

LaTeX2 **Functional** Interfaces for LaTeX3 Programming Layer

Jianrui Lyu (tolvjr@163.com)
<https://github.com/lvjrr/functional>

Version 2022C (2022-04-01)

LaTeX3 programming layer (**expl3**) is very powerful for advanced users, but it is a little complicated for normal users. This **functional** package aims to provide intuitive LaTeX2 functional interfaces for it.

Although there are functions in LaTeX3, the evaluation of them is from outside to inside. With this package, the evaluation of functions is from inside to outside, which is the same as other programming languages such as **JavaScript** or **Lua**. In this way, it is rather easy to debug code too.

Note that many paragraphs in this manual are copied from the documentation of **expl3**.

Contents

| | | |
|----------|---|-----------|
| 1 | Overview of Features | 5 |
| 1.1 | Evaluation from Inside to Outside | 5 |
| 1.2 | Group Scoping of Functions | 5 |
| 1.3 | Tracing Evaluation of Functions | 6 |
| 1.4 | Definitions of Functions | 7 |
| 1.5 | Variants of Arguments | 7 |
| 2 | Functional Programming (Prg) | 9 |
| 2.1 | Defining Functions and Conditionals | 9 |
| 2.2 | Collecting Returned Values | 9 |
| 3 | Argument Using (Use) | 10 |
| 3.1 | Expanding Tokens | 10 |
| 3.2 | Using Tokens | 10 |
| 4 | Control Structures (Bool) | 12 |
| 4.1 | Constant and Scratch Booleans | 12 |
| 4.2 | Creating and Setting Booleans | 12 |
| 4.3 | Viewing Booleans | 13 |
| 4.4 | Booleans and Conditionals | 13 |
| 5 | Token Lists (Tl) | 15 |
| 5.1 | Constant and Scratch Token Lists | 15 |
| 5.2 | Creating and Using Token Lists | 16 |
| 5.3 | Viewing Token Lists | 17 |
| 5.4 | Setting Token List Variables | 17 |
| 5.5 | Replacing Tokens | 18 |
| 5.6 | Working with the Content of Token Lists | 20 |
| 5.7 | Mapping over Token Lists | 21 |
| 5.8 | Token List Conditionals | 22 |
| 5.9 | Token List Case Functions | 25 |
| 6 | Strings (Str) | 28 |
| 6.1 | Constant and Scratch Strings | 28 |
| 6.2 | Creating and Using Strings | 29 |
| 6.3 | Viewing Strings | 29 |
| 6.4 | Setting String Variables | 30 |

| | | |
|-----------|--|-----------|
| 6.5 | Modifying String Variables | 31 |
| 6.6 | Working with the Content of Strings | 32 |
| 6.7 | Mapping over Strings | 33 |
| 6.8 | String Conditionals | 34 |
| 6.9 | String Case Functions | 35 |
| 7 | Integers (Int) | 38 |
| 7.1 | Constant and Scratch Integers | 38 |
| 7.2 | Integer Expressions | 38 |
| 7.3 | Creating and Using Integers | 40 |
| 7.4 | Viewing Integers | 40 |
| 7.5 | Setting Integer Variables | 41 |
| 7.6 | Integer Step Functions | 42 |
| 7.7 | Integer Conditionals | 42 |
| 7.8 | Integer Case Functions | 43 |
| 8 | Floating Point Numbers (Fp) | 45 |
| 8.1 | Constant and Scratch Floating Points | 45 |
| 8.2 | Floating Point Expressions | 46 |
| 8.3 | Creating and Using Floating Points | 47 |
| 8.4 | Viewing Floating Points | 47 |
| 8.5 | Setting Floating Point Variables | 48 |
| 8.6 | Floating Point Step Functions | 48 |
| 8.7 | Float Point Conditionals | 49 |
| 9 | Dimensions (Dim) | 50 |
| 9.1 | Constant and Scratch Dimensions | 50 |
| 9.2 | Dimension Expressions | 50 |
| 9.3 | Creating and Using Dimensions | 51 |
| 9.4 | Viewing Dimensions | 52 |
| 9.5 | Setting Dimension Variables | 52 |
| 9.6 | Dimension Step Functions | 53 |
| 9.7 | Dimension Conditionals | 54 |
| 9.8 | Dimension Case Functions | 55 |
| 10 | Comma Separated Lists (Clist) | 57 |
| 10.1 | Constant and Scratch Comma Lists | 57 |
| 10.2 | Creating and Using Comma Lists | 57 |
| 10.3 | Viewing Comma Lists | 58 |
| 10.4 | Setting Comma Lists | 59 |
| 10.5 | Modifying Comma Lists | 60 |
| 10.6 | Working with the Contents of Comma Lists | 61 |
| 10.7 | Comma Lists as Stacks | 62 |
| 10.8 | Mapping over Comma Lists | 63 |
| 10.9 | Comma List Conditionals | 64 |

| | |
|--|-----------|
| 11 Sequences and Stacks (Seq) | 66 |
| 11.1 Constant and Scratch Sequences | 66 |
| 11.2 Creating and Using Sequences | 66 |
| 11.3 Viewing Sequences | 67 |
| 11.4 Setting Sequences | 67 |
| 11.5 Modifying Sequences | 69 |
| 11.6 Working with the Contents of Sequences | 69 |
| 11.7 Sequences as Stacks | 70 |
| 11.8 Recovering Items from Sequences | 71 |
| 11.9 Mapping over Sequences | 73 |
| 11.10 Sequence Conditionals | 73 |
| 12 Property Lists (Prop) | 75 |
| 12.1 Constant and Scratch Sequences | 75 |
| 12.2 Creating and Using Property Lists | 75 |
| 12.3 Viewing Property Lists | 76 |
| 12.4 Setting Property Lists | 76 |
| 12.5 Recovering Values from Property Lists | 78 |
| 12.6 Mapping over property lists | 79 |
| 12.7 Property List Conditionals | 80 |
| 13 Quarks (Quark) | 81 |
| 13.1 Constant Quarks | 81 |
| 13.2 Quark Conditionals | 81 |
| 14 The Source Code | 82 |
| 14.1 Interfaces for Functional Programming (Prg) | 82 |
| 14.2 Interfaces for Argument Using (Use) | 92 |
| 14.3 Interfaces for Control Structures (Bool) | 92 |
| 14.4 Interfaces for Token Lists (Tl) | 94 |
| 14.5 Interfaces for Strings (Str) | 98 |
| 14.6 Interfaces for Integers (Int) | 101 |
| 14.7 Interfaces for Floating Point Numbers (Fp) | 104 |
| 14.8 Interfaces for Dimensions (Dim) | 106 |
| 14.9 Interfaces for Sorting Functions (Sort) | 109 |
| 14.10 Interfaces for Comma Separated Lists (Clist) | 109 |
| 14.11 Interfaces for Sequences and Stacks (Seq) | 112 |
| 14.12 Interfaces for Property Lists (Prop) | 117 |
| 14.13 Interfaces for Quarks (Quark) | 119 |

Chapter 1

Overview of Features

1.1 Evaluation from Inside to Outside

We will compare our first example with a similar Lua example:

| | |
|--|--|
| <pre>\IgnoreSpacesOn \PrgNewFunction \MathSquare { m } { \IntSet \lTmpaInt { \IntEval {#1 * #1} } \Result { \Value \lTmpaInt } } \IgnoreSpacesOff \MathSquare{5} \MathSquare{\MathSquare{5}}</pre> | <pre>-- define a function -- function MathSquare (arg) local lTmpaInt = arg * arg return lTmpaInt end -- use the function -- print(MathSquare(5)) print(MathSquare(MathSquare(5)))</pre> |
|--|--|

Both examples calculate first the square of 5 and produce 25, then calculate the square of 25 and produce 625. In contrast to `expl3`, this `functional` package does evaluation of functions from inside to outside, which means composition of functions works like other programming languages such as Lua or JavaScript.

You can define new functions with `\PrgNewFunction` command. To make composition of functions work as expected, every function *must not* insert directly any token to the input stream. Instead, a function *must* pass the result (if any) to `functional` package with `\Result` command. And `functional` package is responsible for inserting result tokens to the input stream at the appropriate time.

To remove space tokens inside function code in defining functions, you'd better put function definitions inside `\IgnoreSpacesOn` and `\IgnoreSpacesOff` block. Within this block, `~` is used to input a space.

At the end of this section, we will compare our factorial example with a similar Lua example:

| | |
|--|--|
| <pre>\IgnoreSpacesOn \PrgNewFunction \Fact { m } { \IntCompareTF {#1} = {0} { \Result {1} }{ \Result {\IntMathMult{#1}{\Fact{\IntMathSub{#1}{1}}}} } } \IgnoreSpacesOff \Fact{4}</pre> | <pre>-- define a function -- function Fact (n) if n == 0 then return 1 else return n * Fact(n-1) end end -- use the function -- print(Fact(4))</pre> |
|--|--|

1.2 Group Scoping of Functions

In Lua language, a function or a condition expression makes a block, and the values of local variables will be reset after a block. In `functional` package, a condition expression is in fact a function, and you can make every function become a group by setting `\Functional{scoping=true}`. For example

| | | |
|---|----------|---------------------|
| <code>\Functional{scoping=true}</code> | | -- lua code -- |
| <code>\IgnoreSpacesOn</code> | | -- begin example -- |
| <code>\IntSet \lTmptInt {1}</code> | | local a = 1 |
| <code>\IntVarLog \lTmptInt</code> | % ---- 1 | print(a) ---- 1 |
| <code>\PrgNewFunction \SomeFun { } {</code> | | function SomeFun() |
| <code>\IntSet \lTmptInt {2}</code> | | local a = 2 |
| <code>\IntVarLog \lTmptInt</code> | % ---- 2 | print(a) ---- 2 |
| <code>\IntCompareTF {1} > {0} {</code> | | if 1 > 0 then |
| <code>\IntSet \lTmptInt {3}</code> | | local a = 3 |
| <code>\IntVarLog \lTmptInt</code> | % ---- 3 | print(a) ---- 3 |
| <code>}{ }</code> | | end |
| <code>\IntVarLog \lTmptInt</code> | % ---- 2 | print(a) ---- 2 |
| <code>}</code> | | end |
| <code>\SomeFun</code> | | SomeFun() |
| <code>\IntVarLog \lTmptInt</code> | % ---- 1 | print(a) ---- 1 |
| <code>\IgnoreSpacesOff</code> | | -- end example -- |

Same as `expl3`, the names of local variables *must* start with `l`, while names of global variables *must* start with `g`. The difference is that `functional` package provides only one function for setting both local and global variables of the same type, by checking leading letters of their names. So for integer variables, you can write `\IntSet\lTmptInt{1}` and `\IntSet\gTmptInt{2}`.

The previous example will produce different result if we change variable from `\lTmptInt` to `\gTmptInt`.

| | | |
|---|----------|---------------------|
| <code>\Functional{scoping=true}</code> | | -- lua code -- |
| <code>\IgnoreSpacesOn</code> | | -- begin example -- |
| <code>\IntSet \gTmptInt {1}</code> | | a = 1 |
| <code>\IntVarLog \gTmptInt</code> | % ---- 1 | print(a) ---- 1 |
| <code>\PrgNewFunction \SomeFun { } {</code> | | function SomeFun() |
| <code>\IntSet \gTmptInt {2}</code> | | a = 2 |
| <code>\IntVarLog \gTmptInt</code> | % ---- 2 | print(a) ---- 2 |
| <code>\IntCompareTF {1} > {0} {</code> | | if 1 > 0 then |
| <code>\IntSet \gTmptInt {3}</code> | | a = 3 |
| <code>\IntVarLog \gTmptInt</code> | % ---- 3 | print(a) ---- 3 |
| <code>}{ }</code> | | end |
| <code>\IntVarLog \gTmptInt</code> | % ---- 3 | print(a) ---- 3 |
| <code>}</code> | | end |
| <code>\SomeFun</code> | | SomeFun() |
| <code>\IntVarLog \gTmptInt</code> | % ---- 3 | print(a) ---- 3 |
| <code>\IgnoreSpacesOff</code> | | -- end example -- |

As you can see, the values of global variables will never be reset after a group.

1.3 Tracing Evaluation of Functions

Since every function in `functional` package will pass its return value to the package, it is quite easy to debug your code. You can turn on the tracing by setting `\Functional{tracing=true}`. For example, the tracing log of the first example in this chapter will be the following:

```

[I] \MathSquare{5}
    [I] \IntEval{5*5}
        [I] \Expand{\int_eval:n {5*5}}
        [0] 25
    [I] \Result{25}
    [0] 25
[0] 25
[I] \IntSet{\lTmptInt }{25}
[0]
    [I] \Value{\lTmptInt }
    [0] 25
[I] \Result{25}
[0] 25
[0] 25
[I] \MathSquare{25}
    [I] \IntEval{25*25}
        [I] \Expand{\int_eval:n {25*25}}
        [0] 625
    [I] \Result{625}
    [0] 625
[0] 625
[I] \IntSet{\lTmptInt }{625}
[0]
    [I] \Value{\lTmptInt }
    [0] 625
[I] \Result{625}
[0] 625
[0] 625

```

1.4 Definitions of Functions

Within `expl3`, there are eight commands for defining new functions, which is good for power users.

| | |
|--|---|
| <code>\cs_new:Npn</code> | <code>\cs_new:Nn</code> |
| <code>\cs_new_nopar:Npn</code> | <code>\cs_new_nopar:Nn</code> |
| <code>\cs_new_protected:Npn</code> | <code>\cs_new_protected:Nn</code> |
| <code>\cs_new_protected_nopar:Npn</code> | <code>\cs_new_protected_nopar:Nn</code> |

Within `functional` package, there is only one command (`\PrgNewFunction`) for defining new functions, which is good for normal users. The created functions are always protected and accept `\par` in their arguments.

Since `functional` package gets the results of functions by evaluation (including expansion and execution by $\text{T}_{\text{E}}\text{X}$), it is natural to protect all functions.

1.5 Variants of Arguments

Within `expl3`, there are several expansion variants for arguments, and many expansion functions for expanding them, which are necessary for power users.

| | |
|----------------------------|---------------------------|
| <code>\module_foo:c</code> | <code>\exp_args:Nc</code> |
| <code>\module_bar:e</code> | <code>\exp_args:Ne</code> |
| <code>\module_bar:x</code> | <code>\exp_args:Nx</code> |
| <code>\module_bar:f</code> | <code>\exp_args:Nf</code> |
| <code>\module_bar:o</code> | <code>\exp_args:No</code> |
| <code>\module_bar:V</code> | <code>\exp_args:NV</code> |
| <code>\module_bar:v</code> | <code>\exp_args:Nv</code> |

Within `functional` package, there are only three variants (`c`, `e`, `V`) are provided, and these variants are defined as functions (`\Name`, `\Expand`, `\Value`, respectively), which are easier to use for normal users.

```
\newcommand\test{uvw}  
\Name{test}
```

`uvw`

```
\newcommand\test{uvw}  
\Expand{111\test222}
```

`111uvw222`

```
\IntSet\lTmptInt{123}  
\Value\lTmptInt
```

`123`

The most interesting feature is that you can compose these functions. For example, you can easily get the `v` variant of `expl3` by simply composing `\Name` and `\Value` functions:

```
\IntSet\lTmptInt{123}  
\Value{\Name{\lTmptInt}}
```

`123`

Chapter 2

Functional Programming (Prg)

2.1 Defining Functions and Conditionals

\PrgNewFunction $\langle function \rangle$ $\{\langle argument specification \rangle\}$ $\{\langle code \rangle\}$

Creates protected $\langle function \rangle$ for evaluating the $\langle code \rangle$. Within the $\langle code \rangle$, the parameters ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The returned value must be passed with **\Result** function. The definition is global and an error results if the $\langle function \rangle$ is already defined.

The $\{\langle argument specification \rangle\}$ in a list of letters, where each letter is one of the following argument specifiers (nearly all of them are M or m for functions provided by this package):

- M single-token argument, which will be manipulated first
- m multi-token argument, which will be manipulated first
- N single-token argument, which will not be manipulated first
- n multi-token argument, which will not be manipulated first

The argument manipulation for argument type M or m is: if the argument starts with a function defined with **\PrgNewFunction**, the argument will be evaluated and replaced with the returned value.

\PrgNewConditional $\langle function \rangle$ $\{\langle argument specification \rangle\}$ $\{\langle code \rangle\}$

Creates protected conditional $\langle function \rangle$ for evaluating the $\langle code \rangle$. The returned value of the $\langle function \rangle$ must be either **\cTrueBool** or **\cFalseBool** and be passed with **\Result** function.. The definition is global and an error results if the $\langle function \rangle$ is already defined.

Assume the $\langle function \rangle$ is **\FooIfBar**, then another function **\FooIfBarTF** will be created at the same time. **\FooIfBarTF** function has two extra arguments which are $\{\langle true code \rangle\}$ and $\{\langle false code \rangle\}$.

2.2 Collecting Returned Values

\Result $\{\langle tokens \rangle\}$

Appends $\langle tokens \rangle$ to **\gResultTl**, which holds the returned value of current function. This function is normally used in the $\langle code \rangle$ of **\PrgNewFunction** and **\PrgNewConditional**.

Chapter 3

Argument Using (Use)

3.1 Expanding Tokens

\Name {*<control sequence name>*}

Expands the *<control sequence name>* until only characters remain, then converts this into a control sequence and returns it. The *<control sequence name>* must consist of character tokens when exhaustively expanded.

\Value *<variable>*

Recovers the content of a *<variable>* and returns the value. An error is raised if the variable does not exist or if it is invalid. Note that it is the same as **\TlUse** for *<tl var>*, or **\IntUse** for *<int var>*.

\Expand {*<tokens>*}

Expands the *<tokens>* exhaustively and returns the result.

\UnExpand {*<tokens>*}

Prevents expansion of the *<tokens>* inside the argument of **\Expand** function. The argument of **\UnExpand** must be surrounded by braces.

\OnlyName {*<tokens>*}

Expands the *<tokens>* until only characters remain, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited inside the argument of **\Expand** function.

\OnlyValue *<variable>*

Recovers the content of the *<variable>*, then prevents expansion of this material inside the argument of **\Expand** function.

3.2 Using Tokens

\UseOne {*<argument>*}

\GobbleOne {*<argument>*}

The function **\UseOne** absorbs one argument and returns it. **\GobbleOne** absorbs one argument and returns nothing. For example

```
\UseOne{abc}\GobbleOne{ijk}\UseOne{xyz}
```

```
abcxyz
```

```
\UseGobble {⟨arg1⟩} {⟨arg2⟩}
```

```
\GobbleUse {⟨arg1⟩} {⟨arg2⟩}
```

These functions absorb two arguments. The function `\UseGobble` discards the second argument, and returns the content of the first argument. `\GobbleUse` discards the first argument, and returns the content of the second argument. For example

```
\UseGobble{abc}{uvw}\GobbleUse{abc}{uvw}
```

```
abcuvw
```

Chapter 4

Control Structures (Bool)

4.1 Constant and Scratch Booleans

`\cTrueBool \cFalseBool`

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

```
\BoolVarIfTF \cTrueBool {\Result{True!}} {\Result{False!}}
\BoolVarIfTF \cFalseBool {\Result{True!}} {\Result{False!}}
```

True! False!

`\lTmPaBool \lTmPbBool \lTmPcBool \lTmPiBool \lTmPjBool \lTmPkBool`

Scratch booleans for local assignment. These are never used by the `functional` package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

`\gTmPaBool \gTmPbBool \gTmPcBool \gTmPiBool \gTmPjBool \gTmPkBool`

Scratch booleans for global assignment. These are never used by the `functional` package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

4.2 Creating and Setting Booleans

`\BoolNew` $\langle boolean \rangle$

Creates a new $\langle boolean \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle boolean \rangle$ is initially `false`.

`\BoolConst` $\langle boolean \rangle$ $\{\langle boolexpr \rangle\}$

Creates a new constant $\langle boolean \rangle$ or raises an error if the name is already taken. The value of the $\langle boolean \rangle$ is set globally to the result of evaluating the $\langle boolexpr \rangle$. For example

```
\BoolConst \cFooSomeBool {\IntCompare{3}>{2}}
\BoolVarLog \cFooSomeBool
```

\BoolSet *<boolean>* {*<boolexpr>*}

Evaluates the *<boolean expression>* and sets the *<boolean>* variable to the logical truth of this evaluation. For example

```
\BoolSet \lTmpaBool {\IntCompare{3}<{2}}
\BoolVarLog \lTmpaBool
```

\BoolSetTrue *<boolean>*

Sets *<boolean>* logically **true**.

\BoolSetFalse *<boolean>*

Sets *<boolean>* logically **false**.

\BoolSetEq *<boolean₁>* *<boolean₂>*

Sets *<boolean₁>* to the current value of *<boolean₂>*. For example

```
\BoolSetTrue \lTmpaBool
\BoolSetEq \lTmpbBool \lTmpaBool
\BoolVarLog \lTmpbBool
```

4.3 Viewing Booleans

\BoolLog {*<boolean expression>*}

Writes the logical truth of the *<boolean expression>* in the log file.

\BoolVarLog *<boolean>*

Writes the logical truth of the *<boolean>* in the log file.

\BoolShow {*<boolean expression>*}

Displays the logical truth of the *<boolean expression>* on the terminal.

\BoolVarShow *<boolean>*

Displays the logical truth of the *<boolean>* on the terminal.

4.4 Booleans and Conditionals

\BoolIfExist *<boolean>*

\BoolIfExistTF *<boolean>* {*<true code>*} {*<false code>*}

Tests whether the *<boolean>* is currently defined. This does not check that the *<boolean>* really is a boolean variable. For example

```
\BoolIfExistTF \lTmpaBool {\Result{Yes}} {\Result{No}}
\BoolIfExistTF \lFooUndefinedBool {\Result{Yes}} {\Result{No}}
```

Yes No

```
\BoolVarIf <boolean>
\BoolVarIfTF <boolean> {\<true code>} {\<false code>}
```

Tests the current truth of *<boolean>*, and continues evaluation based on this result. For example

```
\BoolSetTrue \lTmpaBool
\BoolVarIfTF \lTmpaBool {\Result{True!}} {\Result{False!}}
\BoolSetFalse \lTmpaBool
\BoolVarIfTF \lTmpaBool {\Result{True!}} {\Result{False!}}
```

True! False!

```
\BoolVarNot <boolean>
\BoolVarNotTF <boolean> {\<true code>} {\<false code>}
```

Evaluates *<true code>* if *<boolean>* is **false**, and *<false code>* if *<boolean>* is **true**. For example

```
\BoolVarNotTF {\IntCompare{3}>{2}} {\Result{Yes}} {\Result{No}}
```

No

```
\BoolVarAnd <boolean1> <boolean2>
\BoolVarAndTF <boolean1> <boolean2> {\<true code>} {\<false code>}
```

Implements the “And” operation between two booleans, hence is **true** if both are **true**. The *<boolean₂>* is only evaluated if it is needed to determine the result of **\BoolVarAnd**. For example

```
\BoolVarAndTF {\IntCompare{3}>{2}} {\IntCompare{3}>{4}} {\Result{Yes}} {\Result{No}}
```

No

```
\BoolVarOr <boolean1> <boolean2>
\BoolVarOrTF <boolean1> <boolean2> {\<true code>} {\<false code>}
```

Implements the “Or” operation between two booleans, hence is **true** if either one is **true**. The *<boolean₂>* is only evaluated if it is needed to determine the result of **\BoolVarOr**. For example

```
\BoolVarOrTF {\IntCompare{3}>{2}} {\IntCompare{3}>{4}} {\Result{Yes}} {\Result{No}}
```

Yes

```
\BoolVarXor <boolean1> <boolean2>
\BoolVarXorTF <boolean1> <boolean2> {\<true code>} {\<false code>}
```

Implements an “exclusive or” operation between two booleans. For example

```
\BoolVarXorTF {\IntCompare{3}>{2}} {\IntCompare{3}>{4}} {\Result{Yes}} {\Result{No}}
```

Yes

Chapter 5

Token Lists (Tl)

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:

```
\TlFoo {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

```
\TlVarFoo \lSomeTl
```

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 `\UseOne` 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 6 items (Hello, w, o, r, l and d), but 13 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.

5.1 Constant and Scratch Token Lists

`\cSpaceTl`

An explicit space character contained in a token list. For use where an explicit space is required.

`\cEmptyTl`

Constant that is always empty.

`\lTmptaTl` `\lTmpbTl` `\lTmpcTl` `\lTmpiTl` `\lTmpjTl` `\lTmpkTl`

Scratch token lists for local assignment. These are never used by the `functional` package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

```
\gTmptl \gTmptbtl \gTmptctl \gTmptitl \gTmptjtl \gTmptktl
```

Scratch token lists for global assignment. These are never used by the **functional** package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

5.2 Creating and Using Token Lists

```
\TlNew <tl var>
```

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.

