyval_key:w [590](#), [11133](#)
- __prop_from_keyval_loop:w . . . [11133](#)
- __prop_from_keyval_split:Nw . [11133](#)
- __prop_from_keyval_value:n . . [11133](#)
- __prop_from_keyval_value:w
- [590](#), [11133](#)
- __prop_if_in:N [596](#), [11343](#)
- __prop_if_in:nwn [596](#), [11343](#)
- __prop_if_recursion_tail_stop:n
- [11100](#), [11164](#)
- \l__prop_internal_prop . . . [11132](#),
[11135](#), [11137](#), [11138](#), [11143](#), [11145](#),
[11146](#), [11151](#), [11153](#), [11154](#), [11185](#)
- \l__prop_internal_tl
- [594](#), [11091](#), [11094](#),
[11298](#), [11304](#), [11305](#), [11321](#), [11328](#)
- __prop_item:Nn:nwn [593](#)
- __prop_item:Nn:nwn [11249](#)
- __prop_map_function:Nwn [11378](#)
- __prop_map_tokens:nwn [11409](#)
- __prop_pair:wn
- [587](#), [587](#), [587](#), [591](#), [596](#),
[597](#), [597](#), [598](#), [11091](#), [11092](#), [11197](#),
[11200](#), [11252](#), [11255](#), [11300](#), [11323](#),
[11346](#), [11350](#), [11384](#), [11387](#), [11397](#),
[11399](#), [11404](#), [11413](#), [11416](#), [31446](#)
- __prop_put:NNnn [11294](#)
- __prop_put_if_new:NNnn [11315](#)
- __prop_rand_item:w [31436](#)
- __prop_show:NN [11427](#), [11429](#), [11431](#)
- __prop_split:NnTF
- [587](#), [594](#), [595](#), [596](#), [11192](#),

11205, 11211, 11221, 11229, 11238,
 11274, 11284, 11303, 11326, 11369
 _prop_split_aux:NnTF 11192
 _prop_split_aux:w 591, 11192
 _prop_use_i_delimit_by_s_-
 stop:nw 31435, 31449
 \protect 1131, 12811,
 16852, 29426, 29447, 29449, 30798
 \protected 117, 119, 143, 556, 10832, 10834
 \protrudechars 927
 \ProvidesExplClass 7
 \ProvidesExplFile 7, 32090, 32107
 \ProvidesExplFileAux 32093, 32095
 \ProvidesExplPackage 7
 \ProvidesFile 32098, 32099
 pt 219
 \ptexminorversion 1145
 \ptexrevision 1146
 \ptexversion 1147
 \pdximen 928

Q

quark commands:

\q_mark 39, 953, 3223
 \q_nil 21, 21, 39, 39, 39,
 58, 315, 367, 370, 371, 1478, 1481,
 3223, 3277, 3296, 3302, 3317, 3318,
 3324, 3348, 3352, 15507, 15508,
 15510, 15512, 15514, 15516, 15522
 \q_no_value 38, 39, 39, 39, 78, 78, 78,
 78, 78, 78, 85, 85, 85, 118, 127, 146,
 146, 147, 161, 161, 167, 167, 168,
 169, 169, 169, 367, 369, 488, 489,
 549, 592, 592, 3223, 3285, 3306,
 3312, 7663, 7671, 7683, 7709, 9340,
 9723, 9738, 11223, 11234, 11243,
 12519, 12532, 13201, 13336, 13366,
 13509, 13511, 13513, 13515, 13557
 \quark_if_nil:n 370
 \quark_if_nil:NnTF 39, 3275
 \quark_if_nil:NnTF .. 39, 368, 371, 3293
 \quark_if_nil_p:N 39, 3275
 \quark_if_nil_p:n 39, 3293
 \quark_if_no_value:NnTF .. 39, 3275,
 13368, 28460, 28464, 28533, 28537
 \quark_if_no_value:NnTF 39, 3293
 \quark_if_no_value_p:N 39, 3275
 \quark_if_no_value_p:n 39, 3293
 \quark_if_recursion_tail_break:N
 32035
 \quark_if_recursion_tail_break:n
 32037
 \quark_if_recursion_tail_-
 break:NN 40, 371, 3263

\quark_if_recursion_tail_-
 break:nN 40, 371, 3263
 \quark_if_recursion_tail_stop:N .
 40,
 307, 371, 1129, 3231, 29009, 30574,
 30605, 30893, 30905, 30967, 31018
 \quark_if_recursion_tail_stop:n .
 40, 307, 369, 371, 3245, 30325
 \quark_if_recursion_tail_stop_-
 do:Nn 40, 307, 371, 3231
 \quark_if_recursion_tail_stop_-
 do:nn 40, 307, 371, 3245
 \quark_new:N 38, 307,
 308, 373, 3218, 3223, 3224, 3225,
 3226, 3227, 3228, 3230, 3638, 3639,
 3640, 3641, 3642, 4167, 4168, 4556,
 4557, 5203, 8056, 8057, 8969, 8970,
 9571, 9572, 10165, 11098, 11099,
 12644, 13092, 13094, 13095, 14625,
 23552, 23557, 29026, 29028, 29029
 \q_recursion_stop 21,
 21, 40, 40, 40, 40, 40, 41, 315, 368,
 1480, 1484, 3227, 29017, 30547,
 30594, 30627, 30947, 31015, 31233
 \q_recursion_tail
 .. 39, 40, 40, 40, 40, 40, 41, 368,
 368, 3227, 3233, 3239, 3248, 3255,
 3260, 3265, 3272, 29017, 30546,
 30593, 30626, 30946, 31014, 31232
 \q_stop 21,
 21, 33, 38, 38, 39, 39, 54, 315,
 367, 1479, 1482, 3223, 4263, 28895,
 28899, 28910, 28929, 28934, 28945,
 28949, 28961, 28962, 28968, 28970,
 28971, 28973, 28976, 28992, 29000
 \s_stop 42, 42, 376, 3506, 3515
 quark internal commands:
 \s__bool_mark ... 31397, 31418, 31432
 \q__bool_recursion_stop
 8969, 8972, 9065, 9091
 \q__bool_recursion_tail
 8969, 9065, 9091
 \s__bool_stop ... 31397, 31418, 31432
 \q__char_no_value 10165, 10552
 \s__char_stop
 .. 10164, 10506, 10511, 10552, 10554
 \q__clist_mark 558
 \s__clist_mark
 545, 550, 552, 553, 554,
 9567, 9600, 9601, 9618, 9740, 9750,
 9754, 9776, 9832, 9838, 9852, 9864,
 9865, 9866, 9869, 9870, 9871, 9880,
 9881, 9890, 10044, 10045, 10057, 10058

- \q__clist_recursion_stop 16195, 16196, 16198, 16199, 16200, 16202, 16218, 16390, 16395, 16622, 16676, 16685, 16687, 17364, 17522, 18008, 18023, 18047, 18067, 18068, 18200, 18225, 18226, 18240, 18241, 18278, 18279, 18392, 18393, 18394, 18403, 18419, 18423, 18487, 18488, 18491, 18502, 18503, 18511, 18512, 18514, 18515, 18516, 18518, 18519, 18520, 18532, 18535, 18539, 18542, 18562, 18612, 18615, 18618, 18638, 18639, 18641, 18642, 18643, 18651, 18654, 18665, 18666, 18668, 18677, 18753, 18905, 18939, 18940, 18943, 19024, 19162, 19170, 19172, 19349, 19358, 19360, 19365, 19373, 19375, 19377, 19380, 19883, 19895, 19897, 20106, 20123, 20125, 20306, 20325, 20327, 20328, 20331, 20348, 20351, 20354, 20379, 20380, 20382, 20398, 20487, 20500, 20502, 20505, 20510, 20543, 20559, 20642, 20655, 20657, 20670, 20672, 20685, 20687, 20700, 20702, 20715, 20717, 20730, 20740, 21241, 21257, 21258, 21262, 21273, 21380, 21393, 21395, 21411, 21414, 21424, 21447, 21458, 21460, 21474, 21476, 21481, 21543, 21564, 21567, 21597, 21618, 21621, 21671, 21687, 21690, 21765, 21766, 21876, 21878, 21910, 22176, 22184, 22187, 22266
- \q__clist_recursion_tail 555, 9571, 9586, 9985, 10021
- \q__clist_stop 558
- \s__clist_stop 553, 9567, 9569, 9570, 9725, 9728, 9740, 9743, 9751, 9754, 9762, 9776, 9838, 9866, 9869, 9870, 9882, 9890, 10046, 10057, 10058, 10059, 10085, 10119
- \s__color_stop 28644
- \s__cs_mark 324, 357, 358, 1728, 1729, 1732, 1733, 1734, 2791, 2821, 2822, 2824, 2830, 2834, 2856, 2865, 2884, 2912, 2915, 2923, 2938, 2970, 2984, 2988, 2997, 3016, 3025, 3030, 3119, 3122, 3138
- \q__cs_nil 3125
- \q__cs_recursion_stop 2793, 2797, 2808, 3118
- \s__cs_stop 324, 358, 1729, 1732, 1733, 1734, 2791, 2794, 2795, 2825, 2834, 2860, 2912, 2915, 2919, 2927, 2933, 2942, 2948, 2950, 2970, 2992, 2997, 3027, 3030, 3119
- \s__deprecation_mark 31769, 31772, 31774
- \s__deprecation_stop 31769, 31772, 31774, 31794, 31803
- \s__dim_mark 13893, 14054, 14061
- \s__dim_stop 13893, 13895, 14001, 14025, 14054, 14061
- \q__file_nil 13092, 13176, 13190, 13286, 13292
- \q__file_recursion_stop 13094, 13105, 13109, 13118, 13158
- \q__file_recursion_tail 13094, 13105, 13158
- \s__file_stop 659, 13064, 13069, 13091, 13176, 13177, 13182, 13188, 13190, 13191, 13286, 13287, 13292, 13294, 13296, 13730, 13732, 13735, 13736, 13738, 13750, 13826, 13829, 13836, 13838
- \s__fp 720, 721, 723, 727, 728, 751, 757, 758, 761, 775, 776, 778, 807, 810, 811, 812, 814, 820, 822, 911, 15897, 15910, 15911, 15912, 15913, 15914, 15924, 15929, 15931, 15932, 15947, 15960, 15963, 15965, 15975, 15987, 16007, 16024, 16027, 16034, 16041, 16057, 16084, 16190, 16192, 16194, 16195, 16196, 16198, 16199, 16200, 16202, 16218, 16390, 16395, 16622, 16676, 16685, 16687, 17364, 17522, 18008, 18023, 18047, 18067, 18068, 18200, 18225, 18226, 18240, 18241, 18278, 18279, 18392, 18393, 18394, 18403, 18419, 18423, 18487, 18488, 18491, 18502, 18503, 18511, 18512, 18514, 18515, 18516, 18518, 18519, 18520, 18532, 18535, 18539, 18542, 18562, 18612, 18615, 18618, 18638, 18639, 18641, 18642, 18643, 18651, 18654, 18665, 18666, 18668, 18677, 18753, 18905, 18939, 18940, 18943, 19024, 19162, 19170, 19172, 19349, 19358, 19360, 19365, 19373, 19375, 19377, 19380, 19883, 19895, 19897, 20106, 20123, 20125, 20306, 20325, 20327, 20328, 20331, 20348, 20351, 20354, 20379, 20380, 20382, 20398, 20487, 20500, 20502, 20505, 20510, 20543, 20559, 20642, 20655, 20657, 20670, 20672, 20685, 20687, 20700, 20702, 20715, 20717, 20730, 20740, 21241, 21257, 21258, 21262, 21273, 21380, 21393, 21395, 21411, 21414, 21424, 21447, 21458, 21460, 21474, 21476, 21481, 21543, 21564, 21567, 21597, 21618, 21621, 21671, 21687, 21690, 21765, 21766, 21876, 21878, 21910, 22176, 22184, 22187, 22266
- \s__fp_<type> 751
- \s__fp_division 15905
- \s__fp_exact 15905, 15910, 15911, 15912, 15913, 15914, 18487
- \s__fp_expr_mark ... 757, 761, 782, 786, 15900, 17571, 17584, 17666, 17710
- \s__fp_expr_stop 729, 15900, 16098, 17473, 17572, 17576, 17585, 18569, 18580, 18590, 18598
- \s__fp_invalid 15905
- \s__fp_mark 15902, 16047, 16048, 16052
- \s__fp_overflow 15905, 15931
- \s__fp_stop 727, 15902, 15904, 15948, 16024, 16035, 16042, 16048, 16052, 16066, 16085, 16891, 16895, 17403, 17408, 18008, 18030, 18199, 18200, 18225, 18226, 18392, 18393, 18394, 18561, 18562, 20182, 20197, 21517, 21521
- \s__fp_tuple 726, 16008, 16014, 16015, 16092, 16094, 17788, 18000, 18015, 18040, 18042, 18059, 18060, 18062, 18270, 18271,

- 19411, 19412, 19418, 19419, 21493
- \s__fp_underflow [15905](#), [15929](#)
- \s__int_mark [8053](#), [8266](#), [8269](#), [8335](#), [8342](#)
- \q__int_recursion_stop [8056](#), [8753](#), [8770](#), [8813](#), [8840](#)
- \q__int_recursion_tail [8056](#), [8753](#), [8770](#), [8813](#)
- \s__int_stop [505](#), [516](#), [8053](#), [8055](#), [8245](#), [8261](#), [8263](#), [8267](#), [8280](#), [8335](#), [8342](#), [8746](#), [8752](#), [8769](#)
- \s__iow_mark . [12641](#), [12954](#), [12961](#), [12973](#), [13047](#), [13048](#), [13049](#), [13050](#)
- \q__iow_nil [12644](#), [12840](#), [12847](#)
- \s__iow_stop [12641](#), [12643](#), [12840](#), [12881](#), [12939](#), [12977](#), [12990](#), [13047](#), [13050](#)
- \s__kernel_stop [2132](#), [2140](#), [2149](#), [2158](#)
- \s__keys_mark [14622](#), [14668](#), [14671](#), [14679](#), [14686](#), [15284](#), [15287](#), [15292](#), [15298](#), [15532](#), [15535](#), [15544](#), [15546](#), [15551](#), [15554](#)
- \s__keys_nil [14622](#), [14663](#), [14664](#), [14666](#), [14668](#), [14671](#), [14676](#), [14685](#), [14686](#), [14694](#), [15279](#), [15280](#), [15282](#), [15284](#), [15287](#), [15290](#), [15298](#), [15299](#), [15531](#), [15534](#), [15540](#), [15542](#), [15550](#), [15553](#)
- \q__keys_no_value [709](#), [14612](#), [14625](#), [15114](#), [15138](#), [15155](#), [15180](#), [15197](#), [15226](#)
- \s__keys_stop [14622](#), [14704](#), [14714](#), [14866](#), [15267](#), [15277](#), [15464](#), [15484](#)
- \s__keyval_mark [681](#), [681](#), [681](#), [682](#), [685](#), [14403](#), [14411](#), [14417](#), [14419](#), [14420](#), [14421](#), [14422](#), [14428](#), [14429](#), [14432](#), [14437](#), [14438](#), [14442](#), [14443](#), [14448](#), [14449](#), [14454](#), [14455](#), [14458](#), [14459](#), [14463](#), [14464](#), [14467](#), [14470](#), [14471](#), [14479](#), [14480](#), [14485](#), [14486](#), [14489](#), [14492](#), [14496](#), [14497](#), [14503](#), [14512](#), [14513](#), [14516](#), [14521](#), [14530](#), [14531](#), [14536](#), [14537](#), [14540](#), [14541](#), [14542](#), [14543](#), [14544](#), [14561](#), [14562](#), [14568](#), [14572](#), [14574](#), [14576](#), [14587](#)
- \s__keyval_nil . [681](#), [14403](#), [14427](#), [14436](#), [14442](#), [14445](#), [14447](#), [14454](#), [14457](#), [14463](#), [14467](#), [14469](#), [14476](#), [14478](#), [14485](#), [14489](#), [14491](#), [14496](#), [14497](#), [14503](#), [14511](#), [14530](#), [14536](#), [14560](#), [14564](#), [14581](#), [14584](#), [14587](#)
- \s__keyval_stop [14403](#), [14419](#), [14420](#), [14421](#), [14422](#), [14430](#), [14434](#), [14439](#), [14443](#), [14450](#), [14455](#), [14460](#), [14464](#), [14467](#), [14472](#), [14474](#), [14481](#), [14486](#), [14489](#), [14491](#), [14492](#), [14496](#), [14503](#), [14514](#), [14530](#), [14531](#), [14536](#), [14537](#), [14540](#), [14543](#), [14544](#), [14566](#), [14587](#)
- \s__keyval_tail [681](#), [14403](#), [14411](#), [14415](#), [14416](#), [14426](#), [14510](#), [14519](#), [14520](#), [14542](#)
- \s__msg_mark [11447](#), [11801](#), [11866](#), [11867](#), [11872](#), [11875](#)
- \s__msg_stop ... [621](#), [11447](#), [11449](#), [11803](#), [11807](#), [11809](#), [11868](#), [12404](#)
- \s__peek_mark [10911](#), [11059](#), [11060](#), [11067](#)
- \s__peek_stop [10911](#), [10913](#), [11048](#), [11061](#), [11070](#)
- \s__prg_mark [1580](#), [1582](#), [1590](#)
- \q__prg_recursion_stop [321](#), [1521](#), [1586](#), [1668](#)
- \q__prg_recursion_tail [321](#), [1586](#), [1596](#), [1668](#), [1687](#)
- \s__prg_stop [1607](#), [1612](#), [1631](#), [1639](#), [1647](#), [1698](#), [1702](#), [1704](#), [1706](#), [1708](#)
- \s__prop .. [587](#), [587](#), [591](#), [597](#), [598](#), [1175](#), [11091](#), [11091](#), [11092](#), [11095](#), [11197](#), [11200](#), [11252](#), [11255](#), [11301](#), [11324](#), [11346](#), [11350](#), [11384](#), [11387](#), [11399](#), [11413](#), [11416](#), [31446](#), [31451](#)
- \s__prop_mark [590](#), [591](#), [11096](#), [11172](#), [11173](#), [11179](#), [11180](#), [11197](#), [11199](#), [11200](#)
- \q__prop_recursion_stop [11098](#), [11160](#)
- \q__prop_recursion_tail [596](#), [11098](#), [11160](#), [11347](#), [11358](#)
- \s__prop_stop [590](#), [591](#), [11096](#), [11166](#), [11173](#), [11176](#), [11180](#), [11197](#), [11200](#), [31435](#), [31442](#)
- \s__quark [376](#), [3229](#), [3464](#), [3466](#), [3467](#), [3478](#), [3481](#), [3486](#), [3489](#), [3491](#), [3512](#)
- __quark_if_empty_if:n ... [3293](#), [3454](#)
- __quark_if_nil:w [370](#), [3293](#)
- __quark_if_no_value:w [3293](#)
- __quark_if_recursion_tail:w ... [368](#), [373](#), [3245](#), [3272](#)
- __quark_module_name:N [374](#), [3321](#), [3344](#), [3459](#)
- __quark_module_name:w [3459](#)
- __quark_module_name_end:w ... [3459](#)
- __quark_module_name_loop:w .. [3459](#)
- __quark_new_conditional:Nnnn . [3320](#)
- __quark_new_conditional_aux_-do:NNnnn [374](#), [3441](#), [3443](#), [3444](#)
- __quark_new_conditional_-define:NNNNn [374](#), [3444](#)
- __quark_new_conditional_N:Nnnn [3440](#)

- _quark_new_conditional_n:Nnnn [3440](#)
- _quark_new_test:NNNn [3320](#)
- _quark_new_test_aux:Nn
..... [3321](#), [3322](#), [3332](#)
- _quark_new_test_aux:nnNNnnnn [3320](#)
- _quark_new_test_aux_do:nNNnnnnNNn
..... [372](#), [373](#), [3375](#),
[3380](#), [3385](#), [3390](#), [3395](#), [3401](#), [3404](#)
- _quark_new_test_define_break_
ifx:nNNNNn [3402](#), [3417](#)
- _quark_new_test_define_break_
tl:nNNNNn [3386](#), [3417](#)
- _quark_new_test_define_
ifx:nNnNNn [372](#), [373](#), [3391](#), [3396](#), [3417](#)
- _quark_new_test_define_
tl:nNnNNn [372](#), [373](#), [3376](#), [3381](#), [3417](#)
- _quark_new_test_N:Nnnn [3373](#)
- _quark_new_test_n:Nnnn [3373](#)
- _quark_new_test_NN:Nnnn [3373](#)
- _quark_new_test_Nn:Nnnn [3373](#)
- _quark_new_test_nN:Nnnn [3383](#)
- _quark_new_test_nn:Nnnn [3373](#)
- \q_quark_nil [3230](#)
- _quark_quark_conditional_
name:N [375](#), [3343](#), [3481](#)
- _quark_quark_conditional_
name:w [375](#), [3481](#)
- _quark_test_define_aux:NNNNnnNNn
..... [373](#), [3404](#)
- _quark_tmp:w
..... [375](#), [3459](#), [3480](#), [3481](#), [3491](#)
- \q_regex_nil
..... [23557](#), [24802](#), [24820](#), [24821](#)
- \q_regex_recursion_stop
.. [23552](#), [23554](#), [23556](#), [24802](#), [24821](#)
- \s__seq [479](#), [482](#), [483](#), [489](#), [493](#), [495](#),
[1175](#), [7374](#), [7385](#), [7415](#), [7420](#), [7425](#),
[7430](#), [7441](#), [7469](#), [7495](#), [7503](#), [7507](#),
[7730](#), [7778](#), [7982](#), [31456](#), [31462](#), [31493](#)
- \s__seq_mark
..... [7375](#), [7970](#), [7971](#), [7985](#), [7988](#)
- \s__seq_stop [7375](#), [7683](#),
[7686](#), [7694](#), [7696](#), [7777](#), [7778](#), [7972](#),
[7985](#), [7988](#), [7990](#), [31455](#), [31456](#),
[31458](#), [31462](#), [31466](#), [31468](#), [31473](#)
- \s__skip_stop ... [14239](#), [14300](#), [14302](#)
- \s__sort_mark [944](#),
[948](#), [948](#), [950](#), [22651](#), [22846](#), [22850](#),
[22856](#), [22860](#), [22866](#), [22869](#), [22934](#),
[22935](#), [22937](#), [22974](#), [22976](#), [22979](#),
[22983](#), [22986](#), [22989](#), [22991](#), [22994](#)
- \s__sort_stop [947](#),
[948](#), [948](#), [950](#), [22651](#), [22922](#), [22931](#),
[22935](#), [22937](#), [22974](#), [22975](#), [22976](#),
[22981](#), [22983](#), [22987](#), [22989](#), [22997](#)
- \s__str [432](#),
[457](#), [460](#), [5202](#), [5340](#), [5344](#), [5544](#),
[5591](#), [5646](#), [5652](#), [6101](#), [6113](#), [6118](#),
[6128](#), [6133](#), [6138](#), [6141](#), [6156](#), [6169](#),
[6172](#), [6307](#), [6308](#), [6325](#), [6331](#), [6347](#),
[6353](#), [6354](#), [6459](#), [6474](#), [6483](#), [6484](#)
- \s__str_mark
..... [411](#), [417](#), [424](#), [4552](#), [4750](#),
[4781](#), [4788](#), [4861](#), [4878](#), [5126](#), [5128](#)
- \q__str_nil
... [457](#), [5203](#), [6292](#), [6299](#), [6314](#), [6341](#)
- \q__str_recursion_stop
..... [4556](#), [5142](#), [5150](#), [5155](#)
- \q__str_recursion_tail
[415](#), [4556](#), [4795](#), [4804](#), [4821](#), [4841](#), [5142](#)
- \s__str_stop [417](#),
[421](#), [456](#), [460](#), [4552](#), [4554](#), [4555](#),
[4647](#), [4750](#), [4781](#), [4788](#), [4861](#), [4870](#),
[4876](#), [4878](#), [4884](#), [4901](#), [4920](#), [4982](#),
[5039](#), [5051](#), [5089](#), [5105](#), [5112](#), [5120](#),
[5122](#), [5126](#), [5128](#), [5340](#), [5346](#), [5388](#),
[5393](#), [5401](#), [5614](#), [5618](#), [5627](#), [5636](#),
[5660](#), [5669](#), [6018](#), [6020](#), [6028](#), [6114](#),
[6150](#), [6264](#), [6266](#), [6270](#), [6282](#), [6422](#),
[6424](#), [6428](#), [6440](#), [6449](#), [6456](#), [6477](#)
- \q__text_nil [29026](#), [29440](#), [29441](#)
- \q__text_recursion_stop
.... [29028](#), [29031](#), [29211](#), [29225](#),
[29234](#), [29313](#), [29329](#), [29338](#), [29386](#),
[29408](#), [29543](#), [29557](#), [29566](#), [29568](#),
[29634](#), [29650](#), [29659](#), [29703](#), [29777](#),
[29786](#), [29948](#), [29953](#), [30080](#), [30085](#),
[30133](#), [30138](#), [30161](#), [30166](#), [30202](#),
[30211](#), [30648](#), [30661](#), [30670](#), [30688](#),
[30701](#), [30703](#), [30720](#), [30729](#), [30757](#)
- \q__text_recursion_tail
..... [29028](#), [29158](#), [29211](#),
[29312](#), [29386](#), [29408](#), [29543](#), [29568](#),
[29633](#), [29703](#), [30648](#), [30687](#), [30757](#)
- \s__text_stop
.. [29025](#), [29100](#), [29102](#), [29440](#), [29441](#)
- \s__tl [952](#), [953](#), [953](#), [954](#),
[955](#), [962](#), [962](#), [963](#), [23057](#), [23058](#),
[23277](#), [23308](#), [23314](#), [23339](#), [23357](#),
[23362](#), [23376](#), [23388](#), [23411](#), [23414](#)
- \q__tl_act_mark
..... [398](#), [398](#), [398](#), [4167](#), [4172](#), [4189](#)
- \q__tl_act_stop
.... [398](#), [4167](#), [4172](#), [4176](#), [4185](#),
[4187](#), [4193](#), [4198](#), [4201](#), [4205](#), [4208](#)
- \q__tl_mark
... [385](#), [3638](#), [3743](#), [3745](#), [3747](#), [3749](#)

\s__tl_mark 392, 393,
 396, 397, 3985, 3995, 4104, 4105,
 4108, 4111, 4112, 4118, 4121, 4136,
 4137, 4143, 4147, 4149, 4152, 4542
 \q__tl_nil 385, 387, 388, 3638, 3769,
 3845, 3846, 3856, 3857, 4253, 4254
 \s__tl_nil
 397, 4135, 4139, 4157, 4160, 4163, 4542
 \q__tl_recursion_stop 3641
 \q__tl_recursion_tail
 .. 3641, 4000, 4018, 4028, 4043, 4393
 \q__tl_stop 385, 3638, 3768
 \s__tl_stop
 383, 396, 3726, 3728, 3943,
 3949, 3985, 3995, 4106, 4108, 4113,
 4115, 4141, 4163, 4244, 4246, 4264,
 4282, 4299, 4323, 4529, 4539, 4542
 \s__token_stop 577,
 579, 10644, 10748, 10751, 10781,
 10815, 10852, 10856, 10862, 10885
 \quitvmode 694

R

\r 29167, 31006, 31030, 31056, 31180, 31181
 \radical 419
 \raise 420
 rand 218
 randint 218
 \randomseed 929
 \read 421
 \readline 557
 \readpapersizespecial 1148
 \ref 29184, 29191
 regex commands:
 \c_foo_regex 227
 \regex_(g)set:Nn 234
 \regex_const:Nn 227, 234, 26072
 \regex_count:NnN 235, 26115
 \regex_count:nnN 235, 1047, 26115
 \regex_extract_all:NnN ... 235, 26119
 \regex_extract_all:nnN
 228, 235, 969, 26119
 \regex_extract_all:NnNTF . 235, 26119
 \regex_extract_all:nnNTF . 235, 26119
 \regex_extract_once:NnN .. 235, 26119
 \regex_extract_once:nnN
 235, 235, 26119
 \regex_extract_once:NnNTF 235, 26119
 \regex_extract_once:nnNTF
 232, 235, 26119
 \regex_gset:Nn 234, 26072
 \regex_match:NnTF 234, 26105
 \regex_match:nnTF 234, 26105

\regex_new:N 234,
 971, 26066, 26068, 26069, 26070, 26071
 \regex_replace_all:NnN ... 236, 26119
 \regex_replace_all:nnN 228, 236, 26119
 \regex_replace_all:NnNTF . 236, 26119
 \regex_replace_all:nnNTF . 236, 26119
 \regex_replace_once:NnN .. 236, 26119
 \regex_replace_once:nnN
 235, 236, 26119
 \regex_replace_once:NnNTF 236, 26119
 \regex_replace_once:nnNTF 236, 26119
 \regex_set:Nn 234, 26072
 \regex_show:N 234, 26087
 \regex_show:n 227, 234, 26087
 \regex_split:NnN 236, 26119
 \regex_split:nnN 236, 26119
 \regex_split:NnNTF 236, 26119
 \regex_split:nnNTF 236, 26119
 \g_tmpa_regex 236, 26068
 \l_tmpa_regex 236, 26068
 \g_tmpb_regex 236, 26068
 \l_tmpb_regex 236, 26068

regex internal commands:

__regex_action_cost:n
 1014, 1017, 1025,
 25189, 25190, 25198, 25646, 25672
 __regex_action_free:n
 1014, 1025, 25212, 25218,
 25219, 25230, 25289, 25293, 25318,
 25343, 25347, 25350, 25378, 25386,
 25396, 25410, 25441, 25644, 25648
 __regex_action_free_aux:nn .. 25648
 __regex_action_free_group:n ...
 1014, 1025, 25239, 25358, 25361, 25648
 __regex_action_start_wildcard: .
 1014, 25123, 25641
 __regex_action_submatch:n 1014,
 25312, 25313, 25439, 25692, 25694
 __regex_action_success:
 1014, 25126, 25140, 25699
 __regex_action_wildcard: 1029
 \c__regex_all_catcodes_int
 23962, 24084, 24174, 24774
 __regex_anchor:N
 983, 1023, 24403, 24969, 25401
 \c__regex_ascii_lower_int
 23551, 23616, 23622
 \c__regex_ascii_max_control_int .
 23548, 23732
 \c__regex_ascii_max_int
 23548, 23725, 23733, 23914
 \c__regex_ascii_min_int
 23548, 23724, 23731