```
\TlNew \lFooSomeTl
```

```
\TlConst <tl var> {<token list>}
```

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>*.

```
\TlConst \cFooSomeTl {abc}
```

```
\TlUse <tl var>
```

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

```
\TlUse \lTmptbTl
```

```
\TlToStr {<token list>}
```

Converts the *<token list>* to a *<string>*, returning the resulting character tokens. A *<string>* is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space).

```
\TlToStr {12\abc34}
```

```
12\abc 34
```

```
\TlVarToStr <tl var>
```

Converts the content of the *<tl var>* to a string, returning the resulting character tokens. A *<string>* is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space).

```
\TlSet \lTmptl {12\abc34}
\TlVarToStr \lTmptl
```

```
12\abc 34
```


5.3 Viewing Token Lists

\TlLog { $\langle token list \rangle$ }

Writes the $\langle token list \rangle$ in the log file. See also **\TlShow** which displays the result in the terminal.

```
\TlLog {123\abc456}
```

\TlVarLog $\langle tl var \rangle$

Writes the content of the $\langle tl var \rangle$ in the log file. See also **\TlVarShow** which displays the result in the terminal.

```
\TlSet \lTmptl {123\abc456}
\TlVarLog \lTmptl
```

\TlShow { $\langle token list \rangle$ }

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

```
\TlShow {123\abc456}
```

\TlVarShow $\langle tl var \rangle$

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

```
\TlSet \lTmptl {123\abc456}
\TlVarShow \lTmptl
```

5.4 Setting Token List Variables

\TlSet $\langle tl var \rangle$ { $\langle tokens \rangle$ }

Sets $\langle tl var \rangle$ to contain $\langle tokens \rangle$, removing any previous content from the variable.

```
\TlSet \lTmptl {\IntMathMult{4}{5}}
\TlUse \lTmptl
```

20

\TlSetEq $\langle tl var_1 \rangle$ $\langle tl var_2 \rangle$

Sets the content of $\langle tl var_1 \rangle$ equal to that of $\langle tl var_2 \rangle$.

```
\TlSet \lTmptl {abc}
\TlSetEq \lTmptl \lTmptl
\TlUse \lTmptl
```

abc

\TlClear $\langle tl var \rangle$

Clears all entries from the $\langle tl var \rangle$.

```
\TlSet \lTmptl {One}
\TlClear \lTmptl
\TlSet \lTmptl {Two}
\TlUse \lTmptl
```

Two

\TlClearNew $\langle tl\ var \rangle$

Ensures that the $\langle tl\ var \rangle$ exists globally by applying **\TlNew** if necessary, then applies **\TlClear** to leave the $\langle tl\ var \rangle$ empty.

```
\TlClearNew \lFooSomeTl
```

\TlConcat $\langle tl\ var_1 \rangle \langle tl\ var_2 \rangle \langle tl\ var_3 \rangle$

Concatenates the content of $\langle tl\ var_2 \rangle$ and $\langle tl\ var_3 \rangle$ together and saves the result in $\langle tl\ var_1 \rangle$. The $\langle tl\ var_2 \rangle$ is placed at the left side of the new token list.

```
\TlSet \lTmptl {con}
\TlSet \lTmptl {cat}
\TlConcat \lTmptl \lTmptl \lTmptl
\TlUse \lTmptl
```

concat

\TlPutLeft $\langle tl\ var \rangle \{ \langle tokens \rangle \}$

Appends $\langle tokens \rangle$ to the left side of the current content of $\langle tl\ var \rangle$.

```
\TlSet \lTmptl {Functional}
\TlPutLeft \lTmptl {Hello}
\TlUse \lTmptl
```

HelloFunctional

\TlPutRight $\langle tl\ var \rangle \{ \langle tokens \rangle \}$

Appends $\langle tokens \rangle$ to the right side of the current content of $\langle tl\ var \rangle$.

```
\TlSet \lTmptl {Functional}
\TlPutRight \lTmptl {World}
\TlUse \lTmptl
```

FunctionalWorld

5.5 Replacing Tokens

Within token lists, replacement takes place at the top level: there is no recursion into brace groups (more precisely, within a group defined by a category code 1/2 pair).

\TlVarReplaceOnce $\langle tl\ var \rangle \{ \langle old\ tokens \rangle \} \{ \langle new\ tokens \rangle \}$

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

```
\TlSet \lTmptl {1{bc}2bc3}
\TlVarReplaceOnce \lTmptl {bc} {xx}
\TlUse \lTmptl
```

1bc2xx3

\TlVarReplaceAll $\langle tl\ var \rangle \{ \langle old\ tokens \rangle \} \{ \langle new\ tokens \rangle \}$

Replaces all occurrences of $\langle old\ tokens \rangle$ in the $\langle tl\ var \rangle$ with $\langle new\ tokens \rangle$. $\langle Old\ tokens \rangle$ 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 $\langle old\ tokens \rangle$ may remain after the replacement (see **\TlVarRemoveAll** for an example).

```
\TlSet \lTmptl {1{bc}2bc3}
\TlVarReplaceAll \lTmptl {bc} {xx}
\TlUse \lTmptl
```

1bc2xx3

\TlVarRemoveOnce $\langle tl\ var \rangle \{ \langle tokens \rangle \}$

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

```
\TlSet \lTmptl {1{bc}2bc3}
\TlVarRemoveOnce \lTmptl {bc}
\TlUse \lTmptl
```

1bc23

\TlVarRemoveAll $\langle tl\ var \rangle \{ \langle tokens \rangle \}$

Removes all occurrences of $\langle tokens \rangle$ from the $\langle tl\ var \rangle$. $\langle Tokens \rangle$ 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 $\langle tokens \rangle$ may remain after the removal, for instance,

```
\TlSet \lTmptl {abbccd}
\TlVarRemoveAll \lTmptl {bc}
\TlUse \lTmptl
```

abcd

\TlTrimSpaces $\{ \langle token\ list \rangle \}$

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 returns the result.

```
Foo\TlTrimSpaces { 12 34 }Bar
```

Foo12 34Bar

\TlVarTrimSpaces $\langle tl\ var \rangle$

Sets the $\langle tl\ var \rangle$ to contain the result of removing any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from its contents.

```
\TlSet \lTmptl { 12 34 }
\TlVarTrimSpaces \lTmptl
Foo\TlUse \lTmptl Bar
```

Foo12 34Bar

5.6 Working with the Content of Token Lists

\TlCount {*<tokens>*}

Counts the number of *<items>* in *<tokens>* and returns this information. Unbraced tokens count as one element as do each token group (*{...}*). This process ignores any unprotected spaces within *<tokens>*.

```
\TlCount {12\abc34}
```

5

\TlVarCount *<tl var>*

Counts the number of *<items>* in the *<tl var>* and returns this information. Unbraced tokens count as one element as do each token group (*{...}*). This process ignores any unprotected spaces within the *<tl var>*.

```
\TlSet \lTmplTl {12\abc34}
\TlVarCount \lTmplTl
```

5

\TlHead {*<token list>*}

Returns 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

```
\fbox {1\TlHead{ abc }2}
\fbox {1\TlHead{ abc }2}
```

1a2

1a2

If the “head” is a brace group, rather than a single token, the braces are removed, and so

```
\TlHead { { ab} c }
```

yields `\ab`. A blank *<token list>* (see **\TlIfBlank**) results in **\TlHead** returning nothing.

\TlVarHead *<tl var>*

Returns the first *<item>* in the *<tl var>*, discarding the rest of the *<tl var>*. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded.

```
\TlSet \lTmplTl {HELLO}
\TlVarHead \lTmplTl
```

H

\TlTail {*<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 returns the remaining tokens. Thus for example

```
\TlTail { a {bc} d }
```

and

```
\TlTail { a {bc} d }
```

both return `\{bc\}d`. A blank *<token list>* (see **\TlIfBlank**) results in **\TlTail** returning nothing.

\TlVarTail $\langle tl\ var \rangle$

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first $\langle item \rangle$ in the $\langle tl\ var \rangle$, and returns the remaining tokens.

```
\TlSet \lTmptl {HELLO}
\TlVarTail \lTmptl
```

ELLO

\TlItem $\{\langle token\ list \rangle\} \{\langle integer\ expression \rangle\}$

\TlVarItem $\langle tl\ var \rangle \{\langle integer\ expression \rangle\}$

Indexing items in the $\langle token\ list \rangle$ from 1 on the left, this function evaluates the $\langle integer\ expression \rangle$ and returns the appropriate item from the $\langle token\ list \rangle$. 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 returns nothing.

```
\TlItem {abcd} {3}
```

c

\TlRandItem $\{\langle token\ list \rangle\}$

\TlVarRandItem $\langle tl\ var \rangle$

Selects and returns a pseudo-random item of the $\langle token\ list \rangle$. If the $\langle token\ list \rangle$ is blank, the result is empty.

```
\TlRandItem {abcdef}
\TlRandItem {abcdef}
```

b a

5.7 Mapping over 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.

\TlMapInline $\{\langle token\ list \rangle\} \{\langle inline\ function \rangle\}$

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 $\#1$. For example,

```
\IgnoreSpacesOn
\TlClear \lTmptl
\TlMapInline {one} {
  \TlPutRight \lTmptl {[#1]}
}
\Result{\TlUse\lTmptl}
\IgnoreSpacesOff
```

produces [o][n][e].

\TlVarMapInline $\langle tl\ var \rangle \{\langle inline\ function \rangle\}$

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 $\#1$. For example,

```

\IgnoreSpacesOn
\TlClear \lTmptl
\TlSet \lTmptl {one}
\TlVarMapInline \lTmptl {
  \TlPutRight \lTmptl {[#1]}
}
\Result{\TlUse\lTmptl}
\IgnoreSpacesOff

```

produces [o][n][e].

\TlMapVariable {⟨token list⟩} ⟨variable⟩ {⟨code⟩}

Stores each ⟨item⟩ of the ⟨token 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 ⟨tl var⟩, or its original value if the ⟨tl var⟩ is blank.

```

\IgnoreSpacesOn
\TlClear \lTmptl
\TlMapVariable {one} \lTmptl {
  \TlPutRight \lTmptl {\Expand {[ \lTmptl ]}}
}
\Result{\TlUse\lTmptl}
\IgnoreSpacesOff

```

[o][n][e]

\TlVarMapVariable ⟨tl var⟩ ⟨variable⟩ {⟨code⟩}

Stores each ⟨item⟩ of the ⟨tl var⟩ 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 ⟨tl var⟩, or its original value if the ⟨tl var⟩ is blank.

```

\IgnoreSpacesOn
\TlClear \lTmptl
\TlSet \lTmptl {one}
\TlVarMapVariable \lTmptl \lTmptl {
  \TlPutRight \lTmptl {\Expand {[ \lTmptl ]}}
}
\Result{\TlUse\lTmptl}
\IgnoreSpacesOff

```

[o][n][e]

5.8 Token List Conditionals

\TlIfExist ⟨tl var⟩
\TlIfExistTF ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}

Tests whether the ⟨tl var⟩ is currently defined. This does not check that the ⟨tl var⟩ really is a token list variable.

```

\TlIfExistTF \lTmptl {\Result{Yes}} {\Result{No}}
\TlIfExistTF \lFooUndefinedTl {\Result{Yes}} {\Result{No}}

```

Yes No

```
\TlIfEmpty {⟨token list⟩}
\TlIfEmptyTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the *⟨token list⟩* is entirely empty (*i.e.* contains no tokens at all). For example

```
\TlIfEmptyTF {abc} {\Result{Empty}} {\Result{NonEmpty}}
\TlIfEmptyTF {} {\Result{Empty}} {\Result{NonEmpty}}
```

NonEmpty Empty

```
\TlVarIfEmpty ⟨tl var⟩
\TlVarIfEmptyTF ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the *⟨token list variable⟩* is entirely empty (*i.e.* contains no tokens at all). For example

```
\TlSet \lTmptl {abc}
\TlVarIfEmptyTF \lTmptl {\Result{Empty}} {\Result{NonEmpty}}
\TlClear \lTmptl
\TlVarIfEmptyTF \lTmptl {\Result{Empty}} {\Result{NonEmpty}}
```

NonEmpty Empty

```
\TlIfBlank {⟨token list⟩}
\TlIfBlankTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the *⟨token list⟩* consists only of blank spaces (*i.e.* contains no item). The test is **true** if *⟨token list⟩* is zero or more explicit space characters (explicit tokens with character code 32 and category code 10), and is **false** otherwise.

```
\TlIfEmptyTF { } {\Result{Yes}} {\Result{No}}
\TlIfBlankTF { } {\Result{Yes}} {\Result{No}}
```

No Yes

```
\TlIfEq {⟨token list1⟩} {⟨token list2⟩}
\TlIfEqTF {⟨token list1⟩} {⟨token list2⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if *⟨token list₁⟩* and *⟨token list₂⟩* contain the same list of tokens, both in respect of character codes and category codes. See **\StrIfEq** if category codes are not important. For example

```
\TlIfEqTF {abc} {abc} {\Result{Yes}} {\Result{No}}
\TlIfEqTF {abc} {xyz} {\Result{Yes}} {\Result{No}}
```

Yes No

```
\TlVarIfEq ⟨tl var1⟩ ⟨tl var2⟩
\TlVarIfEqTF ⟨tl var1⟩ ⟨tl var2⟩ {⟨true code⟩} {⟨false code⟩}
```

Compares the content of two *⟨token list variables⟩* and is logically **true** if the two contain the same list of tokens (*i.e.* identical in both the list of characters they contain and the category codes of those characters). For example

```
\TlSet \lTmptl {abc}
\TlSet \lTmptl {abc}
\TlSet \lTmptl {xyz}
\TlVarIfEqTF \lTmptl \lTmptl {\Result{Yes}} {\Result{No}}
\TlVarIfEqTF \lTmptl \lTmptl {\Result{Yes}} {\Result{No}}
```

Yes No

See also **\StrVarIfEq** for a comparison that ignores category codes.

```
\TlIfIn {⟨token list1⟩} {⟨token list2⟩}
\TlIfInTF {⟨token list1⟩} {⟨token list2⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if $\langle token list_2 \rangle$ is found inside $\langle token list_1 \rangle$. The $\langle token list_2 \rangle$ cannot contain the tokens $\{, \}$ or $\#$ (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). The search does *not* enter brace (category code 1/2) groups.

```
\TlIfInTF {hello world} {o} {\Result{Yes}} {\Result{No}}
\TlIfInTF {hello world} {a} {\Result{Yes}} {\Result{No}}
```

Yes No

```
\TlVarIfIn ⟨tl var⟩ {⟨token list⟩}
\TlVarIfInTF ⟨tl var⟩ {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the $\langle token list \rangle$ is found in the content of the $\langle tl var \rangle$. The $\langle token list \rangle$ cannot contain the tokens $\{, \}$ or $\#$ (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

```
\TlSet \lTmptl {hello world}
\TlVarIfInTF \lTmptl {o} {\Result{Yes}} {\Result{No}}
\TlVarIfInTF \lTmptl {a} {\Result{Yes}} {\Result{No}}
```

Yes No

```
\TlIfSingle {⟨token list⟩}
\TlIfSingleTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the $\langle token list \rangle$ has exactly one $\langle item \rangle$, *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 **\TlCount**.

```
\TlIfSingleTF {a} {\Result{Yes}} {\Result{No}}
\TlIfSingleTF {abc} {\Result{Yes}} {\Result{No}}
```

Yes No

```
\TlVarIfSingle ⟨tl var⟩
\TlVarIfSingleTF ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the content of the $\langle tl var \rangle$ consists of a single $\langle item \rangle$, *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 **\TlVarCount**.

```
\TlSet \lTmptl {a}
\TlVarIfSingleTF \lTmptl {\Result{Yes}} {\Result{No}}
\TlSet \lTmptl {abc}
\TlVarIfSingleTF \lTmptl {\Result{Yes}} {\Result{No}}
```

Yes No

5.9 Token List Case Functions

```

\TlVarCase <test token list variable>
{
  <token list variable case1> {<code case1>}
  <token list variable case2> {<code case2>}
  ...
  <token list variable casen> {<code casen>}
}

```

This function compares the *<test token list variable>* in turn with each of the *<token list variable cases>*. If the two are equal (as described for **\TlVarIfEq**) then the associated *<code>* is left in the input stream and other cases are discarded. The function does nothing if there is no match.

```

\IgnoreSpacesOn
\TlSet \lTmptl {a}
\TlSet \lTmptb {b}
\TlSet \lTmptc {c}
\TlSet \lTmptk {b}
\TlVarCase \lTmptk {
  \lTmptl {\Result {First}}
  \lTmptb {\Result {Second}}
  \lTmptc {\Result {Third}}
}
\IgnoreSpacesOff

```

Second

```

\TlVarCaseT <test token list variable>
{
  <token list variable case1> {<code case1>}
  <token list variable case2> {<code case2>}
  ...
  <token list variable casen> {<code casen>}
}
{<true code>}

```

This function compares the *<test token list variable>* in turn with each of the *<token list variable cases>*. If the two are equal (as described for **\TlVarIfEq**) 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).

```

\IgnoreSpacesOn
\TlSet \lTmptl {a}
\TlSet \lTmptb {b}
\TlSet \lTmptc {c}
\TlSet \lTmptk {b}
\TlVarCaseT \lTmptk {
  \lTmptl {\IntSet \lTmptkInt {1}}
  \lTmptb {\IntSet \lTmptkInt {2}}
  \lTmptc {\IntSet \lTmptkInt {3}}
}{
  \Result {\IntUse \lTmptkInt}
}
\IgnoreSpacesOff

```

2

```

\TlVarCaseF <test token list variable>
{
  <token list variable case1> {\code case1>}
  <token list variable case2> {\code case2>}
  ...
  <token list variable casen> {\code casen>}
}
{\false code}

```

This function compares the *<test token list variable>* in turn with each of the *<token list variable cases>*. If the two are equal (as described for **\TlVarIfEq**) then the associated *<code>* is left in the input stream and other cases are discarded. If none match then the *<false code>* is inserted into the input stream (after the code for the appropriate case).

```

\IgnoreSpacesOn
\TlSet \lTmptaTl {a}
\TlSet \lTmptbTl {b}
\TlSet \lTmptcTl {c}
\TlSet \lTmptkTl {b}
\TlVarCaseF \lTmptkTl{
  \lTmptaTl {\Result {First}}
  \lTmptbTl {\Result {Second}}
  \lTmptcTl {\Result {Third}}
}{
  \Result {No~Match!}
}
\IgnoreSpacesOff

```

Second

```

\TlVarCaseTF <test token list variable>
{
  <token list variable case1> {\code case1>}
  <token list variable case2> {\code case2>}
  ...
  <token list variable casen> {\code casen>}
}
{\true code}
{\false code}

```

This function compares the *<test token list variable>* in turn with each of the *<token list variable cases>*. If the two are equal (as described for **\TlVarIfEq**) 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 **\TlVarCase**, which does nothing if there is no match, is also available.

```
\IgnoreSpacesOn
\TlSet \lTmptaTl {a}
\TlSet \lTmpbTl {b}
\TlSet \lTmpcTl {c}
\TlSet \lTmpkTl {b}
\TlVarCaseTF \lTmpkTl {
  \lTmptaTl {\IntSet \lTmpkInt {1}}
  \lTmpbTl {\IntSet \lTmpkInt {2}}
  \lTmpcTl {\IntSet \lTmpkInt {3}}
}{
  \Result {\IntUse \lTmpkInt}
}{
  \Result {0}
}
\IgnoreSpacesOff
```

Chapter 6

Strings (Str)

T_EX 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 T_EX sense.

A T_EX 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 T_EX 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 `\TlToStr` 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.

6.1 Constant and Scratch Strings

```
\cAmpersandStr \cAtsignStr \cBackslashStr \cLeftBraceStr \cRightBraceStr  
\cCircumflexStr \cColonStr \cDollarStr \cHashStr \cPercentStr \cTildeStr  
\cUnderscoreStr \cZeroStr
```

Constant strings, containing a single character token, with category code 12.

```
\lTmpaStr \lTmpbStr \lTmpcStr \lTmpiStr \lTmpjStr \lTmpkStr
```

Scratch strings for local assignment. These are never used by the `functional` package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

```
\gTmpaStr \gTmpbStr \gTmpcStr \gTmpiStr \gTmpjStr \gTmpkStr
```

Scratch strings for global assignment. These are never used by the `functional` package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

6.2 Creating and Using Strings

\StrNew $\langle str\ var \rangle$

Creates a new $\langle str\ var \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle str\ var \rangle$ is initially empty.

```
\StrNew \lFooSomeStr
```

\StrConst $\langle str\ var \rangle \{ \langle token\ list \rangle \}$

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

```
\StrConst \cFooSomeStr {12\abc34}
```

\StrUse $\langle str\ var \rangle$

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

```
\StrUse \lTmpaStr
```

6.3 Viewing Strings

\StrLog $\{ \langle token\ list \rangle \}$

Writes $\langle token\ list \rangle$ in the log file.

```
\StrLog {1234\abcd5678}
```

\StrVarLog $\langle str\ var \rangle$

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

```
\StrSet \lTmpiStr {1234\abcd5678}
\StrVarLog \lTmpiStr
```

\StrShow $\{ \langle token\ list \rangle \}$

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

```
\StrShow {1234\abcd5678}
```

\StrVarShow $\langle str\ var \rangle$

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

```
\StrSet \lTmpiStr {1234\abcd5678}
\StrVarShow \lTmpiStr
```

6.4 Setting String Variables

\StrSet $\langle str\ var \rangle \{ \langle token\ list \rangle \}$

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and stores the result in $\langle str\ var \rangle$.

```
\StrSet \lTmpiStr {\IntMathMult{4}{5}}
\StrUse \lTmpiStr
```

20

\StrSetEq $\langle str\ var_1 \rangle \langle str\ var_2 \rangle$

Sets the content of $\langle str\ var_1 \rangle$ equal to that of $\langle str\ var_2 \rangle$.

```
\StrSet \lTmpaStr {abc}
\StrSetEq \lTmpbStr \lTmpaStr
\StrUse \lTmpbStr
```

abc

\StrClear $\langle str\ var \rangle$

Clears the content of the $\langle str\ var \rangle$. For example

```
\StrSet \lTnpjStr {One}
\StrClear \lTnpjStr
\StrSet \lTnpjStr {Two}
\StrUse \lTnpjStr
```

Two

\StrClearNew $\langle str\ var \rangle$

Ensures that the $\langle str\ var \rangle$ exists globally by applying **\StrNew** if necessary, then applies **\StrClear** to leave the $\langle str\ var \rangle$ empty.

```
\StrClearNew \lFooSomeStr
\StrUse \lFooSomeStr
```

\StrConcat $\langle str\ var_1 \rangle \langle str\ var_2 \rangle \langle str\ var_3 \rangle$

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.

```
\StrSet \lTmpbStr {con}
\StrSet \lTnpcStr {cat}
\StrConcat \lTmpaStr \lTmpbStr \lTnpcStr
\StrUse \lTmpaStr
```

concat

\StrPutLeft $\langle str\ var \rangle \{ \langle token\ list \rangle \}$

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.

```
\StrSet \lTmkpStr {Functional}
\StrPutLeft \lTmkpStr {Hello}
\StrUse \lTmkpStr
```

HelloFunctional

\StrPutRight $\langle str\ var \rangle \{ \langle token\ list \rangle \}$

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.

```
\StrSet \lTmkpStr {Functional}
\StrPutRight \lTmkpStr {World}
\StrUse \lTmkpStr
```

FunctionalWorld

6.5 Modifying String Variables

\StrVarReplaceOnce $\langle str\ var \rangle \{ \langle old \rangle \} \{ \langle new \rangle \}$

Converts the $\langle old \rangle$ and $\langle new \rangle$ token lists to strings, then replaces the first (leftmost) occurrence of $\langle old\ string \rangle$ in the $\langle str\ var \rangle$ with $\langle new\ string \rangle$.

```
\StrSet \lTmpaStr {a{bc}bcd}
\StrVarReplaceOnce \lTmpaStr {bc} {xx}
\StrUse \lTmpaStr
```

a{xx}bcd

\StrVarReplaceAll $\langle str\ var \rangle \{ \langle old \rangle \} \{ \langle new \rangle \}$

Converts the $\langle old \rangle$ and $\langle new \rangle$ token lists to strings, then replaces all occurrences of $\langle old\ string \rangle$ in the $\langle str\ var \rangle$ with $\langle new\ string \rangle$. As this function operates from left to right, the pattern $\langle old\ string \rangle$ may remain after the replacement.

```
\StrSet \lTmpaStr {a{bc}bcd}
\StrVarReplaceAll \lTmpaStr {bc} {xx}
\StrUse \lTmpaStr
```

a{xx}xxd

\StrVarRemoveOnce $\langle str\ var \rangle \{ \langle token\ list \rangle \}$

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$ then removes the first (leftmost) occurrence of $\langle string \rangle$ from the $\langle str\ var \rangle$.

```
\StrSet \lTmpaStr {a{bc}bcd}
\StrVarRemoveOnce \lTmpaStr {bc}
\StrUse \lTmpaStr
```

a{ }bcd

\StrVarRemoveAll $\langle str\ var \rangle \{ \langle token\ list \rangle \}$

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$ then removes all occurrences of $\langle string \rangle$ from the $\langle str\ var \rangle$. As this

function operates from left to right, the pattern $\langle string \rangle$ may remain after the removal, for instance,

```
\StrSet \lTmPaStr {abbccd}
\StrVarRemoveAll \lTmPaStr {bc}
\TlUse \lTmPaStr
```

abcd

6.6 Working with the Content of Strings

\StrCount $\{\langle token list \rangle\}$

Returns the number of characters in the string representation of $\langle token list \rangle$, as an integer denotation. All characters including spaces are counted.

```
\StrCount {12\abc34}
```

9

Due to naming conflict, you need to use **\StrSize** instead of **\StrCount** if you want to use **functional** package together with **xstring** package.

\StrVarCount $\langle tl var \rangle$

Returns the number of characters in the string representation of the $\langle tl var \rangle$, as an integer denotation. All characters including spaces are counted.

```
\StrSet \lTmPaStr {12\abc34}
\StrVarCount \lTmPaStr
```

9

\StrHead $\{\langle token list \rangle\}$

Converts the $\langle token list \rangle$ into a $\langle string \rangle$. The first character in the $\langle string \rangle$ is then returned, with category code “other”. If the first character is a space, it returns a space token with category code 10 (blank space). If the $\langle string \rangle$ is empty, then nothing is returned.

```
\StrHead {HELLO}
```

H

\StrVarHead $\langle tl var \rangle$

Converts the $\langle tl var \rangle$ into a $\langle string \rangle$. The first character in the $\langle string \rangle$ is then returned, with category code “other”. If the first character is a space, it returns a space token with category code 10 (blank space). If the $\langle string \rangle$ is empty, then nothing is returned.

```
\StrSet \lTmPaStr {HELLO}
\StrVarHead \lTmPaStr
```

H

\StrTail $\{\langle token list \rangle\}$

Converts the $\langle token list \rangle$ to a $\langle string \rangle$, removes the first character, and returns the remaining characters (if any) with category codes 12 and 10 (for spaces). If the first character is a space, it only trims that space. If the $\langle token list \rangle$ is empty, then nothing is left on the input stream.

```
\StrTail {HELLO}
```

ELLO

\StrVarTail $\langle tl\ var \rangle$

Converts the $\langle tl\ var \rangle$ to a $\langle string \rangle$, removes the first character, and returns the remaining characters (if any) with category codes 12 and 10 (for spaces). If the first character is a space, it only trims that space. If the $\langle token\ list \rangle$ is empty, then nothing is left on the input stream.

```
\StrSet \lTmpaStr {HELLO}
\StrVarTail \lTmpaStr
```

ELLO

\StrItem $\{\langle token\ list \rangle\} \{\langle integer\ expression \rangle\}$

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and returns the character in position $\langle integer\ expression \rangle$ of the $\langle string \rangle$, starting at 1 for the first (left-most) character. All characters including spaces are taken into account. 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.*

```
\StrItem {abcd} {3}
```

c

\StrVarItem $\langle tl\ var \rangle \{\langle integer\ expression \rangle\}$

Converts the $\langle tl\ var \rangle$ to a $\langle string \rangle$, and returns the character in position $\langle integer\ expression \rangle$ of the $\langle string \rangle$, starting at 1 for the first (left-most) character. All characters including spaces are taken into account. 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.*

```
\StrSet \lTmpaStr {abcd}
\StrVarItem \lTmpaStr {3}
```

c

6.7 Mapping over Strings

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.

\StrMapInline $\{\langle token\ list \rangle\} \{\langle inline\ function \rangle\}$
\StrVarMapInline $\langle str\ var \rangle \{\langle inline\ function \rangle\}$

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$ then applies the $\langle inline\ function \rangle$ to every $\langle character \rangle$ in the $\langle str\ var \rangle$ including spaces. The $\langle inline\ function \rangle$ should consist of code which receives the $\langle character \rangle$ as $\#1$. For example,

```
\IgnoreSpacesOn
\StrClear \lTmpaStr
\StrMapInline {one} {
  \StrPutRight \lTmpaStr {[#1]}
}
\Result{\StrUse\lTmpaStr}
\IgnoreSpacesOff
```

produces [o][n][e].

\StrMapVariable $\{\langle token\ list \rangle\} \langle variable \rangle \{\langle code \rangle\}$
\StrVarMapVariable $\langle str\ var \rangle \langle variable \rangle \{\langle code \rangle\}$

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$ then stores each $\langle character \rangle$ in the $\langle string \rangle$ (including spaces) in

turn in the (string or 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 $\langle character \rangle$ in the $\langle string \rangle$, or its original value if the $\langle string \rangle$ is empty.

```
\IgnoreSpacesOn
\StrClear \lTmPaStr
\StrMapVariable {one} \lTmPiStr {
  \StrPutRight \lTmPaStr {\Expand {\lTmPiStr}}
}
\Result{\StrUse\lTmPaStr}
\IgnoreSpacesOff
```

[o][n][e]

6.8 String Conditionals

```
\StrIfExist  $\langle str var \rangle$ 
\StrIfExistTF  $\langle str var \rangle$   $\{\langle true code \rangle\} \{\langle false code \rangle\}$ 
```

Tests whether the $\langle str var \rangle$ is currently defined. This does not check that the $\langle str var \rangle$ really is a string.

```
\StrIfExistTF \lTmPaStr {\Result{Yes}} {\Result{No}}
\StrIfExistTF \lFooUndefinedStr {\Result{Yes}} {\Result{No}}
```

Yes No

```
\StrVarIfEmpty  $\langle str var \rangle$ 
\StrVarIfEmptyTF  $\langle str var \rangle$   $\{\langle true code \rangle\} \{\langle false code \rangle\}$ 
```

Tests if the $\langle string variable \rangle$ is entirely empty (*i.e.* contains no characters at all).

```
\StrSet \lTmPaStr {abc}
\StrVarIfEmptyTF \lTmPaStr {\Result{Empty}} {\Result{NonEmpty}}
\StrClear \lTmPaStr
\StrVarIfEmptyTF \lTmPaStr {\Result{Empty}} {\Result{NonEmpty}}
```

NonEmpty Empty

```
\StrIfEq  $\{\langle tl_1 \rangle\} \{\langle tl_2 \rangle\}$ 
\StrIfEqTF  $\{\langle tl_1 \rangle\} \{\langle tl_2 \rangle\}$   $\{\langle true code \rangle\} \{\langle false code \rangle\}$ 
```

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. See **\TlIfEq** to compare tokens (including their category codes) rather than characters. For example

```
\StrIfEqTF {abc} {abc} {\Result{Yes}} {\Result{No}}
\StrIfEqTF {abc} {xyz} {\Result{Yes}} {\Result{No}}
```

Yes No

```
\StrVarIfEq  $\langle str var_1 \rangle \langle str var_2 \rangle$ 
\StrVarIfEqTF  $\langle str var_1 \rangle \langle str var_2 \rangle$   $\{\langle true code \rangle\} \{\langle false code \rangle\}$ 
```

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 **\TlVarIfEq** to compare tokens (including their category codes) rather than characters.

```

\StrSet \lTmpaStr {abc}
\StrSet \lTmpbStr {abc}
\StrSet \lTmpcStr {xyz}
\StrVarIfEqTF \lTmpaStr \lTmpbStr {\Result{Yes}} {\Result{No}}
\StrVarIfEqTF \lTmpaStr \lTmpcStr {\Result{Yes}} {\Result{No}}

```

Yes No

```

\StrIfInTF {\<tl1>} {\<tl2>}
\StrIfInTF {\<tl1>} {\<tl2>} {\<true code>} {\<false code>}

```

Converts both $\langle token\ lists \rangle$ to $\langle strings \rangle$ and tests whether $\langle string_2 \rangle$ is found inside $\langle string_1 \rangle$.

```

\StrIfInTF {hello world} {o} {\Result{Yes}}{\Result{No}}
\StrIfInTF {hello world} {a} {\Result{Yes}}{\Result{No}}

```

Yes No

```

\StrVarIfInTF \<str var> {\<token list>}
\StrVarIfInTF \<str var> {\<token list>} {\<true code>} {\<false 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 $\langle str\ var \rangle$.

```

\StrSet \lTmpaStr {hello world}
\StrVarIfInTF \lTmpaStr {o} {\Result{Yes}}{\Result{No}}
\StrVarIfInTF \lTmpaStr {a} {\Result{Yes}}{\Result{No}}

```

Yes No

```

\StrCompare {\<tl1>} \<relation> {\<tl2>}
\StrCompareTF {\<tl1>} \<relation> {\<tl2>} {\<true code>} {\<false code>}

```

Compares the two $\langle token\ lists \rangle$ on a character by character basis (namely after converting them to strings) in a lexicographic order according to the character codes of the characters. The $\langle relation \rangle$ can be $<$, $=$, or $>$ and the test is **true** under the following conditions:

- for $<$, if the first string is earlier than the second in lexicographic order;
- for $=$, if the two strings have exactly the same characters;
- for $>$, if the first string is later than the second in lexicographic order.

For example:

```

\StrCompareTF {ab} < {abc} {\Result{Yes}} {\Result{No}}
\StrCompareTF {ab} < {aa} {\Result{Yes}} {\Result{No}}

```

Yes No

Due to naming conflict, you need to use `\StrIfCompare`/`\StrIfCompareTF` as a replacement if you want to use **functional** package together with **xstring** package.

6.9 String Case Functions

```

\StrCase {\<test string>}
{
  {\<string case1>} {\<code case1>}
  {\<string case2>} {\<code case2>}
  ...
  {\<string casen>} {\<code casen>}
}

```

Compares the $\langle test\ string \rangle$ in turn with each of the $\langle string\ cases \rangle$ (all token lists are converted to strings).

If the two are equal (as described for `\StrIfEq`) then the associated `<code>` is left in the input stream and other cases are discarded.

```
\IgnoreSpacesOn
\StrCase {bbb} {
  {aaa} {\Result{First}}
  {bbb} {\Result{Second}}
  {ccb} {\Result{Third}}
}
\IgnoreSpacesOff
```

Second

```
\StrCaseT {<test string>}
{
  {<string case1>} {<code case1>}
  {<string case2>} {<code case2>}
  ...
  {<string casen>} {<code casen>}
}
{<true code>}
```

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 `\StrIfEq`) 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).

```
\IgnoreSpacesOn
\StrCaseT {bbb} {
  {aaa} {\TlSet\lTmkTl{First}}
  {bbb} {\TlSet\lTmkTl{Second}}
  {ccb} {\TlSet\lTmkTl{Third}}
}{
  \Result{\TlUse\lTmkTl}
}
\IgnoreSpacesOff
```

Second

```
\StrCaseF {<test string>}
{
  {<string case1>} {<code case1>}
  {<string case2>} {<code case2>}
  ...
  {<string casen>} {<code casen>}
}
{<false code>}
```

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 `\StrIfEq`) then the associated `<code>` is left in the input stream and other cases are discarded. If none match then the `<false code>` is inserted.

```

\IgnoreSpacesOn
\StrCaseF {bbb} {
  {aaa} {\Result{First}}
  {bbb} {\Result{Second}}
  {ccb} {\Result{Third}}
}{
  \Result{No~Match!}
}
\IgnoreSpacesOff

```

Second

```

\StrCaseTF {⟨test string⟩}
{
  {⟨string case1⟩} {⟨code case1⟩}
  {⟨string case2⟩} {⟨code case2⟩}
  ...
  {⟨string casen⟩} {⟨code casen⟩}
}
{⟨true code⟩}
{⟨false code⟩}

```

Compares the $\langle test\ string \rangle$ in turn with each of the $\langle string\ cases \rangle$ (all token lists are converted to strings). If the two are equal (as described for **\StrIfEq**) 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.

```

\IgnoreSpacesOn
\StrCaseTF {bbb} {
  {aaa} {\TlSet\lTmkTl{First}}
  {bbb} {\TlSet\lTmkTl{Second}}
  {ccb} {\TlSet\lTmkTl{Third}}
}{
  \Result{\TlUse\lTmkTl}
}{
  \Result{No~Match!}
}
\IgnoreSpacesOff

```

Second

Chapter 7

Integers (Int)

7.1 Constant and Scratch Integers

`\cZeroInt \cOneInt`

Integer values used with primitive tests and assignments: their self-terminating nature makes these more convenient and faster than literal numbers.

`\cMaxInt`

The maximum value that can be stored as an integer.

`\cMaxRegisterInt`

Maximum number of registers.

`\cMaxCharInt`

Maximum character code completely supported by the engine.

`\lTmPaInt \lTmPbInt \lTmPcInt \lTmPiInt \lTmPjInt \lTmPkInt`

Scratch integer for local assignment. These are never used by the **functional** package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

`\gTmPaInt \gTmPbInt \gTmPcInt \gTmPiInt \gTmPjInt \gTmPkInt`

Scratch integer for global assignment. These are never used by the **functional** package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

7.2 Integer Expressions

`\IntEval {⟨integer expression⟩}`

Evaluates the *⟨integer expression⟩* and returns the result: 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. For example

```
\IntEval {(1+4)*(2-3)/5}
```

-1

```
\IntMathAdd {⟨integer expression1⟩} {⟨integer expression2⟩}
```

Adds {⟨integer expression₁⟩} and {⟨integer expression₂⟩}, and returns the result. For example

```
\IntMathAdd {7} {3}
```

10

```
\IntMathSub {⟨integer expression1⟩} {⟨integer expression2⟩}
```

Subtracts {⟨integer expression₂⟩} from {⟨integer expression₁⟩}, and returns the result. For example

```
\IntMathSub {7} {3}
```

4

```
\IntMathMult {⟨integer expression1⟩} {⟨integer expression2⟩}
```

Multiplies {⟨integer expression₁⟩} by {⟨integer expression₂⟩}, and returns the result. For example

```
\IntMathMult {7} {3}
```

21

```
\IntMathDiv {⟨integer expression1⟩} {⟨integer expression2⟩}
```

Evaluates the two ⟨integer expressions⟩ 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 / directly in an ⟨integer expression⟩. The result is returned as an ⟨integer denotation⟩. For example

```
\IntMathDiv {8} {3}
```

3

```
\IntMathDivTruncate {⟨integer expression1⟩} {⟨integer expression2⟩}
```

Evaluates the two ⟨integer expressions⟩ as described earlier, then divides the first value by the second, and rounds the result towards zero. Note that division using / rounds to the closest integer instead. The result is returned as an ⟨integer denotation⟩. For example

```
\IntMathDivTruncate {8} {3}
```

2

```
\IntMathSign {⟨intexpr⟩}
```

Evaluates the ⟨integer expression⟩ then leaves 1 or 0 or −1 in the input stream according to the sign of the result.

```
\IntMathAbs {⟨integer expression⟩}
```

Evaluates the ⟨integer expression⟩ as described for `\IntEval` and leaves the absolute value of the result in the input stream as an ⟨integer denotation⟩ after two expansions.

```
\IntMathMax {⟨intexpr1⟩} {⟨intexpr2⟩}
```

```
\IntMathMin {⟨intexpr1⟩} {⟨intexpr2⟩}
```

Evaluates the ⟨integer expressions⟩ as described for `\IntEval` and leaves either the larger or smaller value in the input stream as an ⟨integer denotation⟩ after two expansions.

 $\backslash\text{IntMathMod } \{\langle integer \rangle\} \{\langle integer \rangle\}$

Evaluates the two $\langle integer \rangle$ s as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is obtained by subtracting $\backslash\text{IntMathDivTruncate } \{\langle integer \rangle\} \{\langle integer \rangle\}$ times $\langle integer \rangle$ from $\langle integer \rangle$. Thus, the result has the same sign as $\langle integer \rangle$ and its absolute value is strictly less than that of $\langle integer \rangle$. The result is left in the input stream as an $\langle integer \rangle$ after two expansions.

 $\backslash\text{IntMathRand } \{\langle integer \rangle\} \{\langle integer \rangle\}$

Evaluates the two $\langle integer \rangle$ s and produces a pseudo-random number between the two (with bounds included).

7.3 Creating and Using Integers

 $\backslash\text{IntNew } \langle integer \rangle$

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.

 $\backslash\text{IntConst } \langle integer \rangle \{\langle integer \rangle\}$

Creates a new constant $\langle integer \rangle$ or raises an error if the name is already taken. The value of the $\langle integer \rangle$ is set globally to the $\langle integer \rangle$.

 $\backslash\text{IntUse } \langle integer \rangle$

Recovers the content of an $\langle integer \rangle$ and returns the value. An error is raised if the variable does not exist or if it is invalid.

7.4 Viewing Integers

 $\backslash\text{IntLog } \{\langle integer \rangle\}$

Writes the result of evaluating the $\langle integer \rangle$ in the log file.

 $\backslash\text{IntVarLog } \langle integer \rangle$

Writes the value of the $\langle integer \rangle$ in the log file.

 $\backslash\text{IntShow } \{\langle integer \rangle\}$

Displays the result of evaluating the $\langle integer \rangle$ on the terminal.

 $\backslash\text{IntVarShow } \langle integer \rangle$

Displays the value of the $\langle integer \rangle$ on the terminal.

7.5 Setting Integer Variables

\IntSet $\langle integer \rangle \{ \langle integer expression \rangle \}$

Sets $\langle integer \rangle$ to the value of $\langle integer expression \rangle$, which must evaluate to an integer (as described for **\IntEval**). For example

```
\IntSet \lTmPaInt {3+5}
\IntUse \lTmPaInt
```

8

\IntSetEq $\langle integer_1 \rangle \langle integer_2 \rangle$

Sets the content of $\langle integer_1 \rangle$ equal to that of $\langle integer_2 \rangle$.

\IntZero $\langle integer \rangle$

Sets $\langle integer \rangle$ to 0. For example

```
\IntSet \lTmPaInt {5}
\IntZero \lTmPaInt
\IntUse \lTmPaInt
```

0

\IntZeroNew $\langle integer \rangle$

Ensures that the $\langle integer \rangle$ exists globally by applying **\IntNew** if necessary, then applies **\IntZero** to leave the $\langle integer \rangle$ set to zero.

\IntIncr $\langle integer \rangle$

Increases the value stored in $\langle integer \rangle$ by 1. For example

```
\IntSet \lTmPaInt {5}
\IntIncr \lTmPaInt
\IntUse \lTmPaInt
```

6

\IntDecr $\langle integer \rangle$

Decreases the value stored in $\langle integer \rangle$ by 1. For example

```
\IntSet \lTmPaInt {5}
\IntDecr \lTmPaInt
\IntUse \lTmPaInt
```

4

\IntAdd $\langle integer \rangle \{ \langle integer expression \rangle \}$

Adds the result of the $\langle integer expression \rangle$ to the current content of the $\langle integer \rangle$. For example

```
\IntSet \lTmPaInt {5}
\IntAdd \lTmPaInt {2}
\IntUse \lTmPaInt
```

7

\IntSub $\langle integer \rangle$ $\{\langle integer expression \rangle\}$

Subtracts the result of the $\langle integer expression \rangle$ from the current content of the $\langle integer \rangle$. For example

```
\IntSet \lTmptInt {5}
\IntSub \lTmptInt {3}
\IntUse \lTmptInt
```

2

7.6 Integer Step Functions

\IntStepInline $\{\langle initial value \rangle\} \{\langle step \rangle\} \{\langle final value \rangle\} \{\langle code \rangle\}$

This function first evaluates the $\langle initial value \rangle$, $\langle step \rangle$ and $\langle final value \rangle$, all of which should be 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 ($\#1$). For example

```
\IgnoreSpacesOn
\TlClear \lTmptTl
\IntStepInline {1} {3} {30} {
  \TlPutRight \lTmptTl {[#1]}
}
\Result {\Value\lTmptTl}
\IgnoreSpacesOff
```

produces [1][4][7][10][13][16][19][22][25][28].

\IntStepVariable $\{\langle initial value \rangle\} \{\langle step \rangle\} \{\langle final value \rangle\} \langle tl var \rangle \{\langle code \rangle\}$

This function first evaluates the $\langle initial value \rangle$, $\langle step \rangle$ and $\langle final value \rangle$, all of which should be 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 evaluated, with the $\langle tl var \rangle$ defined as the current $\langle value \rangle$. Thus the $\langle code \rangle$ should make use of the $\langle tl var \rangle$.

7.7 Integer Conditionals

\IntIfExist $\langle integer \rangle$
\IntIfExistTF $\langle integer \rangle \{\langle true code \rangle\} \{\langle false code \rangle\}$

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

\IntIfOdd $\{\langle integer expression \rangle\}$
\IntIfOddTF $\{\langle integer expression \rangle\} \{\langle true code \rangle\} \{\langle false code \rangle\}$

This function first evaluates the $\langle integer expression \rangle$ as described for **\IntEval**. It then evaluates if this is odd or even, as appropriate.

\IntIfEven $\{\langle integer expression \rangle\}$
\IntIfEvenTF $\{\langle integer expression \rangle\} \{\langle true code \rangle\} \{\langle false code \rangle\}$

This function first evaluates the $\langle integer expression \rangle$ as described for **\IntEval**. It then evaluates if this is even or odd, as appropriate.

\IntCompare {<integer expression>} <relation> {<integer expression>}

\IntCompareTF {<integer expression>} <relation> {<integer expression>} {<true code>} {<false code>}

This function first evaluates each of the <integer expressions> as described for **\IntEval**. The two results are then compared using the <relation>:

| | |
|--------------|---|
| Equal | = |
| Greater than | > |
| Less than | < |

For example

```
\IntCompareTF {2} > {1} {\Result{Greater}} {\Result{Less}}
\IntCompareTF {2} > {3} {\Result{Greater}} {\Result{Less}}
```

Greater Less

7.8 Integer Case Functions

\IntCase {<test integer expression>}

```
{
  {<integer case>} {<code case>}
  {<integer case>} {<code case>}
  ...
  {<integer case>} {<code case>}
}
```

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.