__regex_assertion:Nn . [983](#), [1023](#),
 [24403](#), [24428](#), [24437](#), [24963](#), [25401](#)
 __regex_b_test:
 [983](#), [1023](#), [24428](#), [24437](#), [24968](#), [25401](#)
 \l__regex_balance_int [971](#),
 [1036](#), [23546](#), [25475](#), [25488](#), [25603](#),
 [25607](#), [25608](#), [25787](#), [25816](#), [26009](#),
 [26026](#), [26321](#), [26347](#), [26370](#), [26374](#),
 [26375](#), [26382](#), [26383](#), [26387](#), [26388](#)
 \g__regex_balance_intarray
 [969](#), [971](#), [23543](#), [25602](#), [25760](#), [25775](#)
 \l__regex_balance_tl
 [1036](#), [1039](#), [25715](#), [25788](#), [25817](#), [25879](#)
 __regex_branch:n
 . . [983](#), [1000](#), [1019](#), [23540](#), [24089](#),
 [24584](#), [24637](#), [24816](#), [24945](#), [25284](#)
 __regex_break_point:TF
 [972](#), [995](#), [1017](#), [23559](#),
 [23565](#), [25189](#), [25190](#), [25407](#), [25430](#)
 __regex_break_true:w
 . . [972](#), [973](#), [23559](#), [23565](#), [23570](#),
 [23577](#), [23584](#), [23590](#), [23597](#), [23605](#),
 [23652](#), [23664](#), [23681](#), [24378](#), [25422](#)
 __regex_build:N [1046](#),
 [25110](#), [26112](#), [26118](#), [26122](#), [26126](#)
 __regex_build:n [1046](#),
 [25110](#), [26107](#), [26116](#), [26121](#), [26124](#)
 __regex_build_for_cs:n [23676](#), [25128](#)
 __regex_build_new_state:
 [25120](#), [25121](#),
 [25132](#), [25133](#), [25162](#), [25171](#), [25203](#),
 [25237](#), [25242](#), [25286](#), [25301](#), [25306](#),
 [25345](#), [25364](#), [25399](#), [25403](#), [25436](#)
 \l__regex_build_tl [1000](#),
 [23537](#), [24081](#), [24088](#), [24106](#), [24111](#),
 [24114](#), [24115](#), [24118](#), [24119](#), [24122](#),
 [24168](#), [24171](#), [24204](#), [24218](#), [24222](#),
 [24347](#), [24361](#), [24402](#), [24427](#), [24436](#),
 [24446](#), [24478](#), [24491](#), [24495](#), [24577](#),
 [24580](#), [24583](#), [24589](#), [24590](#), [24593](#),
 [24636](#), [24904](#), [24920](#), [24938](#), [24944](#),
 [24999](#), [25002](#), [25007](#), [25037](#), [25052](#),
 [25056](#), [25059](#), [25065](#), [25786](#), [25805](#),
 [25820](#), [25823](#), [25844](#), [25876](#), [25938](#),
 [25941](#), [25956](#), [26000](#), [26016](#), [26052](#)
 __regex_build_transition_-
 left:NNN [25158](#), [25347](#), [25361](#), [25378](#)
 __regex_build_transition_-
 right:nNn [25158](#),
 [25204](#), [25239](#), [25289](#), [25293](#), [25318](#),
 [25343](#), [25350](#), [25358](#), [25386](#), [25396](#)
 __regex_build_transitions_-
 lazyness:NNNNN
 [25169](#), [25211](#), [25217](#), [25229](#)
 \l__regex_capturing_group_int
 [969](#),
 [1013](#), [1053](#), [25109](#), [25118](#), [25255](#),
 [25257](#), [25268](#), [25269](#), [25277](#), [25278](#),
 [25281](#), [25875](#), [26280](#), [26352](#), [26360](#)
 \l__regex_case_changed_char_int
 [974](#),
 [23586](#), [23589](#), [23600](#), [23603](#), [23604](#),
 [23611](#), [23615](#), [23621](#), [25454](#), [25572](#)
 \c__regex_catcode_A_int [23962](#)
 \c__regex_catcode_B_int [23962](#)
 \c__regex_catcode_C_int [23962](#)
 \c__regex_catcode_D_int [23962](#)
 \c__regex_catcode_E_int [23962](#)
 \c__regex_catcode_in_class_mode_-
 int [23952](#),
 [24073](#), [24445](#), [24606](#), [24699](#), [24728](#)
 \g__regex_catcode_intarray
 [969](#), [971](#), [23543](#), [25600](#), [25617](#)
 \c__regex_catcode_L_int [23962](#)
 \c__regex_catcode_M_int [23962](#)
 \c__regex_catcode_mode_int [23952](#),
 [24069](#), [24142](#), [24477](#), [24697](#), [24726](#)
 \c__regex_catcode_O_int [23962](#)
 \c__regex_catcode_P_int [23962](#)
 \c__regex_catcode_S_int [23962](#)
 \c__regex_catcode_T_int [23962](#)
 \c__regex_catcode_U_int [23962](#)
 \l__regex_catcodes_bool
 [23959](#), [24733](#), [24737](#), [24772](#)
 \l__regex_catcodes_int [984](#),
 [23959](#), [24085](#), [24173](#), [24175](#), [24181](#),
 [24464](#), [24481](#), [24581](#), [24594](#), [24693](#),
 [24730](#), [24765](#), [24767](#), [24773](#), [24774](#)
 __regex_char_if_alphanumeric:N
 [23932](#)
 __regex_char_if_alphanumeric:NTF
 [23910](#), [24135](#), [25983](#)
 __regex_char_if_special:N [23910](#)
 __regex_char_if_special:NTF
 [23910](#), [24131](#)
 \g__regex_charcode_intarray
 [969](#), [971](#), [23543](#), [25598](#), [25614](#)
 __regex_chk_c_allowed:TF
 [24054](#), [24686](#)
 __regex_class:NnnnN
 [983](#), [990](#), [992](#), [998](#),
 [23541](#), [24169](#), [24472](#), [24473](#), [24479](#),
 [24833](#), [24912](#), [24922](#), [24960](#), [25183](#)
 \c__regex_class_mode_int
 [23952](#), [24059](#), [24074](#)
 __regex_class_repeat:n
 [1017](#), [25193](#), [25199](#), [25215](#), [25224](#)
 __regex_class_repeat:nN [25194](#), [25208](#)

_regex_class_repeat:nnN	_regex_compile_class_posix:NNNNw
..... 25195, 25222 24505
_regex_command_K:	_regex_compile_class_posix-
..... 983, 24938, 24961, 25434	end:w 24505
_regex_compile:n 24124,	_regex_compile_class_posix-
25112, 26074, 26079, 26084, 26089	loop:w 24505
_regex_compile:w	_regex_compile_class_posix-
..... 989, 24078, 24126, 24779	test:w 24459, 24505
_regex_compile_\$: 24398	_regex_compile_cs_aux:Nn ... 24788
_regex_compile(: 24601	_regex_compile_cs_aux:NNnnN 24788
_regex_compile): 24640	_regex_compile_end:
_regex_compile.: 24369 989, 24078, 24151, 24797
_regex_compile_/A: 24398	_regex_compile_end_cs: 24147, 24788
_regex_compile_/B: 24422	_regex_compile_escaped:N
_regex_compile_/b: 24422 24136, 24153
_regex_compile_/c: 24685	_regex_compile_group_begin:N ..
_regex_compile_/D: 24381	.. 24575, 24623, 24628, 24646, 24648
_regex_compile_/d: 24381	_regex_compile_group_end: ...
_regex_compile_/G: 24398 24575, 24643
_regex_compile_/H: 24381	_regex_compile_lparen:w
_regex_compile_/h: 24381 24610, 24614
_regex_compile_/K: 24935	_regex_compile_one:n
_regex_compile_/N: 24381 24163, 24313, 24319, 24373,
_regex_compile_/S: 24381	24384, 24387, 24397, 24552, 24804
_regex_compile_/s: 24381	_regex_compile_quantifier:w ...
_regex_compile_/u: 24853 24182, 24193, 24451, 24595
_regex_compile_/V: 24381	_regex_compile_quantifier_*:w .
_regex_compile_/v: 24381 24227
_regex_compile_/W: 24381	_regex_compile_quantifier_+:w .
_regex_compile_/w: 24381 24227
_regex_compile_/Z: 24398	_regex_compile_quantifier_?:w .
_regex_compile_/z: 24398 24227
_regex_compile[: 24456	_regex_compile_quantifier_-
_regex_compile]: 24440	abort:nNN
_regex_compile^: 24398	.. 24202, 24237, 24256, 24269, 24292
_regex_compile_abort_tokens:n .	_regex_compile_quantifier_-
..... 24184, 24211, 24561, 24571	braced_auxi:w 24233
_regex_compile_anchor:NTF .. 24398	_regex_compile_quantifier_-
_regex_compile_c[:w 24722	braced_auxii:w 24233
_regex_compile_c_C:NN 24701, 24710	_regex_compile_quantifier_-
_regex_compile_c_lbrack_add:N .	braced_auxiii:w 24233
..... 24722	_regex_compile_quantifier_-
_regex_compile_c_lbrack_end: 24722	lazyness:nnNN 992, 24214, 24228,
_regex_compile_c_lbrack_-	24230, 24232, 24245, 24265, 24287
loop:NN 24722	_regex_compile_quantifier_-
_regex_compile_c_test:NN ... 24685	none: 24198, 24200, 24202
_regex_compile_class:NN 24486	_regex_compile_range:Nw
_regex_compile_class:TFNN 24311, 24324
..... 998, 24471, 24482, 24486	_regex_compile_raw:N
_regex_compile_class_catcode:w 24004, 24132, 24136, 24138,
..... 24463, 24475	24156, 24161, 24189, 24304, 24306,
_regex_compile_class_normal:w .	24326, 24372, 24418, 24454, 24502,
..... 24466, 24469	24522, 24540, 24598, 24603, 24608,
	24624, 24634, 24642, 24660, 24661,

- 24662, 24668, 24679, 24680, 24681,
24689, 24744, 24793, 24865, 24871
- _regex_compile_raw_error:N ...
..... [24301](#),
24409, 24425, 24434, 24856, 24939
- _regex_compile_special:N
..... [985](#), [24132](#), [24153](#),
24195, 24216, 24243, 24248, 24263,
24276, 24310, 24329, 24489, 24507,
24526, 24546, 24547, 24616, 24651,
24669, 24712, 24731, 24858, 24874
- _regex_compile_special_group-
-:w [24649](#)
- _regex_compile_special_group-
:w [24645](#)
- _regex_compile_special_group-
i:w [24649](#)
- _regex_compile_special_group-
l:w [24645](#)
- _regex_compile_u_end:
..... [24877](#), [24883](#), [24888](#)
- _regex_compile_u_in_cs:
..... [24894](#), [24897](#)
- _regex_compile_u_in_cs_aux:n ..
..... [24907](#), [24910](#)
- _regex_compile_u_loop:NN ... [24853](#)
- _regex_compile_u_not_cs:
..... [24892](#), [24916](#)
- _regex_compile_l: [24632](#)
- _regex_compute_case_changed-
char: [23587](#), [23601](#), [23609](#)
- _regex_count:nnN [26116](#), [26118](#), [26163](#)
- \c_regex_cs_in_class_mode_int ..
..... [23952](#), [24785](#)
- \c_regex_cs_mode_int . [23952](#), [24783](#)
- \l_regex_cs_name_tl
..... [23547](#), [23673](#), [23679](#)
- \l_regex_curr_catcode_int [23631](#),
[23650](#), [23658](#), [23670](#), [25454](#), [25616](#)
- \l_regex_curr_char_int
..... [23569](#), [23575](#),
[23576](#), [23583](#), [23595](#), [23596](#), [23611](#),
[23612](#), [23613](#), [23614](#), [23620](#), [23651](#),
[24377](#), [25428](#), [25454](#), [25571](#), [25613](#)
- _regex_curr_cs_to_str:
..... [23523](#), [23661](#), [23673](#)
- \l_regex_curr_pos_int
.. [970](#), [23526](#), [25139](#), [25421](#), [25449](#),
[25476](#), [25478](#), [25481](#), [25489](#), [25496](#),
[25503](#), [25531](#), [25541](#), [25570](#), [25585](#),
[25599](#), [25601](#), [25603](#), [25604](#), [25605](#),
[25615](#), [25618](#), [25697](#), [25706](#), [26215](#)
- \l_regex_curr_state_int
..... [1025](#), [1032](#), [25458](#),
[25623](#), [25624](#), [25626](#), [25631](#), [25634](#),
[25656](#), [25661](#), [25666](#), [25667](#), [25675](#)
- \l_regex_curr_submatches_prop ..
..... [25459](#), [25538](#), [25636](#),
[25668](#), [25669](#), [25687](#), [25696](#), [25708](#)
- \l_regex_default_catcodes_int ..
..... [984](#), [23959](#), [24083](#),
[24085](#), [24181](#), [24481](#), [24581](#), [24594](#)
- _regex_disable_submatches: ...
.. [23675](#), [24780](#), [25689](#), [26157](#), [26166](#)
- \l_regex_empty_success_bool ...
.. [25467](#), [25523](#), [25527](#), [25704](#), [26225](#)
- _regex_escape_␣:w [23798](#)
- _regex_escape_/a:w [23798](#)
- _regex_escape_/break:w [23798](#)
- _regex_escape_/e:w [23798](#)
- _regex_escape_/f:w [23798](#)
- _regex_escape_/n:w [23798](#)
- _regex_escape_/r:w [23798](#)
- _regex_escape_/t:w [23798](#)
- _regex_escape_/x:w [23817](#)
- _regex_escape_\w [23782](#)
- _regex_escape_break:w [23798](#)
- _regex_escape_escaped:N
..... [23768](#), [23792](#), [23795](#)
- _regex_escape_loop:N
..... [978](#), [23775](#), [23782](#), [23817](#),
[23853](#), [23861](#), [23862](#), [23879](#), [23888](#)
- _regex_escape_raw:N
.. [979](#), [23769](#), [23795](#), [23806](#), [23808](#),
[23810](#), [23812](#), [23814](#), [23816](#), [23830](#)
- _regex_escape_unescaped:N
..... [23767](#), [23785](#), [23795](#)
- _regex_escape_use:nnnn
..... [978](#), [989](#), [23763](#), [24129](#), [25789](#)
- _regex_escape_x:N [980](#), [23852](#), [23856](#)
- _regex_escape_x_end:w .. [979](#), [23817](#)
- _regex_escape_x_large:n [23817](#)
- _regex_escape_x_loop:N
..... [980](#), [23849](#), [23865](#)
- _regex_escape_x_loop_error: . [23865](#)
- _regex_escape_x_loop_error:n ..
..... [23868](#), [23880](#), [23885](#)
- _regex_escape_x_test:N
..... [979](#), [23820](#), [23834](#)
- _regex_escape_x_testii:N ... [23834](#)
- \l_regex_every_match_tl
..... [25466](#), [25545](#), [25549](#), [25558](#)
- _regex_extract: ... [1048](#), [26181](#),
[26187](#), [26199](#), [26276](#), [26320](#), [26343](#)
- _regex_extract_all:nnN [26130](#), [26175](#)
- _regex_extract_b:wn [26276](#)
- _regex_extract_e:wn [26276](#)

__regex_extract_once:nnN
 [26128](#), [26175](#)
 __regex_extract_seq_aux:n
 [26241](#), [26259](#)
 __regex_extract_seq_aux:ww .. [26259](#)
 \l_regex_fresh_thread_bool
 [1026](#), [1031](#), [25440](#), [25446](#),
 [25467](#), [25583](#), [25643](#), [25645](#), [25705](#)
 __regex_get_digits:NTFw
 [23990](#), [24235](#), [24250](#)
 __regex_get_digits_loop:nw
 [23993](#), [23996](#), [23999](#)
 __regex_get_digits_loop:w ... [23990](#)
 __regex_group:nnnN ... [983](#), [1000](#),
 [24623](#), [24628](#), [24954](#), [25124](#), [25252](#)
 __regex_group_aux:nnnnN
 ... [1019](#), [25234](#), [25254](#), [25262](#), [25265](#)
 __regex_group_aux:nnnnnN [1019](#)
 __regex_group_end_extract_seq:N
 [26182](#), [26190](#), [26230](#), [26232](#)
 __regex_group_end_replace:N ...
 [26337](#), [26366](#), [26368](#)
 \l_regex_group_level_int
 [23951](#), [24082](#),
 [24100](#), [24102](#), [24104](#), [24582](#), [24588](#)
 __regex_group_no_capture:nnnN ..
 [983](#), [24646](#), [24956](#), [25252](#)
 __regex_group_repeat:nn [25247](#), [25296](#)
 __regex_group_repeat:nnN
 [25248](#), [25336](#)
 __regex_group_repeat:nnnN
 [25249](#), [25367](#)
 __regex_group_repeat_aux:n
 [1020](#), [1021](#), [25303](#), [25316](#), [25354](#), [25371](#)
 __regex_group_resetting:nnnN ...
 [983](#), [24648](#), [24958](#), [25263](#)
 __regex_group_resetting_-
 loop:nnNn [25263](#)
 __regex_group_submatches:nnN ...
 ... [25304](#), [25309](#), [25339](#), [25355](#), [25369](#)
 __regex_hexadecimal_use:N ... [23890](#)
 __regex_hexadecimal_use:NTF ...
 [23851](#), [23860](#), [23870](#), [23890](#)
 __regex_if_end_range:NN [24324](#)
 __regex_if_end_range:NNTF ... [24324](#)
 __regex_if_in_class:TF
 [24014](#), [24093](#), [24166](#),
 [24182](#), [24308](#), [24371](#), [24442](#), [24458](#),
 [24603](#), [24634](#), [24642](#), [26443](#), [26456](#)
 __regex_if_in_class_or_catcode:TF
 ... [24034](#), [24400](#), [24424](#), [24433](#), [24855](#)
 __regex_if_in_cs:TF
 [24022](#), [24791](#), [26441](#), [26450](#)
 __regex_if_match:nn
 [26107](#), [26112](#), [26154](#)
 __regex_if_raw_digit:NN [24002](#)
 __regex_if_raw_digit:NNTF
 [23992](#), [23998](#), [24002](#)
 __regex_if_two_empty_matches:TF
 ... [1026](#), [25467](#), [25528](#), [25534](#), [25701](#)
 __regex_if_within_catcode:TF ...
 [24046](#), [24461](#)
 __regex_int_eval:w
 ... [23491](#), [25728](#), [25729](#), [25740](#), [26297](#)
 \l_regex_internal_a_int
 [992](#), [1039](#), [23529](#),
 [24235](#), [24246](#), [24257](#), [24266](#), [24270](#),
 [24278](#), [24281](#), [24285](#), [24288](#), [24295](#),
 [25216](#), [25219](#), [25225](#), [25230](#), [25305](#),
 [25320](#), [25326](#), [25332](#), [25341](#), [25344](#),
 [25348](#), [25351](#), [25356](#), [25359](#), [25362](#),
 [25377](#), [25385](#), [25394](#), [25891](#), [25912](#)
 \l_regex_internal_a_tl
 ... [978](#), [1008](#), [1008](#), [1009](#), [1050](#),
 [1053](#), [23529](#), [23660](#), [23663](#), [23766](#),
 [23773](#), [23780](#), [24529](#), [24534](#), [24550](#),
 [24555](#), [24560](#), [24564](#), [24570](#), [24571](#),
 [24799](#), [24810](#), [24860](#), [24890](#), [24902](#),
 [24918](#), [24948](#), [24951](#), [25002](#), [25017](#),
 [25059](#), [25066](#), [25153](#), [25154](#), [25155](#),
 [25156](#), [25287](#), [25288](#), [25292](#), [25294](#),
 [26093](#), [26102](#), [26326](#), [26356](#), [26386](#)
 \l_regex_internal_b_int
 ... [23529](#), [24250](#), [24279](#), [24282](#),
 [24283](#), [24285](#), [24289](#), [24296](#), [25321](#),
 [25326](#), [25331](#), [25377](#), [25385](#), [25394](#)
 \l_regex_internal_b_tl [23529](#)
 \l_regex_internal_bool
 ... [23529](#), [24528](#), [24533](#), [24554](#), [24563](#)
 \l_regex_internal_c_int
 ... [23529](#), [25323](#), [25328](#), [25329](#), [25333](#)
 \l_regex_internal_regex
 ... [988](#), [23975](#), [24122](#), [24801](#), [24807](#),
 [25113](#), [26075](#), [26080](#), [26085](#), [26090](#)
 \l_regex_internal_seq ... [23529](#),
 [25083](#), [25084](#), [25089](#), [25096](#), [25097](#),
 [25098](#), [25100](#), [26236](#), [26254](#), [26257](#)
 \g_regex_internal_tl
 ... [23529](#), [23771](#), [23775](#), [24899](#), [24906](#)
 __regex_item_caseful_equal:n ...
 [983](#), [23567](#), [23692](#),
 [23693](#), [23697](#), [23698](#), [23699](#), [23700](#),
 [23701](#), [23710](#), [23715](#), [23733](#), [23751](#),
 [24086](#), [24673](#), [24835](#), [24913](#), [24970](#)
 __regex_item_caseful_range:nn ..
 [983](#), [23567](#), [23689](#),
 [23704](#), [23707](#), [23708](#), [23709](#), [23723](#),

23730, 23737, 23739, 23741, 23744,
 23745, 23746, 23747, 23752, 23755,
 23760, 23761, 24087, 24675, 24972
 _regex_item_caseless_equal:n ..
 983, 23581, 24654, 24977
 _regex_item_caseless_range:nn ..
 983, 23581, 24656, 24979
 _regex_item_catcode: 23628
 _regex_item_catcode:nTF
 983, 998, 23628, 24175, 24483, 24984
 _regex_item_catcode_reverse:nTF
 983, 23628, 24484, 24986
 _regex_item_cs:n
 971, 983, 23668, 24807, 24993
 _regex_item_equal:n
 23626, 24086, 24314, 24320,
 24350, 24363, 24364, 24653, 24672
 _regex_item_exact:nn
 983, 1008, 23648, 24928, 24990
 _regex_item_exact_cs:n
 983, 1005, 23648, 24809, 24925, 24992
 _regex_item_range:nn
 .. 23626, 24087, 24352, 24655, 24674
 _regex_item_reverse:n
 983, 999, 23562, 23647,
 23714, 24388, 24554, 24988, 25431
 \l_regex_last_char_int
 25428, 25454, 25571
 \l_regex_left_state_int
 25105, 25122, 25147, 25154,
 25165, 25172, 25175, 25176, 25178,
 25179, 25205, 25213, 25216, 25240,
 25288, 25290, 25300, 25320, 25340,
 25342, 25370, 25373, 25376, 25379,
 25391, 25404, 25413, 25437, 25444
 \l_regex_left_state_seq
 25105, 25146, 25153, 25287
 _regex_match:n
 25473, 26160, 26170,
 26180, 26189, 26214, 26316, 26346
 \l_regex_match_count_int
 1047, 1048, 26137, 26167, 26168, 26173
 _regex_match_cs:n ... 23679, 25473
 _regex_match_init: 25473
 _regex_match_loop:
 1028, 1031, 25544, 25567
 _regex_match_once:
 1028, 1029, 25484, 25506, 25525, 25563
 _regex_match_one_active:n .. 25567
 \l_regex_match_success_bool ...
 1027,
 25470, 25537, 25553, 25560, 25703
 \l_regex_max_active_int
 1015, 25130, 25462, 25539, 25576,
 25579, 25584, 25681, 25682, 25686
 \l_regex_max_pos_int 1035,
 24413, 24414, 24421, 25077, 25139,
 25449, 25481, 25492, 25503, 25585,
 26215, 26220, 26226, 26335, 26364
 \l_regex_max_state_int
 .. 1013, 1015, 1060, 25102, 25119,
 25131, 25164, 25166, 25167, 25226,
 25238, 25299, 25319, 25321, 25329,
 25373, 25379, 25387, 25397, 25476,
 25491, 25512, 25517, 25521, 26694
 \l_regex_min_active_int
 1015, 25462,
 25517, 25539, 25576, 25578, 25584
 \l_regex_min_pos_int
 1035, 24411, 24420,
 25075, 25449, 25478, 25496, 25519
 \l_regex_min_state_int
 1015, 25102, 25119, 25130,
 25131, 25491, 25512, 25540, 26693
 \l_regex_min_submatch_int
 1047, 1050, 1053, 25520,
 25522, 26140, 26238, 26351, 26359
 \l_regex_mode_int 23952,
 24016, 24024, 24027, 24036, 24039,
 24048, 24056, 24059, 24069, 24070,
 24072, 24074, 24128, 24142, 24144,
 24444, 24448, 24449, 24450, 24477,
 24488, 24605, 24695, 24696, 24724,
 24725, 24781, 24782, 24891, 24937
 _regex_mode_quit_c:
 24067, 24165, 24578
 _regex_msg_repeated:nnN
 25032, 25053, 25063, 26663
 _regex_multi_match:n 1026,
 25547, 26168, 26187, 26195, 26343
 \c_regex_no_match_regex
 23538, 23975, 26067
 \c_regex_outer_mode_int
 23952, 24027, 24039, 24048, 24056,
 24070, 24128, 24144, 24891, 24937
 _regex_pop_lr_states:
 25136, 25144, 25245
 _regex_posix_alnum: 23717
 _regex_posix_alpha: 23717
 _regex_posix_ascii: 23717
 _regex_posix_blank: 23717
 _regex_posix_cntrl: 23717
 _regex_posix_digit: 23717
 _regex_posix_graph: 23717
 _regex_posix_lower: 23717
 _regex_posix_print: 23717
 _regex_posix_punct: 23717
 _regex_posix_space: 23717

_regex_posix_upper: [23717](#)
 _regex_posix_word: [23717](#)
 _regex_posix_xdigit: [23717](#)
 _regex_prop_: [995](#), [24369](#)
 _regex_prop_d: ... [995](#), [23688](#), [23735](#)
 _regex_prop_h: [23688](#), [23727](#)
 _regex_prop_N: [23688](#), [24397](#)
 _regex_prop_s: [23688](#)
 _regex_prop_v: [23688](#)
 _regex_prop_w:
 .. [23688](#), [23756](#), [25429](#), [25431](#), [25432](#)
 _regex_push_lr_states:
 [25134](#), [25144](#), [25243](#)
 _regex_quark_if_nil:N [23558](#)
 _regex_quark_if_nil:NTF
 [24825](#), [24845](#)
 _regex_quark_if_nil:nTF ... [23558](#)
 _regex_quark_if_nil_p:n ... [23558](#)
 _regex_query_get:
 [25543](#), [25573](#), [25611](#)
 _regex_query_range:nn
 [1035](#), [25720](#),
[25725](#), [25744](#), [25829](#), [26330](#), [26363](#)
 _regex_query_range_loop:ww . [25725](#)
 _regex_query_set:nnn
 [1027](#), [25477](#), [25480](#),
[25482](#), [25495](#), [25499](#), [25504](#), [25596](#)
 _regex_query_submatch:n
 [25742](#), [25877](#), [26270](#)
 _regex_replace_all:nnN [26134](#), [26340](#)
 _regex_replace_once:nnN
 [26132](#), [26310](#)
 _regex_replacement:n
 [25783](#), [26315](#), [26345](#)
 _regex_replacement_aux:n ... [25783](#)
 _regex_replacement_balance_
 one_match:n [1034](#),
[1034](#), [25716](#), [25814](#), [26323](#), [26354](#)
 _regex_replacement_c:w [25921](#)
 _regex_replacement_c_A:w
 [1038](#), [26002](#)
 _regex_replacement_c_B:w ... [26005](#)
 _regex_replacement_c_C:w ... [26014](#)
 _regex_replacement_c_D:w ... [26019](#)
 _regex_replacement_c_E:w ... [26022](#)
 _regex_replacement_c_L:w ... [26031](#)
 _regex_replacement_c_M:w ... [26034](#)
 _regex_replacement_c_O:w ... [26037](#)
 _regex_replacement_c_P:w ... [26040](#)
 _regex_replacement_c_S:w ... [26046](#)
 _regex_replacement_c_T:w ... [26054](#)
 _regex_replacement_c_U:w ... [26057](#)
 _regex_replacement_cat:NNN ...
 [25929](#), [25962](#)
 \l_regex_replacement_category_
 seq [25713](#),
[25808](#), [25811](#), [25812](#), [25848](#), [25976](#)
 \l_regex_replacement_category_
 tl [1038](#), [25713](#),
[25843](#), [25849](#), [25855](#), [25977](#), [25978](#)
 _regex_replacement_char:nNN ...
 [1044](#), [25997](#),
[26004](#), [26011](#), [26021](#), [26028](#), [26033](#),
[26036](#), [26039](#), [26043](#), [26056](#), [26059](#)
 \l_regex_replacement_csnames_
 int [1034](#), [25712](#), [25802](#),
[25804](#), [25806](#), [25878](#), [25937](#), [25944](#),
[25955](#), [25957](#), [25967](#), [26008](#), [26025](#)
 _regex_replacement_cu_aux:Nw ...
 [25926](#), [25935](#), [25950](#)
 _regex_replacement_do_one_
 match:n . [25718](#), [25827](#), [26328](#), [26362](#)
 _regex_replacement_error:NNN ...
 [25892](#), [25904](#),
[25915](#), [25930](#), [25933](#), [25951](#), [26061](#)
 _regex_replacement_escaped:N ...
 [25798](#), [25861](#), [25981](#)
 _regex_replacement_exp_not:N ...
 [1040](#), [25724](#), [25926](#)
 _regex_replacement_g:w [25887](#)
 _regex_replacement_g_digits:NN
 [25887](#)
 _regex_replacement_normal:n ...
[25794](#), [25799](#), [25841](#), [25868](#), [25890](#),
[25896](#), [25923](#), [25949](#), [25959](#), [25974](#)
 _regex_replacement_put_
 submatch:n ... [25866](#), [25873](#), [25911](#)
 _regex_replacement_rbrace:N ...
 [25792](#), [25910](#), [25953](#)
 _regex_replacement_u:w [25946](#)
 _regex_return: [1046](#),
[26108](#), [26113](#), [26124](#), [26126](#), [26146](#)
 \l_regex_right_state_int
 [25105](#), [25125](#), [25137](#),
[25149](#), [25156](#), [25165](#), [25166](#), [25205](#),
[25212](#), [25218](#), [25231](#), [25238](#), [25240](#),
[25290](#), [25294](#), [25305](#), [25319](#), [25328](#),
[25340](#), [25344](#), [25348](#), [25351](#), [25356](#),
[25359](#), [25362](#), [25370](#), [25384](#), [25387](#),
[25390](#), [25393](#), [25397](#), [25413](#), [25444](#)
 \l_regex_right_state_seq
 [25105](#), [25148](#), [25155](#), [25292](#)
 \l_regex_saved_success_bool ...
 [1027](#), [23677](#), [23684](#), [25470](#)
 _regex_show:N . [24941](#), [26090](#), [26099](#)
 _regex_show_anchor_to_str:N ...
 [24969](#), [25070](#)

__regex_show_class:NnnnN
 [24960](#), [25034](#)
 __regex_show_group_aux:nnnnN ...
 [24955](#), [24957](#), [24959](#), [25025](#)
 __regex_show_item_catcode:NnTF .
 [24985](#), [24987](#), [25081](#)
 __regex_show_item_exact_cs:n ...
 [24992](#), [25094](#)
 \l__regex_show_lines_int
 .. [23977](#), [25006](#), [25038](#), [25041](#), [25048](#)
 __regex_show_one:n [24949](#),
 [24962](#), [24965](#), [24971](#), [24974](#), [24978](#),
 [24981](#), [24991](#), [24995](#), [25004](#), [25020](#),
 [25027](#), [25031](#), [25044](#), [25060](#), [25099](#)
 __regex_show_pop: [25014](#), [25030](#)
 \l__regex_show_prefix_seq
 [23976](#), [24947](#),
 [24950](#), [24996](#), [25010](#), [25015](#), [25017](#)
 __regex_show_push:n
 [24997](#), [25014](#), [25028](#), [25039](#)
 __regex_show_scope:nn
 [24989](#), [24994](#), [25014](#), [25086](#)
 __regex_single_match: [1026](#),
 [23674](#), [25547](#), [26158](#), [26178](#), [26313](#)
 __regex_split:nnN [26136](#), [26192](#)
 __regex_standard_escapechar: ...
 [23492](#), [23770](#), [24127](#), [25117](#)
 \l__regex_start_pos_int
 [24412](#), [25076](#),
 [25449](#), [25531](#), [25536](#), [25542](#), [26198](#),
 [26210](#), [26223](#), [26226](#), [26300](#), [26364](#)
 \g__regex_state_active_intarray .
 [968](#), [1015](#), [1026](#), [1026](#), [1027](#), [25464](#),
 [25515](#), [25622](#), [25625](#), [25633](#), [25660](#)
 \l__regex_step_int
 [968](#), [25461](#), [25518](#), [25569](#),
 [25623](#), [25627](#), [25635](#), [25649](#), [25651](#)
 __regex_store_state:n
 [25540](#), [25674](#), [25677](#)
 __regex_store_submatches:
 [25677](#), [25691](#)
 __regex_submatch_balance:n
 .. [25717](#), [25748](#), [25818](#), [25881](#), [26262](#)
 \g__regex_submatch_begin_-
 intarray [969](#), [1034](#), [25722](#),
 [25745](#), [25770](#), [25778](#), [25836](#), [26143](#),
 [26205](#), [26208](#), [26221](#), [26282](#), [26306](#)
 \g__regex_submatch_end_intarray .
 . [969](#), [25746](#), [25755](#), [25763](#), [26143](#),
 [26202](#), [26218](#), [26284](#), [26309](#), [26332](#)
 \l__regex_submatch_int
 [969](#), [1047](#), [1049](#), [1050](#), [1053](#),
 [25522](#), [26140](#), [26217](#), [26219](#), [26222](#),
 [26224](#), [26227](#), [26239](#), [26279](#), [26283](#),
 [26285](#), [26287](#), [26288](#), [26353](#), [26361](#)
 \g__regex_submatch_prev_intarray
 .. [969](#), [1047](#), [1051](#), [25721](#), [25832](#),
 [26143](#), [26200](#), [26216](#), [26286](#), [26299](#)
 \g__regex_success_bool [1027](#),
 [23678](#), [23680](#), [23683](#), [25470](#), [25510](#),
 [25552](#), [25561](#), [26148](#), [26278](#), [26317](#)
 \l__regex_success_pos_int
 .. [25449](#), [25519](#), [25536](#), [25706](#), [26198](#)
 \l__regex_success_submatches_-
 prop [1025](#), [1051](#), [25459](#), [25707](#), [26290](#)
 __regex_tests_action_cost:n ...
 [25183](#), [25204](#), [25213](#), [25231](#)
 \g__regex_thread_state_intarray .
 [969](#), [1015](#), [1024](#),
 [1026](#), [1026](#), [1032](#), [25464](#), [25593](#), [25680](#)
 __regex_tmp:w
 [23511](#), [23513](#), [23517](#), [23519](#), [23528](#),
 [24381](#), [24391](#), [24392](#), [24393](#), [24394](#),
 [24395](#), [24406](#), [24411](#), [24412](#), [24413](#),
 [24414](#), [24415](#), [24420](#), [24421](#), [26119](#),
 [26128](#), [26130](#), [26132](#), [26134](#), [26136](#)
 __regex_toks_clear:N . [23495](#), [25164](#)
 __regex_toks_memcpy:NNn [23500](#), [25330](#)
 __regex_toks_put_left:Nn
 [23509](#), [25159](#), [25312](#), [25313](#)
 __regex_toks_put_right:Nn
 [970](#), [23509](#), [25122](#), [25125](#),
 [25137](#), [25161](#), [25172](#), [25404](#), [25437](#)
 __regex_toks_set:Nn
 [23495](#), [25604](#), [25686](#)
 __regex_toks_use:w
 .. [23494](#), [25594](#), [25624](#), [25738](#), [26697](#)
 __regex_trace:nnn [26679](#), [26696](#)
 __regex_trace_pop:nnN [26679](#)
 __regex_trace_push:nnN [26679](#)
 \g__regex_trace_regex_int [26689](#)
 __regex_trace_states:n [26690](#)
 __regex_two_if_eq:NNNN [23978](#)
 __regex_two_if_eq:NNNTF
 [23978](#), [24216](#), [24263](#),
 [24276](#), [24310](#), [24489](#), [24526](#), [24546](#),
 [24547](#), [24616](#), [24651](#), [24668](#), [24669](#),
 [24731](#), [24858](#), [25889](#), [25948](#), [25974](#)
 __regex_use_i_delimit_by_q_-
 recursion_stop:nw .. [23553](#), [24848](#)
 __regex_use_none_delimit_by_q_-
 recursion_stop:w
 [23553](#), [24826](#), [24850](#)
 __regex_use_state:
 [25620](#), [25637](#), [25663](#)
 __regex_use_state_and_submatches:nn
 [1029](#), [25592](#), [25629](#)