\IntCaseT {<test integer expression>}

```
{
  {<integer case>} {<code case>}
  {<integer case>} {<code case>}
  ...
  {<integer case>} {<code case>}
}
{<true code>}
```

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).

\IntCaseF {<test integer expression>}

```
{
  {<integer case>} {<code case>}
  {<integer case>} {<code case>}
  ...
  {<integer case>} {<code case>}
}
{<false code>}
```

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 none match then the <false code> is into the input stream (after the code for the appropriate case). For example

```

\IgnoreSpacesOn
\IntCaseF { 2 * 5 }
{
  { 5 }      { Small }
  { 4 + 6 }  { Medium }
  { -2 * 10 } { Negative }
}
{ No idea! }
\IgnoreSpacesOff

```

Medium

```

\IntCaseTF {⟨test integer expression⟩}
{
  {⟨intexpr case1⟩} {⟨code case1⟩}
  {⟨intexpr case2⟩} {⟨code case2⟩}
  ...
  {⟨intexpr casen⟩} {⟨code casen⟩}
}
{⟨true code⟩}
{⟨false code⟩}

```

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.

Chapter 8

Floating Point Numbers (Fp)

8.1 Constant and Scratch Floating Points

`\cZeroFp` `\cMinusZeroFp`

Zero, with either sign.

`\cOneFp`

One as an `fp`: useful for comparisons in some places.

`\cInfFp` `\cMinusInfFp`

Infinity, with either sign. These can be input directly in a floating point expression as `inf` and `-inf`.

`\cEFp`

The value of the base of the natural logarithm, $e = \exp(1)$.

`\cPiFp`

The value of π . This can be input directly in a floating point expression as `pi`.

`\cOneDegreeFp`

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 `deg`.

`\lTmPaFp` `\lTmPbFp` `\lTmPcFp` `\lTmPiFp` `\lTmPjFp` `\lTmPkFp`

Scratch floating point numbers for local assignment. These are never used by the `functional` package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

`\gTmPaFp` `\gTmPbFp` `\gTmPcFp` `\gTmPiFp` `\gTmPjFp` `\gTmPkFp`

Scratch floating point numbers for global assignment. These are never used by the `functional` package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

8.2 Floating Point Expressions

\FpEval {<floating point expression>}

Evaluates the <floating point expression> and returns 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 **\FpEval** 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. For example

```
\FpEval {(1.2+3.4)*(5.6-7.8)/9}
```

-1.124444444444444

\FpMathAdd {<fpexpr₁>} {<fpexpr₂>}

Adds {<fpexpr₁>} and {<fpexpr₂>}, and returns the result. For example

```
\FpMathAdd {2.8} {3.7}
\FpMathAdd {3.8-1} {2.7+1}
```

6.5 6.5

\FpMathSub {<fpexpr₁>} {<fpexpr₂>}

Subtracts {<fpexpr₂>} from {<fpexpr₁>}, and returns the result. For example

```
\FpMathSub {2.8} {3.7}
\FpMathSub {3.8-1} {2.7+1}
```

-0.9 -0.9

\FpMathMult {<fpexpr₁>} {<fpexpr₂>}

Multiplies {<fpexpr₁>} by {<fpexpr₂>}, and returns the result. For example

```
\FpMathMult {2.8} {3.7}
\FpMathMult {3.8-1} {2.7+1}
```

10.36 10.36

\FpMathDiv {<fpexpr₁>} {<fpexpr₂>}

Divides {<fpexpr₁>} by {<fpexpr₂>}, and returns the result. For example

```
\FpMathDiv {2.8} {3.7}
\FpMathDiv {3.8-1} {2.7+1}
```

0.7567567567567568 0.7567567567567568

\FpMathSign {<fpexpr>}

Evaluates the <fpexpr> and returns the value using **\FpEval{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. For example

```
\FpMathSign {3.5}
\FpMathSign {-2.7}
```

1 -1

\FpMathAbs {*floating point expression*}

Evaluates the *floating point expression* as described for **\FpEval** and returns the absolute value. If the argument is $\pm\infty$, NaN or a tuple, “invalid operation” occurs. Within floating point expressions, **abs()** can be used; it accepts $\pm\infty$ and NaN as arguments.

\FpMathMax {*fp expression*₁} {*fp expression*₂}

\FpMathMin {*fp expression*₁} {*fp expression*₂}

Evaluates the *floating point expressions* as described for **\FpEval** and returns the resulting larger (**max**) or smaller (**min**) value. If the argument is a tuple, “invalid operation” occurs, but no other case raises exceptions. Within floating point expressions, **max()** and **min()** can be used.

8.3 Creating and Using Floating Points

\FpNew *fp var*

Creates a new *fp var* or raises an error if the name is already taken. The declaration is global. The *fp var* is initially +0.

\FpConst *fp var* {*floating point expression*}

Creates a new constant *fp var* or raises an error if the name is already taken. The *fp var* is set globally equal to the result of evaluating the *floating point expression*. For example

```
\FpConst \cMyPiFp {3.1415926}
\FpUse \cMyPiFp
```

3.1415926

\FpUse *fp var*

Recovers the value of the *fp var* and returns the value as a decimal number with no exponent.

8.4 Viewing Floating Points

\FpLog {*floating point expression*}

Evaluates the *floating point expression* and writes the result in the log file.

\FpVarLog *fp var*

Writes the value of *fp var* in the log file.

\FpShow {*floating point expression*}

Evaluates the *floating point expression* and displays the result in the terminal.

\FpVarShow *fp var*

Displays the value of *fp var* in the terminal.

8.5 Setting Floating Point Variables

\FpSet $\langle fp\ var \rangle \{ \langle floating\ point\ expression \rangle \}$

Sets $\langle fp\ var \rangle$ equal to the result of computing the $\langle floating\ point\ expression \rangle$. For example

```
\FpSet \lTmPaFp {4/7}
\FpUse \lTmPaFp
```

0.5714285714285714

\FpSetEq $\langle fp\ var_1 \rangle \langle fp\ var_2 \rangle$

Sets the floating point variable $\langle fp\ var_1 \rangle$ equal to the current value of $\langle fp\ var_2 \rangle$.

\FpZero $\langle fp\ var \rangle$

Sets the $\langle fp\ var \rangle$ to +0. For example

```
\FpSet \lTmPaFp {5.3}
\FpZero \lTmPaFp
\FpUse \lTmPaFp
```

0

\FpZeroNew $\langle fp\ var \rangle$

Ensures that the $\langle fp\ var \rangle$ exists globally by applying **\FpNew** if necessary, then applies **\FpZero** to leave the $\langle fp\ var \rangle$ set to +0.

\FpAdd $\langle fp\ var \rangle \{ \langle floating\ point\ expression \rangle \}$

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. For example

```
\FpSet \lTmPaFp {5.3}
\FpAdd \lTmPaFp {2.11}
\FpUse \lTmPaFp
```

7.41

\FpSub $\langle fp\ var \rangle \{ \langle floating\ point\ expression \rangle \}$

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. For example

```
\FpSet \lTmPaFp {5.3}
\FpSub \lTmPaFp {2.11}
\FpUse \lTmPaFp
```

3.19

8.6 Floating Point Step Functions

\FpStepInline $\{ \langle initial\ value \rangle \} \{ \langle step \rangle \} \{ \langle final\ value \rangle \} \{ \langle code \rangle \}$

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**). For example

```
\IgnoreSpacesOn
\TlClear \lTmptl
\FpStepInline {1} {0.1} {1.5} {
  \TlPutRight \lTmptl {[#1]}
}
\Result {\Value\lTmptl}
\IgnoreSpacesOff
```

produces [1][1.1][1.2][1.3][1.4][1.5].

\FpStepVariable $\langle initial\ value \rangle$ $\langle step \rangle$ $\langle final\ value \rangle$ $\langle tl\ var \rangle$ $\langle code \rangle$

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 $\langle tl\ var \rangle$.

8.7 Float Point Conditionals

\FpIfExist $\langle fp\ var \rangle$
\FpIfExistTF $\langle fp\ var \rangle$ $\langle true\ code \rangle$ $\langle false\ code \rangle$

Tests whether the $\langle fp\ var \rangle$ is currently defined. This does not check that the $\langle fp\ var \rangle$ really is a floating point variable. For example

```
\FpIfExistTF \lTmptl {\Result{Yes}} {\Result{No}}
\FpIfExistTF \lMyUndefinedFp {\Result{Yes}} {\Result{No}}
```

Yes No

\FpCompare $\langle fpexpr_1 \rangle$ $\langle relation \rangle$ $\langle fpexpr_2 \rangle$
\FpCompareTF $\langle fpexpr_1 \rangle$ $\langle relation \rangle$ $\langle fpexpr_2 \rangle$ $\langle true\ code \rangle$ $\langle false\ code \rangle$

Compares the $\langle fpexpr_1 \rangle$ and the $\langle fpexpr_2 \rangle$, and returns **true** if the $\langle relation \rangle$ is obeyed. For example

```
\FpCompareTF {1} > {0.9999} {\Result{Greater}} {\Result{Less}}
\FpCompareTF {1} > {1.0001} {\Result{Greater}} {\Result{Less}}
```

Greater Less

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.

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.

Chapter 9

Dimensions (Dim)

9.1 Constant and Scratch Dimensions

`\cMaxDim`

The maximum value that can be stored as a dimension. This can also be used as a component of a skip.

`\cZeroDim`

A zero length as a dimension. This can also be used as a component of a skip.

`\lTmptaDim \lTmpbDim \lTmpcDim \lTmpiDim \lTmpjDim \lTmkDim`

Scratch dimensions for local assignment. These are never used by the **functional** package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

`\gTmptaDim \gTmpbDim \gTmpcDim \gTmpiDim \gTmpjDim \gTmkDim`

Scratch dimensions for global assignment. These are never used by the **functional** package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

9.2 Dimension Expressions

`\DimEval` $\{\langle dimension\ expression \rangle\}$

Evaluates the $\langle dimension\ expression \rangle$, expanding any dimensions and token list variables within the $\langle expression \rangle$ to their content (without requiring `\DimUse`/`\TlUse`) and applying the standard mathematical rules. The result of the calculation is returned as a $\langle dimension\ denotation \rangle$. For example

```
\DimEval {(1.2pt+3.4pt)/9}
```

0.51111pt

`\DimMathAdd` $\{\langle dimexpr_1 \rangle\} \{\langle dimexpr_2 \rangle\}$

Adds $\{\langle dimexpr_1 \rangle\}$ and $\{\langle dimexpr_2 \rangle\}$, and returns the result. For example

```
\DimMathAdd {2.8pt} {3.7pt}  
\DimMathAdd {3.8pt-1pt} {2.7pt+1pt}
```

6.5pt 6.5pt

 $\backslash\text{DimMathSub}$ $\{\langle\text{dimexpr}_1\rangle\}$ $\{\langle\text{dimexpr}_2\rangle\}$

Subtracts $\{\langle\text{dimexpr}_2\rangle\}$ from $\{\langle\text{dimexpr}_1\rangle\}$, and returns the result. For example

| | |
|---|-----------------|
| $\backslash\text{DimMathSub}$ $\{2.8\text{pt}\}$ $\{3.7\text{pt}\}$ | -0.9pt |
| $\backslash\text{DimMathSub}$ $\{3.8\text{pt}-1\text{pt}\}$ $\{2.7\text{pt}+1\text{pt}\}$ | -0.9pt |

 $\backslash\text{DimMathRatio}$ $\{\langle\text{dimexpr}_1\rangle\}$ $\{\langle\text{dimexpr}_2\rangle\}$

Parses the two $\langle\text{dimension expressions}\rangle$, then calculates the ratio of the two and returns it. The result is a ratio expression between two integers, with all distances converted to scaled points. For example

| | |
|--|-----------------|
| $\backslash\text{DimMathRatio}$ $\{5\text{pt}\}$ $\{10\text{pt}\}$ | $327680/655360$ |
|--|-----------------|

The returned value is suitable for use inside a $\langle\text{dimension expression}\rangle$ such as

| |
|---|
| $\backslash\text{DimSet}$ $\backslash\text{TmpaDim}$ $\{10\text{pt}*\backslash\text{DimMathRatio}\{5\text{pt}\}\{10\text{pt}\}\}$ |
|---|

 $\backslash\text{DimMathSign}$ $\{\langle\text{dimexpr}\rangle\}$

Evaluates the $\langle\text{dimexpr}\rangle$ then returns 1 or 0 or -1 according to the sign of the result. For example

| | |
|--|------|
| $\backslash\text{DimMathSign}$ $\{3.5\text{pt}\}$ | 1 |
| $\backslash\text{DimMathSign}$ $\{-2.7\text{pt}\}$ | -1 |

 $\backslash\text{DimMathAbs}$ $\{\langle\text{dimexpr}\rangle\}$

Converts the $\langle\text{dimexpr}\rangle$ to its absolute value, returning the result as a $\langle\text{dimension denotation}\rangle$. For example

| | |
|---|----------------|
| $\backslash\text{DimMathAbs}$ $\{3.5\text{pt}\}$ | 3.5pt |
| $\backslash\text{DimMathAbs}$ $\{-2.7\text{pt}\}$ | 2.7pt |

 $\backslash\text{DimMathMax}$ $\{\langle\text{dimexpr}_1\rangle\}$ $\{\langle\text{dimexpr}_2\rangle\}$

 $\backslash\text{DimMathMin}$ $\{\langle\text{dimexpr}_1\rangle\}$ $\{\langle\text{dimexpr}_2\rangle\}$

Evaluates the two $\langle\text{dimension expressions}\rangle$ and returns either the maximum or minimum value as appropriate as a $\langle\text{dimension denotation}\rangle$. For example

| | |
|--|-----------------|
| $\backslash\text{DimMathMax}$ $\{3.5\text{pt}\}$ $\{-2.7\text{pt}\}$ | 3.5pt |
| $\backslash\text{DimMathMin}$ $\{3.5\text{pt}\}$ $\{-2.7\text{pt}\}$ | -2.7pt |

9.3 Creating and Using Dimensions

 $\backslash\text{DimNew}$ $\langle\text{dimension}\rangle$

Creates a new $\langle\text{dimension}\rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle\text{dimension}\rangle$ is initially equal to 0 pt.

\DimConst $\langle dimension \rangle$ { $\langle dimension expression \rangle$ }

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 $\langle dimension expression \rangle$. For example

```
\DimConst \cFooSomeDim {1cm}
\DimUse \cFooSomeDim
```

28.45274pt

\DimUse $\langle dimension \rangle$

Recovers the content of a $\langle dimension \rangle$ and returns the value. An error is raised if the variable does not exist or if it is invalid.

9.4 Viewing Dimensions

\DimLog { $\langle dimension expression \rangle$ }

Writes the result of evaluating the $\langle dimension expression \rangle$ in the log file. For example

```
\DimLog {\lFooSomeDim+1cm}
```

\DimVarLog $\langle dimension \rangle$

Writes the value of the $\langle dimension \rangle$ in the log file. For example

```
\DimVarLog \lFooSomeDim
```

\DimShow { $\langle dimension expression \rangle$ }

Displays the result of evaluating the $\langle dimension expression \rangle$ on the terminal. For example

```
\DimShow {\lFooSomeDim+1cm}
```

\DimVarShow $\langle dimension \rangle$

Displays the value of the $\langle dimension \rangle$ on the terminal. For example

```
\DimVarShow \lFooSomeDim
```

9.5 Setting Dimension Variables

\DimSet $\langle dimension \rangle$ { $\langle dimension expression \rangle$ }

Sets $\langle dimension \rangle$ to the value of $\langle dimension expression \rangle$, which must evaluate to a length with units.

\DimSetEq $\langle dimension_1 \rangle$ $\langle dimension_2 \rangle$

Sets the content of $\langle dimension_1 \rangle$ equal to that of $\langle dimension_2 \rangle$. For example

```
\DimSet \lTmpaDim {10pt}
\DimSetEq \lTmpbDim \lTmpaDim
\DimUse \lTmpbDim
```

10.0pt

\DimZero $\langle dimension \rangle$

Sets $\langle dimension \rangle$ to 0 pt. For example

```
\DimSet \lTmpaDim {1em}
\DimZero \lTmpaDim
\DimUse \lTmpaDim
```

0.0pt

\DimZeroNew $\langle dimension \rangle$

Ensures that the $\langle dimension \rangle$ exists globally by applying **\DimNew** if necessary, then applies **\DimZero** to set the $\langle dimension \rangle$ to zero. For example

```
\DimZeroNew \lFooSomeDim
\DimUse \lFooSomeDim
```

0.0pt

\DimAdd $\langle dimension \rangle$ { $\langle dimension expression \rangle$ }

Adds the result of the $\langle dimension expression \rangle$ to the current content of the $\langle dimension \rangle$. For example

```
\DimSet \lTmpaDim {5.3pt}
\DimAdd \lTmpaDim {2.11pt}
\DimUse \lTmpaDim
```

7.41pt

\DimSub $\langle dimension \rangle$ { $\langle dimension expression \rangle$ }

Subtracts the result of the $\langle dimension expression \rangle$ from the current content of the $\langle dimension \rangle$. For example

```
\DimSet \lTmpaDim {5.3pt}
\DimSub \lTmpaDim {2.11pt}
\DimUse \lTmpaDim
```

3.19pt

9.6 Dimension Step Functions

\DimStepInline { $\langle initial value \rangle$ } { $\langle step \rangle$ } { $\langle final value \rangle$ } { $\langle code \rangle$ }

This function first evaluates the $\langle initial value \rangle$, $\langle step \rangle$ and $\langle final value \rangle$, all of which should be 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 #1 replaced by the current $\langle value \rangle$. Thus the $\langle code \rangle$ should define a function of one argument (#1). For example

```

\IgnoreSpacesOn
\TlClear \lTmptl
\DimStepInline {1pt} {0.1pt} {1.5pt} {
  \TlPutRight \lTmptl {[#1]}
}
\Result {\Value\lTmptl}
\IgnoreSpacesOff

```

produces [1.0pt][1.1pt][1.20001pt][1.30002pt][1.40002pt].

\DimStepVariable {<initial value>} {<step>} {<final value>} <tl var> {<code>}

This function first evaluates the <initial value>, <step> and <final value>, all of which should be dimension expressions. Then for each <value> from the <initial value> to the <final value> in turn (using <step> between each <value>), the <code> is inserted into the input stream, with the <tl var> defined as the current <value>. Thus the <code> should make use of the <tl var>.

9.7 Dimension Conditionals

\DimIfExist <dimension>
\DimIfExistTF <dimension> {<true code>} {<false code>}

Tests whether the <dimension> is currently defined. This does not check that the <dimension> really is a dimension variable. For example

```

\DimIfExistTF \lTmptDim {\Result{Yes}} {\Result{No}}
\DimIfExistTF \lFooUndefinedDim {\Result{Yes}} {\Result{No}}

```

Yes No

\DimCompare {<dimexpr₁>} <relation> {<dimexpr₂>}
\DimCompareTF {<dimexpr₁>} <relation> {<dimexpr₂>} {<true code>} {<false code>}

This function first evaluates each of the <dimension expressions> as described for **\DimEval**. The two results are then compared using the <relation>:

| | |
|--------------|---|
| Equal | = |
| Greater than | > |
| Less than | < |

For example

```

\DimCompareTF {1pt} > {0.9999pt} {\Result{Greater}} {\Result{Less}}
\DimCompareTF {1pt} > {1.0001pt} {\Result{Greater}} {\Result{Less}}

```

Greater Less

9.8 Dimension Case Functions

```
\DimCase {⟨test dimension expression⟩}
{
  {⟨dimexpr case1⟩} {⟨code case1⟩}
  {⟨dimexpr case2⟩} {⟨code case2⟩}
  ...
  {⟨dimexpr casen⟩} {⟨code casen⟩}
}
```

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.

```
\DimCaseT {⟨test dimension expression⟩}
{
  {⟨dimexpr case1⟩} {⟨code case1⟩}
  {⟨dimexpr case2⟩} {⟨code case2⟩}
  ...
  {⟨dimexpr casen⟩} {⟨code casen⟩}
}
{⟨true 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).

```
\DimCaseF {⟨test dimension expression⟩}
{
  {⟨dimexpr case1⟩} {⟨code case1⟩}
  {⟨dimexpr case2⟩} {⟨code case2⟩}
  ...
  {⟨dimexpr casen⟩} {⟨code casen⟩}
}
{⟨false 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 none of the cases match then the *⟨false code⟩* is inserted. For example

```
\IgnoreSpacesOn
\DimSet \lTmpaDim {5pt}
\DimCaseF {2\lTmpaDim} {
  {5pt}      {\Result{Small}}
  {4pt+6pt}  {\Result{Medium}}
  {-10pt}    {\Result{Negative}}
}{
  \Result {No Match}
}
\IgnoreSpacesOff
```

Medium

```

\DimCaseTF {⟨test dimension expression⟩}
{
  {⟨dimexpr case1⟩} {⟨code case1⟩}
  {⟨dimexpr case2⟩} {⟨code case2⟩}
  ...
  {⟨dimexpr casen⟩} {⟨code casen⟩}
}
{⟨true code⟩}
{⟨false code⟩}

```

This function evaluates the $\langle test\ dimension\ expression \rangle$ and compares this in turn to each of the $\langle dimension\ expression\ cases \rangle$. If the two are equal 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.

Chapter 10

Comma Separated Lists (Clist)

10.1 Constant and Scratch Comma Lists

\cEmptyClist

Constant that is always empty.

\lTmpaClist \lTmpbClist \lTmpcClist \lTmpiClist \lTnpjClist \lTmpkClist

Scratch comma lists for local assignment. These are never used by the **functional** package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

\gTmpaClist \gTmpbClist \gTmpcClist \gTmpiClist \gTnpjClist \gTmpkClist

Scratch comma lists for global assignment. These are never used by the **functional** package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

10.2 Creating and Using Comma Lists

\ClistNew *<comma list>*

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.

```
\ClistNew \lFooSomeClist
```

\ClistConst *<clist var>* {*<comma list>*}

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

```
\ClistConst \cFooSomeClist {one,two,three}
```

\ClistVarJoin *<clist var>* {*<separator>*}

Returns the contents of the *<clist var>*, with the *<separator>* between the items.

```
\ClistSet \lTmpaClist { a , b , , c , {de} , f }
\ClistVarJoin \lTmpaClist { and }
```

```
a and b and c and de and f
```

\ClistVarJoinExtended *<clist var>* {*<separator between two>*} {*<separator between more than two>*} {*<separator between final two>*}

Returns the contents of the *<clist var>*, 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 joined with the *<separator between two>* and returns.

```
\ClistSet \lTmpaClist { a , b }
\ClistVarJoinExtended \lTmpaClist { and } { , } { , and }
```

```
a and b
```

```
\ClistSet \lTmpaClist { a , b , , c , {de} , f }
\ClistVarJoinExtended \lTmpaClist { and } { , } { , and }
```

```
a, b, c, de, and f
```

\ClistJoin *<comma list>* {*<separator>*}

\ClistJoinExtended *<comma list>* {*<separator between two>*} {*<separator between more than two>*} {*<separator between final two>*}

Returns the contents of the *<comma list>*, with the appropriate *<separator>* between the items. As for **\ClistSet**, 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. The *<separators>* are then inserted in the same way as for **\ClistVarJoin** and **\ClistVarJoinExtended**, respectively.

```
\ClistJoinExtended { a , b } { and } { , } { , and }
```

```
a and b
```

```
\ClistJoinExtended { a , b , , c , {de} , f } { and } { , } { , and }
```

```
a, b, c, de, and f
```

10.3 Viewing Comma Lists

\ClistLog {*<tokens>*}

Writes the entries in the comma list in the log file. See also **\ClistShow** which displays the result in the terminal.

```
\ClistLog {one,two,three}
```

\ClistVarLog *<comma list>*

Writes the entries in the *<comma list>* in the log file. See also **\ClistVarShow** which displays the result in the terminal.

```
\ClistSet \lTmpaClist {one,two,three}
\ClistVarLog \lTmpaClist
```

\ClistShow {*<tokens>*}

Displays the entries in the comma list in the terminal.

```
\ClistShow {one,two,three}
```

```
\ClistVarShow <comma list>
```

Displays the entries in the *<comma list>* in the terminal.

```
\ClistSet \lTmpaClist {one,two,three}
\ClistVarShow \lTmpaClist
```

10.4 Setting Comma Lists

```
\ClistSet <comma list> {<item1>,...,<itemn>}
```

Sets *<comma list>* to contain the *<items>*, 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 *<tokens>* as a single *<item>* even if the *<tokens>* contain commas or spaces, add a set of braces: `\ClistSet <comma list> { {<tokens>} }`.

```
\ClistSet \lTmpaClist {one,two,three}
\ClistVarJoin \lTmpaClist { and }
```

```
one and two and three
```

```
\ClistSetEq <comma list1> <comma list2>
```

Sets the content of *<comma list₁>* equal to that of *<comma list₂>*. To set a token list variable equal to a comma list variable, use `\TlSetEq`. Conversely, setting a comma list variable to a token list is unadvisable unless one checks space-trimming and related issues.

```
\ClistSet \lTmpaClist {one,two,three,four}
\ClistSetEq \lTmpbClist \lTmpaClist
\ClistVarJoin \lTmpbClist { and }
```

```
one and two and three and four
```

```
\ClistSetFromSeq <comma list> <sequence>
```

Converts the data in the *<sequence>* into a *<comma list>*: the original *<sequence>* is unchanged. Items which contain either spaces or commas are surrounded by braces.

```
\SeqPutRight \lTmpaSeq {one}
\SeqPutRight \lTmpaSeq {two}
\ClistSetFromSeq \lTmpaClist \lTmpaSeq
\ClistVarJoin \lTmpaClist { and }
```

```
one and two
```

```
\ClistClear <comma list>
```

Clears all items from the *<comma list>*.

```
\ClistSet \lTmpaClist {one,two,three,four}
\ClistClear \lTmpaClist
```

\ClistClearNew *<comma list>*

Ensures that the *<comma list>* exists globally by applying **\ClistNew** if necessary, then applies **\ClistClear** to leave the list empty.

```
\ClistClearNew \lFooSomeClist
\ClistSet \lFooSomeClist {one,two,three}
\ClistVarJoin \lFooSomeClist { and }
```

one and two and three

\ClistConcat *<comma list₁>* *<comma list₂>* *<comma list₃>*

Concatenates the content of *<comma list₂>* and *<comma list₃>* together and saves the result in *<comma list₁>*. The items in *<comma list₂>* are placed at the left side of the new comma list.

```
\ClistSet \lTmpbClist {one,two}
\ClistSet \lTmpcClist {three,four}
\ClistConcat \lTmpaClist \lTmpbClist \lTmpcClist
\ClistVarJoin \lTmpaClist { + }
```

one + two + three + four

\ClistPutLeft *<comma list>* {*<item₁>*,...,*<item_n>*}

Appends the *<items>* to the left of the *<comma list>*. 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 *<tokens>* as a single *<item>* even if the *<tokens>* contain commas or spaces, add a set of braces: **\ClistPutLeft** *<comma list>* { {*<tokens>*} }.

```
\ClistSet \lTmpaClist {one,two}
\ClistPutLeft \lTmpaClist {zero}
\ClistVarJoin \lTmpaClist { and }
```

zero and one and two

\ClistPutRight *<comma list>* {*<item₁>*,...,*<item_n>*}

Appends the *<items>* to the right of the *<comma list>*. 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 *<tokens>* as a single *<item>* even if the *<tokens>* contain commas or spaces, add a set of braces: **\ClistPutRight** *<comma list>* { {*<tokens>*} }.

```
\ClistSet \lTmpaClist {one,two}
\ClistPutRight \lTmpaClist {three}
\ClistVarJoin \lTmpaClist { and }
```

one and two and three

10.5 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.

\ClistVarRemoveDuplicates *<comma list>*

Removes duplicate items from the *<comma list>*, leaving the left most copy of each item in the *<comma list>*. The *<item>* comparison takes place on a token basis, as for **\TlIfEqTF**.

```
\ClistSet \lTmpaClist {one,two,one,two,three}
\ClistVarRemoveDuplicates \lTmpaClist
\ClistVarJoin \lTmpaClist {,}
```

one,two,three

\ClistVarRemoveAll $\langle comma list \rangle$ $\{\langle item \rangle\}$

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 **\TlIfEqTF**.

```
\ClistSet \lTmpaClist {one,two,one,two,three}
\ClistVarRemoveAll \lTmpaClist {two}
\ClistVarJoin \lTmpaClist {,}
```

one,one,three

\ClistVarReverse $\langle comma list \rangle$

Reverses the order of items stored in the $\langle comma list \rangle$.

```
\ClistSet \lTmpaClist {one,two,one,two,three}
\ClistVarReverse \lTmpaClist
\ClistVarJoin \lTmpaClist {,}
```

three,two,one,two,one

10.6 Working with the Contents of Comma Lists

\ClistCount $\{\langle comma list \rangle\}$
\ClistVarCount $\langle comma list \rangle$

Returns the number of items in the $\langle comma list \rangle$ as an $\langle integer denotation \rangle$. The total number of items in a $\langle comma list \rangle$ includes those which are duplicates, *i.e.* every item in a $\langle comma list \rangle$ is counted.

```
\ClistSet \lTmpaClist {one,two,three,four}
\ClistVarCount \lTmpaClist
```

4

\ClistItem $\{\langle comma list \rangle\} \{\langle integer expression \rangle\}$

Indexing items in the $\langle comma list \rangle$ from 1 at the top (left), this function evaluates the $\langle integer expression \rangle$ and returns the appropriate item from the comma list. If the $\langle integer expression \rangle$ is negative, indexing occurs from the bottom (right) of the comma list. When the $\langle integer expression \rangle$ is larger than the number of items in the $\langle comma list \rangle$ (as calculated by **\ClistCount**) then the function returns nothing.

```
\TlSet \lTmpaTl {\ClistItem {one,two,three,four} {3}}
\TlUse \lTmpaTl
```

three

\ClistVarItem $\langle comma list \rangle \{\langle integer expression \rangle\}$

Indexing items in the $\langle comma list \rangle$ from 1 at the top (left), this function evaluates the $\langle integer expression \rangle$ and returns the appropriate item from the comma list. If the $\langle integer expression \rangle$ is negative, indexing occurs from the bottom (right) of the comma list. When the $\langle integer expression \rangle$ is larger than the number of items in the $\langle comma list \rangle$ (as calculated by **\ClistVarCount**) then the function returns nothing.

```
\ClistSet \lTmpaClist {one,two,three,four}
\TlSet \lTmpaTl {\ClistVarItem \lTmpaClist {3}}
\TlUse \lTmpaTl
```

three

```
\ClistRandItem {<comma list>}
\ClistVarRandItem <clist var>
```

Selects a pseudo-random item of the *<comma list>*. If the *<comma list>* has no item, the result is empty.

```
\TlSet \lTmpaTl {\ClistRandItem {one,two,three,four,five,six}}
\TlUse \lTmpaTl
\TlSet \lTmpaTl {\ClistRandItem {one,two,three,four,five,six}}
\TlUse \lTmpaTl
```

two one

10.7 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.

```
\ClistGet <comma list> <token list variable>
```

Stores the left-most item from the *<comma list>* in the *<token list variable>* without removing it from the *<comma list>*. The *<token list variable>* is assigned locally. If the *<comma list>* is empty the *<token list variable>* is set to the marker value **\qNoValue**.

```
\ClistSet \lTmpaClist {two,three,four}
\ClistGet \lTmpaClist \lTmpaTl
\TlUse \lTmpaTl
```

two

```
\ClistGetT <comma list> <token list variable> <true code>
\ClistGetF <comma list> <token list variable> <false code>
\ClistGetTF <comma list> <token list variable> <true code> <false code>
```

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, stores the left-most item from the *<comma list>* in the *<token list variable>* without removing it from the *<comma list>*. The *<token list variable>* is assigned locally.

```
\ClistSet \lTmpaClist {two,three,four}
\ClistGetTF \lTmpaClist \lTmpaTl {\Result{Yes}} {\Result{No}}
```

Yes

```
\ClistPop <comma list> <token list variable>
```

Pops the left-most item from a *<comma list>* into the *<token list variable>*, *i.e.* removes the item from the comma list and stores it in the *<token list variable>*. The assignment of the *<token list variable>* is local. If the *<comma list>* is empty the *<token list variable>* is set to the marker value **\qNoValue**.

```
\ClistSet \lTmpaClist {two,three,four}
\ClistPop \lTmpaClist \lTmpaTl
\ClistVarJoin \lTmpaClist {,}
```

three,four

```

\clistpopT <comma list> <token list variable> {\true code}
\clistpopF <comma list> <token list variable> {\false code}
\clistpopTF <comma list> <token list variable> {\true code} {\false code}

```

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 *<token list variable>* is assigned locally.

```

\clistset \lTmplist {two,three,four}
\clistpopTF \lTmplist \lTmpl { \Result{Yes} } { \Result{No} }

```

Yes

```

\clistpush <comma list> {\items}

```

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.

```

\clistset \lTmplist {two,three,four}
\clistpush \lTmplist {zero,one}
\clistvarjoin \lTmplist {}

```

zero|one|two|three|four

10.8 Mapping over Comma Lists

When the comma list is given explicitly, 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 a variable, 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 explicit comma lists.

```

\clistmapinline {\comma list} {\inline function}
\clistvarmapinline <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. For example

```

\ignorespaceson
\Tlclear \lTmpl
\clistmapinline {one,two,three} {
  \Tlputright \lTmpl {(#1)}
}
\result {\Tluse\lTmpl}
\ignorespacesoff

```

produces (one)(two)(three).

```

\clistmapvariable {\comma list} <variable> {\code}
\clistvarmapvariable <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.

```

\IgnoreSpacesOn
\clistMapVariable {one,two,three} \lTmpiTl {
  \TlPutRight \gTmptl {\Expand {(\lTmpiTl)}}
}
\TlUse \gTmptl
\IgnoreSpacesOff

```

(one)(two)(three)

10.9 Comma List Conditionals

```

\clistIfExist <comma list>
\clistIfExistTF <comma list> {\true code} {\false code}

```

Tests whether the *<comma list>* is currently defined. This does not check that the *<comma list>* really is a comma list.

```

\clistIfExistTF \lTmptcllist {\Result{Yes}} {\Result{No}}
\clistIfExistTF \lFooUndefinedClist {\Result{Yes}} {\Result{No}}

```

Yes No

```

\clistIfEmpty {\comma list}
\clistIfEmptyTF {\comma list} {\true code} {\false code}

```

Tests if the *<comma list>* is empty (containing no items). The rules for space trimming are as for other n-type comma-list functions, hence the comma list { , , } (without outer braces) is empty, while { ,{},{ } } (without outer braces) contains one element, which happens to be empty: the comma-list is not empty.

```

\clistIfEmptyTF {one,two} {\Result{Empty}} {\Result{NonEmpty}}
\clistIfEmptyTF { , } {\Result{Empty}} {\Result{NonEmpty}}

```

NonEmpty Empty

```

\clistVarIfEmpty <comma list>
\clistVarIfEmptyTF <comma list> {\true code} {\false code}

```

Tests if the *<comma list>* is empty (containing no items).

```

\clistSet \lTmptcllist {one,two}
\clistVarIfEmptyTF \lTmptcllist {\Result{Empty}} {\Result{NonEmpty}}
\clistClear \lTmptcllist
\clistVarIfEmptyTF \lTmptcllist {\Result{Empty}} {\Result{NonEmpty}}

```

NonEmpty Empty

```

\clistIfIn {\comma list} {\item}
\clistIfInTF {\comma list} {\item} {\true code} {\false code}

```

Tests if the *<item>* is present in the *<comma list>*. In the case of an n-type *<comma list>*, the usual rules of space trimming and brace stripping apply. For example

```

\clistIfInTF { a , {b} , {b} , c } {b} {\Result{Yes}} {\Result{No}}
\clistIfInTF { a , {b} , {b} , c } {d} {\Result{Yes}} {\Result{No}}

```

Yes No

 $\backslash\text{ClistVarIfIn}$ $\langle comma\ list \rangle$ $\{\langle item \rangle\}$
 $\backslash\text{ClistVarIfInTF}$ $\langle comma\ list \rangle$ $\{\langle item \rangle\}$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

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.

```
\ClistSet \lTmпаClist {one,two}
```

```
\ClistVarIfInTF \lTmпаClist {one} {\Result{Yes}} {\Result{No}}
```

```
\ClistVarIfInTF \lTmпаClist {three} {\Result{Yes}} {\Result{No}}
```

| |
|--------|
| Yes No |
|--------|

Chapter 11

Sequences and Stacks (Seq)

11.1 Constant and Scratch Sequences

`\cEmptySeq`

Constant that is always empty.

`\lTmPaSeq \lTmPbSeq \lTmPcSeq \lTmPiSeq \lTmPjSeq \lTmPkSeq`

Scratch sequences for local assignment. These are never used by the **functional** package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

`\gTmPaSeq \gTmPbSeq \gTmPcSeq \gTmPiSeq \gTmPjSeq \gTmPkSeq`

Scratch sequences for global assignment. These are never used by the **functional** package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

11.2 Creating and Using Sequences

`\SeqNew` $\langle sequence \rangle$

Creates a new $\langle sequence \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle sequence \rangle$ initially contains no items.

```
\SeqNew \lFooSomeSeq
```

`\SeqConstFromClist` $\langle seq\ var \rangle$ $\{ \langle comma\ list \rangle \}$

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 $\langle comma\ list \rangle$.

```
\SeqConstFromClist \cFooSomeSeq {one,two,three}
```

`\SeqVarJoin` $\langle seq\ var \rangle$ $\{ \langle separator \rangle \}$

Returns the contents of the $\langle seq\ var \rangle$, with the $\langle separator \rangle$ between the items. If the sequence has a single

item, it is returned with no *<separator>*, and an empty sequence returns nothing. An error is raised if the variable does not exist or if it is invalid.

```
\SeqSetSplit \lTmpaSeq {} {a|b|c|{de}|f}
\SeqVarJoin \lTmpaSeq { and }
```

a and b and c and de and f

\SeqVarJoinExtended *<seq var>* {*<separator between two>*} {*<separator between more than two>*} {*<separator between final two>*}

Returns the contents of the *<seq var>*, with the appropriate *<separator>* between the items. Namely, if the sequence 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 sequence has exactly two items, then they are joined with the *<separator between two>* and returned. If the sequence has a single item, it is returned, and an empty sequence returns nothing. An error is raised if the variable does not exist or if it is invalid.

```
\SeqSetSplit \lTmpaSeq {} {a|b|c|{de}|f}
\SeqVarJoinExtended \lTmpaSeq { and } {, } {, and }
```

a, b, c, de, and f

The first separator argument is not used in this case because the sequence has more than 2 items.

11.3 Viewing Sequences

\SeqVarLog *<sequence>*

Writes the entries in the *<sequence>* in the log file.

```
\SeqVarLog \lFooSomeSeq
```

\SeqVarShow *<sequence>*

Displays the entries in the *<sequence>* in the terminal.

```
\SeqVarShow \lFooSomeSeq
```

11.4 Setting Sequences

\SeqSetFromClist *<sequence>* *<comma-list>*

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

```
\SeqSetFromClist \lTmpaSeq {one,two,three}
\SeqVarJoin \lTmpaSeq { and }
```

one and two and three

\SeqSetSplit *<sequence>* {*<delimiter>*} {*<token list>*}

Splits the *<token list>* into *<items>* separated by *<delimiter>*, and assigns the result to the *<sequence>*. Spaces on both sides of each *<item>* are ignored, then one set of outer braces is removed (if any); this space trimming behaviour is identical to that of Clist functions. Empty *<items>* are preserved by **\SeqSetSplit**,

and can be removed afterwards using `\SeqVarRemoveAll` $\langle sequence \rangle$ $\{ \}$. The $\langle delimiter \rangle$ may not contain $\{, \}$ or $\#$ (assuming T_EX'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 $\langle token list \rangle$.

```
\SeqSetSplit \lTmpaSeq {,} {1,2,3}
\SeqVarJoin \lTmpaSeq { and }
```

1 and 2 and 3

\SeqSetEq $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$

Sets the content of $\langle sequence_1 \rangle$ equal to that of $\langle sequence_2 \rangle$.

```
\SeqSetFromClist \lTmpaSeq {one,two,three,four}
\SeqSetEq \lTmpbSeq \lTmpaSeq
\SeqVarJoin \lTmpbSeq { and }
```

one and two and three and four

\SeqClear $\langle sequence \rangle$

Clears all items from the $\langle sequence \rangle$.

```
\SeqClear \lTmpaSeq
```

\SeqClearNew $\langle sequence \rangle$

Ensures that the $\langle sequence \rangle$ exists globally by applying `\SeqNew` if necessary, then applies `\SeqClear` to leave the $\langle sequence \rangle$ empty.

```
\SeqClearNew \lFooSomeSeq
\SeqSetFromClist \lFooSomeSeq {one,two,three}
\SeqVarJoin \lFooSomeSeq { and }
```

one and two and three

\SeqConcat $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$ $\langle sequence_3 \rangle$

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.

```
\SeqSetFromClist \lTmpbSeq {one,two}
\SeqSetFromClist \lTmpcSeq {three,four}
\SeqConcat \lTmpaSeq \lTmpbSeq \lTmpcSeq
\SeqVarJoin \lTmpaSeq {, }
```

one, two, three, four

\SeqPutLeft $\langle sequence \rangle$ $\{ \langle item \rangle \}$

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

```
\SeqSetFromClist \lTmpaSeq {one,two}
\SeqPutLeft \lTmpaSeq {zero}
\SeqVarJoin \lTmpaSeq { and }
```

zero and one and two

\SeqPutRight $\langle sequence \rangle$ $\{ \langle item \rangle \}$

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

```
\SeqSetFromClist \lTmpaSeq {one,two}
\SeqPutRight \lTmpaSeq {three}
\SeqVarJoin \lTmpaSeq { and }
```

```
one and two and three
```

11.5 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.

\SeqVarRemoveDuplicates $\langle sequence \rangle$

Removes duplicate items from the $\langle sequence \rangle$, leaving the left most copy of each item in the $\langle sequence \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for **\TlIfEqTF**.

```
\SeqSetFromClist \lTmpaSeq {one,two,one,two,three}
\SeqVarRemoveDuplicates \lTmpaSeq
\SeqVarJoin \lTmpaSeq {,}
```

```
one,two,three
```

\SeqVarRemoveAll $\langle sequence \rangle \{ \langle item \rangle \}$

Removes every occurrence of $\langle item \rangle$ from the $\langle sequence \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for **\TlIfEqTF**.

```
\SeqSetFromClist \lTmpaSeq {one,two,one,two,three}
\SeqVarRemoveAll \lTmpaSeq {two}
\SeqVarJoin \lTmpaSeq {,}
```

```
one,one,three
```

\SeqVarReverse $\langle sequence \rangle$

Reverses the order of the items stored in the $\langle sequence \rangle$.

```
\SeqSetFromClist \lTmpaSeq {one,two,one,two,three}
\SeqVarReverse \lTmpaSeq
\SeqVarJoin \lTmpaSeq {,}
```

```
three,two,one,two,one
```

11.6 Working with the Contents of Sequences

\SeqVarCount $\langle sequence \rangle$

Returns the number of items in the $\langle sequence \rangle$ 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.

```
\SeqSetFromClist \lTmpaSeq {one,two,three,four}
\SeqVarCount \lTmpaSeq
```

```
4
```

\SeqVarItem $\langle sequence \rangle \{ \langle integer expression \rangle \}$

Indexing items in the $\langle sequence \rangle$ from 1 at the top (left), this function evaluates the $\langle integer expression \rangle$ and returns the appropriate item from the sequence. If the $\langle integer expression \rangle$ is negative, indexing

occurs from the bottom (right) of the sequence. If the *<integer expression>* is larger than the number of items in the *<sequence>* (as calculated by `\SeqVarCount`) then the function returns nothing.

```
\SeqSetFromClist \lTmpaSeq {one,two,three,four}
\TlSet \lTmpaTl {\SeqVarItem \lTmpaSeq {3}}
\TlUse \lTmpaTl
```

three

`\SeqVarRandItem` *<seq var>*

Selects a pseudo-random item of the *<sequence>*. If the *<sequence>* is empty the result is empty.

```
\SeqSetFromClist \lTmpaSeq {one,two,three,four,five,six}
\TlSet \lTmpaTl {\SeqVarRandItem \lTmpaSeq}
\TlUse \lTmpaTl
\TlSet \lTmpaTl {\SeqVarRandItem \lTmpaSeq}
\TlUse \lTmpaTl
```

four two

11.7 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.

`\SeqGet` *<sequence>* *<token list variable>*

Reads the top item from a *<sequence>* into the *<token list variable>* without removing it from the *<sequence>*. The *<token list variable>* is assigned locally. If *<sequence>* is empty the *<token list variable>* is set to the special marker `\qNoValue`.

```
\SeqSetFromClist \lTmpaSeq {two,three,four}
\SeqGet \lTmpaSeq \lTmpaTl
\TlUse \lTmpaTl
```

two

`\SeqGetT` *<sequence>* *<token list variable>* *{<true code>}*

`\SeqGetF` *<sequence>* *<token list variable>* *{<false code>}*

`\SeqGetTF` *<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, stores the top item from a *<sequence>* in the *<token list variable>* without removing it from the *<sequence>*. The *<token list variable>* is assigned locally.

```
\SeqSetFromClist \lTmpaSeq {two,three,four}
\SeqGetTF \lTmpaSeq \lTmpaTl {\Result{Yes}} {\Result{No}}
```

Yes

`\SeqPop` *<sequence>* *<token list variable>*

Pops the top item from a *<sequence>* into the *<token list variable>*. the *<token list variable>* is assigned locally. If *<sequence>* is empty the *<token list variable>* is set to the special marker `\qNoValue`.

```
\SeqSetFromClist \lTmpaSeq {two,three,four}
\SeqPop \lTmpaSeq \lTmpaTl
\SeqVarJoin \lTmpaSeq {,}
```

three,four

```
\SeqPopT <sequence> <token list variable> {\true code}
\SeqPopF <sequence> <token list variable> {\false code}
\SeqPopTF <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 *<token list variable>* is assigned locally.

```
\SeqPopTF \cEmptySeq \lTmpaTl {\Result{Yes}} {\Result{No}}
```

No

```
\SeqPush <sequence> {\item}
```

Adds the *{\item}* to the top of the *<sequence>*.

```
\SeqSetFromClist \lTmpaSeq {two,three,four}
\SeqPush \lTmpaSeq {one}
\SeqVarJoin \lTmpaSeq {||}
```

one|two|three|four

You can only push one item to the *<sequence>* with **\SeqPush**, which is different from **\ClistPush**.

11.8 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.

```
\SeqGetLeft <sequence> <token list variable>
```

Stores the left-most item from a *<sequence>* in the *<token list variable>* without removing it from the *<sequence>*. The *<token list variable>* is assigned locally. If *<sequence>* is empty the *<token list variable>* is set to the special marker **\qNoValue**.