- \l__regex_zeroth_submatch_int . . . 1047, 1051, 26140, 26201, 26203, 26206, 26209, 26279, 26297, 26300, 26324, 26329, 26333
 - \relax 4, 21, 25, 30, 64, 66, 67, 68, 79, 86, 107, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 422
 - \relpenalty 423
 - \RequirePackage 110
 - \resettimer 780
 - reverse commands:
 - \reverse_if:N 23, 424, 505, 506, 760, 1359, 5119, 8144, 8293, 8295, 8297, 8299, 8354, 9119, 14013, 14018, 14022, 14024, 16793, 20368, 21190, 21213, 23575, 23576, 23595, 23596, 23603, 23604, 28913, 28915, 28937, 28961
 - \right 424
 - right commands:
 - \c_right_brace_str 71, 5171, 13147, 23878, 24243, 24263, 24276, 24789, 24793, 24876, 25791, 29286
 - \rightghost 890
 - \righthyphenmin 425
 - \rightmarginkern 695
 - \rightskip 426
 - \rmfamily 30848
 - \romannumeral 427
 - round 215
 - \rPCODE 696
 - \rule 28439, 28440
- S**
- s@ internal commands:
 - \s@_ 955
 - \saveboxresource 933
 - \savecatcodetable 891
 - \saveimageresource 934
 - \savepos 932
 - \savingshyphcodes 558
 - \savingsvdiscards 559
 - scan commands:
 - \scan_align_safe_stop: 32039
 - \scan_new:N 41, 376, 408, 3494, 4542, 4543, 4544, 4552, 4553, 5202, 7374, 7375, 7376, 8053, 8054, 9567, 9568, 10164, 10644, 10911, 10912, 11091, 11096, 11097, 11447, 11448, 12641, 12642, 13091, 13893, 13894, 14239, 14403, 14404, 14405, 14406, 14622, 14623, 14624, 15897, 15900, 15901, 15902, 15903, 15905, 15906, 15907, 15908, 15909, 16008, 22651, 22652, 23058, 28644, 29025, 31397, 31398, 31769, 31770
 - \scan_stop: 9, 17, 18, 18, 18, 41, 41, 91, 159, 173, 304, 308, 308, 324, 325, 327, 336, 340, 341, 352, 376, 378, 381, 390, 395, 424, 505, 510, 524, 577, 577, 584, 586, 587, 597, 621, 665, 665, 666, 671, 757, 760, 761, 762, 766, 983, 1005, 1388, 1740, 1758, 1768, 1786, 1812, 2142, 2151, 2160, 2232, 2667, 2791, 2792, 2807, 2847, 2873, 2897, 2914, 3118, 3124, 3229, 3503, 3506, 3664, 3912, 5120, 5301, 5628, 5629, 5637, 5638, 5670, 5671, 6303, 7728, 8484, 9361, 9365, 9540, 10688, 10760, 10983, 11082, 11085, 11087, 12470, 12475, 12559, 12668, 12671, 13221, 13228, 13686, 13905, 13924, 13926, 13930, 13933, 13936, 13940, 13945, 13949, 14172, 14249, 14267, 14269, 14277, 14279, 14283, 14285, 14306, 14312, 14315, 14341, 14361, 14363, 14371, 14373, 14377, 14379, 14383, 15654, 15661, 15883, 16072, 16791, 16795, 16996, 17013, 17314, 17361, 17362, 17621, 17664, 17694, 17708, 18465, 20278, 20286, 21032, 21035, 21038, 21041, 21044, 21047, 21050, 21053, 21056, 22321, 22348, 22685, 23180, 23220, 23224, 23230, 23232, 23279, 23281, 23661, 23662, 24000, 24818, 25096, 25615, 25618, 25639, 25999, 26051, 26705, 26848, 28441, 29038, 29039, 31815, 31818, 32159, 32170, 32216
- scan internal commands:
 - \g__scan_marks_tl 376, 3493, 3496, 3502, 3507, 3509
- \scantextokens 892
- \scantokens 560
- \scriptbaselineshiftfactor 1149
- \scriptfont 428
- \scriptscriptbaselineshiftfactor 1151
- \scriptscriptfont 429
- \scriptscriptstyle 430
- \scriptsize 30865
- \scriptspace 431
- \scriptstyle 432
- \scrollmode 433
- \scshape 30854
- sec 216

- secd 216
- \selectfont 30826
- seq commands:
 - \c_empty_seq 87, 480, 7385, 7389, 7393, 7396, 7584, 7662, 7670
 - \l_foo_seq 232
 - \seq_clear:N 76, 76, 87, 7392, 7399, 7528, 11864, 11927, 13778, 24996, 25812
 - \seq_clear_new:N 76, 7398
 - \seq_concat:NNN ... 77, 87, 7481, 13784
 - \seq_const_from_clist:Nn ... 77, 7438
 - \seq_count:N 78, 84, 86, 199, 7599, 7789, 7803, 7934, 7962, 25811
 - \seq_elt:w 479
 - \seq_elt_end: 479
 - \seq_gclear:N 76, 942, 7392, 7402, 7616, 22796
 - \seq_gclear_new:N 76, 7398
 - \seq_gconcat:NNN ... 77, 7481, 13797
 - \seq_get:NN ... 85, 8015, 25287, 25292
 - \seq_get:NNTF 85, 8021
 - \seq_get_left:NN 78, 7678, 8015, 8016, 8021, 8022
 - \seq_get_left:NNTF 79, 7748
 - \seq_get_right:NN 78, 7703
 - \seq_get_right:NNTF 79, 7748
 - \seq_gpop:NN ... 85, 8015, 13707, 22445
 - \seq_gpop:NNTF 86, 8021, 12459, 12657, 22416, 22428
 - \seq_gpop_left:NN 78, 7689, 8019, 8020, 8025, 8026
 - \seq_gpop_left:NNTF 79, 7756
 - \seq_gpop_right:NN 78, 7721
 - \seq_gpop_right:NNTF 80, 7756
 - \seq_gpush:Nn . 25, 86, 7995, 12484, 12681, 13692, 22420, 22430, 22439
 - \seq_gput_left:Nn 77, 7491, 8005, 8006, 8007, 8008, 8009, 8010, 8011, 8012, 8013, 8014
 - \seq_gput_right:Nn 77, 7512, 13066, 13073, 13681
 - \seq_gremove_all:Nn 80, 7538
 - \seq_gremove_duplicates:N .. 80, 7522
 - \seq_greverse:N 80, 7564
 - \seq_gset_eq:NN 76, 7396, 7404, 7525, 7596, 22770
 - \seq_gset_filter:NNn ... 270, 31476
 - \seq_gset_from_clist:NN ... 76, 7412
 - \seq_gset_from_clist:Nn ... 76, 7412
 - \seq_gset_from_function:NnN ... 270, 31496
 - \seq_gset_from_inline_x:Nnn ... 270, 7611, 22788, 31486, 31499
 - \seq_gset_map:NNn 83, 7924
 - \seq_gset_map_x:NNn 84, 7914
 - \seq_gset_split:Nnn 77, 7444
 - \seq_gshuffle:N 81, 7592
 - \seq_gsort:Nn 80, 7582, 22766
 - \seq_if_empty:NTF 81, 7582, 7802, 9648, 22479, 25808
 - \seq_if_empty_p:N 81, 7582
 - \seq_if_exist:NTF 77, 7399, 7402, 7487, 7960
 - \seq_if_exist_p:N 77, 7487
 - \seq_if_in:Nn 554
 - \seq_if_in:NnTF 81, 86, 87, 7531, 7639, 12483, 12680
 - \seq_indexed_map_function:NN . 32136
 - \seq_indexed_map_inline:Nn ... 32136
 - \seq_item:Nn 78, 235, 491, 7776, 7803, 11945, 11946, 11951
 - \seq_log:N 88, 8027
 - \seq_map_break: 83, 83, 84, 270, 7806, 7817, 7852, 7860, 7877, 7884, 7893, 15327
 - \seq_map_break:n 83, 491, 7806, 11884, 11898, 13259, 13360, 22767, 22770
 - \seq_map_function:NN 4, 81, 82, 268, 493, 7810, 8037, 9654, 11949, 13787, 25010, 25089
 - \seq_map_indexed_function:NN ... 82, 7881, 32138, 32139
 - \seq_map_indexed_inline:Nn 82, 7881, 32136, 32137
 - \seq_map_inline:Nn 81, 81, 82, 87, 1176, 7529, 7848, 11879, 13359, 15320, 22767, 22770
 - \seq_map_tokens:Nn 81, 82, 7855, 13258
 - \seq_map_variable:NNn 82, 7869
 - \seq_mapthread_function:NNN ... 270, 31454
 - \seq_new:N 4, 76, 76, 7386, 7399, 7402, 7521, 7594, 8041, 8042, 8043, 8044, 9799, 10261, 10264, 11834, 11835, 12411, 12625, 13058, 13083, 13089, 13090, 14616, 22270, 22271, 22272, 22638, 23535, 23976, 25107, 25108, 25714
 - \seq_pop:NN 85, 8015, 25153, 25155, 25848
 - \seq_pop:NNTF 86, 8021
 - \seq_pop_left:NN 78, 7689, 8017, 8018, 8023, 8024
 - \seq_pop_left:NNTF 79, 7756
 - \seq_pop_right:NN 78, 7721, 24947, 25017

- \seq_pop_right:NNTF [80](#), [7756](#)
- \seq_push:Nn
 - . [86](#), [7995](#), [8002](#), [25146](#), [25148](#), [25976](#)
- \seq_put_left:Nn [77](#),
 - [7491](#), [7995](#), [7996](#), [7997](#), [7998](#), [7999](#),
 - [8000](#), [8001](#), [8002](#), [8003](#), [8004](#), [11874](#)
- \seq_put_right:Nn [77](#), [86](#), [87](#),
 - [7512](#), [7532](#), [11935](#), [24950](#), [25015](#), [31974](#)
- \seq_rand_item:N [79](#), [7800](#)
- \seq_remove_all:Nn
 - ... [77](#), [80](#), [86](#), [87](#), [7538](#), [9825](#), [31976](#)
- \seq_remove_duplicates:N
 - [80](#), [86](#), [87](#), [7522](#), [13785](#)
- \seq_reverse:N [80](#), [485](#), [7564](#)
- \seq_set_eq:NN
 - [76](#), [87](#), [7393](#), [7404](#), [7523](#), [7595](#), [22767](#)
- \seq_set_filter:NNn
 - [270](#), [494](#), [25084](#), [31476](#)
- \seq_set_from_clist:NN [76](#), [7412](#), [9824](#)
- \seq_set_from_clist:Nn [76](#),
 - [120](#), [481](#), [7412](#), [13781](#), [13795](#), [15243](#)
- \seq_set_from_function:NnN
 - [270](#), [26236](#), [31496](#)
- \seq_set_from_inline_x:Nnn
 - [270](#), [1177](#), [31486](#), [31497](#)
- \seq_set_map:NNn [83](#), [7924](#)
- \seq_set_map_x:NNn
 - [84](#), [494](#), [7914](#), [25097](#), [26254](#)
- \seq_set_split:Nnn
 - . [77](#), [7444](#), [10262](#), [10265](#), [25083](#), [25096](#)
- \seq_show:N [88](#), [608](#), [8027](#)
- \seq_shuffle:N [81](#), [7592](#)
- \seq_sort:Nn [80](#), [225](#), [7582](#), [22766](#)
- \seq_use:Nn [85](#), [7958](#), [25100](#)
- \seq_use:Nnnn [84](#), [7958](#)
- \g_tmpa_seq [88](#), [8041](#)
- \l_tmpa_seq [88](#), [8041](#)
- \g_tmpb_seq [88](#), [8041](#)
- \l_tmpb_seq [88](#), [8041](#)
- seq internal commands:
 - __seq_count:w [495](#), [7934](#)
 - __seq_count_end:w [495](#), [7934](#)
 - __seq_get_left:wnw [7678](#)
 - __seq_get_right_end:NnN [7703](#)
 - __seq_get_right_loop:nw .. [489](#), [7703](#)
 - __seq_if_in: [7639](#)
 - \l__seq_internal_a_int
 - [7606](#), [7612](#), [7621](#), [7623](#), [7624](#)
 - \l__seq_internal_a_tl
 - [481](#), [7382](#), [7452](#), [7456](#), [7462](#),
 - [7467](#), [7469](#), [7553](#), [7558](#), [7643](#), [7647](#)
 - \l__seq_internal_b_int
 - [7622](#), [7625](#), [7626](#)
 - \l__seq_internal_b_tl
 - [7382](#), [7549](#), [7553](#), [7646](#), [7647](#)
 - \g__seq_internal_seq [7592](#)
 - __seq_item:n
 - [479](#), [479](#), [479](#), [479](#), [483](#), [487](#),
 - [488](#), [489](#), [491](#), [492](#), [492](#), [493](#), [495](#),
 - [495](#), [1175](#), [1176](#), [1176](#), [7377](#), [7495](#),
 - [7503](#), [7513](#), [7515](#), [7520](#), [7570](#), [7571](#),
 - [7573](#), [7578](#), [7608](#), [7644](#), [7683](#), [7686](#),
 - [7696](#), [7711](#), [7714](#), [7727](#), [7728](#), [7739](#),
 - [7783](#), [7792](#), [7816](#), [7819](#), [7829](#), [7834](#),
 - [7840](#), [7844](#), [7859](#), [7863](#), [7904](#), [7906](#),
 - [7920](#), [7930](#), [7941](#), [7942](#), [7943](#), [7944](#),
 - [7945](#), [7946](#), [7947](#), [7948](#), [7953](#), [7954](#),
 - [7969](#), [7984](#), [7987](#), [7990](#), [31492](#), [31493](#)
 - __seq_item:nN [7776](#)
 - __seq_item:nwn [7776](#)
 - __seq_item:wNn [7776](#)
 - __seq_map_function:NNn [7810](#)
 - __seq_map_function:Nw
 - [7813](#), [7819](#), [7823](#)
 - __seq_map_indexed:NN [7883](#), [7891](#), [7896](#)
 - __seq_map_indexed:nNN [7881](#)
 - __seq_map_indexed:Nw [493](#), [7881](#)
 - __seq_map_tokens:nw [7855](#)
 - __seq_mapthread_function:Nnnwnn
 - [31454](#)
 - __seq_mapthread_function:wNN . [31454](#)
 - __seq_mapthread_function:wNw . [31454](#)
 - __seq_pop:NNNN
 - [7660](#), [7690](#), [7692](#), [7722](#), [7724](#)
 - __seq_pop_item_def:
 - [479](#), [479](#), [7560](#), [7610](#), [7826](#),
 - [7852](#), [7877](#), [7922](#), [7932](#), [31484](#), [31494](#)
 - __seq_pop_left:NNN . [7689](#), [7758](#), [7761](#)
 - __seq_pop_left:wnNNN [7689](#)
 - __seq_pop_right:NNN
 - [484](#), [7721](#), [7764](#), [7767](#)
 - __seq_pop_right_loop:nn [7721](#)
 - __seq_pop_TF:NNNN [490](#), [7660](#),
 - [7749](#), [7751](#), [7758](#), [7761](#), [7764](#), [7767](#)
 - __seq_push_item_def: ... [7607](#), [7826](#)
 - __seq_push_item_def:n
 - [479](#), [479](#), [7544](#), [7826](#),
 - [7850](#), [7871](#), [7920](#), [7930](#), [31482](#), [31492](#)
 - __seq_put_left_aux:w [483](#), [7491](#)
 - __seq_remove_all_aux:NNn [7538](#)
 - __seq_remove_duplicates:NN .. [7522](#)
 - \l__seq_remove_seq
 - [7521](#), [7528](#), [7531](#), [7532](#), [7534](#)
 - __seq_reverse:NN [7564](#)
 - __seq_reverse_item:nw [485](#)
 - __seq_reverse_item:nwn [7564](#)
 - __seq_set_filter:NNNn [31476](#)

- __seq_set_from_inline_x:NNnn . 31486
- __seq_set_map:NNNn . 7924
- __seq_set_map_x:NNNn . 7914
- __seq_set_split:NNnn . 7444
- __seq_set_split_auxi:w . 481, 7444
- __seq_set_split_auxii:w . 481, 7444
- __seq_set_split_end: . 481, 7444
- __seq_show:NN . 8027
- __seq_shuffle:NN . 7592
- __seq_shuffle_item:n . 7592
- __seq_tmp:w . 7384, 7570, 7573, 7727, 7739
- __seq_use:NNnNnn . 7958
- __seq_use:nwwn . 7958
- __seq_use:nwwwwnwn . 7958
- __seq_use_setup:w . 7958
- __seq_wrap_item:n . 481, 1176, 7415, 7420, 7425, 7430, 7441, 7453, 7478, 7520, 7556, 31482
- \setbox . 434
- \setfontid . 893
- \setlanguage . 435
- \setrandomseed . 935
- \sfcode . 436
- \sffamily . 28419, 30849
- \shapemode . 894
- \shellescape . 781
- \Shipout . 1212
- \shipout . 437, 1199, 1200
- \ShortText . 49, 96
- \show . 438
- \showbox . 439
- \showboxbreadth . 440
- \showboxdepth . 441
- \showgroups . 561
- \showifs . 562
- \showlists . 442
- \showmode . 1153
- \showthe . 443
- \ShowTokens . 226
- \showtokens . 563
- sign . 215
- sin . 216
- sind . 216
- \sjis . 1154
- \skewchar . 444
- \skip . 445, 10838
- skip commands:
 - \c_max_skip . 182, 14326
 - \skip_add:Nn . 180, 14276
 - \skip_const:Nn . 180, 678, 14246, 14326, 14327
 - \skip_eval:n . 181, 181, 181, 181, 14249, 14290, 14305, 14321, 14325
 - \skip_gadd:Nn . 180, 14276
 - .skip_gset:N . 191, 15076
 - \skip_gset:Nn . 180, 674, 14266
 - \skip_gset_eq:NN . 180, 14272
 - \skip_gsub:Nn . 180, 14276
 - \skip_gzero:N . 180, 14252, 14259
 - \skip_gzero_new:N . 180, 14256
 - \skip_horizontal:N . 182, 14310
 - \skip_horizontal:n . 182, 14310
 - \skip_if_eq:nnTF . 181, 14288
 - \skip_if_eq_p:nn . 181, 14288
 - \skip_if_exist:NTF . 180, 14257, 14259, 14262
 - \skip_if_exist_p:N . 180, 14262
 - \skip_if_finite:nTF . 181, 14294
 - \skip_if_finite_p:n . 181, 14294
 - \skip_log:N . 182, 14322
 - \skip_log:n . 182, 14322
 - \skip_new:N . 179, 180, 14240, 14248, 14257, 14259, 14328, 14329, 14330, 14331
 - .skip_set:N . 191, 15076
 - \skip_set:Nn . 180, 14266
 - \skip_set_eq:NN . 180, 14272
 - \skip_show:N . 181, 14318
 - \skip_show:n . 181, 677, 14320
 - \skip_sub:Nn . 180, 14276
 - \skip_use:N . 181, 181, 14299, 14306, 14307
 - \skip_vertical:N . 183, 14310
 - \skip_vertical:n . 183, 14310
 - \skip_zero:N . 180, 180, 183, 14252, 14257
 - \skip_zero_new:N . 180, 14256
 - \g_tmpa_skip . 182, 14328
 - \l_tmpa_skip . 182, 14328
 - \g_tmpb_skip . 182, 14328
 - \l_tmpb_skip . 182, 14328
 - \c_zero_skip . 182, 665, 13908, 13910, 14252, 14253, 14326
- skip internal commands:
 - __skip_if_finite:wwNw . 14294
 - __skip_tmp:w . 14294, 14304
- \skipdef . 446
- \slshape . 30855
- \small . 30866
- sort commands:
 - \sort_ordered: . 32041
 - \sort_return_same: . 225, 225, 944, 22849, 32042
 - \sort_return_swapped: . 225, 225, 944, 22849, 32044
 - \sort_reversed: . 32043
- sort internal commands:
 - __sort:nnNnn . 946, 947

- \l__sort_A_int
 - . 944, 22648, 22655, 22662, 22665,
 - 22674, 22813, 22818, 22821, 22841,
 - 22873, 22880, 22895, 22897, 22898
- \l__sort_B_int
 - . 943, 944, 22648, 22818, 22822,
 - 22830, 22832, 22833, 22885, 22886,
 - 22895, 22896, 22905, 22906, 22908
- \l__sort_begin_int
 - 938, 944, 22646, 22810, 22898, 22908
- \l__sort_block_int
 - . 938, 939, 942, 22645, 22657,
 - 22662, 22666, 22669, 22674, 22675,
 - 22740, 22801, 22804, 22811, 22814
- \l__sort_C_int
 - . 943, 944, 22648, 22819, 22823,
 - 22830, 22831, 22842, 22874, 22881,
 - 22885, 22887, 22888, 22905, 22907
- __sort_compare:nn
 - . 941, 945, 22739, 22840
- __sort_compute_range:
 - . 938, 939, 939, 22679, 22727
- __sort_copy_block: 943, 22820, 22828
- __sort_disable_toksdef: 22726, 23005
- __sort_disabled_toksdef:n ... 23005
- \l__sort_end_int 938, 943, 943, 944,
 - 22646, 22802, 22810, 22811, 22812,
 - 22813, 22814, 22815, 22816, 22833
- __sort_error: ... 22999, 23011, 23029
- __sort_i:nnnnNn 948
- \g__sort_internal_seq
 - 941, 942, 22638, 22788, 22795, 22796
- \g__sort_internal_tl
 - . 22638, 22751, 22754, 22755
- \l__sort_length_int
 - . 938, 938, 22640, 22737, 22801
- __sort_level:
 - . 941, 951, 22741, 22799, 23003
- __sort_loop:wNn 947
- __sort_main:NNNn
 - . 941, 942, 22724, 22750, 22787
- \l__sort_max_int
 - . 938, 939, 22640, 22659, 22731
- \c__sort_max_length_int 22679
- __sort_merge_blocks:
 - . 22803, 22808, 23002
- __sort_merge_blocks_aux:
 - 943, 22824, 22838, 22891, 22901, 23001
- __sort_merge_blocks_end:
 - . 945, 22899, 22903
- \l__sort_min_int ... 938, 939, 941,
 - 941, 22640, 22656, 22664, 22681,
 - 22697, 22705, 22718, 22728, 22738,
 - 22752, 22791, 22802, 23027, 23028
- __sort_quick_cleanup:w 22913
- __sort_quick_end:nnTFNn
 - . 949, 950, 22933, 22973
- __sort_quick_only_i:NnnnnNn . 22938
- __sort_quick_only_i_end:nnwnw .
 - . 22949, 22973
- __sort_quick_only_ii:NnnnnNn . 22938
- __sort_quick_only_ii_end:nnwnw
 - . 22956, 22973
- __sort_quick_prepare:Nnnn ... 22913
- __sort_quick_prepare_end:NNNnw .
 - . 22913
- __sort_quick_single_end:nnwnw .
 - . 22942, 22973
- __sort_quick_split:NnNn
 - . 948, 948, 22933,
 - 22938, 22978, 22985, 22991, 22993
- __sort_quick_split_end:nnwnw ..
 - . 22963, 22970, 22973
- __sort_quick_split_i:NnnnnNn ...
 - . 947, 22938
- __sort_quick_split_ii:NnnnnNn 22938
- __sort_redefine_compute_range: .
 - . 22679
- __sort_return_mark:w
 - . 944, 22844, 22845, 22849
- __sort_return_none_error:
 - . 944, 22847, 22849, 22883, 22893
- __sort_return_same:w
 - . 944, 22857, 22875, 22883
- __sort_return_swapped:w 22867, 22893
- __sort_return_two_error: 944, 22849
- __sort_seq:NNNn 941, 22766
- __sort_shrink_range: 939,
 - 940, 22653, 22683, 22699, 22707, 22720
- __sort_shrink_range_loop: ... 22653
- __sort_tl:NNn 941, 22743
- __sort_tl_toks:w 941, 22743
- __sort_too_long_error:NNw
 - . 22732, 23022
- \l__sort_top_int 938, 941, 941, 943,
 - 944, 22640, 22728, 22731, 22734,
 - 22735, 22738, 22760, 22791, 22812,
 - 22815, 22816, 22819, 22888, 23028
- \l__sort_true_max_int
 - . 938, 939, 22640, 22656, 22669,
 - 22682, 22698, 22706, 22719, 23027
- sp 219
- spac commands:
 - \spac_directions_normal_body_dir
 - . 1349
 - \spac_directions_normal_page_dir
 - . 1350
 - \spacefactor 447

- \spaceskip 448
- \span 449
- \special 450
- \splitbotmark 451
- \splitbotmarks 564
- \splitdiscards 565
- \splitfirstmark 452
- \splitfirstmarks 566
- \splitmaxdepth 453
- \splittopskip 454
- sqr t 217
- \SS 29180, 30591, 30948
- \ss 29180, 30591, 30944
- str commands:
 - \c_ampersand_str 71, 5171
 - \c_atsign_str 71, 5171
 - \c_backslash_str
 - . 71, 5171, 5869, 5871, 5894, 5923,
 - 5925, 5957, 5966, 5970, 23788, 24360
 - \c_circumflex_str 71, 5171
 - \c_colon_str
 - . 71, 5171, 10751, 10856, 10862, 14714
 - \c_dollar_str 71, 5171
 - \l_foo_str 72
 - \c_hash_str
 - . 71, 5171, 5837, 5940, 6513, 6514,
 - 6517, 6520, 28961, 28992, 28993, 28997
 - \c_percent_str .. 71, 5171, 5839, 5993
 - str_byte 5221
 - \str_case:nn
 - . 64, 4729, 12301, 13543, 24509, 31503
 - \str_case:nnn 32045, 32047
 - \str_case:nnTF 64, 668,
 - 4729, 4734, 4739, 9285, 9307, 9544,
 - 12238, 14889, 25073, 32046, 32048
 - \str_case_e:nn 64, 4729, 32050
 - \str_case_e:nnTF
 - . 64, 2602, 4729, 4765, 4770, 5892,
 - 24241, 32052, 32054, 32056, 32058
 - \str_case_x:nn 32049
 - \str_case_x:nnn 32051
 - \str_case_x:nnTF . 32053, 32055, 32057
 - \str_clear:N 61,
 - 61, 4560, 14696, 14849, 15265, 15266
 - \str_clear_new:N 61, 4560
 - \str_concat:NNN 61, 4560
 - \str_const:Nn
 - 60, 4587, 5171, 5172, 5173,
 - 5174, 5175, 5176, 5177, 5178, 5179,
 - 5180, 5181, 5182, 5933, 5934, 5956,
 - 9245, 9276, 9510, 13866, 13873,
 - 13877, 13881, 14593, 14594, 14595,
 - 14596, 14597, 14598, 14599, 31501
 - \str_convert_pdfname:n 74, 6489
 - \str_count:N 66,
 - 5061, 11568, 11569, 11748, 11749,
 - 11770, 11771, 12740, 12818, 23472
 - \str_count:n 66, 5061, 23466
 - \str_count_ignore_spaces:n
 - 66, 422, 5061, 23087
 - \str_count_spaces:N 66, 5041
 - \str_count_spaces:n 66, 422, 5041, 5067
 - \str_declare_eight_bit_encoding:nnn
 - . 74, 439, 5598, 6528, 6535, 6599,
 - 6641, 6698, 6799, 6886, 6972, 7046,
 - 7059, 7112, 7210, 7273, 7311, 7326
 - str_end 6392
 - str_error 5221
 - \str_fold_case:n 32124
 - \str_foldcase:n
 - 69, 70, 131, 262, 5129,
 - 16982, 32132, 32133, 32134, 32135
 - \str_gclear:N 61, 4560
 - \str_gclear_new:N 4560
 - \str_gconcat:NNN 61, 4560
 - \str_gput_left:Nn 61, 4587
 - \str_gput_right:Nn 62, 4587
 - \str_gremove_all:Nn 62, 4657
 - \str_gremove_once:Nn 62, 4651
 - \str_greplace_all:Nnn 62, 4611, 4660
 - \str_greplace_once:Nnn 62, 4611, 4654
 - \str_gset:Nn
 - 61, 4587, 13713, 13714, 13715
 - \str_gset_convert:Nnnn 72, 5360
 - \str_gset_convert:NnnnTF ... 74, 5360
 - \str_gset_eq:NN
 - 61, 4560, 13700, 13701, 13702
 - \str_head:N 67, 423, 5099
 - \str_head:n
 - 67, 401, 423, 4284, 4331, 5099
 - \str_head_ignore_spaces:n .. 67, 5099
 - \str_if_empty:NnTF
 - . 63, 4663, 13376, 14652, 14675, 15490
 - \str_if_empty_p:N 63, 4663
 - \str_if_eq:NN 413
 - \str_if_eq:nn 145, 587, 596
 - \str_if_eq:NNTF 63, 4707
 - \str_if_eq:nnTF 47, 47, 47, 63, 64,
 - 64, 148, 149, 484, 578, 1995, 2623,
 - 3926, 4693, 4756, 4784, 6274, 6277,
 - 6432, 6435, 7546, 9269, 9299, 9420,
 - 10754, 10810, 11184, 11257, 11352,
 - 11895, 11938, 13756, 13819, 13834,
 - 14290, 14757, 15324, 16852, 16925,
 - 23836, 23858, 23867, 26535, 26665,
 - 28972, 28991, 28995, 29426, 29742,
 - 30798, 31891, 32062, 32064, 32066
- \str_if_eq_p:NN 63, 4707