```
\SeqSetFromClist \lTmpaSeq {two,three,four}
\SeqGetLeft \lTmpaSeq \lTmpaTl
\TlUse \lTmpaTl
```

two

```
\SeqGetLeftT <sequence> <token list variable> {\true code}
\SeqGetLeftF <sequence> <token list variable> {\false code}
\SeqGetLeftTF <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, stores the left-most item from the *<sequence>* in the *<token list variable>* without removing it from the *<sequence>*, then leaves the *<true code>* in the input stream. The *<token list variable>* is assigned locally.

```
\SeqSetFromClist \lTmpaSeq {two,three,four}
\SeqGetLeftTF \lTmpaSeq \lTmpaTl {\Result{Yes}} {\Result{No}}
```

Yes

\SeqGetRight *<sequence>* *<token list variable>*

Stores the right-most item from a *<sequence>* in the *<token list variable>* without removing it from the *<sequence>*. The *<token list variable>* is assigned locally. If *<sequence>* is empty the *<token list variable>* is set to the special marker **\qNoValue**.

```
\SeqSetFromClist \lTmpaSeq {two,three,four}
\SeqGetRight \lTmpaSeq \lTmpaTl
\TlUse \lTmpaTl
```

four

\SeqGetRightT *<sequence>* *<token list variable>* *{<true code>}*
\SeqGetRightF *<sequence>* *<token list variable>* *{<false code>}*
\SeqGetRightTF *<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, stores the right-most item from the *<sequence>* in the *<token list variable>* without removing it from the *<sequence>*, then leaves the *<true code>* in the input stream. The *<token list variable>* is assigned locally.

```
\SeqSetFromClist \lTmpaSeq {two,three,four}
\SeqGetRightTF \lTmpaSeq \lTmpaTl {\Result{Yes}} {\Result{No}}
```

Yes

\SeqPopLeft *<sequence>* *<token list variable>*

Pops the left-most item from a *<sequence>* into the *<token list variable>*, *i.e.* removes the item from the sequence and stores it in the *<token list variable>*. The assignment of the *<token list variable>* is local. If *<sequence>* is empty the *<token list variable>* is set to the special marker **\qNoValue**.

```
\SeqSetFromClist \lTmpaSeq {two,three,four}
\SeqPopLeft \lTmpaSeq \lTmpaTl
\SeqVarJoin \lTmpaSeq {,}
```

three,four

\SeqPopLeftT *<sequence>* *<token list variable>* *{<true code>}*
\SeqPopLeftF *<sequence>* *<token list variable>* *{<false code>}*
\SeqPopLeftTF *<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 left-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 *<token list variable>* is assigned locally.

```
\SeqPopLeftTF \cEmptySeq \lTmpaTl {\Result{Yes}} {\Result{No}}
```

No

\SeqPopRight *<sequence>* *<token list variable>*

Pops the right-most item from a *<sequence>* into the *<token list variable>*, *i.e.* removes the item from the sequence and stores it in the *<token list variable>*. The assignment of the *<token list variable>* is local. If *<sequence>* is empty the *<token list variable>* is set to the special marker **\qNoValue**.

```
\SeqSetFromClist \lTmpaSeq {two,three,four}
\SeqPopRight \lTmpaSeq \lTmpaTl
\SeqVarJoin \lTmpaSeq {,}
```

two,three

```

\SeqPopRightT <sequence> <token list variable> {\true code}
\SeqPopRightF <sequence> <token list variable> {\false code}
\SeqPopRightTF <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 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 *<token list variable>* is assigned locally.

```
\SeqPopRightTF \cEmptySeq \lTmplTl {\Result{Yes}} {\Result{No}}
```

No

11.9 Mapping over Sequences

```
\SeqVarMapInline <sequence> {\inline function}
```

Applies *<inline function>* to every *<item>* stored within the *<sequence>*. The *<inline function>* should consist of code which will receive the *<item>* as #1. The *<items>* are returned from left to right. For example,

```

\IgnoreSpacesOn
\SeqSetFromClist \lTmplSeq {one,two,three}
\TlClear \lTmplTl
\SeqVarMapInline \lTmplSeq {
  \TlPutRight \lTmplTl {(#1)}
}
\Result {\TlUse\lTmplTl}
\IgnoreSpacesOff

```

produces (one)(two)(three).

```
\SeqVarMapVariable <sequence> <variable> {\code}
```

Stores each *<item>* of the *<sequence>* 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 *<sequence>*, or its original value if the *<sequence>* is empty. The *<items>* are returned from left to right.

```

\IgnoreSpacesOn
\IntZero \lTmplInt
\SeqSetFromClist \lTmplSeq {1,3,7}
\SeqVarMapVariable \lTmplSeq \lTmplTl {
  \IntAdd \lTmplInt {\lTmplTl*\lTmplTl}
}
\Result {\IntUse\lTmplInt}
\IgnoreSpacesOff

```

59

11.10 Sequence Conditionals

```

\SeqIfExist <sequence>
\SeqIfExistTF <sequence> {\true code} {\false code}

```

Tests whether the *<sequence>* is currently defined. This does not check that the *<sequence>* really is a sequence variable.

```
\SeqIfExistTF \lTmpaSeq {\Result{Yes}} {\Result{No}}
\SeqIfExistTF \lFooUndefinedSeq {\Result{Yes}} {\Result{No}}
```

Yes No

```
\SeqVarIfEmpty <sequence>
\SeqVarIfEmptyTF <sequence> {\true code} {\false code}
```

Tests if the *<sequence>* is empty (containing no items).

```
\SeqSetFromClist \lTmpaSeq {one,two}
\SeqVarIfEmptyTF \lTmpaSeq {\Result{Empty}} {\Result{NonEmpty}}
\SeqClear \lTmpaSeq
\SeqVarIfEmptyTF \lTmpaSeq {\Result{Empty}} {\Result{NonEmpty}}
```

NonEmpty Empty

```
\SeqVarIfIn <sequence> {\item}
\SeqVarIfInTF <sequence> {\item} {\true code} {\false code}
```

Tests if the *<item>* is present in the *<sequence>*.

```
\SeqSetFromClist \lTmpaSeq {one,two}
\SeqVarIfInTF \lTmpaSeq {one} {\Result{Yes}} {\Result{Not}}
\SeqVarIfInTF \lTmpaSeq {three} {\Result{Yes}} {\Result{Not}}
```

Yes Not

Chapter 12

Property Lists (Prop)

L^AT_EX3 implements a “property list” data type, which contain an unordered list of entries each of which consists of a *⟨key⟩* and an associated *⟨value⟩*. The *⟨key⟩* and *⟨value⟩* may both be any *⟨balanced text⟩*, the *⟨key⟩* is processed using `\TlToStr`, meaning that category codes are ignored. 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 *⟨key⟩*: if an entry is added to a property list which already contains the *⟨key⟩* then the new entry overwrites the existing one. The *⟨keys⟩* are compared on a string basis, using the same method as `\StrIfEq`.

12.1 Constant and Scratch Sequences

`\cEmptyProp`

Constant that is always empty.

`\lTmпаProp \lTmрbProp \lTmрcProp \lTmрiProp \lTmрjProp \lTmрkProp`

Scratch property lists for local assignment. These are never used by the `functional` package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

`\gTmпаProp \gTmрbProp \gTmрcProp \gTmрiProp \gTmрjProp \gTmрkProp`

Scratch property lists for global assignment. These are never used by the `functional` package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

12.2 Creating and Using Property Lists

`\PropNew` *⟨property list⟩*

Creates a new *⟨property list⟩* or raises an error if the name is already taken. The declaration is global. The *⟨property list⟩* initially contains no entries.

```
\PropNew \lFooSomeProp
```

```
\PropConstFromKeyval <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, processed in the way described for **\PropSetFromKeyval**. If duplicate keys appear only the last of the values is kept. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

```
\PropConstFromKeyval \cFooSomeProp {key1=one,key2=two,key3=three}
```

```
\PropToKeyval <property list>
```

Returns the *<property list>* in a key–value notation. Keep in mind that a *<property list>* is *unordered*, while key–value interfaces don’t necessarily are, so this can’t be used for arbitrary interfaces.

```
\PropToKeyval \lTmPaProp
```

12.3 Viewing Property Lists

```
\PropVarLog <property list>
```

Writes the entries in the *<property list>* in the log file.

```
\PropVarLog \lTmPaProp
```

```
\PropVarShow <property list>
```

Displays the entries in the *<property list>* in the terminal.

```
\PropVarShow \lTmPaProp
```

12.4 Setting Property Lists

```
\PropSetFromKeyval <prop var>
{
  <key1> = <value1> ,
  <key2> = <value2> , ...
}
```

Sets *<prop var>* to contain key–value pairs given in the second argument. If duplicate keys appear only the last of the values is kept.

Spaces are trimmed around every *<key>* and every *<value>*, and if the result of trimming spaces consists of a single brace group then a set of outer braces is removed. This enables both the *<key>* and the *<value>* to contain spaces, commas or equal signs. The *<key>* is then processed by **\TlToStr**. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

```
\PropSetFromKeyval \lTmPaProp {key1=one,key2=two}
```

\PropSetEq $\langle \text{property list}_1 \rangle$ $\langle \text{property list}_2 \rangle$

Sets the content of $\langle \text{property list}_1 \rangle$ equal to that of $\langle \text{property list}_2 \rangle$.

```
\PropSetFromKeyval \lTmPaProp {key1=one,key2=two,key3=three}
\PropSetEq \lTmPbProp \lTmPaProp
\PropVarLog \lTmPbProp
```

\PropClear $\langle \text{property list} \rangle$

Clears all entries from the $\langle \text{property list} \rangle$.

```
\PropClear \lTmPaProp
```

\PropClearNew $\langle \text{property list} \rangle$

Ensures that the $\langle \text{property list} \rangle$ exists globally by applying **\PropNew** if necessary, then applies **\PropClear** to leave the list empty.

```
\PropClearNew \lFooSomeProp
```

\PropConcat $\langle \text{prop var}_1 \rangle$ $\langle \text{prop var}_2 \rangle$ $\langle \text{prop var}_3 \rangle$

Combines the key–value pairs of $\langle \text{prop var}_2 \rangle$ and $\langle \text{prop var}_3 \rangle$, and saves the result in $\langle \text{prop var}_1 \rangle$. If a key appears in both $\langle \text{prop var}_2 \rangle$ and $\langle \text{prop var}_3 \rangle$ then the last value, namely the value in $\langle \text{prop var}_3 \rangle$ is kept.

```
\PropSetFromKeyval \lTmPbProp {key1=one,key2=two}
\PropSetFromKeyval \lTmPcProp {key3=three,key4=four}
\PropConcat \lTmPaProp \lTmPbProp \lTmPcProp
\PropVarLog \lTmPaProp
```

\PropPut $\langle \text{property list} \rangle$ $\{ \langle \text{key} \rangle \}$ $\{ \langle \text{value} \rangle \}$

Adds an entry to the $\langle \text{property list} \rangle$ which may be accessed using the $\langle \text{key} \rangle$ and which has $\langle \text{value} \rangle$. If the $\langle \text{key} \rangle$ is already present in the $\langle \text{property list} \rangle$, the existing entry is overwritten by the new $\langle \text{value} \rangle$. Both the $\langle \text{key} \rangle$ and $\langle \text{value} \rangle$ may contain any $\langle \text{balanced text} \rangle$. The $\langle \text{key} \rangle$ is stored after processing with **\TlToStr**, meaning that category codes are ignored.

```
\PropSetFromKeyval \lTmPaProp {key1=one,key2=two}
\PropPut \lTmPaProp {key1} {newone}
\PropVarLog \lTmPaProp
```

\PropPutIfNew $\langle \text{property list} \rangle$ $\{ \langle \text{key} \rangle \}$ $\{ \langle \text{value} \rangle \}$

If the $\langle \text{key} \rangle$ is present in the $\langle \text{property list} \rangle$ then no action is taken. Otherwise, a new entry is added as described for **\PropPut**.

```
\PropSetFromKeyval \lTmPaProp {key1=one,key2=two}
\PropPutIfNew \lTmPaProp {key1} {newone}
\PropVarLog \lTmPaProp
```

```
\PropPutFromKeyval <prop var>
{
  <key1> = <value1> ,
  <key2> = <value2> , ...
}
```

Updates the *<prop var>* by adding entries for each key–value pair given in the second argument. The addition is done through **\PropPut**, hence if the *<prop var>* already contains some of the keys, the corresponding values are discarded and replaced by those given in the key–value list. If duplicate keys appear in the key–value list then only the last of the values is kept.

```
\PropSetFromKeyval \lTmPaProp {key1=one,key2=two}
\PropPutFromKeyval \lTmPaProp {key1=newone,key3=three}
\PropVarLog \lTmPaProp
```

```
\PropVarRemove <property list> {<key>}
```

Removes the entry listed under *<key>* from the *<property list>*. If the *<key>* is not found in the *<property list>* no change occurs, *i.e.* there is no need to test for the existence of a key before deleting it.

```
\PropSetFromKeyval \lTmPaProp {key1=one,key2=two,key3=three}
\PropVarRemove \lTmPaProp {key2}
\PropVarLog \lTmPaProp
```

12.5 Recovering Values from Property Lists

```
\PropVarCount <property list>
```

Returns the number of key–value pairs in the *<property list>* as an *<integer denotation>*.

```
\PropSetFromKeyval \lTmPaProp {key1=one,key2=two,key3=three}
\PropVarCount \lTmPaProp
```

3

```
\PropVarItem <property list> {<key>}
```

Returns the *<value>* corresponding to the *<key>* in the *<property list>*. If the *<key>* is missing, nothing is returned.

```
\PropSetFromKeyval \lTmPaProp {key1=one,key2=two,key3=three}
\TlSet \lTmPaTl {\PropVarItem \lTmPaProp {key2}}
\TlUse \lTmPaTl
```

two

```
\PropGet <property list> {<key>} <token list variable>
```

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 **\qNoValue**. The assignment of the *<token list variable>* is local.

```
\PropSetFromKeyval \lTmPaProp {key1=one,key2=two,key3=three}
\PropGet \lTmPaProp {key2} \lTmPaTl
\TlUse \lTmPaTl
```

two

```

\PropGetT <property list> {<key>} <token list variable> {<true code>}
\PropGetF <property list> {<key>} <token list variable> {<false code>}
\PropGetTF <property list> {<key>} <token list variable> {<true code>} {<false code>}

```

If the <key> is not present in the <property list>, 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 <key> is present in the <property list>, stores the corresponding <value> in the <token list variable> without removing it from the <property list>, then leaves the <true code> in the input stream. The <token list variable> is assigned locally.

```

\PropSetFromKeyval \lTmPaProp {key1=one,key2=two,key3=three}
\PropGetTF \lTmPaProp {key2} \lTmPaTl {\Result{Yes}} {\Result{No}}

```

Yes

```

\PropPop <property list> {<key>} <token list variable>

```

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 **\qNoValue**. The <key> and <value> are then deleted from the property list. The assignment of the <token list variable> is local.

```

\PropSetFromKeyval \lTmPaProp {key1=one,key2=two,key3=three}
\PropPop \lTmPaProp {key2} \lTmPaTl
Pop: \TlUse \lTmPaTl.
Count: \PropVarCount \lTmPaProp.

```

Pop: two. Count: 2.

```

\PropPopT <property list> {<key>} <token list variable> {<true code>}
\PropPopF <property list> {<key>} <token list variable> {<false code>}
\PropPopTF <property list> {<key>} <token list variable> {<true code>} {<false code>}

```

If the <key> is not present in the <property list>, 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 <key> is present in the <property list>, pops the corresponding <value> in the <token list variable>, *i.e.* removes the item from The <token list variable> is assigned locally.

```

\PropSetFromKeyval \lTmPaProp {key1=one,key2=two,key3=three}
\PropPopTF \lTmPaProp {key2} \lTmPaTl {\Result{Yes}} {\Result{No}}

```

Yes

12.6 Mapping over property lists

```

\PropVarMapInline <property list> {<inline function>}

```

Applies <inline function> to every <entry> stored within the <property list>. The <inline function> should consist of code which receives the <key> as #1 and the <value> as #2. The order in which <entries> are returned is not defined and should not be relied upon. For example,

```

\IgnoreSpacesOn
\PropSetFromKeyval \lTmPkProp {key1=one,key2=two,key3=three}
\TlClear \lTmPaTl
\PropVarMapInline \lTmPkProp {
  \TlPutRight \lTmPaTl {(#1=#2)}
}
\Result {\TlUse\lTmPaTl}
\IgnoreSpacesOff

```

produces (key1=one)(key2=two)(key3=three).

12.7 Property List Conditionals

\PropIfExist $\langle property\ list \rangle$

\PropIfExistTF $\langle property\ list \rangle \{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$

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.

```
\PropIfExistTF \lTmпаProp {\Result{Yes}} {\Result{No}}
\PropIfExistTF \lFooUndefinedProp {\Result{Yes}} {\Result{No}}
```

Yes No

\PropVarIfEmpty $\langle property\ list \rangle$

\PropVarIfEmptyTF $\langle property\ list \rangle \{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$

Tests if the $\langle property\ list \rangle$ is empty (containing no entries).

```
\PropSetFromKeyval \lTmпаProp {key1=one,key2=two}
\PropVarIfEmptyTF \lTmпаProp {\Result{Empty}} {\Result{NonEmpty}}
\PropClear \lTmпаProp
\PropVarIfEmptyTF \lTmпаProp {\Result{Empty}} {\Result{NonEmpty}}
```

NonEmpty Empty

\PropVarIfIn $\langle property\ list \rangle \{ \langle key \rangle \} \{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$

Tests if the $\langle key \rangle$ is present in the $\langle property\ list \rangle$, making the comparison using the method described by **\StrIfEqTF**.

```
\PropSetFromKeyval \lTmпаProp {key1=one,key2=two}
\PropVarIfInTF \lTmпаProp {key1} {\Result{Yes}} {\Result{Not}}
\PropVarIfInTF \lTmпаProp {key3} {\Result{Yes}} {\Result{Not}}
```

Yes Not

Chapter 13

Quarks (Quark)

Quarks are control sequences (and in fact, token lists) that expand to themselves and should therefore never be executed directly in the code. This would result in an endless loop!

Quarks can be used as error return values for functions that receive erroneous input. For example, in the function `\PropGet` 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 `\qNoValue`. 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.

13.1 Constant Quarks

`\qNoValue`

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 `\PropGet` if there is no data to return.

13.2 Quark Conditionals

`\QuarkVarIfNoValue` *<token>*
`\QuarkVarIfNoValueTF` *<token>* *{<true code>}* *{<false code>}*

Tests if the *<token>* is equal to `\qNoValue`.

```
\ClistGet \cEmptyClist \lTmptl
\QuarkVarIfNoValueTF \lTmptl {\Result{NoValue}} {\Result{SomeValue}}
```

NoValue

```
\SeqPop \cEmptySeq \lTmptl
\QuarkVarIfNoValueTF \lTmptl {\Result{NoValue}} {\Result{SomeValue}}
```

NoValue

```
\PropSetFromKeyval \lTmptProp {key1=one,key2=two}
\PropGet \lTmptProp {key3} \lTmptl
\QuarkVarIfNoValueTF \lTmptl {\Result{NoValue}} {\Result{SomeValue}}
```

NoValue

Chapter 14

The Source Code

```
%% -----  
%% Functional: LaTeX2 functional interfaces for LaTeX3 programming layer  
%% Copyright : 2022 (c) Jianrui Lyu <tolvjvr@163.com>  
%% Repository: https://github.com/lvjvr/functional  
%% Repository: https://bitbucket.org/lvjvr/functional  
%% License   : The LaTeX Project Public License 1.3c  
%% -----
```

14.1 Interfaces for Functional Programming (Prg)

```
\NeedsTeXFormat{LaTeX2e}[2018-04-01]  
  
\RequirePackage{expl3}  
\ProvidesExplPackage{functional}{2022-04-01}{2022C}  
  {^^JLaTeX2 functional interfaces for LaTeX3 programming layer}  
  
\cs_generate_variant:Nn \iow_log:n { V }  
\cs_generate_variant:Nn \str_set:Nn { Ne }  
\cs_generate_variant:Nn \tl_log:n { e }  
\cs_generate_variant:Nn \tl_set:Nn { Ne }  
  
\prg_generate_conditional_variant:Nnn \str_if_eq:nn { Ve } { TF }  
  
\tl_new:N \gResultTl  
\int_new:N \l__fun_arg_count_int  
\tl_new:N \l__fun_parameters_defined_tl  
\tl_const:Nn \c__fun_parameter_defined_i__tl { } % no argument  
\tl_const:Nn \c__fun_parameter_defined_i_i_tl { #1 }  
\tl_const:Nn \c__fun_parameter_defined_i_ii_tl { #1 #2 }  
\tl_const:Nn \c__fun_parameter_defined_i_iii_tl { #1 #2 #3 }  
\tl_const:Nn \c__fun_parameter_defined_i_iv_tl { #1 #2 #3 #4 }  
\tl_const:Nn \c__fun_parameter_defined_i_v_tl { #1 #2 #3 #4 #5 }  
\tl_const:Nn \c__fun_parameter_defined_i_vi_tl { #1 #2 #3 #4 #5 #6 }  
\tl_const:Nn \c__fun_parameter_defined_i_vii_tl { #1 #2 #3 #4 #5 #6 #7 }  
\tl_const:Nn \c__fun_parameter_defined_i_viii_tl { #1 #2 #3 #4 #5 #6 #7 #8 }  
\tl_const:Nn \c__fun_parameter_defined_i_ix_tl { #1 #2 #3 #4 #5 #6 #7 #8 #9 }  
\tl_new:N \l__fun_parameters_called_tl  
\tl_const:Nn \c__fun_parameter_called_i_i_tl { {#1} }  
\tl_const:Nn \c__fun_parameter_called_i_ii_tl { {#1}{#2} }  
\tl_const:Nn \c__fun_parameter_called_i_iii_tl { {#1}{#2}{#3} }
```

```

\tl_const:Nn \c__fun_parameter_called_i_iv_tl { {#1}{#2}{#3}{#4} }
\tl_const:Nn \c__fun_parameter_called_i_v_tl { {#1}{#2}{#3}{#4}{#5} }
\tl_const:Nn \c__fun_parameter_called_i_vi_tl { {#1}{#2}{#3}{#4}{#5}{#6} }
\tl_const:Nn \c__fun_parameter_called_i_vii_tl { {#1}{#2}{#3}{#4}{#5}{#6}{#7} }
\tl_new:N \l__fun_parameters_true_tl
\tl_new:N \l__fun_parameters_false_tl
\tl_const:Nn \c__fun_parameter_called_i_tl { {#1} }
\tl_const:Nn \c__fun_parameter_called_ii_tl { {#2} }
\tl_const:Nn \c__fun_parameter_called_iii_tl { {#3} }
\tl_const:Nn \c__fun_parameter_called_iv_tl { {#4} }
\tl_const:Nn \c__fun_parameter_called_v_tl { {#5} }
\tl_const:Nn \c__fun_parameter_called_vi_tl { {#6} }
\tl_const:Nn \c__fun_parameter_called_vii_tl { {#7} }
\tl_const:Nn \c__fun_parameter_called_viii_tl { {#8} }
\tl_const:Nn \c__fun_parameter_called_ix_tl { {#9} }

%% #1: function name; #2: argument specification; #3 function body
\cs_new_protected:Npn \__fun_new_function:Nnn #1 #2 #3
{
  \int_set:Nn \l__fun_arg_count_int { \tl_count:n {#2} } % spaces are ignored
  \tl_set_eq:Nc \l__fun_parameters_defined_tl
    { c__fun_parameter_defined_i_ \int_to_roman:n { \l__fun_arg_count_int } _tl }
  \exp_last_unbraced:NcV \cs_new_protected:Npn
    { __fun_defined_ \cs_to_str:N #1 : w }
    \l__fun_parameters_defined_tl
    {
      \__fun_group_begin:
      \tl_gclear:N \gResultTl
      #3
      \__fun_tracing_log:e { [0] ~ \exp_not:V \gResultTl }
      \__fun_group_end:
    }
  \use:c { __fun_new_with_arg_ \int_to_roman:n { \l__fun_arg_count_int } :NnV }
    #1 {#2} \l__fun_parameters_defined_tl
}
\cs_generate_variant:Nn \__fun_new_function:Nnn { cne }

\cs_set_eq:NN \PrgNewFunction \__fun_new_function:Nnn

\tl_new:N \g__fun_last_result_tl

%% #1: function name; #2: argument specification; #3 function body
\cs_new_protected:Npn \__fun_new_conditional:Nnn #1 #2 #3
{
  \__fun_new_function:Nnn #1 { #2 } { #3 }
  \tl_set_eq:Nc \l__fun_parameters_called_tl
    { c__fun_parameter_called_i_ \int_to_roman:n { \l__fun_arg_count_int } _tl }
  \tl_set_eq:Nc \l__fun_parameters_true_tl
    { c__fun_parameter_called_ \int_to_roman:n { \l__fun_arg_count_int + 1 } _tl }
  \tl_set_eq:Nc \l__fun_parameters_false_tl
    { c__fun_parameter_called_ \int_to_roman:n { \l__fun_arg_count_int + 2 } _tl }
  \__fun_new_function:cne { \cs_to_str:N #1 TF } { #2 n n }
  {
    #1 \exp_not:V \l__fun_parameters_called_tl
    \exp_not:n
    {
      \tl_set_eq:NN \g__fun_last_result_tl \gResultTl
    }
  }
}