- \str_if_eq_p:nn 63,
4693, 9260, 9518, 9520, 13885, 32060
- \str_if_eq_x:nnTF 32061, 32063, 32065
- \str_if_eq_x_p:nn 32059
- \str_if_exist:NTF 63, 4663, 9267
- \str_if_exist_p:N 63, 4663
- \str_if_in:NnTF 63, 4715
- \str_if_in:nnTF 63, 3154, 4715, 22523
- \str_item:Nn 67, 4903
- \str_item:nn 67, 418, 422, 4903
- \str_item_ignore_spaces:nn
..... 67, 418, 4903
- \str_log:N 70, 5187
- \str_log:n 70, 5187
- \str_lower_case:n 32124
- \str_lowercase:n 69,
262, 5129, 32124, 32125, 32126, 32127
- \str_map_break: 65, 4790
- \str_map_break:n ... 65, 66, 3158, 4790
- \str_map_function:NN
..... 64, 64, 65, 65, 4790
- \str_map_function:nN
..... 64, 64, 415, 4790, 6492
- \str_map_inline:Nn .. 65, 65, 65, 4790
- \str_map_inline:nn
..... 65, 3152, 4790, 25497
- \str_map_variable:NNn 65, 4790
- \str_map_variable:nNn 65, 4790
- \str_new:N 60, 61, 4560, 5183, 5184,
5185, 5186, 11445, 11446, 13055,
13056, 13057, 13086, 13087, 13088,
14603, 14605, 14608, 14610, 14613
- str_overflow 6392
- \str_put_left:Nn 61, 4587
- \str_put_right:Nn 62, 4587
- \str_range:Nnn 68, 4964
- \str_range:nnn 68, 168, 422, 4964, 23469
- \str_range_ignore_spaces:nnn 68, 4964
- \str_remove_all:Nn 62, 62, 4657
- \str_remove_once:Nn 62, 4651
- \str_replace_all:Nnn . 62, 4611, 4658
- \str_replace_once:Nnn 62, 4611, 4652
- \str_set:Nn ... 61, 62, 4587, 4849,
11531, 11532, 11736, 11737, 11758,
11759, 13769, 13770, 13771, 14631,
14633, 15131, 15133, 15274, 15399
- \str_set_convert:Nnnn
..... 72, 74, 74, 74, 431, 441, 5360
- \str_set_convert:NnnnTF 74, 431, 5360
- \str_set_eq:NN 61, 4560
- \str_show:N 70, 5187
- \str_show:n 70, 5187
- \str_tail:N 67, 5114
- \str_tail:n 67, 957, 5114, 29074
- \str_tail_ignore_spaces:n .. 67, 5114
- \str_upper_case:n 32124
- \str_uppercase:n 69,
262, 5129, 32128, 32129, 32130, 32131
- \str_use:N 66, 4560
- \c_tilde_str 71, 5171
- \g_tmpa_str 71, 5183
- \l_tmpa_str 62, 71, 5183
- \g_tmpb_str 71, 5183
- \l_tmpb_str 71, 5183
- \c_underscore_str 71, 5171
- str internal commands:
- \g__str_alias_prop .. 434, 5204, 5431
- \c__str_byte_1_tl 5274
- \c__str_byte_0_tl 5274
- \c__str_byte_1_tl 5274
- \c__str_byte_255_tl 5274
- \c__str_byte_⟨number⟩_tl 429
- __str_case:nnTF 4729
- __str_case:nw 4729
- __str_case_e:nnTF 4729
- __str_case_e:nw 4729
- __str_case_end:nw 4729
- __str_change_case:nn 5129
- __str_change_case_aux:nn 5129
- __str_change_case_char:nN ... 5129
- __str_change_case_end:nw 5129
- __str_change_case_end:wn 5148, 5166
- __str_change_case_loop:nw ... 5129
- __str_change_case_output:nw ... 5129
- __str_change_case_result:n .. 5129
- __str_change_case_space:n ... 5129
- __str_collect_delimit_by_q_-
stop:w 4992, 5015
- __str_collect_end:nnnnnnnw ...
..... 421, 5015
- __str_collect_end:wn 5015
- __str_collect_loop:wn 5015
- __str_collect_loop:wnNNNNNNN . 5015
- __str_convert:nnn
..... 433, 433, 5403, 5404, 5418
- __str_convert:nnnn 434, 5418
- __str_convert:NNnNN 5400
- __str_convert:nNNnnn 5360
- __str_convert:wwwnn
..... 433, 5387, 5392, 5400
- __str_convert_decode: .. 5391, 5541
- __str_convert_decode_clist: . 5581
- __str_convert_decode_eight_-
bit:n 5602, 5608
- __str_convert_decode_utf16: . 6263
- __str_convert_decode_utf16be: 6263
- __str_convert_decode_utf16le: 6263
- __str_convert_decode_utf32: . 6421

- `__str_convert_decode_utf32be:` [6421](#)
- `__str_convert_decode_utf32le:` [6421](#)
- `__str_convert_decode_utf8:` [6082](#)
- `__str_convert_encode:` [5396](#), [5545](#)
- `__str_convert_encode_clist:` [5592](#)
- `__str_convert_encode_eight_-`
`bit:n` [5604](#), [5654](#)
- `__str_convert_encode_utf16:` [6178](#)
- `__str_convert_encode_utf16be:` [6178](#)
- `__str_convert_encode_utf16le:` [6178](#)
- `__str_convert_encode_utf32:` [6361](#)
- `__str_convert_encode_utf32be:` [6361](#)
- `__str_convert_encode_utf32le:` [6361](#)
- `__str_convert_encode_utf8:` [6009](#)
- `__str_convert_escape:` [5539](#)
- `__str_convert_escape_bytes:` [5539](#)
- `__str_convert_escape_hex:` [5929](#)
- `__str_convert_escape_name:` [448](#), [5933](#)
- `__str_convert_escape_string:` [5956](#)
- `__str_convert_escape_url:` [5988](#)
- `__str_convert_gmap:N` [5318](#),
[5542](#), [5621](#), [5930](#), [5936](#), [5959](#), [5989](#)
- `__str_convert_gmap_internal:N` [5334](#), [5552](#), [5560](#), [5594](#),
[5663](#), [6010](#), [6191](#), [6363](#), [6367](#), [6369](#)
- `__str_convert_gmap_internal_-`
`loop:Nw` [5334](#)
- `__str_convert_gmap_internal_-`
`loop:Nww` [5338](#), [5344](#), [5348](#)
- `__str_convert_gmap_loop:NN` [5318](#)
- `__str_convert_lowercase_-`
`alphanum:n` [5423](#), [5455](#)
- `__str_convert_lowercase_-`
`alphanum_loop:N` [5455](#)
- `__str_convert_pdfname:n` [6489](#)
- `__str_convert_pdfname_bytes:n` [6489](#)
- `__str_convert_pdfname_bytes_-`
`aux:n` [6489](#)
- `__str_convert_pdfname_bytes_-`
`aux:nnn` [6489](#)
- `__str_convert_pdfname_bytes_-`
`aux:nnnn` [6510](#), [6511](#)
- `__str_convert_unescape:` [5523](#)
- `__str_convert_unescape_bytes:` [5523](#)
- `__str_convert_unescape_hex:` [5745](#)
- `__str_convert_unescape_name:` [443](#), [5791](#)
- `__str_convert_unescape_string:` [5841](#)
- `__str_convert_unescape_url:` [5791](#)
- `__str_count:n` [422](#), [4919](#), [4979](#), [5061](#)
- `__str_count_aux:n` [5061](#)
- `__str_count_loop:NNNNNNNN` [5061](#)
- `__str_count_spaces_loop:w` [5041](#)
- `__str_decode_clist_char:n` [5581](#)
- `__str_decode_eight_bit_char:N` [5608](#)
- `__str_decode_eight_bit_load:nn` [5608](#)
- `__str_decode_eight_bit_load_-`
`missing:n` [5608](#)
- `__str_decode_native_char:N` [5541](#)
- `__str_decode_utf_viii_aux:wNnnwN` [6082](#)
- `__str_decode_utf_viii_continuation:wwN` [6082](#)
- `__str_decode_utf_viii_end:` [6082](#)
- `__str_decode_utf_viii_overflow:w` [6082](#)
- `__str_decode_utf_viii_start:N` [6082](#)
- `__str_decode_utf_xvi:Nw` [456](#), [6263](#)
- `__str_decode_utf_xvi_bom:NN` [6263](#)
- `__str_decode_utf_xvi_error:nnN` [6297](#)
- `__str_decode_utf_xvi_extra:NNw` [6297](#)
- `__str_decode_utf_xvi_pair:NN` [456](#), [457](#), [6291](#), [6297](#)
- `__str_decode_utf_xvi_pair_-`
`end:Nw` [6297](#)
- `__str_decode_utf_xvi_quad:NNwNN` [6297](#)
- `__str_decode_utf_xxxii:Nw` [460](#), [6421](#)
- `__str_decode_utf_xxxii_bom:NNNN` [6421](#)
- `__str_decode_utf_xxxii_end:w` [6421](#)
- `__str_decode_utf_xxxii_loop:NNNN` [6421](#)
- `__str_encode_clist_char:n` [5592](#)
- `__str_encode_eight_bit_char:n` [5654](#)
- `__str_encode_eight_bit_char_-`
`aux:n` [5654](#)
- `__str_encode_eight_bit_load:nn` [5654](#)
- `__str_encode_native_char:n` [5545](#)
- `__str_encode_utf_vii_loop:wwnnw` [449](#)
- `__str_encode_utf_viii_char:n` [6009](#)
- `__str_encode_utf_viii_loop:wwnnw` [6009](#)
- `__str_encode_utf_xvi_aux:N` [6178](#)
- `__str_encode_utf_xvi_be:nn` [454](#)
- `__str_encode_utf_xvi_char:n` [6178](#)
- `__str_encode_utf_xxxii_be:n` [6361](#)
- `__str_encode_utf_xxxii_be_-`
`aux:nn` [6361](#)
- `__str_encode_utf_xxxii_le:n` [6361](#)
- `__str_encode_utf_xxxii_le_-`
`aux:nn` [6361](#)
- `\l__str_end_flag` [6212](#)
- `\g__str_error_bool` [5220](#), [5357](#), [5367](#), [5371](#), [5376](#), [5380](#)
- `__str_escape:n` [4671](#)
- `__str_escape_hex_char:N` [5929](#)

- __str_escape_name_char:n [5933](#), [6502](#), [6525](#)
- \c__str_escape_name_not_str [447](#), [5933](#)
- \c__str_escape_name_str ... [447](#), [5933](#)
- __str_escape_string_char:N .. [5956](#)
- \c__str_escape_string_str [5956](#)
- __str_escape_url_char:n [5988](#)
- \l__str_extra_flag [6031](#), [6212](#)
- __str_filter_bytes:n [5499](#), [5533](#), [5811](#), [5873](#)
- __str_filter_bytes_aux:N [5499](#)
- __str_head:w [423](#), [5099](#)
- __str_hexadecimal_use:N [5255](#)
- __str_hexadecimal_use:NTF [443](#), [5255](#), [5765](#), [5775](#), [5814](#), [5816](#)
- __str_if_contains_char:Nn ... [5223](#)
- __str_if_contains_char:nn ... [5232](#)
- __str_if_contains_char:NnTF ... [5223](#), [5945](#), [5951](#), [5964](#)
- __str_if_contains_char:nnTF ... [427](#), [5223](#), [5998](#), [6004](#)
- __str_if_contains_char_aux:nn [5223](#)
- __str_if_contains_char_auxi:nN [5223](#)
- __str_if_contains_char_true: . [5223](#)
- __str_if_eq:nn [4671](#), [4696](#), [4704](#), [4710](#)
- __str_if_escape_name:n [5942](#)
- __str_if_escape_name:nTF [5933](#)
- __str_if_escape_string:N [5976](#)
- __str_if_escape_string:NTF .. [5956](#)
- __str_if_escape_url:n [5995](#)
- __str_if_escape_url:nTF [5988](#)
- __str_if_flag_error:nnn [431](#), [432](#), [5350](#), [5369](#), [5378](#), [5534](#), [5561](#), [5622](#), [5664](#), [5759](#), [5805](#), [5806](#), [5864](#), [5865](#), [6095](#), [6192](#), [6295](#), [6452](#)
- __str_if_flag_no_error:nnn [431](#), [5350](#), [5369](#), [5378](#)
- __str_if_flag_times:nTF [5358](#), [6040](#), [6041](#), [6042](#), [6043](#), [6227](#), [6228](#), [6229](#), [6399](#), [6400](#)
- __str_if_recursion_tail_break:NN [4558](#), [4830](#), [4848](#)
- __str_if_recursion_tail_stop_do:Nn [4558](#), [5165](#)
- \l__str_internal_int [5196](#), [5611](#), [5628](#), [5629](#), [5630](#), [5631](#), [5637](#), [5638](#), [5639](#), [5641](#), [5647](#), [5657](#), [5670](#), [5671](#), [5672](#), [5674](#), [5682](#)
- \l__str_internal_tl [434](#), [5196](#), [5275](#), [5276](#), [5278](#), [5431](#), [5432](#), [5433](#), [5435](#), [5439](#), [5443](#), [5450](#), [5600](#)
- __str_item:nn [418](#), [4903](#)
- __str_item:w [418](#), [4903](#)
- __str_load_catcodes: ... [5438](#), [5481](#)
- __str_map_function:Nn [415](#), [4790](#)
- __str_map_function:w [415](#), [4790](#)
- __str_map_inline:NN [4790](#)
- __str_map_variable:NnN [4790](#)
- \c__str_max_byte_int [5201](#), [5566](#), [5693](#)
- \l__str_missing_flag [6031](#), [6212](#)
- __str_octal_use:N [5247](#)
- __str_octal_use:NTF [428](#), [428](#), [5247](#), [5876](#), [5878](#), [5880](#)
- __str_output_byte:n [459](#), [5286](#), [5315](#), [5316](#), [5476](#), [5673](#), [5696](#), [6023](#), [6029](#), [6379](#), [6388](#)
- __str_output_byte:w [443](#), [5286](#), [5752](#), [5778](#), [5813](#), [5875](#)
- __str_output_byte_pair:nnN .. [5302](#)
- __str_output_byte_pair_be:n ... [5302](#), [6180](#), [6184](#), [6378](#)
- __str_output_byte_pair_le:n ... [5302](#), [6186](#), [6389](#)
- __str_output_end: [443](#), [5286](#), [5757](#), [5777](#), [5827](#), [5909](#), [5913](#)
- __str_output_hexadecimal:n [5286](#), [5932](#), [5940](#), [5993](#), [6513](#), [6514](#), [6517](#), [6520](#)
- \l__str_overflow_flag [6031](#)
- \l__str_overlong_flag [6031](#)
- __str_range:nnn [4964](#)
- __str_range:nnw [4964](#)
- __str_range:w [4964](#)
- __str_range_normalize:nn [4987](#), [4988](#), [4996](#)
- __str_replace:NNNnn [4611](#)
- __str_replace_aux:NNNnnn [4611](#)
- __str_replace_next:w [4611](#)
- \c__str_replacement_char_int ... [5200](#), [5640](#), [6107](#), [6131](#), [6145](#), [6165](#), [6172](#), [6202](#), [6354](#), [6463](#), [6468](#), [6484](#)
- \g__str_result_tl [426](#), [430](#), [430](#), [432](#), [436](#), [438](#), [443](#), [456](#), [459](#), [460](#), [5199](#), [5320](#), [5324](#), [5336](#), [5340](#), [5386](#), [5398](#), [5532](#), [5533](#), [5583](#), [5584](#), [5587](#), [5595](#), [5750](#), [5754](#), [5799](#), [5801](#), [5852](#), [5855](#), [5858](#), [5861](#), [6089](#), [6091](#), [6181](#), [6264](#), [6266](#), [6270](#), [6289](#), [6364](#), [6422](#), [6424](#), [6427](#), [6446](#)
- __str_skip_end:NNNNNNNN .. [419](#), [4943](#)
- __str_skip_end:w [4943](#)
- __str_skip_exp_end:w [419](#), [421](#), [4930](#), [4939](#), [4943](#), [4994](#)
- __str_skip_loop:wNNNNNNNN ... [4943](#)
- __str_tail_auxi:w [5114](#)
- __str_tail_auxii:w [424](#), [5114](#)
- __str_tmp:n [4561](#), [4567](#), [4570](#), [4588](#), [4598](#), [4601](#)

- _str_tmp:w [443](#), [454](#), [456](#),
[460](#), [5196](#), [5791](#), [5837](#), [5839](#), [5844](#),
[5869](#), [6190](#), [6197](#), [6202](#), [6204](#), [6207](#),
[6208](#), [6288](#), [6303](#), [6308](#), [6319](#), [6322](#),
[6328](#), [6329](#), [6445](#), [6460](#), [6465](#), [6471](#)
- _str_to_other_end:w [417](#), [4858](#)
- _str_to_other_fast_end:w ... [4881](#)
- _str_to_other_fast_loop:w
..... [4883](#), [4892](#), [4899](#)
- _str_to_other_loop:w [417](#), [4858](#)
- _str_unescape_hex_auxi:N ... [5745](#)
- _str_unescape_hex_auxii:N .. [5745](#)
- _str_unescape_name_loop:wNN . [5791](#)
- _str_unescape_string_loop:wNNN
..... [5841](#)
- _str_unescape_string_newlines:wN
..... [5841](#)
- _str_unescape_string_repeat:NNNNN
..... [5841](#)
- _str_unescape_url_loop:wNN . [5791](#)
- _str_use_i_delimit_by_s_-
stop:nw [423](#),
[4554](#), [4929](#), [4938](#), [5057](#), [5108](#), [5111](#)
- _str_use_none_delimit_by_s_-
stop:w
... [4554](#), [4645](#), [4927](#), [4936](#), [5095](#),
[5346](#), [5627](#), [5636](#), [5669](#), [6024](#), [6114](#)
- \strcmp [22](#)
- \string [455](#)
- \suppressfontnotfounderror [712](#)
- \suppressifcsnameerror [895](#)
- \suppresslongerror [896](#)
- \suppressmathparerror [897](#)
- \suppressoutererror [898](#)
- \suppressprimitiveerror [899](#)
- \synctex [697](#)
- sys commands:
- \c_sys_backend_str [119](#), [9264](#)
- \c_sys_day_int [116](#), [9415](#)
- \c_sys_engine_str ... [116](#), [9245](#), [31503](#)
- \c_sys_engine_version_str [271](#), [31501](#)
- \sys_everyjob: [9405](#), [9501](#)
- \sys_finalise: [119](#), [9266](#), [9499](#)
- \sys_get_shell:nnN [118](#), [9337](#)
- \sys_get_shell:nnN(TF) [267](#)
- \sys_get_shell:nnNTF [118](#), [9337](#), [9339](#)
- \sys_gset_rand_seed:n [117](#), [218](#), [9461](#)
- \c_sys_hour_int [116](#), [9415](#)
- \sys_if_engine luatex:TF [116](#), [257](#),
[2609](#), [9245](#), [9377](#), [9379](#), [9392](#), [9480](#),
[10329](#), [10331](#), [12473](#), [13226](#), [13239](#),
[13416](#), [13478](#), [13500](#), [13569](#), [13628](#),
[13684](#), [13864](#), [16255](#), [22277](#), [22315](#),
[22354](#), [22367](#), [22393](#), [22461](#), [22506](#),
[28667](#), [28870](#), [32012](#), [32014](#), [32016](#)
- \sys_if_engine luatex_p:
.... [116](#), [5501](#), [5525](#), [5547](#), [5714](#),
[6495](#), [9245](#), [12620](#), [13318](#), [13352](#),
[13398](#), [13534](#), [22556](#), [28861](#), [29754](#),
[29768](#), [29804](#), [29844](#), [29892](#), [29928](#),
[30022](#), [30029](#), [30107](#), [30185](#), [30263](#),
[30286](#), [30320](#), [30888](#), [30973](#), [32010](#)
- \sys_if_engine pdftex:TF
.... [116](#), [9245](#), [32026](#), [32028](#), [32030](#)
- \sys_if_engine pdftex_p:
..... [116](#), [9245](#), [32024](#)
- \sys_if_engine ptex:TF [116](#), [9245](#)
- \sys_if_engine ptex_p:
..... [116](#), [9245](#), [23105](#)
- \sys_if_engine uptex:TF ... [116](#), [9245](#)
- \sys_if_engine uptex_p:
..... [116](#), [9245](#), [23106](#)
- \sys_if_engine xetex:TF
..... [5](#), [116](#), [2608](#), [9245](#),
[9283](#), [9527](#), [10330](#), [32084](#), [32086](#), [32088](#)
- \sys_if_engine xetex_p:
[116](#), [5502](#), [5526](#), [5548](#), [5715](#), [6496](#),
[9245](#), [9426](#), [22556](#), [28861](#), [29755](#),
[29769](#), [29805](#), [29845](#), [29893](#), [29929](#),
[30023](#), [30030](#), [30108](#), [30186](#), [30264](#),
[30287](#), [30321](#), [30889](#), [30974](#), [32082](#)
- \sys_if_output_dvi:TF [117](#), [9508](#)
- \sys_if_output_dvi_p: [117](#), [9508](#)
- \sys_if_output_pdf:TF
..... [117](#), [9297](#), [9508](#), [9530](#)
- \sys_if_output_pdf_p: [117](#), [9508](#)
- \sys_if_platform_unix:TF
..... [117](#), [9264](#), [13882](#)
- \sys_if_platform_unix_p:
..... [117](#), [9264](#), [13882](#)
- \sys_if_platform_windows:TF
..... [117](#), [9264](#), [13882](#)
- \sys_if_platform_windows_p:
..... [117](#), [9264](#), [13882](#)
- \sys_if_rand_exist:TF . [271](#), [535](#),
[9262](#), [9449](#), [9463](#), [15815](#), [21805](#), [21829](#)
- \sys_if_rand_exist_p: [271](#), [9262](#)
- \sys_if_shell: [118](#)
- \sys_if_shell:TF [118](#), [9344](#), [9488](#), [31557](#)
- \sys_if_shell_p: [118](#), [9488](#)
- \sys_if_shell_restricted:TF [118](#), [9488](#)
- \sys_if_shell_restricted_p: [118](#), [9488](#)
- \sys_if_shell_unrestricted:TF ...
..... [118](#), [9488](#)
- \sys_if_shell_unrestricted_p: ...
..... [118](#), [9488](#)

- \c_sys_jobname_str
..... 116, 166, 542, 9413, 31914
- \sys_load_backend:n .. 119, 119, 9264
- \sys_load_debug: 119, 9323
- \sys_load_deprecation: 119, 9323
- \c_sys_minute_int 116, 9415
- \c_sys_month_int 116, 9415
- \c_sys_output_str 117, 9508
- \c_sys_platform_str
..... 117, 9264, 13864, 13885
- \sys_rand_seed: ... 81, 117, 218, 9447
- \c_sys_shell_escape_int
..... 118, 9476, 9491, 9493, 9495
- \sys_shell_now:n 118, 9379
- \sys_shell_shipout:n 118, 9392
- \c_sys_year_int 116, 9415
- sys internal commands:
 - \g__sys_backend_tl
..... 9274, 9275, 9276, 9522
 - __sys_const:nn 9229, 9259,
9262, 9490, 9492, 9494, 9517, 9519
 - \g__sys_debug_bool .. 9321, 9325, 9327
 - \g__sys_deprecation_bool
..... 1192, 9321, 9331, 9333
 - __sys_everyjob:n 9405, 9413,
9415, 9447, 9461, 9476, 9488, 9497
 - \g__sys_everyjob_tl 9405
 - __sys_finalise:n
..... 9499, 9508, 9523, 9536
 - \g__sys_finalise_tl 9499
 - __sys_get:nnN 9337
 - __sys_get_do:Nw 9337
 - \l_sys_internal_tl 9335
 - __sys_load_backend_check:N .. 9264
 - \c_sys_marker_tl ... 9336, 9360, 9372
 - \c_sys_shell_stream_int
..... 9377, 9389, 9402
 - __sys_tmp:w
... 9418, 9439, 9441, 9442, 9443, 9444
- syst commands:
 - \c_syst_catcodes_n 22512, 22516
 - \c_syst_last_allocated_toks .. 22712
- T**
- \t 29167, 31013
- \tabskip 456
- \tagcode 698
- \tan 216
- \tand 216
- \tate 1155
- \tbaselineshift 1156
- TEX and L^AT_EX 2_ε commands:
 - \@ 5172
 - \@@@hyph 299
 - \@@end 1185, 1186
 - \@hyph 1189, 1192
 - \@input 1187
 - \@italiccorr 1193
 - \@shipout 1195, 1196
 - \@tracingfonts 300, 1231
 - \@underline 1194
 - \@addtofilelist 13680
 - \@changed@cmd 29457
 - \@classoptionslist .. 9538, 9540, 9542
 - \@current@cmd 29454
 - \@currnamestack
..... 643, 13077, 13079, 13080
 - \@expl@finalise@setup@@ 22547, 22548
 - \@filelist 170, 644, 658, 660,
661, 13679, 13779, 13782, 13791, 13796
 - \@firstofone 19
 - \@firstoftwo 19, 314
 - \@gobbbletwo 20
 - \@gobble 20
 - \@secondoftwo 19, 314
 - \@tempa 104, 106, 1203, 1217, 1220
 - \@tfor 299, 1203
 - \@uclclist 1156, 30625
 - \@unexpandable@protect 761
 - \@unusedoptionlist 9557
 - \AtBeginDocument 299
 - \botmark 579
 - \box 247
 - \catcodetable 927, 931, 933
 - \char 143
 - \chardef 138, 138, 501, 524, 1123
 - \color 1107
 - \conditionally@traceoff
..... 636, 11838, 12798
 - \conditionally@tracelon 11856
 - \copy 240
 - \count 143, 438, 939
 - \cr 534
 - \CROP@shipout 1204
 - \csname 17
 - \csstring 323
 - \currentgrouplevel ... 335, 930, 1173
 - \currentgrouptype 335, 1173
 - \def 143
 - \detokenize 51
 - \dimen 578
 - \dimendef 578
 - \directlua 257
 - \dp 241, 762, 763
 - \dup@shipout 1205
 - \e@alloc@ccodetable@count 22510
 - \e@alloc@top 940, 22698
 - \edef 1, 4, 377

- `\end` 299, 606
- `\endcsname` 17
- `\endinput` 155
- `\endlinechar` 46, 46,
161, 381, 382, 579, 927, 928, 928, 930
- `\endtemplate` 115, 534
- `\errhelp` 602, 602
- `\errmessage` 602, 602, 603
- `\errorcontextlines` 305, 408, 603, 1065
- `\escapechar` 51, 323, 335, 635, 969
- `\everyeof` 383
- `\everyjob` 539
- `\everypar` 24, 337, 355
- `\expandafter` 33, 35
- `\expanded` 4, 20, 28,
30, 339, 342, 348, 350, 355, 362, 381
- `\fi` 142
- `\firstmark` 356, 579
- `\font` 142
- `\fontdimen`
... 200, 238, 714, 715, 716, 716, 717
- `\frozen@everydisplay` 1190
- `\frozen@everymath` 1191
- `\futurelet`
... 534, 582, 584, 953, 955, 957, 957
- `\global` 279
- `\GPTorg@shipout` 1206
- `\halign` 115, 337, 534, 568
- `\hskip` 182
- `\ht` 241, 762, 763
- `\hyphen` 579
- `\hyphenchar` 714
- `\ifcase` 102
- `\ifdim` 185
- `\ifeof` 166
- `\iffalse` 108
- `\ifhbox` 250
- `\ifnum` 102
- `\ifodd` 103, 586
- `\iftrue` 108
- `\ifvbox` 250
- `\ifvoid` 250
- `\ifx` 23, 277
- `\indent` 337
- `\infty` 211
- `\input` 299
- `\input@path`
167, 648, 13260, 13262, 13361, 13363
- `\italiccorr` 579
- `\jobname` 116, 539
- `\lastnamedcs` 326
- `\lccode` 520, 956, 960, 1118
- `\leavevmode` 24
- `\let` 279
- `\letcharcode` 564
- `\LL@shipout` 1207
- `\loctoks` 939, 940
- `\long` 3, 144, 350
- `\lower` 1171
- `\lowercase` 1042, 1043, 1043
- `\luaescapestring` 257
- `\makeatletter` 7
- `\MakeUppercase` 1154
- `\mathchar` 143
- `\mathchardef` 138, 501, 1123
- `\meaning` 15,
135, 143, 144, 577, 578, 584, 586, 955
- `\mem@oldshipout` 1208
- `\message` 28
- `\newcatcodetable` 927
- `\newif` 108, 264
- `\newlinechar` 46,
46, 305, 326, 381, 382, 408, 603, 633
- `\newread` 624
- `\newtoks` 225, 951, 968
- `\newwrite` 631
- `\noexpand` 34, 142, 349, 350, 350, 351, 352
- `\nullfont` 579, 580
- `\number` 102, 815
- `\numexpr` 304, 353
- `\opem@shipout` 1209
- `\or` 102
- `\outer` 144,
277, 586, 624, 631, 1186, 1186, 1188
- `\parindent` 24
- `\pdfescapehex` 442
- `\pdfescapename` 73, 442
- `\pdfescapestring` 73, 442
- `\pdffilesize` 648
- `\pdfmapfile` 301
- `\pdfmapline` 301
- `\pdfstrcmp` *xiii*, 275, 276, 277, 290, 1117
- `\pdfuniformdeviate` 218
- `\pgfpages@originalshipout` 1210
- `\pi` 211
- `\pr@shipout` 1211
- `\primitive` 299, 349, 350, 350, 540
- `\protect` 636, 760, 761, 1160
- `\protected` 144, 350
- `\ProvidesClass` 7
- `\ProvidesFile` 7
- `\ProvidesPackage` 7
- `\quitvmode` 337
- `\read` 161, 629
- `\readline` 161, 629
- `\relax` 22, 142, 277, 319,
324, 335, 521, 521, 720, 723, 746, 777
- `\RequirePackage` 7, 277, 643

- `\reserveinserts` 277
- `\romannumeral` 36, 720
- `\savecatcodetable` 930
- `\scantokens` 74, 381, 647
- `\shipout` 299
- `\show` 16, 58, 335
- `\showbox` 1064
- `\showthe` 335, 519, 673, 677, 680
- `\showtokens` 58, 408, 608
- `\sin` 211
- `\skip` 960, 961
- `\space` 579
- `\splitbotmark` 579
- `\splitfirstmark` 579
- `\SS` 1163
- `\strcmp` 275, 290
- `\string` 135, 955, 957, 958
- `\tenrm` 142
- `\tex_lowercase:D` 567
- `\tex_unexpanded:D` 347
- `\the` 93, 142, 177, 181, 184, 340, 349, 350, 350, 353
- `\toks` *xxii*, 81, 102, 225, 353, 354, 355, 486, 938, 939, 939, 939, 941, 941, 942, 943, 943, 945, 945, 946, 951, 955, 956, 958, 960, 961, 963, 968, 969, 969, 970, 970, 970, 971, 975, 1014, 1015, 1020, 1021, 1025, 1025, 1026, 1026, 1027, 1030, 1032, 1035, 1042, 1060
- `\toks@` 355
- `\toksdef` 951
- `\topmark` 143, 579
- `\tracingfonts` 300
- `\tracingnesting` 381, 647
- `\tracingonline` 1065
- `\typeout` 636
- `\uccode` 1118
- `\Ucharcat` 567
- `\unexpanded` 34, 52, 52, 52, 56, 56, 57, 78, 79, 84, 85, 123, 126, 127, 128, 128, 147, 269, 272, 349, 350, 350, 352, 377, 400, 400, 505
- `\unhbox` 247
- `\unhcopy` 244
- `\uniformdeviate` 218
- `\unless` 23
- `\unvbox` 247
- `\unvcopy` 246
- `\uppercase` 1042
- `\usepackage` 643
- `\valign` 534
- `\verso@orig@shipout` 1213
- `\vskip` 183
- `\vtop` 1083
- `\wd` 241, 762, 763
- `\write` 164, 633
- tex commands:
 - `\tex_above:D` 189
 - `\tex_abovedisplayshortskip:D` .. 190
 - `\tex_abovedisplayskip:D` 191
 - `\tex_abovewithdelims:D` 192
 - `\tex_accent:D` 193
 - `\tex_adjdemerits:D` 194
 - `\tex_adjustspacing:D` 649, 907
 - `\tex_advance:D` .. 195, 8206, 8208, 8210, 8212, 8218, 8220, 8222, 8224, 13936, 13939, 13945, 13948, 14277, 14279, 14283, 14285, 14371, 14373, 14377, 14379, 22804, 22811, 22814, 23214, 23216, 23249, 23251, 24448
 - `\tex_afterassignment:D` 196, 10920, 23155, 23198
 - `\tex_aftergroup:D` 197, 933, 1394
 - `\tex_alignmark:D` 782, 1236
 - `\tex_aligntab:D` 783, 1237
 - `\tex_atop:D` 198
 - `\tex_atopwithdelims:D` 199
 - `\tex_attribute:D` 784, 1238
 - `\tex_attributedef:D` 785, 1239
 - `\tex_automaticdiscretionary:D` .. 787
 - `\tex_automatichyphenmode:D` 788
 - `\tex_automatichyphenpenalty:D` .. 790
 - `\tex_autospacing:D` 1106
 - `\tex_autoxspacing:D` 1107
 - `\tex_badness:D` 200
 - `\tex_baselineskip:D` 201
 - `\tex_batchmode:D` 202, 11716
 - `\tex_begincsname:D` 791
 - `\tex_begingroup:D` 203, 1198, 1300, 1389
 - `\tex_beginL:D` 511
 - `\tex_beginR:D` 512
 - `\tex_belowdisplayshortskip:D` .. 204
 - `\tex_belowdisplayskip:D` 205
 - `\tex_binoppenalty:D` 206
 - `\tex_bodydir:D` 792, 1275, 1349
 - `\tex_bodydirection:D` 793
 - `\tex_botmark:D` 207
 - `\tex_botmarks:D` 513
 - `\tex_box:D` 208, 26731, 26733, 26773, 32110, 32113
 - `\tex_boxdir:D` 794, 1276
 - `\tex_boxdirection:D` 795
 - `\tex_boxmaxdepth:D` 209
 - `\tex_breakafterdirmode:D` 796
 - `\tex_brokenpenalty:D` 210
 - `\tex_catcode:D` 211, 2505, 10168, 10170, 28877, 28884

- `\tex_catcodetable:D` 797, 1240, 22371, 22378
- `\tex_char:D` 212
- `\tex_chardef:D` 213, 312, 1382, 1411, 1413, 1710, 1711, 8181, 8938, 8960, 8965, 10841, 12470, 12668, 29193, 29195, 29197
- `\tex_cleaders:D` 214
- `\tex_clearmarks:D` 798, 1241
- `\tex_closein:D` 215, 12481
- `\tex_closeout:D` 216, 12678
- `\tex_clubpenalties:D` 514
- `\tex_clubpenalty:D` 217
- `\tex_copy:D` 218, 26725, 26727, 26748, 26757, 26766, 26774
- `\tex_copyfont:D` 650, 908
- `\tex_count:D` 219, 12418, 12420, 12632, 12634, 22681, 22697, 22705, 22706
- `\tex_countdef:D` 220
- `\tex_cr:D` 221
- `\tex_crampeddisplaystyle:D` 799, 1242
- `\tex_crampedscriptscriptstyle:D` 801, 1243
- `\tex_crampedscriptstyle:D` . 802, 1245
- `\tex_crampedtextstyle:D` ... 803, 1246
- `\tex_crcr:D` 222
- `\tex_creationdate:D` 772
- `\tex_csname:D` 223, 1376
- `\tex_csstring:D` 804
- `\tex_currentcjktoken:D` ... 1108, 1165
- `\tex_currentgrouplevel:D` 515, 932, 22360, 22440, 22449, 22456
- `\tex_currentgroupstype:D` 516
- `\tex_currentifbranch:D` 517
- `\tex_currentiflevel:D` 518
- `\tex_currentifttype:D` 519
- `\tex_currentspacingmode:D` 1109
- `\tex_currentxspacingmode:D` ... 1110
- `\tex_day:D` 224, 1307, 1311
- `\tex_deadcycles:D` 225
- `\tex_def:D` 226, 701, 702, 703, 1395, 1397, 1399, 1400, 1421, 1423, 1424, 1425, 1427, 1428, 1429, 1431, 1432, 1433
- `\tex_defaultthyphenchar:D` 227
- `\tex_defaultskewchar:D` 228
- `\tex_delcode:D` 229
- `\tex_delimiter:D` 230
- `\tex_delimiterfactor:D` 231
- `\tex_delimitershortfall:D` 232
- `\tex_detokenize:D` 520, 1385, 1387, 28865
- `\tex_dimen:D` 233, 5628, 5637, 5647, 5648, 5649, 5670, 5682, 5683, 5684
- `\tex_dimendef:D` 234
- `\tex_dimexpr:D` 521, 13891, 26703
- `\tex_directlua:D` . 805, 1229, 1230, 4675, 9482, 13867, 16259, 28656, 28872
- `\tex_disablecjktoken:D` 1166
- `\tex_discretionary:D` 235
- `\tex_disinhibitglue:D` 1111
- `\tex_displayindent:D` 236
- `\tex_displaylimits:D` 237
- `\tex_displaystyle:D` 238
- `\tex_displaywidowpenalties:D` .. 522
- `\tex_displaywidowpenalty:D` 239
- `\tex_displaywidth:D` 240
- `\tex_divide:D` 241, 22675, 24449
- `\tex_doublehyphenemerits:D` ... 242
- `\tex_dp:D` 243, 26741
- `\tex_draftmode:D` 651, 909
- `\tex_dtou:D` 1112
- `\tex_dump:D` 244
- `\tex_dviextension:D` 806
- `\tex_dvifeedback:D` 807
- `\tex_dvivariable:D` 808
- `\tex_eachlinedepth:D` 652
- `\tex_eachlineheight:D` 653
- `\tex_edef:D` 245, 1199, 1200, 1216, 1301, 1302, 1307, 1308, 1313, 1314, 1319, 1320, 1422, 1426, 1430, 1434, 12890, 12948, 31877
- `\tex_efcode:D` 689
- `\tex_elapsedtime:D` 654, 773
- `\tex_else:D` 246, 1202, 1228, 1304, 1310, 1316, 1322, 1362, 1414, 1417, 1459
- `\tex_emergencystretch:D` 247
- `\tex_enablecjktoken:D` ... 1167, 9251
- `\tex_end:D` 248, 1186, 1332, 1821
- `\tex_endcsname:D` 249, 1377
- `\tex_endgroup:D` 250, 1184, 1224, 1325, 1390
- `\tex_endinput:D` ... 251, 11725, 13665
- `\tex_endL:D` 523
- `\tex_endlinechar:D` 158, 159, 173, 252, 3661, 3662, 3663, 3697, 5497, 12537, 12539, 12540, 22322, 22326, 22339, 22373, 22374, 22389, 22538, 22553, 22586
- `\tex_endR:D` 524
- `\tex_epTeXinputencoding:D` 1113
- `\tex_epTeXversion:D` 1114, 31521, 31544
- `\tex_eqno:D` 253
- `\tex_errhelp:D` 254, 11584
- `\tex_errmessage:D` ... 255, 1813, 11604

- `\tex_errorcontextlines:D`
256, 4532, 11599, 11618, 11818, 26837
- `\tex_errorstopmode:D` 257
- `\tex_escapechar:D` 258, 2124, 5749,
5798, 5851, 12559, 12752, 12799,
12805, 23120, 23182, 23183, 23493
- `\tex_eTeXglueshrinkorder:D` 809
- `\tex_eTeXgluestretchorder:D` ... 810
- `\tex_eTeXrevision:D` 525
- `\tex_eTeXversion:D` 526
- `\tex_etoksapp:D` 811
- `\tex_etokspre:D` 812
- `\tex_euc:D` 1115
- `\tex_everycr:D` 259
- `\tex_everydisplay:D` 260, 1190
- `\tex_everyeof:D`
..... 527, 3670, 3722, 9360, 13219
- `\tex_everyhbox:D` 261
- `\tex_everyjob:D` 262, 1333
- `\tex_everymath:D` 263, 1191
- `\tex_everypar:D` 264
- `\tex_everyvbox:D` 265
- `\tex_exceptionpenalty:D` 813
- `\tex_exhyphenpenalty:D` 266
- `\tex_expandafter:D` 267, 706,
1203, 1217, 1219, 1220, 1378, 28865
- `\tex_expanded:D`
..... 362, 815, 1342, 1458, 1459,
2195, 2198, 2265, 2268, 2301, 2307,
2391, 2394, 2415, 2418, 2483, 2486,
2514, 2991, 3031, 3039, 10347, 12270
- `\tex_explicitdiscretionary:D` .. 816
- `\tex_explicithyphenpenalty:D` .. 814
- `\tex_fam:D` 268
- `\tex_fi:D` 269,
707, 1188, 1197, 1221, 1223, 1232,
1233, 1234, 1292, 1294, 1295, 1299,
1306, 1312, 1318, 1324, 1330, 1343,
1351, 1356, 1363, 1419, 1420, 1461
- `\tex_filedump:D` 655, 774, 13473, 13498
- `\tex_filemoddate:D`
..... 656, 775, 13636, 13637
- `\tex_filesize:D`
..... 650, 651, 657, 776, 13238,
13317, 13351, 13397, 13498, 13533
- `\tex_finalhyphenemerits:D` 270
- `\tex_firstlineheight:D` 658
- `\tex_firstmark:D` 271
- `\tex_firstmarks:D` 528
- `\tex_firstvalidlanguage:D` 817
- `\tex_floatingpenalty:D` 272
- `\tex_font:D` 273, 15653
- `\tex_fontchardp:D` 529
- `\tex_fontcharht:D` 530
- `\tex_fontcharic:D` 531
- `\tex_fontcharwd:D` 532
- `\tex_fontdimen:D` 274, 15642
- `\tex_fontexpand:D` 659, 910
- `\tex_fontid:D` 818, 1247
- `\tex_fontname:D` 275
- `\tex_fontsize:D` 660
- `\tex_forcecjktoken:D` 1168
- `\tex_formatname:D` 819, 1248
- `\tex_futurelet:D`
..... 276, 10915, 10917, 23128, 23186
- `\tex_gdef:D` 277, 1435, 1438, 1442, 1446
- `\tex_gleaders:D` 825, 1249
- `\tex_global:D` 179, 183, 278, 666, 708,
1219, 1305, 1311, 1317, 1323, 1890,
1897, 8159, 8165, 8169, 8188, 8199,
8210, 8212, 8222, 8224, 8232, 8938,
8965, 10647, 10649, 10659, 10917,
12470, 12668, 13905, 13910, 13926,
13933, 13939, 13948, 14249, 14253,
14269, 14274, 14279, 14285, 14341,
14347, 14363, 14368, 14373, 14379,
15653, 26727, 26733, 26803, 26856,
26868, 26881, 26901, 26946, 26958,
26970, 26983, 27004, 27019, 32113
- `\tex_globaldefs:D` 279
- `\tex_glueexpr:D` 533, 14267,
14269, 14277, 14279, 14283, 14285,
14299, 14306, 14312, 14315, 21739
- `\tex_glueshrink:D` 534
- `\tex_glueshrinkorder:D` 535
- `\tex_gluestretch:D` . 536, 23340, 23346
- `\tex_gluestretchorder:D` 537
- `\tex_gluetomu:D` 538
- `\tex_halign:D` 280
- `\tex_hangafter:D` 281
- `\tex_hangindent:D` 282
- `\tex_hbadness:D` 283
- `\tex_hbox:D` 284, 26848, 26851,
26856, 26863, 26868, 26875, 26881,
26895, 26901, 26909, 26914, 28358
- `\tex_hfi:D` 1116
- `\tex_hfil:D` 285
- `\tex_hfill:D` 286
- `\tex_hfilneg:D` 287
- `\tex_hfuzz:D` 288
- `\tex_hjcode:D` 820
- `\tex_hoffset:D` 289, 1345
- `\tex_holdinginserts:D` 290
- `\tex_hpack:D` 821
- `\tex_hrule:D` 291
- `\tex_hsize:D`
..... 292, 27509, 27531, 27533, 27582
- `\tex_hskip:D` 293, 14310

- `\tex_hss:D` 294, 26918, 26920, 27363, 27372
- `\tex_ht:D` 295, 26740
- `\tex_hyphen:D` 188, 1192
- `\tex_hyphenation:D` 296
- `\tex_hyphenationbounds:D` 822
- `\tex_hyphenationmin:D` 823
- `\tex_hyphenchar:D` 297, 15643
- `\tex_hyphenpenalty:D` 298
- `\tex_hyphenpenaltymode:D` 824
- `\tex_if:D` 130, 299, 1365, 1366
- `\tex_ifabsdim:D` 646, 911
- `\tex_ifabsnum:D` 647, 912, 15712, 15716
- `\tex_ifcase:D` 300, 8052
- `\tex_ifcat:D` 301, 1367
- `\tex_ifcondition:D` 826
- `\tex_ifcsname:D` 539, 1375
- `\tex_ifdbbox:D` 1117
- `\tex_ifddir:D` 1118
- `\tex_ifdefined:D` 540, 705, 1185, 1189, 1195, 1226, 1229, 1235, 1294, 1295, 1326, 1331, 1344, 1352, 1374, 1412, 1415, 1459
- `\tex_ifdim:D` 302, 13890
- `\tex_ifeof:D` 303, 12501
- `\tex_iffalse:D` 304, 1360
- `\tex_iffontchar:D` 541
- `\tex_ifhbox:D` 305, 26785
- `\tex_ifhmode:D` 306, 1371
- `\tex_ifincsname:D` 690
- `\tex_ifinner:D` 307, 1373
- `\tex_ifjfont:D` 1119
- `\tex_ifmbox:D` 1120
- `\tex_ifmdir:D` 1121
- `\tex_ifmmode:D` 308, 1370
- `\tex_ifnum:D` 309, 1293, 1392
- `\tex_ifodd:D` ... 310, 1369, 8051, 8931
- `\tex_ifprimitive:D` 648, 778
- `\tex_iftbox:D` 1122
- `\tex_iftdir:D` 1124
- `\tex_iftfont:D` 1123
- `\tex_iftrue:D` 311, 1359
- `\tex_ifvbox:D` 312, 26786
- `\tex_ifvmode:D` 313, 1372
- `\tex_ifvoid:D` 314, 26787
- `\tex_ifx:D` 315, 1201, 1218, 1303, 1309, 1315, 1321, 1368
- `\tex_ifybox:D` 1125
- `\tex_ifydir:D` 1126
- `\tex_ignoredimen:D` 661
- `\tex_ignoreligaturesinfont:D` .. 913
- `\tex_ignorespaces:D` 316
- `\tex_immediate:D` 317, 1830, 1832, 12670, 12678, 12719
- `\tex_immediateassigned:D` 827
- `\tex_immediateassignment:D` 828
- `\tex_indent:D` 318, 2179
- `\tex_inhibitglue:D` 1127
- `\tex_inhibitxspcode:D` 1128
- `\tex_initcatcodetable:D` 829, 1250, 22287
- `\tex_input:D` 319, 1187, 1334, 9365, 13225, 13683
- `\tex_inputlineno:D` .. 320, 1828, 11522
- `\tex_insert:D` 321
- `\tex_insertht:D` 662, 914
- `\tex_insertpenalties:D` 322
- `\tex_interactionmode:D` 542, 26821, 26824, 26826
- `\tex_interlinepenalties:D` 543
- `\tex_interlinepenalty:D` 323
- `\tex_italiccorrection:D` 187, 1193, 1346
- `\tex_jcharwidowpenalty:D` 1129
- `\tex_jfam:D` 1130
- `\tex_jfont:D` 1131
- `\tex_jis:D` 1132
- `\tex_jobname:D` 324, 9414, 9498, 13068, 13069
- `\tex_kanjiskip:D` 1133, 9249
- `\tex_kansuji:D` 1134
- `\tex_kansujichar:D` 1135
- `\tex_kcatcode:D` 1136
- `\tex_kchar:D` 1169
- `\tex_kchardef:D` 1170
- `\tex_kern:D` 325, 27071, 27361, 27370, 27932, 28192, 28197, 28279, 28280, 28576, 28577, 31265, 31267, 31316, 31318
- `\tex_kuten:D` 1137, 1171
- `\tex_language:D` 326, 1335
- `\tex_lastbox:D` 327, 26801, 26803
- `\tex_lastkern:D` 328
- `\tex_lastlinedepth:D` 663
- `\tex_lastlinefit:D` 544
- `\tex_lastnamedcs:D` 830
- `\tex_lastnodechar:D` 1138
- `\tex_lastnodesubtype:D` 1139
- `\tex_lastnodetype:D` 545
- `\tex_lastpenalty:D` 329
- `\tex_lastskip:D` 330
- `\tex_lastxpos:D` 664, 921
- `\tex_lastypos:D` 665, 922
- `\tex_latelua:D` 831, 1251, 28657
- `\tex_lateluafunction:D` 832
- `\tex_lccode:D` 331, 3563, 3564, 3565, 4864, 4865, 4887, 4888,

- 10244, 10246, 23101, 23111, 23180,
 23182, 23185, 23215, 25999, 26051
 \tex_leaders:D 332
 \tex_left:D 333, 1353
 \tex_leftghost:D 833, 1277
 \tex_lefthyphenmin:D 334
 \tex_leftmarginkern:D 691
 \tex_leftskip:D 335
 \tex_leqno:D 336
 \tex_let:D 180, 183,
 337, 708, 1186, 1186, 1187, 1190,
 1191, 1192, 1193, 1194, 1196, 1219,
 1225, 1227, 1231, 1236, 1237, 1238,
 1239, 1240, 1241, 1242, 1243, 1245,
 1246, 1247, 1248, 1249, 1250, 1251,
 1252, 1253, 1254, 1255, 1256, 1257,
 1258, 1259, 1260, 1261, 1262, 1263,
 1264, 1265, 1266, 1268, 1269, 1271,
 1272, 1273, 1275, 1276, 1277, 1278,
 1279, 1281, 1282, 1283, 1284, 1285,
 1286, 1287, 1288, 1289, 1290, 1291,
 1297, 1298, 1305, 1311, 1317, 1323,
 1327, 1328, 1329, 1332, 1333, 1334,
 1335, 1336, 1337, 1338, 1339, 1340,
 1341, 1342, 1345, 1346, 1347, 1348,
 1349, 1350, 1353, 1354, 1355, 1359,
 1360, 1361, 1362, 1363, 1364, 1365,
 1366, 1367, 1368, 1369, 1370, 1371,
 1372, 1373, 1374, 1375, 1376, 1377,
 1378, 1379, 1380, 1381, 1383, 1384,
 1385, 1386, 1387, 1388, 1389, 1390,
 1392, 1393, 1394, 1410, 1421, 1422,
 1435, 1436, 1886, 10647, 10649,
 10659, 23102, 23112, 31815, 31818
 \tex_letcharcode:D 834
 \tex_letterspacefont:D 692
 \tex_limits:D 338
 \tex_linedir:D 835
 \tex_linedirection:D 836
 \tex_linepenalty:D 339
 \tex_lineskip:D 340
 \tex_lineskiplimit:D 341
 \tex_localbrokenpenalty:D . 837, 1278
 \tex_localinterlinepenalty:D ...
 838, 1279
 \tex_localleftbox:D 843, 1281
 \tex_localrightbox:D 844, 1282
 \tex_long:D 342, 701, 702,
 703, 1395, 1397, 1400, 1423, 1424,
 1425, 1426, 1427, 1429, 1431, 1432,
 1433, 1434, 1438, 1440, 1446, 1448
 \tex_looseness:D 343
 \tex_lower:D 344, 26784
 \tex_lowercase:D
 345, 1183, 3566, 4866, 4889,
 10275, 10393, 11591, 23102, 23112,
 23181, 26000, 26052, 31714, 32076
 \tex_lpcode:D 693
 \tex_luabytecode:D 839
 \tex_luabytecodecall:D 840
 \tex_luacopyinputnodes:D 841
 \tex_luaedef:D 842
 \tex_luaescapestring:D 412,
 845, 1252, 4674, 16263, 16264, 28655
 \tex_luafunction:D 846, 1253
 \tex_luafunctioncall:D 847
 \tex_luatexbanner:D 848
 \tex_luatexrevision:D ... 849, 31528
 \tex_luatexversion:D
 850, 1295, 1326, 1412, 4672,
 8176, 8855, 9247, 10636, 12621, 31526
 \tex_mag:D 346
 \tex_mapfile:D 666, 1297
 \tex_mapline:D 667, 1298
 \tex_mark:D 347
 \tex_marks:D 546
 \tex_mathaccent:D 348
 \tex_mathbin:D 349
 \tex_mathchar:D 350
 \tex_mathchardef:D 312, 351,
 1418, 8184, 8185, 29194, 29196, 29198
 \tex_mathchoice:D 352
 \tex_mathclose:D 353
 \tex_mathcode:D ... 354, 10238, 10240
 \tex_mathdelimitersmode:D 851
 \tex_mathdir:D 852, 1283
 \tex_mathdirection:D 853
 \tex_mathdisplayskipmode:D 854
 \tex_matheqnogapstep:D 855
 \tex_mathinner:D 355
 \tex_mathnolimitsmode:D 856
 \tex_mathop:D 356, 1336
 \tex_mathopen:D 357
 \tex_mathoption:D 857
 \tex_mathord:D 358
 \tex_mathpenaltiesmode:D 858
 \tex_mathpunct:D 359
 \tex_mathrel:D 360
 \tex_mathrulesfam:D 859
 \tex_mathscriptboxmode:D 861
 \tex_mathscriptcharmode:D 862
 \tex_mathscriptsmode:D 860
 \tex_mathstyle:D 863, 1254
 \tex_mathsurround:D 361
 \tex_mathsurroundmode:D 864
 \tex_mathsurroundskip:D 865
 \tex_maxdeadcycles:D 362

<code>\tex_maxdepth:D</code>	363	<code>\tex_outputbox:D</code>	871, 1257
<code>\tex_mdffivesum:D</code>	668, 777, 13428	<code>\tex_outputpenalty:D</code>	392
<code>\tex_meaning:D</code> ..	364, 1200, 1217, 1301, 1307, 1313, 1319, 1383, 1384	<code>\tex_over:D</code>	393, 1339
<code>\tex_medmuskip:D</code>	365	<code>\tex_overfullrule:D</code>	394
<code>\tex_message:D</code>	366	<code>\tex_overline:D</code>	395
<code>\tex_middle:D</code>	547, 1354	<code>\tex_overwithdelims:D</code>	396
<code>\tex_mkern:D</code>	367	<code>\tex_pagebottomoffset:D</code> ...	872, 1284
<code>\tex_month:D</code> ...	368, 1313, 1317, 1337	<code>\tex_pagedepth:D</code>	397
<code>\tex_moveleft:D</code>	369, 26778	<code>\tex_pagedir:D</code>	873, 1285, 1350
<code>\tex_moveright:D</code>	370, 26780	<code>\tex_pagedirection:D</code>	874
<code>\tex_mskip:D</code>	371	<code>\tex_pagediscards:D</code>	551
<code>\tex_muexpr:D</code> ..	548, 14361, 14363, 14371, 14373, 14377, 14379, 14383	<code>\tex_pagefillllstretch:D</code>	398
<code>\tex_multiply:D</code>	372	<code>\tex_pagefillstretch:D</code>	399
<code>\tex_muskip:D</code>	373	<code>\tex_pagefilstretch:D</code>	400
<code>\tex_muskipdef:D</code>	374	<code>\tex_pagefistretch:D</code>	1142
<code>\tex_mutogluue:D</code>	303, 549	<code>\tex_pagegoal:D</code>	401
<code>\tex_newlinechar:D</code> 375, 1812, 3663, 3689, 3693, 4530, 11597, 11816, 12718		<code>\tex_pageheight:D</code> ...	671, 925, 1286
<code>\tex_noalign:D</code>	376	<code>\tex_pageleftoffset:D</code> ...	875, 1258
<code>\tex_noautospadding:D</code>	1140	<code>\tex_pagerightoffset:D</code> ...	876, 1287
<code>\tex_noautoxspacing:D</code>	1141	<code>\tex_pageshrink:D</code>	402
<code>\tex_noboundary:D</code>	377	<code>\tex_pagestretch:D</code>	403
<code>\tex_noexpand:D</code>	378, 1379	<code>\tex_pagetopoffset:D</code> ...	877, 1259
<code>\tex_nohrule:D</code>	866	<code>\tex_pagetotal:D</code>	404
<code>\tex_noindent:D</code>	379	<code>\tex_pagewidth:D</code> ...	672, 926, 1288
<code>\tex_nokerns:D</code>	867, 1255	<code>\tex_par:D</code>	405
<code>\tex_noligatures:D</code>	669	<code>\tex_pardir:D</code>	878, 1289
<code>\tex_noligs:D</code>	868, 1256	<code>\tex_pardirection:D</code>	879
<code>\tex_nolimits:D</code>	380	<code>\tex_parfillskip:D</code>	406
<code>\tex_nonscript:D</code>	381	<code>\tex_parindent:D</code>	407
<code>\tex_nonstopmode:D</code>	382	<code>\tex_parshape:D</code>	408
<code>\tex_normaldeviate:D</code>	670, 923	<code>\tex_parshapedimen:D</code>	552
<code>\tex_nospaces:D</code>	869	<code>\tex_parshapeindent:D</code>	553
<code>\tex_novrule:D</code>	870	<code>\tex_parshapelength:D</code>	554
<code>\tex_nulldelimiterspace:D</code>	383	<code>\tex_parskip:D</code>	409
<code>\tex_nullfont:D</code>	384, 10870	<code>\tex_patterns:D</code>	410
<code>\tex_number:D</code> .	385, 8048, 27412, 28876	<code>\tex_pausing:D</code>	411
<code>\tex_numexpr:D</code> 550, 8049, 15882, 23491		<code>\tex_pdfannot:D</code>	577
<code>\tex_odelcode:D</code>	1175	<code>\tex_pdfcatalog:D</code>	578
<code>\tex_odelimiter:D</code>	1176	<code>\tex_pdfcolorstack:D</code>	580
<code>\tex_omathaccent:D</code>	1177	<code>\tex_pdfcolorstackinit:D</code>	581
<code>\tex_omathchar:D</code>	1178	<code>\tex_pdfcompresslevel:D</code>	579
<code>\tex_omathchardef:D</code>		<code>\tex_pdfcreationdate:D</code>	582
..	1179, 1415, 1416, 8177, 8179, 8180	<code>\tex_pdfdecimaldigits:D</code>	583
<code>\tex_omathcode:D</code>	1180	<code>\tex_pdfdest:D</code>	584
<code>\tex_omit:D</code>	386	<code>\tex_pdfdestmargin:D</code>	585
<code>\tex_openin:D</code>	387, 12472	<code>\tex_pdfendlink:D</code>	586
<code>\tex_openout:D</code>	388, 12670	<code>\tex_pdfendthread:D</code>	587
<code>\tex_or:D</code>	389, 1361	<code>\tex_pdfextension:D</code>	880
<code>\tex_oradical:D</code>	1181	<code>\tex_pdffeedback:D</code>	881
<code>\tex_outer:D</code>	390, 1338, 31877	<code>\tex_pdffontattr:D</code>	588
<code>\tex_output:D</code>	391	<code>\tex_pdffontname:D</code>	589
		<code>\tex_pdffontobjnum:D</code>	590
		<code>\tex_pdfgamma:D</code>	591
		<code>\tex_pdfgentounicode:D</code>	594

<code>\tex_pdfglyphtounicode:D</code>	595	<code>\tex_pdfximagebbox:D</code>	645
<code>\tex_pdfhorigin:D</code>	596	<code>\tex_penalty:D</code>	412
<code>\tex_pdfimageapplygamma:D</code>	592	<code>\tex_pkmode:D</code>	673
<code>\tex_pdfimagegamma:D</code>	593	<code>\tex_pkresolution:D</code>	674
<code>\tex_pdfimagehicolor:D</code>	597	<code>\tex_postbreakpenalty:D</code>	1143
<code>\tex_pdfimageresolution:D</code>	598	<code>\tex_postdisplaypenalty:D</code>	413
<code>\tex_pdfincludechars:D</code>	599	<code>\tex_postexhyphenchar:D</code> ...	883, 1260
<code>\tex_pdfinclusioncopyfonts:D</code> ..	600	<code>\tex_postthyphenchar:D</code>	884, 1261
<code>\tex_pdfinclusionerrorlevel:D</code> ..	602	<code>\tex_prebinoppenalty:D</code>	885
<code>\tex_pdfinfo:D</code>	603	<code>\tex_prebreakpenalty:D</code>	1144
<code>\tex_pdflastannot:D</code>	604	<code>\tex_predisplaydirection:D</code>	555
<code>\tex_pdflastlink:D</code>	605	<code>\tex_predisplaygapfactor:D</code>	886
<code>\tex_pdflastobj:D</code>	606	<code>\tex_predisplaypenalty:D</code>	414
<code>\tex_pdflastxform:D</code>	607, 916	<code>\tex_predisplaysize:D</code>	415
<code>\tex_pdflastximage:D</code>	608, 918	<code>\tex_preexhyphenchar:D</code>	887, 1262
<code>\tex_pdflastximagecolordepth:D</code> .	610	<code>\tex_prehyphenchar:D</code>	888, 1263
<code>\tex_pdflastximagepages:D</code> ..	611, 920	<code>\tex_prerelpenalty:D</code>	889
<code>\tex_pdflinkmargin:D</code>	612	<code>\tex_pretolerance:D</code>	416
<code>\tex_pdfliteral:D</code>	613	<code>\tex_prevdepth:D</code>	417
<code>\tex_pdfmajorversion:D</code>	614	<code>\tex_prevgraf:D</code>	418
<code>\tex_pdfminorversion:D</code>	615	<code>\tex_primitive:D</code>	
<code>\tex_pdfnames:D</code>	616	350, 675, 779, 2565, 9423, 9433
<code>\tex_pdfobj:D</code>	617	<code>\tex_protected:D</code>	
<code>\tex_pdfobjcompresslevel:D</code>	618	556, 1423, 1425, 1427,
<code>\tex_pdfoutline:D</code>	619		1428, 1429, 1430, 1431, 1432, 1433,
<code>\tex_pdfoutput:D</code>	620, 924, 9513		1434, 1442, 1444, 1446, 1448, 31877
<code>\tex_pdfpageattr:D</code>	621	<code>\tex_protrudechars:D</code>	676, 927
<code>\tex_pdfpagebox:D</code>	623	<code>\tex_ptexminorversion:D</code>	
<code>\tex_pdfpageref:D</code>	624	1145, 31518, 31537
<code>\tex_pdfpageresources:D</code>	625	<code>\tex_ptexrevision:D</code>	1146, 31519, 31538
<code>\tex_pdfpagesattr:D</code>	622, 626	<code>\tex_ptexversion:D</code>	
<code>\tex_pdfrefobj:D</code>	627	...	1147, 31513, 31516, 31532, 31535
<code>\tex_pdfrefxform:D</code>	628, 930	<code>\tex_pxdimen:D</code>	677, 928
<code>\tex_pdfrefximage:D</code>	629, 931	<code>\tex_quitvmode:D</code>	694
<code>\tex_pdfrestore:D</code>	630	<code>\tex_radical:D</code>	419
<code>\tex_pdfretval:D</code>	631	<code>\tex_raise:D</code>	420, 26782
<code>\tex_pdfsave:D</code>	632	<code>\tex_randomseed:D</code>	678, 929, 9450
<code>\tex_pdfsetmatrix:D</code>	633	<code>\tex_read:D</code>	421, 11717, 12521
<code>\tex_pdfstartlink:D</code>	634	<code>\tex_readline:D</code>	557, 12538
<code>\tex_pdfstartthread:D</code>	635	<code>\tex_readpapersizespecial:D</code> ..	1148
<code>\tex_pdfsuppressptexinfo:D</code>	636	<code>\tex_relax:D</code>	
<code>\tex_pdftexbanner:D</code>	686, 1327	303, 422, 723, 1388, 8050, 13892
<code>\tex_pdftexrevision:D</code>	687, 1328, 31509	<code>\tex_relpennalty:D</code>	423
<code>\tex_pdftexversion:D</code>		<code>\tex_resettimer:D</code>	679, 780
...	302, 688, 1294, 1329, 9248, 31507	<code>\tex_right:D</code>	424, 1355
<code>\tex_pdfthread:D</code>	637	<code>\tex_rightghost:D</code>	890, 1290
<code>\tex_pdfthreadmargin:D</code>	638	<code>\tex_righthyphenmin:D</code>	425
<code>\tex_pdftrailer:D</code>	639	<code>\tex_rightmarginern:D</code>	695
<code>\tex_pdfuniqueresname:D</code>	640	<code>\tex_rightskip:D</code>	426
<code>\tex_pdfvariable:D</code>	882	<code>\tex_romannumeral:D</code>	
<code>\tex_pdfvorigin:D</code>	641	323, 323, 348, 427,
<code>\tex_pdfxform:D</code>	642, 933		1381, 1393, 1715, 10287, 15884, 22360
<code>\tex_pdfxformname:D</code>	643	<code>\tex_rpcode:D</code>	696
<code>\tex_pdfximage:D</code>	644, 934		