```

```

        \tl_gclear:N \gResultTl
        \exp_last_unbraced:NV \bool_if:NTF \g__fun_last_result_tl
      }
      \exp_not:V \l__fun_parameters_true_tl
      \exp_not:V \l__fun_parameters_false_tl
    }
  }

\cs_set_eq:NN \PrgNewConditional \__fun_new_conditional:Nnn

\int_new:N \g__fun_nesting_level_int

%% #1: function name; #2: argument specifications; #3 parameters tl defined
%% Some times we need to create a function without arguments
\cs_new_protected:Npn \__fun_new_with_arg:Nnn #1 #2 #3
{
  \cs_new_protected:Npn #1 #3
  {
    \int_gincr:N \g__fun_nesting_level_int
    \__fun_evaluate:Nn #1 {#2}
    \int_gdecr:N \g__fun_nesting_level_int
    \__fun_return_result:
  }
}

\cs_generate_variant:Nn \__fun_new_with_arg:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_i:Nnn #1 #2 #3
{
  \cs_new_protected:Npn #1 #3
  {
    \int_gincr:N \g__fun_nesting_level_int
    \__fun_one_argument_gset:nn { 1 } { ##1 }
    \__fun_evaluate:Nn #1 {#2}
    \int_gdecr:N \g__fun_nesting_level_int
    \__fun_return_result:
  }
}

\cs_generate_variant:Nn \__fun_new_with_arg_i:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_ii:Nnn #1 #2 #3
{
  \cs_new_protected:Npn #1 #3
  {
    \int_gincr:N \g__fun_nesting_level_int
    \__fun_one_argument_gset:nn { 1 } { ##1 }
    \__fun_one_argument_gset:nn { 2 } { ##2 }
    \__fun_evaluate:Nn #1 {#2}
    \int_gdecr:N \g__fun_nesting_level_int
    \__fun_return_result:
  }
}

\cs_generate_variant:Nn \__fun_new_with_arg_ii:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_iii:Nnn #1 #2 #3

```

```

{
  \cs_new_protected:Npn #1 #3
  {
    \int_gincr:N \g__fun_nesting_level_int
    \__fun_one_argument_gset:nn { 1 } { ##1 }
    \__fun_one_argument_gset:nn { 2 } { ##2 }
    \__fun_one_argument_gset:nn { 3 } { ##3 }
    \__fun_evaluate:Nn #1 {#2}
    \int_gdecr:N \g__fun_nesting_level_int
    \__fun_return_result:
  }
}

\cs_generate_variant:Nn \__fun_new_with_arg_iii:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_iv:Nnn #1 #2 #3
{
  \cs_new_protected:Npn #1 #3
  {
    \int_gincr:N \g__fun_nesting_level_int
    \__fun_one_argument_gset:nn { 1 } { ##1 }
    \__fun_one_argument_gset:nn { 2 } { ##2 }
    \__fun_one_argument_gset:nn { 3 } { ##3 }
    \__fun_one_argument_gset:nn { 4 } { ##4 }
    \__fun_evaluate:Nn #1 {#2}
    \int_gdecr:N \g__fun_nesting_level_int
    \__fun_return_result:
  }
}

\cs_generate_variant:Nn \__fun_new_with_arg_iv:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_v:Nnn #1 #2 #3
{
  \cs_new_protected:Npn #1 #3
  {
    \int_gincr:N \g__fun_nesting_level_int
    \__fun_one_argument_gset:nn { 1 } { ##1 }
    \__fun_one_argument_gset:nn { 2 } { ##2 }
    \__fun_one_argument_gset:nn { 3 } { ##3 }
    \__fun_one_argument_gset:nn { 4 } { ##4 }
    \__fun_one_argument_gset:nn { 5 } { ##5 }
    \__fun_evaluate:Nn #1 {#2}
    \int_gdecr:N \g__fun_nesting_level_int
    \__fun_return_result:
  }
}

\cs_generate_variant:Nn \__fun_new_with_arg_v:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_vi:Nnn #1 #2 #3
{
  \cs_new_protected:Npn #1 #3
  {
    \int_gincr:N \g__fun_nesting_level_int
    \__fun_one_argument_gset:nn { 1 } { ##1 }
    \__fun_one_argument_gset:nn { 2 } { ##2 }
    \__fun_one_argument_gset:nn { 3 } { ##3 }

```

```

    \__fun_one_argument_gset:nn { 4 } { ##4 }
    \__fun_one_argument_gset:nn { 5 } { ##5 }
    \__fun_one_argument_gset:nn { 6 } { ##6 }
    \__fun_evaluate:Nn #1 {#2}
    \int_gdecr:N \g__fun_nesting_level_int
    \__fun_return_result:
  }
}
\cs_generate_variant:Nn \__fun_new_with_arg_vi:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_vii:Nnn #1 #2 #3
{
  \cs_new_protected:Npn #1 #3
  {
    \int_gincr:N \g__fun_nesting_level_int
    \__fun_one_argument_gset:nn { 1 } { ##1 }
    \__fun_one_argument_gset:nn { 2 } { ##2 }
    \__fun_one_argument_gset:nn { 3 } { ##3 }
    \__fun_one_argument_gset:nn { 4 } { ##4 }
    \__fun_one_argument_gset:nn { 5 } { ##5 }
    \__fun_one_argument_gset:nn { 6 } { ##6 }
    \__fun_one_argument_gset:nn { 7 } { ##7 }
    \__fun_evaluate:Nn #1 {#2}
    \int_gdecr:N \g__fun_nesting_level_int
    \__fun_return_result:
  }
}
\cs_generate_variant:Nn \__fun_new_with_arg_vii:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_viii:Nnn #1 #2 #3
{
  \cs_new_protected:Npn #1 #3
  {
    \int_gincr:N \g__fun_nesting_level_int
    \__fun_one_argument_gset:nn { 1 } { ##1 }
    \__fun_one_argument_gset:nn { 2 } { ##2 }
    \__fun_one_argument_gset:nn { 3 } { ##3 }
    \__fun_one_argument_gset:nn { 4 } { ##4 }
    \__fun_one_argument_gset:nn { 5 } { ##5 }
    \__fun_one_argument_gset:nn { 6 } { ##6 }
    \__fun_one_argument_gset:nn { 7 } { ##7 }
    \__fun_one_argument_gset:nn { 8 } { ##8 }
    \__fun_evaluate:Nn #1 {#2}
    \int_gdecr:N \g__fun_nesting_level_int
    \__fun_return_result:
  }
}
\cs_generate_variant:Nn \__fun_new_with_arg_viii:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_ix:Nnn #1 #2 #3
{
  \cs_new_protected:Npn #1 #3
  {
    \int_gincr:N \g__fun_nesting_level_int
    \__fun_one_argument_gset:nn { 1 } { ##1 }

```

```

    \__fun_one_argument_gset:nn { 2 } { ##2 }
    \__fun_one_argument_gset:nn { 3 } { ##3 }
    \__fun_one_argument_gset:nn { 4 } { ##4 }
    \__fun_one_argument_gset:nn { 5 } { ##5 }
    \__fun_one_argument_gset:nn { 6 } { ##6 }
    \__fun_one_argument_gset:nn { 7 } { ##7 }
    \__fun_one_argument_gset:nn { 8 } { ##8 }
    \__fun_one_argument_gset:nn { 9 } { ##9 }
    \__fun_evaluate:Nn #1 {#2}
    \int_gdecr:N \g__fun_nesting_level_int
    \__fun_return_result:
  }
}

\cs_generate_variant:Nn \__fun_new_with_arg_ix:Nnn { NnV }

\tl_new:N \l__fun_argtype_tl
\tl_const:Nn \c__fun_argtype_m_tl { m }
\tl_const:Nn \c__fun_argtype_M_tl { M }
\tl_const:Nn \c__fun_argtype_n_tl { n }
\tl_const:Nn \c__fun_argtype_N_tl { N }
\tl_new:N \l__fun_argument_tl

%% #1: function name; #2: argument specifications
\cs_new_protected:Npn \__fun_evaluate:Nn #1 #2
{
  \__fun_argtype_index_gzero:
  \__fun_arguments_gclear:
  \tl_map_variable:nNn { #2 } \l__fun_argtype_tl % spaces are ignored
  {
    \__fun_argtype_index_gincr:
    \__fun_one_argument_get:eN { \__fun_argtype_index_use: } \l__fun_argument_tl
    \tl_case:Nn \l__fun_argtype_tl
    {
      \c__fun_argtype_m_tl
      {
        \__fun_evaluate_and_put_argument:N \l__fun_argument_tl
      }
      \c__fun_argtype_M_tl
      {
        \__fun_evaluate_and_put_argument:N \l__fun_argument_tl
      }
      \c__fun_argtype_n_tl
      {
        \__fun_arguments_gput:e { { \exp_not:V \l__fun_argument_tl } }
      }
      \c__fun_argtype_N_tl
      {
        \__fun_arguments_gput:e { \exp_not:V \l__fun_argument_tl }
      }
    }
  }
  \__fun_arguments_log:N #1
  \__fun_arguments_called:c { __fun_defined_ \cs_to_str:N #1 : w }
}

\cs_new_protected:Npn \__fun_evaluate_and_put_argument:N #1
{
  \cs_if_exist:cTF

```

```

{
  __fun_defined_ \exp_last_unbraced:Ne \cs_to_str:N { \tl_head:N #1 } : w
}
{
  #1
  \__fun_arguments_gput:e { { \exp_not:V \gResultTl } }
}
{
  \__fun_arguments_gput:e { { \exp_not:V #1 } }
}
}

%% #1: argument number; #2: token lists
\cs_new_protected:Npn \__fun_one_argument_gset:nn #1 #2
{
  \tl_gset:cn
  { g__fun_one_argument_ \int_use:N \g__fun_nesting_level_int _#1_tl } { #2 }
  %\__fun_one_argument_log:nn { #1 } { set }
}

%% #1: argument number; #2: variable of token lists
\cs_new_protected:Npn \__fun_one_argument_get:nN #1 #2
{
  \tl_set_eq:Nc
  #2 { g__fun_one_argument_ \int_use:N \g__fun_nesting_level_int _#1_tl }
  %\__fun_one_argument_log:nn { #1 } { get }
}
\cs_generate_variant:Nn \__fun_one_argument_get:nN { eN }

%% #1: argument number; #2: get or set
\cs_new_protected:Npn \__fun_one_argument_log:nn #1 #2
{
  \tl_log:e
  {
    #2 ~ level _ \int_use:N \g__fun_nesting_level_int _ arg _ #1 ~ = ~
    \exp_not:v
    { g__fun_one_argument_ \int_use:N \g__fun_nesting_level_int _#1_tl }
  }
}

\int_new:c { g__fun_argtype_index_ 1 _int }
\int_new:c { g__fun_argtype_index_ 2 _int }
\int_new:c { g__fun_argtype_index_ 3 _int }
\int_new:c { g__fun_argtype_index_ 4 _int }
\int_new:c { g__fun_argtype_index_ 5 _int }

\cs_new_protected:Npn \__fun_argtype_index_gzero:
{
  \int_gzero_new:c
  { g__fun_argtype_index_ \int_use:N \g__fun_nesting_level_int _int }
}

\cs_new_protected:Npn \__fun_argtype_index_gincr:
{
  \int_gincr:c
  { g__fun_argtype_index_ \int_use:N \g__fun_nesting_level_int _int }
}

```



```

\cs_new:Npn \__fun_argtype_index_use:
{
  \int_use:c { g__fun_argtype_index_ \int_use:N \g__fun_nesting_level_int _int }
}

\cs_new_protected:Npn \__fun_arguments_called:N #1
{
  \exp_last_unbraced:Nv
    #1 { g__fun_arguments_ \int_use:N \g__fun_nesting_level_int _tl }
}
\cs_generate_variant:Nn \__fun_arguments_called:N { c }

\cs_new_protected:Npn \__fun_arguments_gclear:
{
  \tl_gclear:c { g__fun_arguments_ \int_use:N \g__fun_nesting_level_int _tl }
}

\cs_new_protected:Npn \__fun_arguments_log:N #1
{
  \__fun_tracing_log:e
  {
    [I] ~ \token_to_str:N #1
    \exp_not:v { g__fun_arguments_ \int_use:N \g__fun_nesting_level_int _tl }
  }
}

\cs_new_protected:Npn \__fun_arguments_gput:n #1
{
  \tl_gput_right:cn
    { g__fun_arguments_ \int_use:N \g__fun_nesting_level_int _tl } { #1 }
}
\cs_generate_variant:Nn \__fun_arguments_gput:n { e }

\cs_set_eq:NN \Break \prg_break:
\cs_set_eq:NN \PrgBreak \prg_break:

\cs_set_eq:NN \BreakDo \prg_break:n
\cs_set_eq:NN \PrgBreakDo \prg_break:n

\cs_new_protected:Npn \__fun_put_result:n #1
{
  \tl_gput_right:Nn \gResultTl { #1 }
}
\cs_generate_variant:Nn \__fun_put_result:n { e, V }

\PrgNewFunction \Result { m }
{
  \__fun_put_result:n { #1 }
}

\cs_new_protected:Npn \__fun_return_result:
{
  \int_compare:nNnT { \g__fun_nesting_level_int } = { 0 }
  { \tl_use:N \gResultTl }
}

```

```

\str_new:N \l__fun_variable_name_str
\str_new:N \l__fun_variable_name_a_str
\str_new:N \l__fun_variable_name_b_str

\prg_new_protected_conditional:Npnn \__fun_if_global_variable:N #1 { TF }
{
  \str_set:Ne \l__fun_variable_name_str { \cs_to_str:N #1 }
  \str_set:Ne \l__fun_variable_name_b_str
    { \str_item:Nn \l__fun_variable_name_str { 2 } }
  \str_if_eq:VeTF
    \l__fun_variable_name_b_str
    { \str_uppercase:f { \l__fun_variable_name_b_str } }
  {
    \str_set:Ne \l__fun_variable_name_a_str
      { \str_head:N \l__fun_variable_name_str }
    \str_case:VnF \l__fun_variable_name_a_str
      {
        { l } { \prg_return_false: }
        { g } { \prg_return_true: }
      }
    { \__fun_if_set_local: }
  }
  { \__fun_if_set_local: }
}

\bool_new:N \g__fun_variable_local_bool

\cs_new:Npn \__fun_if_set_local:
{
  \bool_if:NTF \g__fun_variable_local_bool
  {
    \bool_gset_false:N \g__fun_variable_local_bool
    \prg_return_false:
  }
  { \prg_return_true: }
}

\PrgNewFunction \Local { } { \bool_gset_true:N \g__fun_variable_local_bool }

%% We must not put an assignment inside a group
\cs_new_protected:Npn \__fun_do_assignment:Nnn #1 #2 #3
{
  \__fun_group_end:
  \__fun_if_global_variable:NTF #1 { #2 } { #3 }
  \__fun_group_begin:
}

\bool_new:N \l__fun_scoping_bool

\cs_new_protected:Npn \__fun_scoping_true:
{
  \cs_set_eq:NN \__fun_group_begin: \group_begin:
  \cs_set_eq:NN \__fun_group_end: \group_end:
}

\cs_new_protected:Npn \__fun_scoping_false:

```

```

{
  \cs_set_eq:NN \__fun_group_begin: \scan_stop:
  \cs_set_eq:NN \__fun_group_end: \scan_stop:
}

\cs_new_protected:Npn \__fun_scoping_set:
{
  \bool_if:NTF \l__fun_scoping_bool
  { \__fun_scoping_true: } { \__fun_scoping_false: }
}

\bool_new:N \l__fun_tracing_bool
\tl_new:N \l__tracing_text_tl

\cs_new_protected:Npn \__fun_tracing_log_on:n #1
{
  \tl_set:Nx \l__tracing_text_tl
  {
    \prg_replicate:nn
    { \int_eval:n { (\g__fun_nesting_level_int - 1) * 4 } } { ~ }
  }
  \tl_put_right:Nn \l__tracing_text_tl { #1 }
  \iow_log:V \l__tracing_text_tl
}
\cs_generate_variant:Nn \__fun_tracing_log_on:n { e, V }

\cs_new_protected:Npn \__fun_tracing_log_off:n #1 { }
\cs_new_protected:Npn \__fun_tracing_log_off:e #1 { }
\cs_new_protected:Npn \__fun_tracing_log_off:V #1 { }

\cs_new_protected:Npn \__fun_tracing_true:
{
  \cs_set_eq:NN \__fun_tracing_log:n \__fun_tracing_log_on:n
  \cs_set_eq:NN \__fun_tracing_log:e \__fun_tracing_log_on:e
  \cs_set_eq:NN \__fun_tracing_log:V \__fun_tracing_log_on:V
}

\cs_new_protected:Npn \__fun_tracing_false:
{
  \cs_set_eq:NN \__fun_tracing_log:n \__fun_tracing_log_off:n
  \cs_set_eq:NN \__fun_tracing_log:e \__fun_tracing_log_off:e
  \cs_set_eq:NN \__fun_tracing_log:V \__fun_tracing_log_off:V
}

\cs_new_protected:Npn \__fun_tracing_set:
{
  \bool_if:NTF \l__fun_tracing_bool
  { \__fun_tracing_true: } { \__fun_tracing_false: }
}

\keys_define:nn { functional }
{
  scoping .bool_set:N = \l__fun_scoping_bool,
  tracing .bool_set:N = \l__fun_tracing_bool,
}

```

```

\NewDocumentCommand \Functional { m }
{
  \keys_set:nn { functional } { #1 }
  \__fun_scoping_set:
  \__fun_tracing_set:
}

\Functional { scoping = false, tracing = false }

\cs_new_protected:Npn \__fun_ignore_spaces_on:
{
  \ExplSyntaxOn
  \char_set_catcode_math_subscript:N \_
  \char_set_catcode_other:N \:
}
\cs_set_eq:NN \IgnoreSpacesOn \__fun_ignore_spaces_on:
\cs_set_eq:NN \IgnoreSpacesOff \ExplSyntaxOff

```

14.2 Interfaces for Argument Using (Use)

```

\PrgNewFunction \Name { m }
{
  \exp_args:Nc \__fun_put_result:n { #1 }
}

\PrgNewFunction \Value { M }
{
  \__fun_put_result:V #1
}

\PrgNewFunction \Expand { m }
{
  \__fun_put_result:e { #1 }
}

\cs_set_eq:NN \UnExpand \exp_not:n
\cs_set_eq:NN \NoExpand \exp_not:N
\cs_set_eq:NN \OnlyName \exp_not:c
\cs_set_eq:NN \OnlyValue \exp_not:V
\cs_set_eq:NN \OnlyExpandF \exp_not:f
\cs_set_eq:NN \OnlyExpandO \exp_not:o

\PrgNewFunction \UseOne { n } { \Result { #1 } }

\PrgNewFunction \GobbleOne { n } { \Result { } }

\PrgNewFunction \UseGobble { n n } { \UseOne { #1 } }

\PrgNewFunction \GobbleUse { n n } { \UseOne { #2 } }

```

14.3 Interfaces for Control Structures (Bool)

```

\bool_const:Nn \cTrueBool { \c_true_bool }

```

```

\bool_const:Nn \cFalseBool { \c_false_bool }

\bool_new:N \lTmPaBool   \bool_new:N \lTmPbBool   \bool_new:N \lTmPcBool
\bool_new:N \lTmPiBool   \bool_new:N \lTmPjBool   \bool_new:N \lTmPkBool
\bool_new:N \l@Funx@Bool \bool_new:N \l@Funy@Bool \bool_new:N \l@Funz@Bool

\bool_new:N \gTmPaBool   \bool_new:N \gTmPbBool   \bool_new:N \gTmPcBool
\bool_new:N \gTmPiBool   \bool_new:N \gTmPjBool   \bool_new:N \gTmPkBool
\bool_new:N \g@Funx@Bool \bool_new:N \g@Funy@Bool \bool_new:N \g@Funz@Bool

\PrgNewFunction \BoolNew { M } { \bool_new:N #1 }

\PrgNewFunction \BoolConst { M m } { \bool_const:Nn #1 { #2 } }

\PrgNewFunction \BoolSet { M m } {
  \__fun_do_assignment:Nnn #1
  { \bool_gset:Nn #1 { #2 } } { \bool_set:Nn #1 { #2 } }
}

\PrgNewFunction \BoolSetTrue { M }
{
  \__fun_do_assignment:Nnn #1 { \bool_gset_true:N #1 } { \bool_set_true:N #1 }
}

\PrgNewFunction \BoolSetFalse { M }
{
  \__fun_do_assignment:Nnn #1 { \bool_gset_false:N #1 } { \bool_set_false:N #1 }
}

\PrgNewFunction \BoolSetEq { M M }
{
  \__fun_do_assignment:Nnn #1
  { \bool_gset_eq:NN #1 #2 } { \bool_set_eq:NN #1 #2 }
}

\PrgNewFunction \BoolLog { m } { \bool_log:n { #1 } }

\PrgNewFunction \BoolVarLog { M } { \bool_log:N #1 }

\PrgNewFunction \BoolShow { m } { \bool_show:n { #1 } }

\PrgNewFunction \BoolVarShow { M } { \bool_show:N #1 }

\PrgNewConditional \BoolIfExist { M }
{
  \bool_if_exist:Ntf #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrgNewConditional \BoolVarIf { M } { \Result { #1 } }

\PrgNewConditional \BoolVarNot { M }
{
  \bool_if:Ntf #1
  { \Result { \cFalseBool } } { \Result { \cTrueBool } }
}

```

```

}

\PrGNewConditional \BoolVarAnd { M M }
{
  \bool_lazy_and:nnTF {#1} {#2}
  { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrGNewConditional \BoolVarOr { M M }
{
  \bool_lazy_or:nnTF {#1} {#2}
  { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrGNewConditional \BoolVarXor { M M }
{
  \bool_xor:nnTF {#1} {#2}
  { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