- `\tex_savecatcodetable:D`
..... 891, 1264, 22321, 22377
- `\tex_savepos:D` 680, 932
- `\tex_savinghyphcodes:D` 558
- `\tex_savingvdiscards:D` 559
- `\tex_scantextokens:D` 892, 1265
- `\tex_scantokens:D` 560, 3675, 3736
- `\tex_scriptbaselineshiftfactor:D`
..... 1150
- `\tex_scriptfont:D` 428
- `\tex_scriptscriptbaselineshiftfactor:D`
..... 1152
- `\tex_scriptscriptfont:D` 429
- `\tex_scriptscriptstyle:D` 430
- `\tex_scriptspace:D` 431
- `\tex_scriptstyle:D` 432
- `\tex_scrollmode:D` 433
- `\tex_setbox:D` .. 434, 26725, 26727,
26731, 26733, 26748, 26757, 26766,
26801, 26803, 26851, 26856, 26863,
26868, 26875, 26881, 26895, 26901,
26941, 26946, 26953, 26958, 26965,
26970, 26977, 26983, 26998, 27004,
27015, 27019, 28358, 32110, 32113
- `\tex_setfontid:D` 893
- `\tex_setlanguage:D` 435
- `\tex_setrandomseed:D` . 682, 935, 9466
- `\tex_sfcode:D` 436, 10256, 10258
- `\tex_shapemode:D` 894
- `\tex_shellescape:D` ... 683, 781, 9485
- `\tex_shipout:D` 437, 1196, 1220
- `\tex_show:D` 438
- `\tex_showbox:D` 439, 26838
- `\tex_showboxbreadth:D` ... 440, 26834
- `\tex_showboxdepth:D` 441, 26835
- `\tex_showgroups:D` 561
- `\tex_showifs:D` 562
- `\tex_showlists:D` 442
- `\tex_showmode:D` 1153
- `\tex_showthe:D` 443
- `\tex_showtokens:D`
..... 408, 563, 1348, 4534, 11820
- `\tex_sjis:D` 1154
- `\tex_skewchar:D` 444
- `\tex_skip:D` 445,
5629, 5638, 5648, 5671, 5683, 23218,
23247, 23266, 23324, 23340, 23346
- `\tex_skipdef:D` 446
- `\tex_space:D` 186
- `\tex_spacefactor:D` 447
- `\tex_spaceskip:D` 448
- `\tex_span:D` 449
- `\tex_special:D` 450
- `\tex_splitbotmark:D` 451
- `\tex_splitbotmarks:D` 564
- `\tex_splitdiscards:D` 565
- `\tex_splitfirstmark:D` 452
- `\tex_splitfirstmarks:D` 566
- `\tex_splitmaxdepth:D` 453
- `\tex_splittopskip:D` 454
- `\tex_strcmp:D` . 681, 4671, 13568, 16269
- `\tex_string:D` 455, 1199,
1203, 1302, 1308, 1314, 1320, 1386
- `\tex_suppressfontnotfounderror:D`
..... 713, 1273
- `\tex_suppressifcurnameerror:D` ...
..... 895, 1266
- `\tex_suppresslongerror:D` .. 896, 1268
- `\tex_suppressmathparerror:D` 897, 1269
- `\tex_suppressoutererror:D` . 898, 1271
- `\tex_suppressprimitiveerror:D` .. 900
- `\tex_synctex:D` 697
- `\tex_tabskip:D` 456
- `\tex_tagcode:D` 698
- `\tex_tate:D` 1155
- `\tex_tbaselineshift:D` 1156
- `\tex_textbaselineshiftfactor:D` 1158
- `\tex_textdir:D` 901, 1291
- `\tex_textdirection:D` 902
- `\tex_textfont:D` 457
- `\tex_textstyle:D` 458
- `\tex_TeXxTstate:D` 567
- `\tex_tfont:D` 1159
- `\tex_the:D` 159,
303, 341, 459, 757, 762, 763, 1828,
2110, 2238, 2242, 2564, 2606, 2697,
2716, 2731, 2736, 5649, 5684, 7613,
8235, 8237, 9450, 10170, 10240,
10246, 10252, 10258, 14122, 14123,
14124, 14194, 14196, 14307, 14309,
14384, 16887, 17378, 22761, 22793,
22841, 22842, 22873, 22874, 22880,
22881, 23339, 23449, 23494, 23513,
23519, 23522, 23526, 26821, 28877
- `\tex_thickmuskip:D` 460
- `\tex_thinmuskip:D` 461
- `\tex_time:D` 462, 1301, 1305
- `\tex_toks:D`
463, 2707, 2736, 5630, 5639, 5649,
5672, 5684, 7613, 7624, 7625, 7626,
22734, 22761, 22793, 22830, 22841,
22842, 22873, 22874, 22880, 22881,
22885, 22895, 22905, 23163, 23181,
23339, 23494, 23497, 23504, 23512,
23513, 23518, 23519, 23522, 23526
- `\tex_toksapp:D` 903
- `\tex_toksdef:D` 464, 23012
- `\tex_tokspre:D` 904

<code>\tex_tolerance:D</code>	465	<code>\tex_Umathcloseopspacing:D</code>	969
<code>\tex_topmark:D</code>	466	<code>\tex_Umathcloseordspacing:D</code> ...	970
<code>\tex_topmarks:D</code>	568	<code>\tex_Umathclosepunctspacing:D</code> ..	972
<code>\tex_topskip:D</code>	467	<code>\tex_Umathcloserelspacing:D</code> ...	973
<code>\tex_tpack:D</code>	905	<code>\tex_Umathcode:D</code>	974
<code>\tex_tracingassigns:D</code>	569	<code>\tex_Umathcodenum:D</code>	975
<code>\tex_tracingcommands:D</code>	468	<code>\tex_Umathconnectoroverlapmin:D</code>	977
<code>\tex_tracingfonts:D</code>		<code>\tex_Umathfractiondelsize:D</code> ...	978
..... 684, 936, 1225, 1227, 1231		<code>\tex_Umathfractiondenomdown:D</code> ..	980
<code>\tex_tracinggroups:D</code>	570	<code>\tex_Umathfractiondenomvgap:D</code> ..	982
<code>\tex_tracingifs:D</code>	571	<code>\tex_Umathfractionnumup:D</code>	983
<code>\tex_tracinglostchars:D</code>	469	<code>\tex_Umathfractionnumvgap:D</code> ...	984
<code>\tex_tracingmacros:D</code>	470	<code>\tex_Umathfractionrule:D</code>	985
<code>\tex_tracingnesting:D</code>		<code>\tex_Umathinnerbinspacing:D</code> ...	986
..... 572, 3660, 9359, 13218		<code>\tex_Umathinnerclosespacing:D</code> ..	988
<code>\tex_tracingonline:D</code>	471, 26836	<code>\tex_Umathinnerinnerspacing:D</code> ...	990
<code>\tex_tracingoutput:D</code>	472	<code>\tex_Umathinneropenspacing:D</code> ..	991
<code>\tex_tracingpages:D</code>	473	<code>\tex_Umathinneropspacing:D</code>	992
<code>\tex_tracingparagraphs:D</code>	474	<code>\tex_Umathinnerordspacing:D</code> ...	993
<code>\tex_tracingrestores:D</code>	475	<code>\tex_Umathinnerpunctspacing:D</code> ..	995
<code>\tex_tracingscantokens:D</code>	573	<code>\tex_Umathinnerrelspacing:D</code> ...	996
<code>\tex_tracingstats:D</code>	476	<code>\tex_Umathlimitabovebgap:D</code>	997
<code>\tex_uccode:D</code>	477, 10250, 10252	<code>\tex_Umathlimitabovekern:D</code>	998
<code>\tex_Uchar:D</code>	938, 1272, 28865	<code>\tex_Umathlimitabovevgap:D</code>	999
<code>\tex_Ucharcat:D</code> ...	939, 10345, 28882	<code>\tex_Umathlimitbelowbgap:D</code> ...	1000
<code>\tex_uchyph:D</code>	478	<code>\tex_Umathlimitbelowkern:D</code> ...	1001
<code>\tex_ucs:D</code>	1172	<code>\tex_Umathlimitbelowvgap:D</code> ...	1002
<code>\tex_Udelcode:D</code>	940	<code>\tex_Umathnolimitsubfactor:D</code> .	1003
<code>\tex_Udelcodenum:D</code>	941	<code>\tex_Umathnolimitsupfactor:D</code> ..	1004
<code>\tex_Udelimiter:D</code>	942	<code>\tex_Umathopbinspacing:D</code>	1005
<code>\tex_Udelimiterover:D</code>	943	<code>\tex_Umathopclosespacing:D</code> ...	1006
<code>\tex_Udelimiterunder:D</code>	944	<code>\tex_Umathopenbinspacing:D</code> ...	1007
<code>\tex_Uhextensible:D</code>	945	<code>\tex_Umathopenclosespacing:D</code> .	1008
<code>\tex_Umathaccent:D</code>	946	<code>\tex_Umathopeninnerspacing:D</code> .	1009
<code>\tex_Umathaxis:D</code>	947	<code>\tex_Umathopenopenspacing:D</code> ..	1010
<code>\tex_Umathbinbinspacing:D</code>	948	<code>\tex_Umathopenopspacing:D</code>	1011
<code>\tex_Umathbinclosespacing:D</code> ...	949	<code>\tex_Umathopenordspacing:D</code> ...	1012
<code>\tex_Umathbininnerspacing:D</code> ...	950	<code>\tex_Umathopenpunctspacing:D</code> .	1013
<code>\tex_Umathbinopenspacing:D</code>	951	<code>\tex_Umathopenrelspacing:D</code> ...	1014
<code>\tex_Umathbinopspacing:D</code>	952	<code>\tex_Umathoperatorsize:D</code>	1015
<code>\tex_Umathbinordspacing:D</code>	953	<code>\tex_Umathopinnerspacing:D</code> ...	1016
<code>\tex_Umathbinpunctspacing:D</code> ...	954	<code>\tex_Umathopopenspacing:D</code>	1017
<code>\tex_Umathbinrelspacing:D</code>	955	<code>\tex_Umathopopspacing:D</code>	1018
<code>\tex_Umathchar:D</code>	956	<code>\tex_Umathopordspacing:D</code>	1019
<code>\tex_Umathcharclass:D</code>	957	<code>\tex_Umathoppunctspacing:D</code> ...	1020
<code>\tex_Umathchardef:D</code>	958	<code>\tex_Umathoprelspacing:D</code>	1021
<code>\tex_Umathcharfam:D</code>	959	<code>\tex_Umathordbinspacing:D</code>	1022
<code>\tex_Umathcharnum:D</code>	960	<code>\tex_Umathordclosespacing:D</code> ..	1023
<code>\tex_Umathcharnumdef:D</code>	961	<code>\tex_Umathordinnerspacing:D</code> ..	1024
<code>\tex_Umathcharslot:D</code>	962	<code>\tex_Umathordopenspacing:D</code> ...	1025
<code>\tex_Umathclosebinspacing:D</code> ...	963	<code>\tex_Umathordopspacing:D</code>	1026
<code>\tex_Umathcloseclosespacing:D</code> ..	965	<code>\tex_Umathordordspacing:D</code>	1027
<code>\tex_Umathcloseinnerspacing:D</code> ..	967	<code>\tex_Umathordpunctspacing:D</code> ..	1028
<code>\tex_Umathcloseopspacing:D</code> ..	968	<code>\tex_Umathordrelspacing:D</code>	1029

- `\tex_Umathoverbarkern:D` 1030
- `\tex_Umathoverbarrule:D` 1031
- `\tex_Umathoverbarvgap:D` 1032
- `\tex_Umathoverdelimterbgap:D` . 1034
- `\tex_Umathoverdelimtervgap:D` . 1036
- `\tex_Umathpunctbinspacing:D` .. 1037
- `\tex_Umathpunctclosespacing:D` . 1039
- `\tex_Umathpunctinnerspacing:D` . 1041
- `\tex_Umathpunctopenspacing:D` . 1042
- `\tex_Umathpuncttopspacing:D` ... 1043
- `\tex_Umathpunctordspacing:D` .. 1044
- `\tex_Umathpunctpunctspacing:D` . 1046
- `\tex_Umathpunctrelspacing:D` .. 1047
- `\tex_Umathquad:D` 1048
- `\tex_Umathradicaldegreeafter:D` 1050
- `\tex_Umathradicaldegreebefore:D` 1052
- `\tex_Umathradicaldegreeraise:D` 1054
- `\tex_Umathradicalkern:D` 1055
- `\tex_Umathradicalrule:D` 1056
- `\tex_Umathradicalvgap:D` 1057
- `\tex_Umathrelbinspacing:D` 1058
- `\tex_Umathrelclosespacing:D` .. 1059
- `\tex_Umathrelinnerspacing:D` .. 1060
- `\tex_Umathreloppenspacing:D` ... 1061
- `\tex_Umathreloppspacing:D` 1062
- `\tex_Umathrelordspacing:D` 1063
- `\tex_Umathrelrelspacing:D` ... 1064
- `\tex_Umathrelrelspacing:D` 1065
- `\tex_Umathskewedfractionvgap:D` 1067
- `\tex_Umathskewedfractionvgap:D` 1069
- `\tex_Umathspaceafterscript:D` . 1070
- `\tex_Umathstackdenomdown:D` ... 1071
- `\tex_Umathstacknumup:D` 1072
- `\tex_Umathstackvgap:D` 1073
- `\tex_Umathsubshiftdown:D` 1074
- `\tex_Umathsubshiftdrop:D` 1075
- `\tex_Umathsubsupshiftdown:D` .. 1076
- `\tex_Umathsubsupvgap:D` 1077
- `\tex_Umathsubtopmax:D` 1078
- `\tex_Umathsupbottommin:D` 1079
- `\tex_Umathsupshiftdrop:D` 1080
- `\tex_Umathsupshiftup:D` 1081
- `\tex_Umathsupsubbottommax:D` .. 1082
- `\tex_Umathunderbarkern:D` 1083
- `\tex_Umathunderbarrule:D` 1084
- `\tex_Umathunderbarvgap:D` 1085
- `\tex_Umathunderdelimterbgap:D` 1087
- `\tex_Umathunderdelimtervgap:D` 1089
- `\tex_undefined:D` 579, 580,
1225, 1297, 1298, 1305, 1311, 1317,
1323, 1327, 1328, 1329, 1903, 1911,
8880, 14816, 14831, 14884, 14905,
22684, 23102, 23112, 23190, 23290
- `\tex_underline:D` 479, 1194
- `\tex_unexpanded:D`
..... 574, 1341, 1380, 2480, 28868
- `\tex_unhbox:D` 480, 26922
- `\tex_unhcopy:D` 481, 26921
- `\tex_uniformdeviate:D`
... 486, 685, 914, 915, 937, 7592,
7623, 9263, 21834, 21835, 22016, 22019
- `\tex_unkern:D` 482
- `\tex_unless:D` 575, 1364
- `\tex_Unosubscript:D` 1090
- `\tex_Unosuperscript:D` 1091
- `\tex_unpenalty:D` 483
- `\tex_unskip:D` 484
- `\tex_unvbox:D` 485, 27011
- `\tex_unvcopy:D` 486, 27010
- `\tex_Uoverdelimter:D` 1092
- `\tex_uppercase:D` 487, 32078
- `\tex_uptexrevision:D` ... 1173, 31542
- `\tex_uptexversion:D` ... 1174, 31541
- `\tex_Uradical:D` 1093
- `\tex_Uroot:D` 1094
- `\tex_Uskewed:D` 1095
- `\tex_Uskewedwithdelims:D` 1096
- `\tex_Ustack:D` 1097
- `\tex_Ustartdisplaymath:D` 1098
- `\tex_Ustartmath:D` 1099
- `\tex_Ustopdisplaymath:D` 1100
- `\tex_Ustopmath:D` 1101
- `\tex_Usubscript:D` 1102
- `\tex_Usuperscript:D` 1103
- `\tex_Uunderdelimter:D` 1104
- `\tex_Uvextensible:D` 1105
- `\tex_vadjust:D` 488
- `\tex_valign:D` 489
- `\tex_vbadness:D` 490
- `\tex_vbox:D` 491, 26926,
26931, 26936, 26941, 26946, 26965,
26970, 26977, 26983, 26998, 27004
- `\tex_vcenter:D` 492, 1340
- `\tex_vfi:D` 1164
- `\tex_vfil:D` 493
- `\tex_vfill:D` 494
- `\tex_vfilneg:D` 495
- `\tex_vfuzz:D` 496
- `\tex_voffset:D` 497, 1347
- `\tex_vpack:D` 906
- `\tex_vrule:D` 498, 28441
- `\tex_vsize:D` 499
- `\tex_vskip:D` 500, 14313
- `\tex_vsplit:D` 501, 27015, 27020
- `\tex_vss:D` 502
- `\tex_vtop:D` .. 503, 26928, 26953, 26958
- `\tex_wd:D` 504, 26742
- `\tex_widowpenalties:D` 576

- `\tex_widowpenalty:D` 505
- `\tex_write:D`
 . 506, 1830, 1832, 12699, 12702, 12719
- `\tex_xdef:D` 507, 1436, 1440, 1444, 1448
- `\tex_XeTeXcharclass:D` 714
- `\tex_XeTeXcharglyph:D` 715
- `\tex_XeTeXcountfeatures:D` 716
- `\tex_XeTeXcountglyphs:D` 717
- `\tex_XeTeXcountselectors:D` 718
- `\tex_XeTeXcountvariations:D` ... 719
- `\tex_XeTeXdashbreakstate:D` 721
- `\tex_XeTeXdefaultencoding:D` ... 720
- `\tex_XeTeXfeaturecode:D` 722
- `\tex_XeTeXfeaturename:D` 723
- `\tex_XeTeXfindfeaturebyname:D` .. 725
- `\tex_XeTeXfindselectorbyname:D` . 727
- `\tex_XeTeXfindvariationbyname:D` 729
- `\tex_XeTeXfirstfontchar:D` 730
- `\tex_XeTeXfonttype:D` 731
- `\tex_XeTeXgenerateactualtext:D` . 733
- `\tex_XeTeXglyph:D` 734
- `\tex_XeTeXglyphbounds:D` 735
- `\tex_XeTeXglyphindex:D` 736
- `\tex_XeTeXglyphname:D` 737
- `\tex_XeTeXinputencoding:D` 738
- `\tex_XeTeXinputnormalization:D` . 740
- `\tex_XeTeXinterchartokenstate:D` 742
- `\tex_XeTeXinterchartoks:D` 743
- `\tex_XeTeXisdefaultselector:D` .. 745
- `\tex_XeTeXisexclusivefeature:D` . 747
- `\tex_XeTeXlastfontchar:D` 748
- `\tex_XeTeXlinebreaklocale:D` ... 750
- `\tex_XeTeXlinebreakpenalty:D` .. 751
- `\tex_XeTeXlinebreakskip:D` 749
- `\tex_XeTeXOTcountfeatures:D` ... 752
- `\tex_XeTeXOTcountlanguages:D` .. 753
- `\tex_XeTeXOTcountscripts:D` 754
- `\tex_XeTeXOTfeaturetag:D` 755
- `\tex_XeTeXOTlanguagetag:D` 756
- `\tex_XeTeXOTscripttag:D` 757
- `\tex_XeTeXpdfpfile:D` 758
- `\tex_XeTeXpdfpagecount:D` 759
- `\tex_XeTeXpicfile:D` 760
- `\tex_XeTeXrevision:D` 761, 9429, 31550
- `\tex_XeTeXselectorname:D` 762
- `\tex_XeTeXtracingfonts:D` 763
- `\tex_XeTeXupwardsmode:D` 764
- `\tex_XeTeXuseglyphmetrics:D` ... 765
- `\tex_XeTeXvariation:D` 766
- `\tex_XeTeXvariationdefault:D` .. 767
- `\tex_XeTeXvariationmax:D` 768
- `\tex_XeTeXvariationmin:D` 769
- `\tex_XeTeXvariationname:D` 770
- `\tex_XeTeXversion:D`
 . 771, 8178, 8856, 9255, 10637, 31549
- `\tex_xkanjiskip:D` 1160
- `\tex_xleaders:D` 508
- `\tex_xspaceskip:D` 509
- `\tex_xspcode:D` 1161
- `\tex_ybaselineshift:D` 1162
- `\tex_year:D` 510, 1319, 1323
- `\tex_yoko:D` 1163
- `\text` 30831
- text commands:
 - `\l_text_accents_tl`
 260, 263, 29165, 29378, 30962
 - `\l_text_case_exclude_arg_tl`
 262, 263, 29183, 29696
 - `\text_declare_expand_equivalent:Nn`
 260, 29502
 - `\text_declare_purify_equivalent:Nn`
 263, 263,
 30811, 30824, 30825, 30826, 30827,
 30844, 30869, 30870, 30871, 30872,
 30875, 30876, 30882, 30884, 30885,
 30886, 30894, 30906, 30948, 30963
 - `\text_expand` 262
 - `\text_expand:n`
 260, 263, 29199, 29535, 30641
 - `\l_text_expand_exclude_tl`
 260, 263, 29189, 29379
 - `\l_text_letterlike_tl`
 260, 263, 29165, 29407
 - `\text_lowercase:n`
 ... 69, 133, 262, 29513, 32175, 32177
 - `\text_lowercase:nn`
 262, 29513, 32178, 32180
 - `\l_text_math_arg_tl` 260,
 263, 263, 29185, 29377, 29695, 30757
 - `\l_text_math_delims_tl` . 260, 263,
 263, 263, 29187, 29311, 29628, 30686
 - `\text_purify:n` 263, 30635
 - `\text_titlecase:n`
 ... 69, 131, 262, 29513, 32187, 32189
 - `\text_titlecase:nn`
 262, 29513, 32190, 32192
 - `\l_text_titlecase_check_letter_-`
 bool 262, 263, 29511, 29801
 - `\text_titlecase_first:n` .. 262, 29513
 - `\text_titlecase_first:nn` . 262, 29513
 - `\text_uppercase:n`
 69, 131, 133, 262, 29513, 32181, 32183
 - `\text_uppercase:nn`
 262, 29513, 32184, 32186
- text internal commands:
 - `__text_change_case:nnn`
 29514, 29516, 29518, 29520,

- [29522, 29524, 29526, 29528, 29529](#)
`_text_change_case_aux:nnn` .. [29529](#)
`_text_change_case_break:w` .. [29529](#)
`_text_change_case_char:nnnN` ...
 [29529](#),
 [29904, 29915, 29924, 29938, 30026,](#)
 [30126, 30159, 30241, 30255, 30278](#)
`_text_change_case_char_-`
 aux:nnnN .. [29529](#)
`_text_change_case_char_-`
 lower:nnN .. [29529](#)
`_text_change_case_char_next_-`
 end:nn .. [29529](#)
`_text_change_case_char_next_-`
 lower:nn .. [29529](#)
`_text_change_case_char_next_-`
 title:nn .. [29529](#)
`_text_change_case_char_next_-`
 titleonly:nn .. [29529](#)
`_text_change_case_char_next_-`
 upper:nn .. [29529](#)
`_text_change_case_char_-`
 title:nN .. [29529](#)
`_text_change_case_char_-`
 title:nnN .. [29529](#)
`_text_change_case_char_-`
 title:nnnN .. [29529](#)
`_text_change_case_char_-`
 titleonly:nN .. [29529](#)
`_text_change_case_char_-`
 titleonly:nnN .. [29529](#)
`_text_change_case_char_-`
 upper:nnN .. [29529](#)
`_text_change_case_char_-`
 UTFviii:nnnn .. [29529](#)
`_text_change_case_char_-`
 UTFviii:nnnNN .. [29529](#)
`_text_change_case_char_-`
 UTFviii:nnnNNN .. [29529](#)
`_text_change_case_char_-`
 UTFviii:nnnNNNN .. [29858](#)
`_text_change_case_char_-`
 UTFviii:nnnNNNNN .. [29529](#)
`_text_change_case_cs_check:nnN`
 .. [29529](#)
`_text_change_case_end:w` [29529](#)
`_text_change_case_exclude:nnN` .
 .. [29529](#)
`_text_change_case_exclude:nnNN`
 .. [29529](#)
`_text_change_case_exclude:nnNn`
 .. [29529](#)
`_text_change_case_exclude:nnnN`
 .. [29529](#)
- `_text_change_case_group_-`
 lower:nnn .. [29529](#)
`_text_change_case_group_-`
 title:nnn .. [29529](#)
`_text_change_case_group_-`
 titleonly:nnn .. [29529](#)
`_text_change_case_group_-`
 upper:nnn .. [29529](#)
`_text_change_case_if_greek:n` [30000](#)
`_text_change_case_if_greek:nTF`
 .. [29927](#)
`_text_change_case_letterlike:nnnnN`
 .. [29529](#)
`_text_change_case_letterlike_-`
 lower:nnN .. [29529](#)
`_text_change_case_letterlike_-`
 title:nnN .. [29529](#)
`_text_change_case_letterlike_-`
 titleonly:nnN .. [29529](#)
`_text_change_case_letterlike_-`
 upper:nnN .. [29529](#)
`_text_change_case_loop:nnw` ...
 .. [29529, 29952, 29958, 29993,](#)
 [29994, 30067, 30084, 30103, 30197,](#)
 [30209, 30221, 30226, 30236, 30253](#)
`_text_change_case_lower_-`
 az:nnnN .. [30280](#)
`_text_change_case_lower_lt:nnN`
 .. [30028](#)
`_text_change_case_lower_-`
 lt:nnnN .. [30032](#)
`_text_change_case_lower_lt:nnw`
 .. [30028](#)
`_text_change_case_lower_lt_-`
 auxi:nnnN .. [30034, 30045](#)
`_text_change_case_lower_lt_-`
 auxii:nnnN .. [30049, 30070](#)
`_text_change_case_lower_-`
 sigma:NnnN .. [29529](#)
`_text_change_case_lower_-`
 sigma:nnnN ... [29529, 30073, 30199](#)
`_text_change_case_lower_-`
 sigma:nnNw .. [29529](#)
`_text_change_case_lower_-`
 tr:NnnN .. [30184](#)
`_text_change_case_lower_-`
 tr:nnnN .. [30184, 30281](#)
`_text_change_case_lower_-`
 tr:nnnNN .. [30184](#)
`_text_change_case_lower_-`
 tr:nnNw .. [30184](#)
`_text_change_case_math_-`
 group:nnNn .. [29529](#)

<code>_text_change_case_math_-</code>		
<code>loop:nnNw</code>	29529
<code>_text_change_case_math_N_-</code>		
<code>type:nnNN</code>	29529
<code>_text_change_case_math_-</code>		
<code>search:nnNNN</code>	29529
<code>_text_change_case_math_-</code>		
<code>space:nnNw</code>	29529
<code>_text_change_case_N_type:nnN</code>		29529
<code>_text_change_case_N_type:nnnN</code>		29529
.....		29529
<code>_text_change_case_N_type_-</code>		
<code>aux:nnN</code>	29529
<code>_text_change_case_result:n</code>		29529
<code>_text_change_case_setup:NN</code>		...
.....		30572 , 30579 , 30581
<code>_text_change_case_setup:Nn</code>		...
.....		30603 , 30623 , 30625
<code>_text_change_case_space:nnw</code>		29529
<code>_text_change_case_store:n</code>	
.....		29529 , 29899 , 29921 , 29944 , 29962 , 29984 , 30061 , 30075 , 30100 , 30128 , 30155 , 30178 , 30195 , 30207 , 30219 , 30224 , 30235 , 30248 , 30266 , 30273
<code>_text_change_case_store:nw</code>		29529
<code>_text_change_case_title_-</code>		
<code>el:nnnN</code>	30021
<code>_text_change_case_title_nl:nnN</code>	
.....		30149
<code>_text_change_case_title_-</code>		
<code>nl:nnnN</code>	30149
<code>_text_change_case_title_nl:nnw</code>	
.....		30149
<code>_text_change_case_upper_-</code>		
<code>az:nnnN</code>	30280
<code>_text_change_case_upper_-</code>		
<code>de-alt:nnnN</code>	29891
<code>_text_change_case_upper_-</code>		
<code>de-alt:nnnN</code>	29891
<code>_text_change_case_upper_el:nnN</code>	
.....		29927
<code>_text_change_case_upper_-</code>		
<code>el:nnnN</code>	29927
<code>_text_change_case_upper_-</code>		
<code>el:nnnn</code>	29927
<code>_text_change_case_upper_el_-</code>		
<code>aux:nnnN</code>	29927
<code>_text_change_case_upper_el_-</code>		
<code>loop:nnw</code>	29927
<code>_text_change_case_upper_lt:nnN</code>	
.....		30106
<code>_text_change_case_upper_-</code>		
<code>lt:nnnN</code>	30110
<code>_text_change_case_upper_lt:nnw</code>	
.....		30106
<code>_text_change_case_upper_lt_-</code>		
<code>aux:nnnN</code>	30112 , 30123
<code>_text_change_case_upper_-</code>		
<code>tr:nnnN</code>	30258 , 30283
<code>_text_change_cases_lower_-</code>		
<code>lt:nnnN</code>	30028
<code>_text_change_cases_lower_lt_-</code>		
<code>auxi:nnnN</code>	30028
<code>_text_change_cases_lower_lt_-</code>		
<code>auxii:nnnN</code>	30028
<code>_text_change_cases_upper_-</code>		
<code>lt:nnnN</code>	30106
<code>_text_change_cases_upper_lt_-</code>		
<code>aux:nnnN</code>	30106
<code>_text_char_catcode:N</code>	
.....		29118 , 29783 , 29793 , 29794 , 29900 , 29987 , 30063 , 30064 , 30065 , 30076 , 30101 , 30129 , 30156 , 30179 , 30196 , 30208 , 30220 , 30225 , 30269
<code>\c_text_chardef_group_begin_-</code>		
<code>token</code>	29193 , 29275
<code>\c_text_chardef_group_end_token</code>	
.....		29193 , 29283
<code>\c_text_chardef_space_token</code>		...
.....		29193 , 29262
<code>\c_text_dotless_i_tl</code>		30235 , 30284
<code>\c_text_dotted_I_tl</code>		30274 , 30284
<code>_text_expand:n</code>		29199
<code>_text_expand_cs:N</code>		29199
<code>_text_expand_cs_expand:N</code>		29199
<code>_text_expand_encoding:N</code>		29199
<code>_text_expand_encoding_escape:N</code>	
.....		29199
<code>_text_expand_encoding_escape:NN</code>	
.....		29455 , 29460 , 29465
<code>_text_expand_end:w</code>		29199
<code>_text_expand_exclude:N</code>		29199
<code>_text_expand_exclude:NN</code>		29199
<code>_text_expand_exclude:Nn</code>		29199
<code>_text_expand_exclude:nN</code>		29199
<code>_text_expand_explicit:N</code>		29199
<code>_text_expand_group:n</code>		29199
<code>_text_expand_implicit:N</code>		29199
<code>_text_expand_letterlike:N</code>		29199
<code>_text_expand_letterlike:NN</code>		29199
<code>_text_expand_loop:w</code>		29199
<code>_text_expand_math_group:Nn</code>		29199
<code>_text_expand_math_loop:Nw</code>		29199
<code>_text_expand_math_N_type:NN</code>		29199
<code>_text_expand_math_search:NNN</code>		29199
<code>_text_expand_math_space:Nw</code>		29199
<code>_text_expand_N_type:N</code>		29199

<code>__text_expand_N_type_auxi:N</code>	29199	<code>__text_purify_math_result:n</code>	...
<code>__text_expand_N_type_auxii:N</code>	29199	30704 ,
<code>__text_expand_N_type_auxiii:N</code>	29199	30708 , 30709 , 30710 , 30715 , 30771	
<code>__text_expand_noexpand:nn</code>	...	30635	
<code>__text_expand_noexpand:w</code>	<code>__text_purify_math_search:NNN</code>	30635
.....	29487 , 29495	<code>__text_purify_math_space:NNw</code>	30635
<code>__text_expand_protect:N</code>	<code>__text_purify_math_start:NNw</code>	30635
<code>__text_expand_protect:nN</code>	<code>__text_purify_math_stop:Nw</code>	...
<code>__text_expand_protect:Nw</code>	30715 , 30734
<code>__text_expand_replace:N</code>	<code>__text_purify_math_store:n</code>	30635
<code>__text_expand_replace:n</code>	<code>__text_purify_math_store:nw</code>	30635
<code>__text_expand_result:n</code>	<code>__text_purify_N_type:N</code>	30635
<code>__text_expand_space:w</code>	<code>__text_purify_N_type_aux:N</code>	30635
<code>__text_expand_store:n</code>	<code>__text_purify_protect:N</code>	30635
<code>__text_expand_store:nw</code>	<code>__text_purify_replace:N</code>	30635
<code>\c_text_grosses_Eszett_tl</code>	<code>__text_purify_replace:n</code>	30635
.....	29921 , 30284	<code>__text_purify_result:n</code>
<code>\c_text_I_ogonek_tl</code>	30649 , 30653 , 30654 , 30655
<code>\c_text_i_ogonek_tl</code>	<code>__text_purify_space:w</code>	30635
<code>__text_if_expandable:NTF</code>	<code>__text_purify_store:n</code>	30635
.....	29150 , 29484 , 30801	<code>__text_purify_store:nw</code>	30635
<code>__text_if_recursion_tail_stop:N</code>	<code>__text_quark_if_nil:nTF</code>	29027 , 29443
.....	30634	<code>__text_quark_if_nil_p:n</code>	29027
<code>__text_if_recursion_tail_stop-</code>		<code>__text_tmp:n</code>
<code>do:Nn</code>	30913 , 30969 , 30976 , 30983 , 31021	
.....	29032 , 29255 , 29317 , 29342 , 29390 ,	<code>__text_tmp:nnnn</code>	...
.....	29412 , 29621 , 29638 , 29663 , 29707 ,	30334 , 30365 ,	
.....	30680 , 30692 , 30733 , 30761 , 30808	30366 , 30913 , 30914 , 30988 , 30989	
<code>__text_loop:Nn</code>	30891 , 30899 , 30903 ,	<code>__text_tmp:w</code>
.....	30911 , 30922 , 30965 , 30970 , 30997	30289 ,	
<code>__text_loop:nn</code>	30323 , 30332 , 30368	30292 , 30308 , 30311 , 30312 , 30313 ,	
<code>__text_loop:NNn</code>	31016 , 31022 , 31024	30314 , 30315 , 30328 , 30350 , 30548 ,	
<code>\l_text_math_mode_tl</code>	30551 , 30563 , 30566 , 30567 , 30568	
<code>\c_text_mathchardef_group-</code>		<code>__text_tmp_aux:n</code>
<code>begin_token</code>	30985 , 30988	
.....	29193 , 29276	<code>__text_token_to_explicit:N</code>
<code>\c_text_mathchardef_group_end-</code>		29033 , 30790
<code>token</code>	<code>__text_token_to_explicit:n</code>	29033
.....	29193 , 29284	<code>__text_token_to_explicit_auxi:w</code>
<code>\c_text_mathchardef_space_token</code>	29033
.....	29193 , 29266	<code>__text_token_to_explicit-</code>	
<code>__text_purify:n</code>	<code>auxii:w</code>
<code>__text_purify_accent:NN</code>	29033
<code>__text_purify_end:w</code>	<code>__text_token_to_explicit-</code>	
<code>__text_purify_expand:N</code>	<code>auxiii:w</code>
<code>__text_purify_group:n</code>	29033
<code>__text_purify_loop:w</code>	<code>__text_token_to_explicit_char:N</code>
<code>__text_purify_math_cmd:N</code>	29033
<code>__text_purify_math_cmd:n</code>	<code>__text_token_to_explicit_cs:N</code>	29033
.....	30766 , 30770	<code>__text_token_to_explicit_cs-</code>	
<code>__text_purify_math_cmd:NN</code>	...	<code>aux:N</code>
<code>__text_purify_math_cmd:Nn</code>	29033
<code>__text_purify_math_end:w</code>	<code>__text_use_i_delimit_by_q-</code>	
<code>__text_purify_math_group:NNn</code>	30635	<code>recursion_stop:nw</code>
<code>__text_purify_math_loop:NNw</code>	30635	29030 , 29321 , 29394 ,
<code>__text_purify_math_N_type:NNN</code>	30635	29416 , 29642 , 29711 , 30696 , 30765	
		<code>\textbaselineshiftfactor</code>
		1157
		<code>\textbf</code>
		30836
		<code>\textdir</code>
		901
		<code>\textdirection</code>
		902
		<code>\textfont</code>
		457

- \textit 30838
- \textmd 30837
- \textnormal 30832
- \textrm 30833
- \textsc 30841
- \textsf 30834
- \textsl 30839
- \textstyle 458
- \texttt 18383, 30835
- \textulc 30842
- \textup 30840
- \TeXeTstate 567
- \tfont 1159
- \TH 29181, 30592, 30932
- \th 29181, 30592, 30945
- \the 41, 120, 121, 122,
123, 124, 125, 126, 127, 128, 129, 459
- \thickmuskip 460
- \thinmuskip 461
- thousand commands:
 - \c_one_thousand 31959
 - \c_ten_thousand 31961
- \time 462, 1302, 9439, 9441
- \tiny 28420, 30867
- tl commands:
 - \c_empty_tl 58,
513, 545, 2588, 3522, 3538, 3540,
3561, 3835, 8579, 8585, 9565, 9584,
11732, 28913, 31621, 31658, 31677
 - \l_my_tl 227, 233
 - \c_novalue_tl 48, 58, 3562, 3934
 - \c_space_tl
58, 3571, 4205, 5158, 10020,
10029, 11526, 13150, 13152, 28203,
28247, 28314, 28926, 28997, 29248,
29269, 29355, 29613, 29678, 30673,
30749, 30882, 31373, 31375, 31893
 - \tl_analysis_map_inline:Nn
226, 23365, 24918
 - \tl_analysis_map_inline:nn
226, 23365, 25479
 - \tl_analysis_show:N 226, 23392, 32068
 - \tl_analysis_show:n 226, 23392, 32070
 - \tl_build_begin:N 272, 273,
273, 273, 1009, 1178, 24081, 24580,
24944, 25037, 25786, 31577, 31592
 - \tl_build_clear:N 272, 31592
 - \tl_build_end:N 272, 273,
1009, 1178, 1179, 24111, 24119,
24590, 24999, 25056, 25820, 31665
 - \tl_build_gbegin:N
272, 273, 273, 273, 31577, 31593
 - \tl_build_gclear:N 272, 31592
 - \tl_build_gend:N 273, 31665
 - \tl_build_get:N 273
 - \tl_build_get:NN 273, 31651
 - \tl_build_gput_left:Nn 273, 31634
 - \tl_build_gput_right:Nn 273, 31594
 - \tl_build_put_left:Nn 273, 31634
 - \tl_build_put_right:Nn 273, 1037,
1180, 24088, 24106, 24114, 24118,
24168, 24171, 24204, 24218, 24222,
24347, 24361, 24402, 24427, 24436,
24446, 24478, 24491, 24495, 24577,
24583, 24589, 24593, 24636, 24904,
24920, 24938, 25007, 25052, 25065,
25805, 25844, 25876, 25938, 25941,
25956, 26000, 26016, 26052, 31594
 - \tl_case:Nn 48, 3964
 - \tl_case:nn 414
 - \tl_case:nn(TF) 507, 1174
 - \tl_case:Nnn 32071, 32073
 - \tl_case:NnTF
48, 3964, 3969, 3974, 32072, 32074
 - \tl_clear:N
43, 44, 3537, 3544, 3649, 5443,
9626, 9627, 12856, 12857, 12860,
12869, 12979, 12982, 13042, 13538,
15354, 23766, 25788, 31668, 31729
 - \tl_clear_new:N
44, 3543, 9630, 9631, 13805, 13806,
13807, 13808, 13809, 29504, 30813
 - \tl_concat:NNN 44, 3553, 4583
 - \tl_const:Nn 43, 523, 3525, 3561,
3569, 3571, 3645, 5280, 5285, 5605,
5606, 7385, 7440, 9336, 9624, 10395,
10642, 10665, 11095, 11153, 11450,
11451, 11490, 11495, 11497, 11499,
11501, 11503, 11508, 11509, 11516,
12753, 12759, 13197, 15910, 15911,
15912, 15913, 15914, 15922, 16011,
18124, 19427, 19874, 19875, 19876,
19877, 19878, 19879, 19880, 19881,
19882, 23479, 23538, 26085, 28901,
28916, 28939, 28951, 28980, 29010,
29011, 29012, 30294, 30336, 30352,
30553, 30575, 30577, 30595, 30596,
30611, 30618, 30968, 31019, 31714
 - \tl_count:N 27, 48, 51, 52, 4072
 - \tl_count:n 27, 48, 51, 52, 331,
422, 500, 727, 1560, 1564, 1952,
2002, 4072, 4399, 4414, 4426, 24836
 - \tl_count_tokens:n 52, 4085
 - \tl_gclear:N 43, 941, 3537, 3546,
9408, 9503, 9628, 9629, 22755, 31673
 - \tl_gclear_new:N 44, 3543, 9632, 9633
 - \tl_gconcat:NNN 44, 3553, 4584
 - \tl_gput_left:Nn 44, 3590, 6181, 6364

- \tl_gput_right:Nn 44, 1512, 1513, 3502,
3614, 7515, 9411, 9506, 22103, 22548
- \tl_gremove_all:Nn 45, 3817
- \tl_gremove_once:Nn 45, 3811
- \tl_greplace_all:Nnn . 45, 3742, 3820
- \tl_greplace_once:Nnn 45, 3742, 3814
- \tl_greverse:N 52, 4235
- .tl_gset:N 191, 15084
- \tl_gset:Nn
44, 77, 273, 384, 1181, 3556, 3572,
3655, 3745, 3749, 4126, 4238, 4614,
4618, 5320, 5336, 5386, 5532, 5584,
5595, 5750, 5799, 5852, 5858, 6089,
6289, 6446, 7424, 7429, 7447, 7484,
7501, 7541, 7567, 7692, 7724, 7761,
7767, 7917, 7927, 9293, 9303, 9316,
9525, 9547, 9549, 9551, 9553, 9555,
9645, 9672, 9691, 9734, 9770, 9819,
9858, 11212, 11241, 11287, 11295,
11318, 18122, 22751, 23257, 23771,
24899, 31479, 31489, 31672, 32171
- \tl_gset_eq:NN 44,
3540, 3549, 4580, 5363, 5379, 7408,
7409, 7410, 7411, 8954, 9638, 9639,
9640, 9641, 11124, 11125, 11126,
11127, 18129, 22745, 26080, 32150
- \tl_gset_from_file:Nnn 32141
- \tl_gset_from_file_x:Nnn 32141
- \tl_gset_rescan:Nnn 46, 3646
- .tl_gset_x:N 191, 15084
- \tl_gsort:Nn 53, 4167, 22743
- \tl_gtrim_spaces:N 53, 4117
- \tl_head:N 54, 4241
- \tl_head:n . . 54, 54, 349, 400, 400,
405, 2546, 4241, 4423, 28936, 28944
- \tl_head:w
.... 54, 401, 402, 4241, 28961, 28992
- \tl_if_blank:nTF 46, 54, 54,
54, 3823, 4269, 4413, 4563, 4590,
6515, 6518, 9273, 10028, 10107,
10515, 11643, 12560, 13062, 13254,
13256, 13274, 13307, 13413, 13497,
13598, 13607, 15261, 15467, 22917,
25500, 28892, 28905, 28955, 28984,
30047, 30072, 30125, 30300, 31889
- \tl_if_blank_p:n 46, 3823, 13441
- \tl_if_empty:N 4667, 4669, 9873, 9875
- \tl_if_empty:NTF
... 47, 3833, 11542, 11552, 12873,
12963, 12998, 13079, 13343, 13528,
13564, 15289, 15294, 15443, 25843
- \tl_if_empty:nTF
..... 47, 388, 390, 391, 545, 554,
1600, 1691, 2525, 2644, 3247, 3254,
3271, 3421, 3756, 3843, 3854, 3915,
3955, 4362, 4383, 4625, 5411, 6479,
7450, 9017, 9578, 9597, 9606, 9609,
9886, 9919, 10858, 11077, 11182,
11813, 11905, 11920, 12041, 12045,
12119, 12181, 12277, 12278, 12287,
12294, 12300, 12307, 12867, 13734,
13740, 13742, 13744, 13862, 14715,
14812, 15635, 16626, 17483, 21782,
21850, 23483, 23484, 26660, 28978
- \tl_if_empty_p:N 47, 3833
- \tl_if_empty_p:n 47, 3843, 3854
- \tl_if_eq:NN 413
- \tl_if_eq:nn(TF) 123, 123
- \tl_if_eq:NNTF 38, 47, 47,
47, 48, 63, 484, 3867, 3988, 7553,
11338, 11886, 11940, 28499, 28502
- \tl_if_eq:NnTF 47, 3879
- \tl_if_eq:nnTF
..... 47, 63, 80, 80, 484, 3892, 9907
- \tl_if_eq_p:NN 47, 3867
- \tl_if_exist:N 4663, 4665
- \tl_if_exist:NTF
..... 44, 3544, 3546, 3559, 4065,
10527, 10538, 10617, 10625, 23394
- \tl_if_exist_p:N 44, 3559
- \tl_if_head_eq_catcode:nN 401
- \tl_if_head_eq_catcode:nNTF 55, 4275
- \tl_if_head_eq_catcode_p:nN 55, 4275
- \tl_if_head_eq_charcode:nN 401
- \tl_if_head_eq_charcode:nNTF ...
..... 55, 4275, 28894
- \tl_if_head_eq_charcode_p:nN 55, 4275
- \tl_if_head_eq_meaning:nN 402
- \tl_if_head_eq_meaning:nNTF 55, 4275
- \tl_if_head_eq_meaning_p:nN
..... 55, 4275, 24835
- \tl_if_head_is_group:nTF
..... 55, 2522, 2641, 4181, 4302,
4339, 4365, 9604, 13114, 29230,
29334, 29562, 29655, 30666, 30725
- \tl_if_head_is_group_p:n ... 55, 4365
- \tl_if_head_is_N_type:n 401
- \tl_if_head_is_N_type:nTF ... 55,
2519, 2583, 2629, 2635, 2680, 2695,
2740, 3952, 4178, 4279, 4296, 4315,
4348, 4484, 13111, 29227, 29331,
29559, 29652, 29779, 29950, 30082,
30135, 30163, 30204, 30663, 30722
- \tl_if_head_is_N_type_p:n .. 55, 4348
- \tl_if_head_is_space:nTF
..... 55, 4376, 4466, 4475, 5152
- \tl_if_head_is_space_p:n ... 55, 4376

- \tl_if_in:Nn 554
- \tl_if_in:nn 390, 390
- \tl_if_in:NnTF
 - 47, 3496, 3778, 3905, 3905, 3906
- \tl_if_in:nnTF 47, 390,
 - 414, 3692, 3762, 3764, 3905, 3906,
 - 3907, 3910, 4718, 4726, 9350, 11075,
 - 11799, 11801, 23662, 28294, 31563
- \tl_if_novalue:nTF 48, 3921
- \tl_if_novalue_p:n 48, 3921
- \tl_if_single:n 391
- \tl_if_single:NnTF 48, 3935, 3936, 3937
- \tl_if_single:nTF
 - 48, 526, 3936, 3937, 3938, 3939
- \tl_if_single_p:N 48, 3935
- \tl_if_single_p:n 48, 3935, 3939
- \tl_if_single_token:nTF
 - 48, 3950, 30606
- \tl_if_single_token_p:n 48, 3950
- \tl_item:Nn 56, 4388
- \tl_item:nn 56, 405, 4388, 4414
- \tl_log:N 58, 4516, 5191
- \tl_log:n 58,
 - 335, 335, 797, 2102, 2118, 4518,
 - 4540, 5190, 8891, 8991, 18154, 31370
- \tl_lower_case:n 32175
- \tl_lower_case:nn 32175
- \tl_map_break: 50,
 - 238, 964, 4001, 4007, 4019, 4029,
 - 4036, 4044, 4050, 4056, 23388, 23389
- \tl_map_break:n
 - ... 50, 50, 4056, 13263, 13364, 22750
- \tl_map_function:NN 49,
 - 49, 49, 49, 270, 270, 3997, 4080, 24906
- \tl_map_function:nN
 - 49, 49, 49, 1996,
 - 3997, 4075, 5492, 5494, 7453, 24188
- \tl_map_inline:Nn 49, 49, 49,
 - 4011, 5276, 5278, 13363, 22750, 30962
- \tl_map_inline:nn 40,
 - 49, 49, 4011, 9257, 11010, 11012,
 - 11014, 11024, 12756, 17536, 20615,
 - 22712, 30817, 30828, 30845, 30873
- \tl_map_tokens:Nn ... 49, 4025, 13262
- \tl_map_tokens:nn 49, 4025
- \tl_map_variable:NNn 49, 4040
- \tl_map_variable:nNn .. 49, 394, 4040
- \tl_mixed_case:n 32175
- \tl_mixed_case:nn 32175
- \tl_new:N ... 43, 44, 135, 378, 3519,
 - 3544, 3546, 3877, 3878, 4545, 4546,
 - 4547, 4548, 5197, 5199, 7382, 7383,
 - 9335, 9412, 9507, 9522, 9566, 9621,
 - 9622, 10323, 10906, 11094, 11444,
 - 11831, 11832, 12409, 12412, 12434,
 - 12626, 12651, 12730, 12732, 12745,
 - 12747, 12748, 12750, 13054, 13084,
 - 13085, 14601, 14604, 14609, 14611,
 - 14617, 14618, 14620, 14621, 22099,
 - 22274, 22275, 22639, 23065, 23529,
 - 23530, 23536, 23537, 23547, 25466,
 - 25713, 25715, 27381, 27406, 27407,
 - 28412, 28649, 29165, 29168, 29183,
 - 29185, 29187, 29189, 29192, 31717
- \tl_put_left:Nn 44, 3590
- \tl_put_right:Nn
 - 44, 1179, 3614, 7513, 10365,
 - 10367, 10370, 10372, 10373, 10375,
 - 10377, 10379, 10380, 10382, 10384,
 - 10386, 10388, 13004, 13007, 13012,
 - 13381, 23773, 25879, 31756, 31761
- \tl_rand_item:N 56, 4411
- \tl_rand_item:n 56, 4411
- \tl_range:Nnn 57, 4418
- \tl_range:nnn 57, 68, 272, 4418
- \tl_range_braced:Nnn 272, 31683
- \tl_range_braced:nnn . 57, 272, 31683
- \tl_range_unbraced:Nnn ... 272, 31683
- \tl_range_unbraced:nnn 57, 272, 31683
- \tl_remove_all:Nn 45, 45, 3817
- \tl_remove_once:Nn 45, 3811
- \tl_replace_all:Nnn
 - 45, 481, 552, 3742, 3818, 7462
- \tl_replace_once:Nnn
 - 45, 3742, 3812, 10402
- \tl_rescan:nn .. 46, 46, 223, 381, 3646
- \tl_reverse:N 52, 52, 4235
- \tl_reverse:n
 - 52, 52, 52, 4215, 4236, 4238
- \tl_reverse_items:n . 52, 52, 52, 4101
- .tl_set:N 191, 15084
- \tl_set:Nn 44, 45,
 - 46, 77, 191, 273, 273, 356, 379, 384,
 - 610, 690, 1181, 3554, 3572, 3653,
 - 3743, 3747, 3882, 3895, 3896, 4051,
 - 4124, 4236, 4529, 4612, 4616, 5275,
 - 5432, 5600, 7414, 7419, 7445, 7452,
 - 7456, 7467, 7482, 7493, 7539, 7549,
 - 7558, 7565, 7643, 7646, 7663, 7671,
 - 7680, 7690, 7699, 7705, 7722, 7736,
 - 7758, 7764, 7873, 7915, 7925, 8476,
 - 9274, 9340, 9375, 9643, 9670, 9689,
 - 9723, 9729, 9732, 9738, 9745, 9768,
 - 9817, 9856, 9997, 10363, 10368,
 - 10943, 11206, 11222, 11223, 11231,
 - 11232, 11234, 11240, 11243, 11276,
 - 11277, 11286, 11294, 11298, 11316,
 - 11321, 11371, 11581, 11815, 11843,

- 11926, 12463, 12519, 12532, 12565,
 12654, 12661, 12737, 12812, 12815,
 12816, 12821, 12832, 12837, 12858,
 12995, 13015, 13034, 13036, 13201,
 13236, 13336, 13341, 13356, 13366,
 13391, 13509, 13511, 13513, 13515,
 13526, 13557, 13562, 13818, 13821,
 13822, 13823, 13824, 13831, 13832,
 13833, 13835, 13839, 14164, 14612,
 14793, 15113, 15122, 15151, 15152,
 15164, 15172, 15193, 15194, 15206,
 15214, 15225, 15234, 15245, 15259,
 15360, 15457, 18120, 18457, 22398,
 22408, 23660, 23673, 24122, 24529,
 24534, 24799, 24860, 24890, 25002,
 25059, 25549, 25558, 25636, 25668,
 25978, 26257, 26326, 26356, 26380,
 27649, 28295, 28296, 28417, 28650,
 29166, 29169, 29184, 29186, 29188,
 29190, 29505, 30814, 31477, 31487,
 31652, 31667, 31728, 32160, 32162
 \tl_set_eq:NN 44, 524, 3538,
 3549, 4579, 5361, 5370, 7404, 7405,
 7406, 7407, 8953, 9634, 9635, 9636,
 9637, 11120, 11121, 11122, 11123,
 11889, 11897, 13384, 14691, 15269,
 15349, 15370, 18128, 22743, 26075
 \tl_set_from_file:Nnn 32141
 \tl_set_from_file_x:Nnn 32141
 \tl_set_rescan:Nnn
 46, 46, 223, 381, 647, 3646
 .tl_set_x:N 191, 15084
 \tl_show:N 58, 58, 965, 4516, 5188, 23400
 \tl_show:n
 .. 58, 58, 268, 335, 335, 408, 408,
 797, 1173, 2098, 2115, 4516, 4525,
 5187, 8890, 8989, 10172, 10242,
 10248, 10254, 10260, 18152, 31368
 \tl_show_analysis:N 32067
 \tl_show_analysis:n 32069
 \tl_sort:Nn 53, 4167, 22743
 \tl_sort:nN . 53, 946, 948, 4167, 22913
 \tl_tail:N .. 54, 438, 4241, 5595, 24810
 \tl_tail:n 54, 4241
 \tl_to_lowercase:n 32075
 \tl_to_str:N
 51, 60, 165, 411, 637, 1040,
 4061, 4633, 4710, 4718, 5754, 10618,
 10626, 12816, 12827, 13782, 13796
 \tl_to_str:n 46, 46,
 51, 51, 60, 69, 70, 118, 146, 146,
 165, 187, 232, 233, 309, 318, 391,
 411, 417, 423, 578, 595, 596, 1040,
 1040, 1385, 1408, 1499, 1504, 1585,
 1667, 2132, 2806, 2820, 2823, 2830,
 2834, 3118, 3150, 3168, 3480, 3491,
 3665, 3759, 3846, 4060, 4526, 4541,
 4595, 4634, 4718, 4726, 4861, 4883,
 4907, 4914, 4968, 4975, 5049, 5068,
 5079, 5104, 5112, 5120, 5126, 5138,
 5149, 5275, 5388, 5393, 5458, 6491,
 8751, 8768, 8812, 8897, 9345, 9384,
 9397, 10632, 10640, 10747, 10751,
 10781, 10782, 10815, 10830, 10832,
 10834, 10852, 11070, 11075, 11193,
 11251, 11252, 11300, 11323, 11345,
 11346, 11678, 11679, 11978, 11979,
 12345, 12346, 12347, 12348, 12381,
 12382, 12383, 12384, 12738, 12754,
 13271, 13301, 13392, 14004, 14205,
 14304, 15530, 16048, 16052, 16069,
 16277, 16278, 16889, 16890, 16895,
 16899, 21538, 21592, 21666, 22173,
 22472, 24188, 25942, 26092, 28456,
 28529, 29116, 29872, 29875, 31373,
 31375, 31379, 31381, 31386, 31388,
 31558, 31842, 31870, 31881, 31884
 \tl_to_uppercase:n 32077
 \tl_trim_spaces:N 53, 4117
 \tl_trim_spaces:n 52,
 711, 4117, 7474, 13182, 13184, 31889
 \tl_trim_spaces_apply:nN
 ... 53, 686, 4117, 9580, 10120, 11170
 \tl_trim_spaces:n 397
 \tl_upper_case:n 32175
 \tl_upper_case:nn 32175
 \tl_use:N .. 51, 177, 181, 184, 4063,
 5613, 5617, 5659, 9407, 9502, 14883
 \g_tmpa_tl 59, 4545
 \l_tmpa_tl 5, 45, 59,
 1199, 1201, 1218, 1301, 1303, 1307,
 1309, 1313, 1315, 1319, 1321, 4547
 \g_tmpb_tl 59, 4545
 \l_tmpb_tl 59, 1200,
 1201, 1216, 1218, 1302, 1303, 1308,
 1309, 1314, 1315, 1320, 1321, 4547
 tl internal commands:
 __tl_act:NNNnn
 398, 398, 399, 4089, 4169, 4220
 __tl_act_count_group:nn 4085
 __tl_act_count_normal:nN 4085
 __tl_act_count_space:n 4085
 __tl_act_end:w 4169
 __tl_act_end:wn 395, 4190, 4196
 __tl_act_group:nwnNNN 4169
 __tl_act_loop:w 4169
 __tl_act_normal:NwnNNN 4169
 __tl_act_output:n 399, 4169

__tl_act_result:n
 398, 4174, 4196, 4211, 4212, 4213, 4214
 __tl_act_reverse 399
 __tl_act_reverse_output:n
 4169, 4230, 4232, 4234
 __tl_act_space:wwnNNN 398, 4169
 __tl_analysis:n
 956, 965, 23088, 23367, 23396, 23404
 __tl_analysis_a:n 23092, 23117
 __tl_analysis_a_bgroup:w
 23148, 23170
 __tl_analysis_a_cs:ww 23227
 __tl_analysis_a_egroup:w
 23150, 23170
 __tl_analysis_a_group:nw 23170
 __tl_analysis_a_group_aux:w . 23170
 __tl_analysis_a_group_auxii:w 23170
 __tl_analysis_a_group_test:w . 23170
 __tl_analysis_a_loop:w .. 23124,
 23127, 23168, 23210, 23224, 23242
 __tl_analysis_a_safe:N
 23149, 23191, 23227
 __tl_analysis_a_space:w 23147, 23153
 __tl_analysis_a_space_test:w ...
 958, 23153
 __tl_analysis_a_store:
 958, 23164, 23206, 23212
 __tl_analysis_a_type:w 23128, 23129
 __tl_analysis_b:n 23093, 23255
 __tl_analysis_b_char:Nww
 23282, 23288
 __tl_analysis_b_cs:Nww 23284, 23312
 __tl_analysis_b_cs_test:ww .. 23312
 __tl_analysis_b_loop:w
 964, 23255, 23358, 23363
 __tl_analysis_b_normal:wwN
 23268, 23333
 __tl_analysis_b_normals:ww
 962, 963, 23265, 23268, 23309, 23319
 __tl_analysis_b_special:w
 23271, 23330
 __tl_analysis_b_special_char:wN
 23330
 __tl_analysis_b_special_space:w
 23330
 \l__tl_analysis_char_token
 953, 958,
 959, 23059, 23157, 23162, 23200, 23205
 __tl_analysis_cs_space_count:NN
 23072, 23241, 23315
 __tl_analysis_cs_space_count:w .
 23072
 __tl_analysis_cs_space_count_-
 end:w 23072
 __tl_analysis_disable:n
 23097, 23119, 23185, 23238
 __tl_analysis_extract_charcode:
 23066, 23180
 __tl_analysis_extract_charcode_-
 aux:w 23066
 \l__tl_analysis_index_int
 960, 961,
 23062, 23122, 23125, 23163, 23181,
 23218, 23221, 23247, 23249, 23336
 __tl_analysis_map_inline_aux:Nn
 23365
 __tl_analysis_map_inline_-
 aux:nnn 23365
 \l__tl_analysis_nesting_int
 957, 23063, 23123, 23214, 23223
 \l__tl_analysis_normal_int
 23061, 23121, 23166, 23208,
 23219, 23222, 23239, 23248, 23253
 \g__tl_analysis_result_tl
 964, 23065, 23257, 23387, 23410
 __tl_analysis_show:
 23398, 23406, 23408
 __tl_analysis_show_active:n ...
 23423, 23452
 __tl_analysis_show_cs:n 23419, 23452
 \c__tl_analysis_show_etc_str ...
 967, 23472, 23474, 23479
 __tl_analysis_show_long:nn .. 23452
 __tl_analysis_show_long_-
 aux:nnnn 23452, 23458
 __tl_analysis_show_loop:wNw . 23408
 __tl_analysis_show_normal:n ...
 23426, 23432
 __tl_analysis_show_value:N
 23437, 23461
 \l__tl_analysis_token
 953, 954, 957, 958,
 959, 23059, 23069, 23128, 23132,
 23135, 23138, 23186, 23190, 23205
 \l__tl_analysis_type_int
 957, 960, 23064,
 23131, 23146, 23214, 23216, 23220
 __tl_build_begin:NN .. 31577, 31622
 __tl_build_begin:NNN .. 1179, 31577
 __tl_build_end_loop:NN 31665
 __tl_build_get:NNN
 31651, 31667, 31672
 __tl_build_get:w 31651
 __tl_build_get_end:w 31651
 __tl_build_last:NNn
 1179, 1179, 31589, 31594, 31655
 __tl_build_put:nn 1179, 31594, 31646
 __tl_build_put:nw 1179, 31594

- _tl_build_put_left:NNn [31634](#)
- _tl_case:NnTF [3967](#), [3972](#), [3977](#), [3982](#), [3984](#)
- _tl_case:nnTF [3964](#)
- _tl_case:Nw [3964](#)
- _tl_case_end:nw [3964](#)
- _tl_count:n [395](#), [4072](#)
- _tl_head_auxi:nw [4241](#)
- _tl_head_auxii:n [4241](#)
- _tl_if_blank_p:NNw [3823](#)
- _tl_if_empty_if:n [370](#), [387](#), [388](#), [3825](#), [3854](#), [3953](#), [3957](#)
- _tl_if_head_eq_meaning_-normal:nN [4316](#), [4320](#)
- _tl_if_head_eq_meaning_-special:nN [4317](#), [4329](#)
- _tl_if_head_is_N_type:w . [403](#), [4348](#)
- _tl_if_head_is_space:w [4376](#)
- _tl_if_novalue:w [3921](#)
- _tl_if_recursion_tail_break:nN . . . [404](#), [3643](#), [4007](#), [4036](#), [4050](#), [4404](#)
- _tl_if_recursion_tail_stop:nTF [3643](#)
- _tl_if_recursion_tail_stop_p:n [3643](#)
- _tl_if_single:nnw . [391](#), [3941](#), [3949](#)
- _tl_if_single:nTF [3939](#)
- _tl_if_single_p:n [3939](#)
- \l_tl_internal_a_tl [408](#), [3648](#), [3649](#), [3650](#), [3877](#), [3895](#), [3899](#), [4529](#), [4535](#), [32149](#), [32150](#), [32158](#), [32160](#), [32162](#), [32169](#), [32171](#)
- \l_tl_internal_b_tl [3877](#), [3882](#), [3885](#), [3896](#), [3899](#)
- _tl_item:nn [4388](#)
- _tl_item_aux:nn [4388](#)
- _tl_map_function:Nn [393](#), [3997](#), [4016](#)
- _tl_map_tokens:nn [4025](#)
- _tl_map_variable:Nnn [4040](#)
- _tl_quark_if_nil:n [3644](#)
- _tl_quark_if_nil:nTF [3767](#)
- _tl_range:Nnnn . . [4418](#), [31685](#), [31690](#)
- _tl_range:nnNn [4418](#)
- _tl_range:nnnNn [4418](#)
- _tl_range:w [405](#), [4418](#)
- _tl_range_braced:w [405](#), [1182](#), [31683](#)
- _tl_range_collect:nn . . . [1182](#), [4418](#)
- _tl_range_collect_braced:w . . . [405](#), [1182](#), [31683](#)
- _tl_range_collect_group:nN . [4418](#)
- _tl_range_collect_group:nn . . . [4486](#), [4495](#)
- _tl_range_collect_N:nN [4418](#)
- _tl_range_collect_space:nw . [4418](#)
- _tl_range_collect_unbraced:w [31683](#)
- _tl_range_items:nnNn [405](#)
- _tl_range_normalize:nn [4433](#), [4437](#), [4497](#)
- _tl_range_skip:w [405](#), [4418](#)
- _tl_range_skip_spaces:n [4418](#)
- _tl_range_unbraced:w [31683](#)
- _tl_replace:NnNNNnn [384](#), [385](#), [3743](#), [3745](#), [3747](#), [3749](#), [3754](#)
- _tl_replace_auxi:NnnNNNnn [385](#), [3754](#)
- _tl_replace_auxii:nNNNnn [384](#), [385](#), [386](#), [3754](#)
- _tl_replace_next:w [384](#), [386](#), [3747](#), [3749](#), [3754](#)
- _tl_replace_wrap:w [384](#), [386](#), [3743](#), [3745](#), [3754](#)
- _tl_rescan:NNw [381](#), [3646](#), [3728](#), [3733](#)
- \c_tl_rescan_marker_tl [383](#), [3645](#), [3670](#), [3678](#), [3708](#), [3740](#)
- _tl_reverse_group_preserve:nn [4215](#)
- _tl_reverse_items:nwNwn [4101](#)
- _tl_reverse_items:wn [4101](#)
- _tl_reverse_normal:nN [4215](#)
- _tl_reverse_space:n [4215](#)
- _tl_set_rescan:nNN [381](#), [382](#), [3665](#), [3687](#)
- _tl_set_rescan:NNnn [381](#), [3646](#)
- _tl_set_rescan_multi:nNN [381](#), [382](#), [383](#), [3646](#), [3695](#), [3717](#)
- _tl_set_rescan_single:nnNN [382](#), [3687](#)
- _tl_set_rescan_single:NNww . . [383](#)
- _tl_set_rescan_single_aux:nnnNN [3687](#)
- _tl_set_rescan_single_aux:w [383](#), [3687](#)
- _tl_show:n [4525](#)
- _tl_show:NN [4516](#)
- _tl_show:w [4525](#)
- _tl_tl_head:w [4241](#), [4282](#), [4299](#), [4323](#)
- _tl_tmp:w [390](#), [396](#), [3914](#), [3915](#), [3921](#), [3934](#), [4129](#), [4166](#)
- _tl_trim_spaces:nn [686](#), [4118](#), [4121](#), [4129](#)
- _tl_trim_spaces_auxi:w . . [397](#), [4129](#)
- _tl_trim_spaces_auxii:w . [397](#), [4129](#)
- _tl_trim_spaces_auxiii:w [397](#), [4129](#)
- _tl_trim_spaces_auxiv:w . [397](#), [4129](#)
- token commands:
 - \c_alignment_token [135](#), [574](#), [10584](#), [10645](#), [10684](#), [23298](#), [29123](#)
 - \c_parameter_token [135](#), [574](#), [1035](#), [10645](#), [10688](#), [10691](#)
 - \g_peek_token . [139](#), [139](#), [10903](#), [10917](#)

- \l_peek_token [139](#), [139](#), [584](#),
[586](#), [1183](#), [10903](#), [10915](#), [10932](#),
[10971](#), [10983](#), [11003](#), [11053](#), [11054](#),
[11055](#), [11058](#), [31744](#), [31745](#), [31746](#)
- \c_space_token [34](#), [55](#), [58](#), [135](#), [142](#),
[274](#), [402](#), [575](#), [2664](#), [4304](#), [4341](#),
[10593](#), [10645](#), [10708](#), [10932](#), [11055](#),
[12928](#), [23132](#), [23162](#), [23304](#), [23838](#),
[23873](#), [29095](#), [29132](#), [29259](#), [31746](#)
- \token_get_arg_spec:N [32193](#)
- \token_get_prefix_spec:N [32193](#)
- \token_get_replacement_spec:N . [32193](#)
- \token_if_active:NTF [137](#), [10721](#), [29835](#)
- \token_if_active_p:N
[137](#), [10721](#), [13132](#), [29473](#), [29810](#), [30779](#)
- \token_if_alignment:NTF
. [136](#), [136](#), [10682](#)
- \token_if_alignment_p:N . . [136](#), [10682](#)
- \token_if_chardef:NTF
. [138](#), [10793](#), [23441](#)
- \token_if_chardef_p:N
. [138](#), [10793](#), [29063](#)
- \token_if_cs:NTF
[137](#), [10758](#), [29362](#), [29686](#), [29957](#), [30786](#)
- \token_if_cs_p:N [137](#), [10758](#), [29472](#),
[30090](#), [30143](#), [30171](#), [30216](#), [30778](#)
- \token_if_dim_register:NTF
. [138](#), [10793](#), [23443](#)
- \token_if_dim_register_p:N [138](#), [10793](#)
- \token_if_eq_catcode:NNTF . . [137](#),
[139](#), [140](#), [140](#), [140](#), [274](#), [2664](#), [10731](#)
- \token_if_eq_catcode_p:NN [137](#), [10731](#)
- \token_if_eq_charcode:NNTF
. [137](#), [140](#), [140](#), [140](#), [141](#),
[274](#), [10736](#), [12240](#), [12928](#), [20137](#),
[23873](#), [23878](#), [24501](#), [24701](#), [24714](#),
[24716](#), [24754](#), [24876](#), [25846](#), [25925](#)
- \token_if_eq_charcode_p:NN [137](#), [10736](#)
- \token_if_eq_meaning:NNTF
. [137](#), [141](#), [141](#), [141](#),
[141](#), [274](#), [2667](#), [2678](#), [10726](#), [12980](#),
[16396](#), [17411](#), [17470](#), [18405](#), [18407](#),
[18412](#), [18476](#), [18662](#), [20735](#), [24195](#),
[24507](#), [24540](#), [24689](#), [24712](#), [24744](#),
[24871](#), [24874](#), [25896](#), [25923](#), [25964](#),
[25981](#), [29294](#), [29300](#), [29319](#), [29345](#),
[29486](#), [29640](#), [29666](#), [30694](#), [30735](#)
- \token_if_eq_meaning_p:NN
. [137](#), [10726](#), [29158](#), [29259](#), [29261](#),
[29265](#), [29275](#), [29276](#), [29283](#), [29284](#)
- \token_if_expandable:NTF
. [137](#), [10763](#), [23439](#), [29152](#)
- \token_if_expandable_p:N
. [137](#), [10763](#), [13124](#)
- \token_if_group_begin:NTF [136](#), [10667](#)
- \token_if_group_begin_p:N [136](#), [10667](#)
- \token_if_group_end:NTF . . [136](#), [10672](#)
- \token_if_group_end_p:N . . [136](#), [10672](#)
- \token_if_int_register:NTF
. [138](#), [10793](#), [23444](#)
- \token_if_int_register_p:N [138](#), [10793](#)
- \token_if_letter:N [577](#)
- \token_if_letter:NTF
. [137](#), [10711](#), [29792](#), [29806](#)
- \token_if_letter_p:N [137](#), [10711](#), [29809](#)
- \token_if_long_macro:NTF . [137](#), [10793](#)
- \token_if_long_macro_p:N . [137](#), [10793](#)
- \token_if_macro:NTF
. [137](#), [2137](#), [2146](#), [2155](#), [10741](#), [10847](#)
- \token_if_macro_p:N [137](#), [10741](#)
- \token_if_math_subscript:NTF
. [136](#), [10701](#)
- \token_if_math_subscript_p:N
. [136](#), [10701](#)
- \token_if_math_superscript:NTF
. [136](#), [10695](#)
- \token_if_math_superscript_p:N
. [136](#), [10695](#)
- \token_if_math_toggle:NTF [136](#), [10677](#)
- \token_if_math_toggle_p:N [136](#), [10677](#)
- \token_if_mathchardef:NTF
. [138](#), [10793](#), [23442](#)
- \token_if_mathchardef_p:N
. [138](#), [10793](#), [29064](#)
- \token_if_muskip_register:NTF
. [138](#), [10793](#)
- \token_if_muskip_register_p:N
. [138](#), [10793](#)
- \token_if_other:NTF [137](#), [10716](#)
- \token_if_other_p:N [137](#), [10716](#)
- \token_if_parameter:NTF . . [136](#), [10687](#)
- \token_if_parameter_p:N . . [136](#), [10687](#)
- \token_if_primitive:NTF . . [139](#), [10841](#)
- \token_if_primitive_p:N . . [139](#), [10841](#)
- \token_if_protected_long_macro:NTF [138](#), [2561](#), [10793](#)
- \token_if_protected_long_macro_p:N [138](#), [10793](#), [13131](#), [29157](#)
- \token_if_protected_macro:NTF
. [137](#), [2560](#), [10793](#)
- \token_if_protected_macro_p:N
. [137](#), [10793](#), [13130](#), [29156](#)
- \token_if_skip_register:NTF
. [138](#), [10793](#), [23445](#)
- \token_if_skip_register_p:N
. [138](#), [10793](#)
- \token_if_space:NTF [136](#), [10706](#)
- \token_if_space_p:N [136](#), [10706](#)

- \token_if_toks_register:NTF 139, 355, 2763, 10793, 23446
 - \token_if_toks_register_p:N 139, 10793
 - \token_new:Nn 32079
 - \token_to_meaning:N 15, 135, 143, 576, 579, 1383, 1399, 1838, 2140, 2149, 2158, 2769, 2820, 10645, 10747, 10814, 10851, 11058, 23069, 23435, 23460, 29100
 - \token_to_str:N 5, 17, 60, 135, 143, 165, 323, 403, 505, 578, 758, 760, 1038, 1385, 1399, 1399, 1563, 1572, 1604, 1627, 1675, 1680, 1695, 1716, 1717, 1737, 1838, 1964, 1999, 2006, 2094, 2114, 2127, 2746, 2840, 2925, 2940, 2955, 2962, 2988, 2997, 3048, 3114, 3135, 3326, 3350, 3354, 3369, 3499, 3645, 4353, 4369, 4523, 4831, 5249, 5257, 5260, 7602, 8036, 8274, 8996, 9051, 9336, 9420, 10145, 10501, 10505, 10527, 10530, 10538, 10541, 10617, 10618, 10625, 10626, 10645, 10828, 10829, 10834, 10835, 10836, 10837, 10838, 10839, 11436, 12800, 12801, 12802, 12803, 12804, 12811, 13138, 13197, 13328, 13542, 15690, 15732, 15806, 16047, 16062, 16283, 16284, 16773, 16774, 16803, 16970, 17021, 17053, 17073, 17088, 17100, 17101, 17114, 17115, 17140, 17149, 17151, 17176, 17179, 17204, 17206, 17220, 17236, 17254, 17324, 17334, 17335, 17350, 17351, 17684, 17728, 17920, 18159, 22147, 22467, 22496, 22502, 23010, 23026, 23077, 23158, 23201, 23231, 23280, 23291, 23293, 23295, 23305, 23341, 23352, 23398, 23434, 23459, 23480, 23784, 23791, 23892, 23896, 24619, 25869, 26101, 26680, 26682, 26843, 27431, 27648, 28599, 29074, 29075, 29469, 29477, 29504, 29505, 29731, 29734, 29743, 30575, 30577, 30595, 30596, 30609, 30612, 30616, 30619, 30775, 30783, 30813, 30814, 30952, 30955, 30959, 30968, 31020, 31841, 31869, 31881, 31884
 - token internal commands:
 - \c_token_A_int 10841, 10878
 - __token_delimit_by_char":w . . 10775
 - __token_delimit_by_count:w . . 10775
 - __token_delimit_by_dimen:w . . 10775
 - __token_delimit_by_macro:w . . 10775
 - __token_delimit_by_muskip:w . 10775
 - __token_delimit_by_skip:w . . 10775
 - __token_delimit_by_toks:w . . 10775
 - __token_if_macro_p:w 10741
 - __token_if_primitive:NNw 10841
 - __token_if_primitive:Nw 10841
 - __token_if_primitive_loop:N . 10841
 - __token_if_primitive_nullfont:N 10841
 - __token_if_primitive_space:w . 10841
 - __token_if_primitive_undefined:N 10841
 - __token_tmp:w 578, 10776, 10785, 10786, 10787, 10788, 10789, 10790, 10791, 10794, 10828, 10829, 10830, 10831, 10833, 10835, 10836, 10837, 10838, 10839
 - \toks 463, 10839
 - \toksapp 903
 - \toksdef 464, 23006
 - \tokspre 904
 - \tolerance 465
 - \topmark 466
 - \topmarks 568
 - \topskip 467
 - \tpack 905
 - \tracingassigns 569
 - \tracingcommands 468
 - \tracingfonts 936
 - \tracinggroups 570
 - \tracingifs 571
 - \tracinglostchars 469
 - \tracingmacros 470
 - \tracingnesting 572
 - \tracingonline 471
 - \tracingoutput 472
 - \tracingpages 473
 - \tracingparagraphs 474
 - \tracingrestores 475
 - \tracingscantokens 573
 - \tracingstats 476
 - true 219
 - trunc 215
 - \ttfamily 30850
 - two commands:
 - \c_thirty_two 31951
 - \c_two_hundred_fifty_five . . . 31955
 - \c_two_hundred_fifty_six 31957
- U
- \u xxii, 1007, 29167, 31003, 31084, 31085, 31100, 31101, 31110, 31111, 31124, 31125, 31126, 31152, 31153, 31178, 31179

<code>\uccode</code>	477	<code>\Umathlimitabovekern</code>	998
<code>\Uchar</code>	938	<code>\Umathlimitabovevgap</code>	999
<code>\Ucharcat</code>	939	<code>\Umathlimitbelowbgap</code>	1000
<code>\uchyph</code>	478	<code>\Umathlimitbelowkern</code>	1001
<code>\ucs</code>	1172	<code>\Umathlimitbelowvgap</code>	1002
<code>\Udelcode</code>	940	<code>\Umathnolimitsubfactor</code>	1003
<code>\Udelcodenum</code>	941	<code>\Umathnolimitsupfactor</code>	1004
<code>\Udelimiter</code>	942	<code>\Umathopbinspacing</code>	1005
<code>\Udelimiterover</code>	943	<code>\Umathopcloseespacing</code>	1006
<code>\Udelimiterunder</code>	944	<code>\Umathopenbinspacing</code>	1007
<code>\Uhextensible</code>	945	<code>\Umathopencloseespacing</code>	1008
<code>\Umathaccent</code>	946	<code>\Umathopeninnerspacing</code>	1009
<code>\Umathaxis</code>	947	<code>\Umathopenopenspacing</code>	1010
<code>\Umathbinbinspacing</code>	948	<code>\Umathopenopspacing</code>	1011
<code>\Umathbincloseespacing</code>	949	<code>\Umathopenordspacing</code>	1012
<code>\Umathbininnerspacing</code>	950	<code>\Umathopenpunctspacing</code>	1013
<code>\Umathbinopenspacing</code>	951	<code>\Umathopenrelspacing</code>	1014
<code>\Umathbinopspacing</code>	952	<code>\Umathoperatorsize</code>	1015
<code>\Umathbinordspacing</code>	953	<code>\Umathopinnerspacing</code>	1016
<code>\Umathbinpunctspacing</code>	954	<code>\Umathopopenspacing</code>	1017
<code>\Umathbinrelspacing</code>	955	<code>\Umathopopspacing</code>	1018
<code>\Umathchar</code>	956	<code>\Umathopordspacing</code>	1019
<code>\Umathcharclass</code>	957	<code>\Umathoppunctspacing</code>	1020
<code>\Umathchardef</code>	958	<code>\Umathoprelspacing</code>	1021
<code>\Umathcharfam</code>	959	<code>\Umathordbinspacing</code>	1022
<code>\Umathcharnum</code>	960	<code>\Umathordcloseespacing</code>	1023
<code>\Umathcharnumdef</code>	961	<code>\Umathordinnerspacing</code>	1024
<code>\Umathcharslot</code>	962	<code>\Umathordopenspacing</code>	1025
<code>\Umathclosebinspacing</code>	963	<code>\Umathordopspacing</code>	1026
<code>\Umathclosecloseespacing</code>	964	<code>\Umathordordspacing</code>	1027
<code>\Umathcloseinnerspacing</code>	966	<code>\Umathordpunctspacing</code>	1028
<code>\Umathcloseopenspacing</code>	968	<code>\Umathordrelspacing</code>	1029
<code>\Umathcloseopspacing</code>	969	<code>\Umathoverbarkern</code>	1030
<code>\Umathcloseordspacing</code>	970	<code>\Umathoverbarrule</code>	1031
<code>\Umathclosepunctspacing</code>	971	<code>\Umathoverbarvgap</code>	1032
<code>\Umathcloserelspacing</code>	973	<code>\Umathoverdelimiterbgap</code>	1033
<code>\Umathcode</code>	974	<code>\Umathoverdelimitervgap</code>	1035
<code>\Umathcodenum</code>	975	<code>\Umathpunctbinspacing</code>	1037
<code>\Umathconnectoroverlapmin</code>	976	<code>\Umathpunctcloseespacing</code>	1038
<code>\Umathfractiondelsize</code>	978	<code>\Umathpunctinnerspacing</code>	1040
<code>\Umathfractiondenomdown</code>	979	<code>\Umathpunctopenspacing</code>	1042
<code>\Umathfractiondenomvgap</code>	981	<code>\Umathpunctopspacing</code>	1043
<code>\Umathfractionnumup</code>	983	<code>\Umathpunctordspacing</code>	1044
<code>\Umathfractionnumvgap</code>	984	<code>\Umathpunctpunctspacing</code>	1045
<code>\Umathfractionrule</code>	985	<code>\Umathpunctrelspacing</code>	1047
<code>\Umathinnerbinspacing</code>	986	<code>\Umathquad</code>	1048
<code>\Umathinnercloseespacing</code>	987	<code>\Umathradicaldegreeafter</code>	1049
<code>\Umathinnerinnerspacing</code>	989	<code>\Umathradicaldegreebefore</code>	1051
<code>\Umathinneropenspacing</code>	991	<code>\Umathradicaldegreeraise</code>	1053
<code>\Umathinneropspacing</code>	992	<code>\Umathradicalkern</code>	1055
<code>\Umathinnerordspacing</code>	993	<code>\Umathradicalrule</code>	1056
<code>\Umathinnerpunctspacing</code>	994	<code>\Umathradicalvgap</code>	1057
<code>\Umathinnerrelspacing</code>	996	<code>\Umathrelbinspacing</code>	1058
<code>\Umathlimitabovebgap</code>	997	<code>\Umathrelcloseespacing</code>	1059

<code>\Umathrelinnerspacing</code>	1060	11551, 11742, 11764, 11781, 11787,
<code>\Umathreloopenspacing</code>	1061	11794, 11848, 12885, 13414, 13527,
<code>\Umathreloppspacing</code>	1062	14007, 14705, 14711, 14935, 24096,
<code>\Umathrelordspacing</code>	1063	25852, 25991, 28648, 29563, 29688,
<code>\Umathrelpunctspacing</code>	1064	29708, 29735, 29745, 29812, 29815,
<code>\Umathrelrelspacing</code>	1065	29839, 29841, 29878, 29901, 29922,
<code>\Umathskewedfractionhgap</code>	1066	29944, 30137, 30145, 30146, 30165,
<code>\Umathskewedfractionvgap</code>	1068	30180, 30182, 30276, 30870, 30871
<code>\Umathspaceafterscript</code>	1070	<code>\use:n</code> 19, 20,
<code>\Umathstackdenomdown</code>	1071	20, 43, 142, 313, 362, 381, 493, 557,
<code>\Umathstacknumup</code>	1072	620, 708, 758, 938, 948, 1026, 1065,
<code>\Umathstackvgap</code>	1073	1452, 1458, 1460, 1463, 1532, 1549,
<code>\Umathsubshiftdown</code>	1074	1575, 1635, 1644, 1661, 1907, 1984,
<code>\Umathsubshiftdrop</code>	1075	2129, 2377, 2801, 2850, 2987, 3018,
<code>\Umathsubsupshiftdown</code>	1076	3104, 3452, 4037, 4052, 4332, 4630,
<code>\Umathsubsupvgap</code>	1077	4690, 4717, 4725, 4815, 4836, 4850,
<code>\Umathsubtopmax</code>	1078	5522, 7866, 9233, 9240, 9998, 10382,
<code>\Umathsubbottommin</code>	1079	10659, 10741, 10778, 10796, 10842,
<code>\Umathsupshiftdrop</code>	1080	11419, 11675, 11692, 11975, 11992,
<code>\Umathsupshiftup</code>	1081	12331, 12404, 12535, 12707, 13358,
<code>\Umathsupsubbottommax</code>	1082	13585, 14202, 14492, 14543, 15108,
<code>\Umathunderbarkern</code>	1083	15159, 15201, 15220, 15425, 15459,
<code>\Umathunderbarrule</code>	1084	15480, 16319, 16327, 16336, 16353,
<code>\Umathunderbarvgap</code>	1085	16361, 16389, 16854, 18397, 22469,
<code>\Umathunderdelimiterbgap</code>	1086	22549, 23449, 23641, 24057, 24060,
<code>\Umathunderdelimitervgap</code>	1088	24186, 24718, 24952, 25010, 25089,
undefine commands:		25469, 25534, 25574, 25654, 26377,
<code>.undefine:</code>	191, 15100	26394, 26838, 27857, 28415, 28425,
<code>\underline</code>	479	28970, 28971, 28973, 29611, 29675,
<code>\unexpanded</code>	574, 2648, 2672	30307, 30326, 30562, 30844, 30879
<code>\unhbox</code>	480	<code>\use:nn</code> 19,
<code>\unhcopy</code>	481	1463, 2210, 3677, 3738, 5305, 9370,
<code>\uniformdeviate</code>	937	9982, 10566, 13231, 14003, 16884,
<code>\unkern</code>	482	16893, 16897, 20317, 22192, 23417,
<code>\unless</code>	575	29053, 31378, 31380, 31385, 31387
<code>\Unosubscript</code>	1090	<code>\use:nnn</code> 19, 1463, 1961, 10572
<code>\Unosuperscript</code>	1091	<code>\use:nnnn</code> 19, 1463
<code>\unpenalty</code>	483	<code>\use_i:nn</code> 19, 312, 317, 318, 319, 374,
<code>\unskip</code>	484	591, 598, 879, 882, 896, 900, 901,
<code>\unvbox</code>	485	1403, 1467, 1517, 1593, 1615, 1753,
<code>\unvcopy</code>	486	1781, 1940, 2659, 2689, 2702, 2747,
<code>\Uoverdelimiter</code>	1092	3021, 3092, 3441, 3741, 4255, 5820,
<code>\uppercase</code>	487	5825, 5906, 5910, 7482, 7484, 7858,
<code>\upshape</code>	30856	7901, 7938, 9235, 11199, 11412,
<code>\uptexrevision</code>	1173	15684, 15686, 16028, 16664, 16854,
<code>\uptexversion</code>	1174	18225, 18561, 18856, 19344, 19627,
<code>\Uradical</code>	1093	20146, 20312, 20625, 20635, 20639,
<code>\Uroot</code>	1094	21147, 21352, 21913, 21938, 22834,
use commands:		22889, 22899, 22909, 23233, 24017,
<code>\use:N</code>	16,	24028, 24037, 24040, 24049, 31789
104, 319, 1451, 1599, 1690, 1803,		<code>\use_i:nnn</code> 19, 443, 1469, 2140,
1805, 1807, 1809, 5168, 8272, 8708,		3126, 5291, 5828, 6332, 7731, 8884,
8718, 8823, 8827, 8829, 8831, 8832,		16823, 18813, 20287, 22126, 25900
8836, 9027, 9049, 11538, 11548,		<code>\use_i:nnnn</code> 19, 303, 527, 528, 1469,

- 9022, 9024, 9038, 9043, 9059, 9061,
 18395, 18831, 18838, 19031, 21924
 \use_i_delimit_by_q_nil:nw . 21, 1481
 \use_i_delimit_by_q_recursion_-
 stop:nw 21, 1481, 3240, 3256
 \use_i_delimit_by_q_recursion_-
 stop:w 40, 40
 \use_i_delimit_by_q_stop:nw 21, 1481
 \use_ii:nnn . 20, 318, 319, 1469,
 1584, 2236, 3470, 7707, 7812, 11380
 \use_ii:nn 19, 114, 312, 317,
 374, 591, 701, 706, 879, 882, 896,
 900, 901, 912, 1011, 1405, 1467,
 1519, 1617, 1636, 1652, 1755, 1783,
 1938, 2164, 2661, 2704, 2749, 3443,
 3690, 4257, 9241, 11200, 16229,
 16252, 16666, 18038, 18225, 18226,
 18858, 20148, 20631, 20637, 20641,
 21149, 21354, 21784, 21915, 23235,
 24019, 24025, 24030, 24042, 24051,
 24548, 24670, 24841, 25029, 31800
 \use_ii:nnn . 19, 320, 1469, 1652, 2149
 \use_ii:nnnn . 19, 527, 528, 1469, 9038
 \use_ii_i:nn 20, 433, 1477, 5310, 5395
 \use_iii:nnn 19, 1469, 2158, 2169, 16034
 \use_iii:nnnn 19,
 527, 528, 1469, 9038, 9060, 9062, 9063
 \use_iv:nnnn
 19, 527, 528, 1469, 9038, 9058, 18026
 \use_none:n 20, 375, 387,
 396, 397, 452, 550, 553, 603, 637,
 702, 708, 754, 755, 759, 759, 964,
 1485, 1583, 1635, 1636, 1909, 1965,
 2549, 2680, 3059, 3060, 3242, 3257,
 3381, 3397, 3467, 3724, 3803, 3825,
 3953, 4164, 4254, 4271, 4335, 4352,
 4362, 4363, 4368, 4383, 4386, 4475,
 4484, 5246, 5269, 5285, 5297, 5330,
 5463, 5513, 5764, 5774, 5812, 5874,
 6038, 6120, 6225, 6397, 7380, 7716,
 8578, 8584, 8874, 9234, 9239, 9429,
 9609, 9653, 9750, 9845, 9872, 9911,
 10891, 11607, 11814, 12041, 12045,
 12858, 12913, 12969, 13292, 13756,
 13759, 14807, 15402, 15784, 16023,
 16172, 16176, 16180, 16184, 17485,
 17738, 17745, 17762, 17781, 17804,
 17872, 17913, 18038, 18053, 18074,
 18075, 18287, 18288, 18832, 18835,
 19815, 21508, 21793, 23231, 23280,
 23378, 23416, 23643, 23905, 24063,
 24715, 30824, 30876, 31470, 31471
 \use_none:nn
 20, 386, 391, 401, 402, 489,
 1485, 1565, 1573, 1644, 3158, 3786,
 3943, 4116, 4282, 4299, 5354, 5685,
 7554, 7738, 9017, 9578, 9886, 12914,
 12958, 16088, 16171, 16175, 16179,
 16183, 21503, 25277, 25913, 30825
 \use_none:nnn
 . 20, 402, 1485, 2926, 2941, 3370,
 4323, 12915, 14921, 14930, 16170,
 16174, 16178, 16182, 16823, 23447
 \use_none:nnnn
 20, 1485, 12916, 14134, 30827
 \use_none:nnnnn 20, 315,
 638, 639, 740, 1485, 12917, 12927,
 16314, 16348, 16374, 16382, 18421
 \use_none:nnnnnn
 20, 1485, 1697, 12918, 31678
 \use_none:nnnnnnn
 20, 740, 1485, 16316,
 16350, 16376, 16384, 16707, 18872
 \use_none:nnnnnnnn
 20, 319, 1485, 1606, 3007
 \use_none:nnnnnnnnn 20, 1485
 \use_none_delimit_by_q_nil:w 21, 1478
 \use_none_delimit_by_q_recursion_-
 stop:w
 21, 40, 40, 317, 1478, 3234, 3249
 \use_none_delimit_by_q_stop:w . . .
 21, 376, 451, 1478
 \use_none_delimit_by_s_stop:w . . .
 42, 42, 3515
 \useboxresource 930
 \usefont 30827
 \useimageresource 931
 \Uskewed 1095
 \Uskewedwithdelims 1096
 \Ustack 1097
 \Ustartdisplaymath 1098
 \Ustartmath 1099
 \Ustopdisplaymath 1100
 \Ustopmath 1101
 \Usubscript 1102
 \Usuperscript 1103
 \Uunderdelimater 1104
 \Uvextensible 1105

 V
 \v 29167, 31008,
 31094, 31095, 31096, 31097, 31106,
 31107, 31140, 31141, 31148, 31149,
 31160, 31161, 31168, 31169, 31172,
 31173, 31195, 31196, 31197, 31198,
 31199, 31200, 31201, 31202, 31203,
 31204, 31205, 31206, 31207, 31208,
 31209, 31212, 31213, 31222, 31223

\adjust	488	W	
\valign	489	\wd	504
value commands:		\widowpenalties	576
.value_forbidden:n	191, 15102	\widowpenalty	505
.value_required:n	191, 15102	\write	506
\vbadness	490		
\vbox	491	X	
vbox commands:		\xdef	507
\vbox:n	241, 241, 245, 26925	xetex commands:	
\vbox_gset:Nn	245, 26939, 27500	\xetex_if_engine:TF	
\vbox_gset:Nw	246, 26975, 27572		32083, 32085, 32087
\vbox_gset_end:	246, 26975, 27574	\xetex_if_engine_p:	32081
\vbox_gset_split_to_ht:Nnn	246, 27014	\XeTeXcharclass	714
\vbox_gset_to_ht:Nnn	245, 26963	\XeTeXcharglyph	715
\vbox_gset_to_ht:Nnw	246, 26996	\XeTeXcountfeatures	716
\vbox_gset_top:Nn	245, 26951	\XeTeXcountglyphs	717
\vbox_set:Nn	245, 246, 26939, 27494	\XeTeXcountselectors	718
\vbox_set:Nw	246, 26975, 27565	\XeTeXcountvariations	719
\vbox_set_end:	246, 246, 26975, 27567	\XeTeXdashbreakstate	721
\vbox_set_split_to_ht:Nnn	246, 27014	\XeTeXdefaultencoding	720
\vbox_set_to_ht:Nnn	245, 246, 26963	\XeTeXfeaturecode	722
\vbox_set_to_ht:Nnw	246, 26996	\XeTeXfeaturename	723
\vbox_set_top:Nn		\XeTeXfindfeaturebyname	724
	245, 26951, 27514, 27588	\XeTeXfindselectorbyname	726
\vbox_to_ht:nn	245, 26929	\XeTeXfindvariationbyname	728
\vbox_to_zero:n	245, 26929	\XeTeXfirstfontchar	730
\vbox_top:n	245, 26925	\XeTeXfonttype	731
\vbox_unpack:N	246, 27010, 27514, 27588	\XeTeXgenerateactualtext	732
\vbox_unpack_clear:N	32120	\XeTeXglyph	734
\vbox_unpack_drop:N		\XeTeXglyphbounds	735
	247, 27010, 32120, 32122	\XeTeXglyphindex	736
\vcenter	492	\XeTeXglyphname	737
vcoffin commands:		\XeTeXinputencoding	738
\vcoffin_gset:Nnn	252, 27491	\XeTeXinputnormalization	739
\vcoffin_gset:Nnw	252, 27563	\XeTeXinterchartokenstate	741
\vcoffin_gset_end:	252, 27563	\XeTeXinterchartoks	743
\vcoffin_set:Nnn	252, 27491	\XeTeXisdefaultselector	744
\vcoffin_set:Nnw	252, 27563	\XeTeXisexclusivefeature	746
\vcoffin_set_end:	252, 27563	\XeTeXlastfontchar	748
\vfi	1164	\XeTeXlinebreaklocale	750
\vfil	493	\XeTeXlinebreakpenalty	751
\vfill	494	\XeTeXlinebreakskip	749
\vfilneg	495	\XeTeXOTcountfeatures	752
\vfuzz	496	\XeTeXOTcountlanguages	753
\voffset	497	\XeTeXOTcountscripts	754
\vpack	906	\XeTeXOTfeaturetag	755
\vrule	498	\XeTeXOTlanguagetag	756
\vsize	499	\XeTeXOTscripttag	757
\vskip	500	\XeTeXpdffile	758
\vsplit	501	\XeTeXpdfpagecount	759
\vss	502	\XeTeXpicfile	760
\vtop	503	\XeTeXrevision	761
		\XeTeXselectorname	762
		\XeTeXtracingfonts	763

<code>\XeTeXupwardsmode</code>	764	<code>\xkanjiskip</code>	1160
<code>\XeTeXuseglypmetrics</code>	765	<code>\xleaders</code>	508
<code>\XeTeXvariation</code>	766	<code>\xspaceskip</code>	509
<code>\XeTeXvariationdefault</code>	767	<code>\xspcode</code>	1161
<code>\XeTeXvariationmax</code>	768		
<code>\XeTeXvariationmin</code>	769	Y	
<code>\XeTeXvariationname</code>	770	<code>\ybaselineshift</code>	1162
<code>\XeTeXversion</code>	771	<code>\year</code>	510, 1320, 9444
		<code>\yoko</code>	1163