```

14.4 Interfaces for Token Lists (Tl)

```

\tl_set_eq:NN \cEmptyTl \c_empty_tl
\tl_set_eq:NN \cSpaceTl \c_space_tl
\tl_set_eq:NN \cNoValueTl \c_novalue_tl

\tl_new:N \lTmptl \tl_new:N \lTmptbTl \tl_new:N \lTmptcTl
\tl_new:N \lTmptiTl \tl_new:N \lTmptjTl \tl_new:N \lTmptkTl
\tl_new:N \l@Funx@Tl \tl_new:N \l@Funy@Tl \tl_new:N \l@Funz@Tl

\tl_new:N \gTmptl \tl_new:N \gTmptbTl \tl_new:N \gTmptcTl
\tl_new:N \gTmptiTl \tl_new:N \gTmptjTl \tl_new:N \gTmptkTl
\tl_new:N \g@Funx@Tl \tl_new:N \g@Funy@Tl \tl_new:N \g@Funz@Tl

\PrGNewFunction \TlNew { M } { \tl_new:N #1 }

\PrGNewFunction \TlLog { m } { \tl_log:n { #1 } }

\PrGNewFunction \TlVarLog { M } { \tl_log:N #1 }

\PrGNewFunction \TlShow { m } { \tl_show:n { #1 } }

\PrGNewFunction \TlVarShow { M } { \tl_show:N #1 }

\PrGNewFunction \TlUse { M } { \Result { \Value #1 } }

\PrGNewFunction \TlToStr { m } { \Expand { \tl_to_str:n { #1 } } }

\PrGNewFunction \TlVarToStr { M } { \Expand { \tl_to_str:N #1 } }

\PrGNewFunction \TlConst { M m } { \tl_const:Nn #1 { #2 } }

\PrGNewFunction \TlSet { M m }

```

```

{
  \__fun_do_assignment:Nnn #1 { \tl_gset:Nn #1 {#2} } { \tl_set:Nn #1 {#2} }
}

\PrgNewFunction \TlSetEq { M M }
{
  \__fun_do_assignment:Nnn #1 { \tl_gset_eq:NN #1 #2 } { \tl_set_eq:NN #1 #2 }
}

\PrgNewFunction \TlConcat { M M M }
{
  \__fun_do_assignment:Nnn #1
  { \tl_gconcat:NNN #1 #2 #3 } { \tl_concat:NNN #1 #2 #3 }
}

\PrgNewFunction \TlClear { M }
{
  \__fun_do_assignment:Nnn #1 { \tl_gclear:N #1 } { \tl_clear:N #1 }
}

\PrgNewFunction \TlClearNew { M }
{
  \__fun_do_assignment:Nnn #1 { \tl_gclear_new:N #1 } { \tl_clear_new:N #1 }
}

\PrgNewFunction \TlPutLeft { M m }
{
  \__fun_do_assignment:Nnn #1
  { \tl_gput_left:Nn #1 {#2} } { \tl_put_left:Nn #1 {#2} }
}

\PrgNewFunction \TlPutRight { M m }
{
  \__fun_do_assignment:Nnn #1
  { \tl_gput_right:Nn #1 {#2} } { \tl_put_right:Nn #1 {#2} }
}

\PrgNewFunction \TlVarReplaceOnce { M m m }
{
  \__fun_do_assignment:Nnn #1
  { \tl_greplace_once:Nnn #1 {#2} {#3} } { \tl_replace_once:Nnn #1 {#2} {#3} }
}

\PrgNewFunction \TlVarReplaceAll { M m m }
{
  \__fun_do_assignment:Nnn #1
  { \tl_greplace_all:Nnn #1 {#2} {#3} } { \tl_replace_all:Nnn #1 {#2} {#3} }
}

\PrgNewFunction \TlVarRemoveOnce { M m }
{
  \__fun_do_assignment:Nnn #1
  { \tl_gremove_once:Nn #1 {#2} } { \tl_remove_once:Nn #1 {#2} }
}

\PrgNewFunction \TlVarRemoveAll { M m }

```

```

{
  \__fun_do_assignment:Nnn #1
  { \tl_gremove_all:Nn #1 {#2} } { \tl_remove_all:Nn #1 {#2} }
}

\PrgNewFunction \TlTrimSpaces { m } { \Expand { \tl_trim_spaces:n { #1 } } }

\PrgNewFunction \TlVarTrimSpaces { M }
{
  \__fun_do_assignment:Nnn #1 { \tl_gtrim_spaces:N #1 } { \tl_trim_spaces:N #1 }
}

\PrgNewFunction \TlCount { m } { \Expand { \tl_count:n { #1 } } }

\PrgNewFunction \TlVarCount { M } { \Expand { \tl_count:N #1 } }

\PrgNewFunction \TlHead { m } { \Expand { \tl_head:n { #1 } } }

\PrgNewFunction \TlVarHead { M } { \Expand { \tl_head:N #1 } }

\PrgNewFunction \TlTail { m } { \Expand { \tl_tail:n { #1 } } }

\PrgNewFunction \TlVarTail { M } { \Expand { \tl_tail:N #1 } }

\PrgNewFunction \TlItem { m m } { \Expand { \tl_item:nn {#1} {#2} } }

\PrgNewFunction \TlVarItem { M m } { \Expand { \tl_item:Nn #1 {#2} } }

\PrgNewFunction \TlRandItem { m } { \Expand { \tl_rand_item:n {#1} } }

\PrgNewFunction \TlVarRandItem { M } { \Expand { \tl_rand_item:N #1 } }

\PrgNewFunction \TlVarCase { M m } { \tl_case:Nn {#1} {#2} }
\PrgNewFunction \TlVarCaseT { M m n } { \tl_case:NnT {#1} {#2} {#3} }
\PrgNewFunction \TlVarCaseF { M m n } { \tl_case:NnF {#1} {#2} {#3} }
\PrgNewFunction \TlVarCaseTF { M m n n } { \tl_case:NnTF {#1} {#2} {#3} {#4} }

\PrgNewFunction \TlMapInline { m n }
{
  \tl_map_inline:nn {#1} {#2}
}

\PrgNewFunction \TlVarMapInline { M n }
{
  \tl_map_inline:Nn #1 {#2}
}

\PrgNewFunction \TlMapVariable { m M n }
{
  \tl_map_variable:nNn {#1} #2 {#3}
}

\PrgNewFunction \TlVarMapVariable { M M n }

```



```

{
  \tl_map_variable:NNn #1 #2 {#3}
}

\PrgNewConditional \TlIfExist { M }
{
  \tl_if_exist:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrgNewConditional \TlIfEmpty { m }
{
  \tl_if_empty:nTF {#1} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrgNewConditional \TlVarIfEmpty { M }
{
  \tl_if_empty:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrgNewConditional \TlIfBlank { m }
{
  \tl_if_blank:nTF {#1} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrgNewConditional \TlIfEq { m m }
{
  \tl_if_eq:nnTF {#1} {#2} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrgNewConditional \TlVarIfEq { M M }
{
  \tl_if_eq:NNTF #1 #2 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrgNewConditional \TlIfIn { m m }
{
  \tl_if_in:nnTF {#1} {#2} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrgNewConditional \TlVarIfIn { M m }
{
  \tl_if_in:NnTF #1 {#2} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrgNewConditional \TlIfSingle { m }
{
  \tl_if_single:nTF {#1} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrgNewConditional \TlVarIfSingle { M }
{
  \tl_if_single:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

```

14.5 Interfaces for Strings (Str)

```

\str_set_eq:NN \cAmpersandStr \c_ampersand_str
\str_set_eq:NN \cAttignStr \c_atsign_str
\str_set_eq:NN \cBackslashStr \c_backslash_str
\str_set_eq:NN \cLeftBraceStr \c_left_brace_str
\str_set_eq:NN \cRightBraceStr \c_right_brace_str
\str_set_eq:NN \cCircumflexStr \c_circumflex_str
\str_set_eq:NN \cColonStr \c_colon_str
\str_set_eq:NN \cDollarStr \c_dollar_str
\str_set_eq:NN \cHashStr \c_hash_str
\str_set_eq:NN \cPercentStr \c_percent_str
\str_set_eq:NN \cTildeStr \c_tilde_str
\str_set_eq:NN \cUnderscoreStr \c_underscore_str
\str_set_eq:NN \cZeroStr \c_zero_str

\str_new:N \lTmptStr \str_new:N \lTmptbStr \str_new:N \lTmptcStr
\str_new:N \lTmptiStr \str_new:N \lTmptjStr \str_new:N \lTmptkStr
\str_new:N \l@Funx@Str \str_new:N \l@Funy@Str \str_new:N \l@Funz@Str

\str_new:N \gTmptStr \str_new:N \gTmptbStr \str_new:N \gTmptcStr
\str_new:N \gTmptiStr \str_new:N \gTmptjStr \str_new:N \gTmptkStr
\str_new:N \g@Funx@Str \str_new:N \g@Funy@Str \str_new:N \g@Funz@Str

\PrgNewFunction \StrNew { M } { \str_new:N #1 }

\PrgNewFunction \StrLog { m } { \str_log:n { #1 } }

\PrgNewFunction \StrVarLog { M } { \str_log:N #1 }

\PrgNewFunction \StrShow { m } { \str_show:n { #1 } }

\PrgNewFunction \StrVarShow { M } { \str_show:N #1 }

\PrgNewFunction \StrUse { M } { \Result { \Value #1 } }

\PrgNewFunction \StrConst { M m } { \str_const:Nn #1 {#2} }

\PrgNewFunction \StrSet { M m }
{
  \__fun_do_assignment:Nnn #1 { \str_gset:Nn #1 {#2} } { \str_set:Nn #1 {#2} }
}

\PrgNewFunction \StrSetEq { M M }
{
  \__fun_do_assignment:Nnn #1 { \str_gset_eq:NN #1 #2 } { \str_set_eq:NN #1 #2 }
}

\PrgNewFunction \StrConcat { M M M }
{
  \__fun_do_assignment:Nnn #1
  { \str_gconcat:NNN #1 #2 #3 } { \str_concat:NNN #1 #2 #3 }
}

\PrgNewFunction \StrClear { M }

```

```

{
  \__fun_do_assignment:Nnn #1 { \str_gclear:N #1 } { \str_clear:N #1 }
}

\PrgNewFunction \StrClearNew { M }
{
  \__fun_do_assignment:Nnn #1 { \str_gclear_new:N #1 } { \str_clear_new:N #1 }
}

\PrgNewFunction \StrPutLeft { M m }
{
  \__fun_do_assignment:Nnn #1
  { \str_gput_left:Nn #1 {#2} } { \str_put_left:Nn #1 {#2} }
}

\PrgNewFunction \StrPutRight { M m }
{
  \__fun_do_assignment:Nnn #1
  { \str_gput_right:Nn #1 {#2} } { \str_put_right:Nn #1 {#2} }
}

\PrgNewFunction \StrVarReplaceOnce { M m m }
{
  \__fun_do_assignment:Nnn #1
  { \str_greplace_once:Nnn #1 {#2} {#3} } { \str_replace_once:Nnn #1 {#2} {#3} }
}

\PrgNewFunction \StrVarReplaceAll { M m m }
{
  \__fun_do_assignment:Nnn #1
  { \str_greplace_all:Nnn #1 {#2} {#3} } { \str_replace_all:Nnn #1 {#2} {#3} }
}

\PrgNewFunction \StrVarRemoveOnce { M m }
{
  \__fun_do_assignment:Nnn #1
  { \str_gremove_once:Nn #1 {#2} } { \str_remove_once:Nn #1 {#2} }
}

\PrgNewFunction \StrVarRemoveAll { M m }
{
  \__fun_do_assignment:Nnn #1
  { \str_gremove_all:Nn #1 {#2} } { \str_remove_all:Nn #1 {#2} }
}

%% Avoid naming conflict with xstring package
\cs_if_exist:NF \StrCount
{ \PrgNewFunction \StrCount { m } { \Expand { \str_count:n { #1 } } } }

%% Provide another name for \StrCount function
\PrgNewFunction \StrSize { m } { \Expand { \str_count:n { #1 } } }

\PrgNewFunction \StrVarCount { M } { \Expand { \str_count:N #1 } }

\PrgNewFunction \StrHead { m } { \Expand { \str_head:n { #1 } } }

```

```

\PrGNewFunction \StrVarHead { M } { \Expand { \str_head:N #1 } }

\PrGNewFunction \StrTail { m } { \Expand { \str_tail:n { #1 } } }

\PrGNewFunction \StrVarTail { M } { \Expand { \str_tail:N #1 } }

\PrGNewFunction \StrItem { m m } { \Expand { \str_item:nn {#1} {#2} } }

\PrGNewFunction \StrVarItem { M m } { \Expand { \str_item:Nn #1 {#2} } }

\PrGNewFunction \StrCase { m m } { \str_case:nn {#1} {#2} }
\PrGNewFunction \StrCaseT { m m n } { \str_case:nnT {#1} {#2} {#3} }
\PrGNewFunction \StrCaseF { m m n } { \str_case:nnF {#1} {#2} {#3} }
\PrGNewFunction \StrCaseTF { m m n n } { \str_case:nnTF {#1} {#2} {#3} {#4} }

\PrGNewFunction \StrMapInline { m n }
{
  \str_map_inline:nn {#1} {#2}
}

\PrGNewFunction \StrVarMapInline { M n }
{
  \str_map_inline:Nn #1 {#2}
}

\PrGNewFunction \StrMapVariable { m M n }
{
  \str_map_variable:nNn {#1} #2 {#3}
}

\PrGNewFunction \StrVarMapVariable { M M n }
{
  \str_map_variable:NNn #1 #2 {#3}
}

\PrGNewConditional \StrIfExist { M }
{
  \str_if_exist:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrGNewConditional \StrVarIfEmpty { M }
{
  \str_if_empty:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrGNewConditional \StrIfEq { m m }
{
  \str_if_eq:nnTF {#1} {#2} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrGNewConditional \StrVarIfEq { M M }
{
  \str_if_eq:NNTF #1 #2 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

```

```

\PrGNewConditional \StrIfIn { m m }
{
  \str_if_in:nnTF {#1} {#2} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrGNewConditional \StrVarIfIn { M m }
{
  \str_if_in:NnTF #1 {#2} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

%% Avoid naming conflict with xstring package
\cs_if_exist:NF \StrCompare
{
  \PrGNewConditional \StrCompare { m N m }
  {
    \str_compare:nNnTF {#1} #2 {#3}
    { \Result { \cTrueBool } }
    { \Result { \cFalseBool } }
  }
}

%% Provide another name for \StrCompare function
\PrGNewConditional \StrIfCompare { m N m }
{
  \str_compare:nNnTF {#1} #2 {#3}
  { \Result { \cTrueBool } }
  { \Result { \cFalseBool } }
}

```

14.6 Interfaces for Integers (Int)

```

\cs_set_eq:NN \cZeroInt      \c_zero_int
\cs_set_eq:NN \cOneInt       \c_one_int
\cs_set_eq:NN \cMaxInt        \c_max_int
\cs_set_eq:NN \cMaxRegisterInt \c_max_register_int
\cs_set_eq:NN \cMaxCharInt    \c_max_char_in

\int_new:N \lTmpaInt   \int_new:N \lTmpbInt   \int_new:N \lTmpcInt
\int_new:N \lTmpiInt   \int_new:N \lTmpjInt   \int_new:N \lTmpkInt
\int_new:N \l@Funx@Int \int_new:N \l@Funy@Int \int_new:N \l@Funz@Int

\int_new:N \gTmpaInt   \int_new:N \gTmpbInt   \int_new:N \gTmpcInt
\int_new:N \gTmpiInt   \int_new:N \gTmpjInt   \int_new:N \gTmpkInt
\int_new:N \g@Funx@Int \int_new:N \g@Funy@Int \int_new:N \g@Funz@Int

\PrGNewFunction \IntEval { m }
{
  \Result { \Expand { \int_eval:n { #1 } } }
}

\PrGNewFunction \IntMathAdd { m m }
{
  \int_set:Nn \l@Funx@Int { \int_eval:n { (#1) + (#2) } }
  \Result { \Value \l@Funx@Int }
}

```

```

\PrGNewFunction \IntMathSub { m m }
{
  \int_set:Nn \l@Funx@Int { \int_eval:n { (#1) - (#2) } }
  \Result { \Value \l@Funx@Int }
}

\PrGNewFunction \IntMathMult { m m }
{
  \int_set:Nn \l@Funx@Int { \int_eval:n { (#1) * (#2) } }
  \Result { \Value \l@Funx@Int }
}

\PrGNewFunction \IntMathDiv { m m }
{
  \Expand { \int_div_round:nn { #1 } { #2 } }
}

\PrGNewFunction \IntMathDivTruncate { m m }
{
  \Expand { \int_div_truncate:nn { #1 } { #2 } }
}

\PrGNewFunction \IntMathSign { m } { \Expand { \int_sign:n { #1 } } }

\PrGNewFunction \IntMathAbs { m } { \Expand { \int_abs:n { #1 } } }

\PrGNewFunction \IntMathMax { m m } { \Expand { \int_max:nn { #1 } { #2 } } }

\PrGNewFunction \IntMathMin { m m } { \Expand { \int_min:nn { #1 } { #2 } } }

\PrGNewFunction \IntMathMod { m m } { \Expand { \int_mod:nn { #1 } { #2 } } }

\PrGNewFunction \IntMathRand { m m } { \Expand { \int_rand:nn { #1 } { #2 } } }

\PrGNewFunction \IntNew { M } { \int_new:N #1 }

\PrGNewFunction \IntConst { M m } { \int_const:Nn #1 { #2 } }

\PrGNewFunction \IntLog { m } { \int_log:n { #1 } }

\PrGNewFunction \IntVarLog { M } { \int_log:N #1 }

\PrGNewFunction \IntShow { m } { \int_show:n { #1 } }

\PrGNewFunction \IntVarShow { M } { \int_show:N #1 }

\PrGNewFunction \IntUse { M } { \Result { \Value #1 } }

\PrGNewFunction \IntSet { M m }
{
  \__fun_do_assignment:Nnn #1 { \int_gset:Nn #1 {#2} } { \int_set:Nn #1 {#2} }
}

```

```

\PrNewFunction \IntZero { M }
{
  \__fun_do_assignment:Nnn #1 { \int_gzero:N #1 } { \int_zero:N #1 }
}

\PrNewFunction \IntZeroNew { M }
{
  \__fun_do_assignment:Nnn #1 { \int_gzero_new:N #1 } { \int_zero_new:N #1 }
}

\PrNewFunction \IntSetEq { M M }
{
  \__fun_do_assignment:Nnn #1 { \int_gset_eq:NN #1 #2 } { \int_set_eq:NN #1 #2 }
}

\PrNewFunction \IntIncr { M }
{
  \__fun_do_assignment:Nnn #1 { \int_gincr:N #1 } { \int_incr:N #1 }
}

\PrNewFunction \IntDecr { M }
{
  \__fun_do_assignment:Nnn #1 { \int_gdecr:N #1 } { \int_decr:N #1 }
}

\PrNewFunction \IntAdd { M m }
{
  \__fun_do_assignment:Nnn #1 { \int_gadd:Nn #1 {#2} } { \int_add:Nn #1 {#2} }
}

\PrNewFunction \IntSub { M m }
{
  \__fun_do_assignment:Nnn #1 { \int_gsub:Nn #1 {#2} } { \int_sub:Nn #1 {#2} }
}

\PrNewFunction \IntStepInline { m m m n }
{
  \int_step_inline:nnnn { #1 } { #2 } { #3 } { #4 }
}

\PrNewFunction \IntStepVariable { m m m M n }
{
  \int_step_variable:nnnNn { #1 } { #2 } { #3 } #4 { #5 }
}

\PrNewConditional \IntIfExist { M }
{
  \int_if_exist:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrNewConditional \IntIfOdd { m }
{
  \int_if_odd:NTF { #1 } { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrNewConditional \IntIfEven { m }

```

```

{
  \int_if_even:nTF { #1 } { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrgNewConditional \IntCompare { m N m }
{
  \int_compare:nNnTF {#1} #2 {#3}
  { \Result { \cTrueBool } }
  { \Result { \cFalseBool } }
}

\PrgNewFunction \IntCase { m m } { \int_case:nn {#1} {#2} }
\PrgNewFunction \IntCaseT { m m n } { \int_case:nnT {#1} {#2} {#3} }
\PrgNewFunction \IntCaseF { m m n } { \int_case:nnF {#1} {#2} {#3} }
\PrgNewFunction \IntCaseTF { m m n n } { \int_case:nnTF {#1} {#2} {#3} {#4} }

```

14.7 Interfaces for Floating Point Numbers (Fp)

```

\fp_set_eq:NN \cZeroFp \c_zero_fp
\fp_set_eq:NN \cMinusZeroFp \c_minus_zero_fp
\fp_set_eq:NN \cOneFp \c_one_fp
\fp_set_eq:NN \cInfFp \c_inf_fp
\fp_set_eq:NN \cMinusInfFp \c_minus_inf_fp
\fp_set_eq:NN \cEFp \c_e_fp
\fp_set_eq:NN \cPiFp \c_pi_fp
\fp_set_eq:NN \cOneDegreeFp \c_one_degree_fp

\fp_new:N \lTmпаFp \fp_new:N \lTmрbFp \fp_new:N \lTmрcFp
\fp_new:N \lTmрiFp \fp_new:N \lTmрjFp \fp_new:N \lTmрkFp
\fp_new:N \l@Funx@Fp \fp_new:N \l@Funy@Fp \fp_new:N \l@Funz@Fp

\fp_new:N \gTmпаFp \fp_new:N \gTmрbFp \fp_new:N \gTmрcFp
\fp_new:N \gTmрiFp \fp_new:N \gTmрjFp \fp_new:N \gTmрkFp
\fp_new:N \g@Funx@Fp \fp_new:N \g@Funy@Fp \fp_new:N \g@Funz@Fp

\PrgNewFunction \FpEval { m }
{
  \Result { \Expand { \fp_eval:n { #1 } } }
}

\PrgNewFunction \FpMathAdd { m m }
{
  \fp_set:Nn \l@Funx@Fp { \fp_eval:n { (#1) + (#2) } }
  \Result { \FpUse \l@Funx@Fp }
}

\PrgNewFunction \FpMathSub { m m }
{
  \fp_set:Nn \l@Funx@Fp { \fp_eval:n { (#1) - (#2) } }
  \Result { \FpUse \l@Funx@Fp }
}

\PrgNewFunction \FpMathMult { m m }
{
  \fp_set:Nn \l@Funx@Fp { \fp_eval:n { (#1) * (#2) } }
}

```



```

    \Result { \FpUse \l@Funx@Fp }
}

\PrGNewFunction \FpMathDiv { m m }
{
    \fp_set:Nn \l@Funx@Fp { \fp_eval:n { (#1) / (#2) } }
    \Result { \FpUse \l@Funx@Fp }
}

\PrGNewFunction \FpMathSign { m }
{
    \Result { \Expand { \fp_sign:n { #1 } } }
}

\PrGNewFunction \FpMathAbs { m }
{
    \Result { \Expand { \fp_abs:n { #1 } } }
}

\PrGNewFunction \FpMathMax { m m }
{
    \Result { \Expand { \fp_max:nn { #1 } { #2 } } }
}

\PrGNewFunction \FpMathMin { m m }
{
    \Result { \Expand { \fp_min:nn { #1 } { #2 } } }
}

\PrGNewFunction \FpNew { M } { \fp_new:N #1 }

\PrGNewFunction \FpConst { M m } { \fp_const:Nn #1 {#2} }

\PrGNewFunction \FpUse { M } { \Result { \Expand { \fp_use:N #1 } } }

\PrGNewFunction \FpLog { m } { \fp_log:n { #1 } }

\PrGNewFunction \FpVarLog { M } { \fp_log:N #1 }

\PrGNewFunction \FpShow { m } { \fp_show:n { #1 } }

\PrGNewFunction \FpVarShow { M } { \fp_show:N #1 }

\PrGNewFunction \FpSet { M m }
{
    \__fun_do_assignment:Nnn #1 { \fp_gset:Nn #1 {#2} } { \fp_set:Nn #1 {#2} }
}

\PrGNewFunction \FpSetEq { M M }
{
    \__fun_do_assignment:Nnn #1 { \fp_gset_eq:NN #1 #2 } { \fp_set_eq:NN #1 #2 }
}

\PrGNewFunction \FpZero { M }

```

```

{
  \__fun_do_assignment:Nnn #1 { \fp_gzero:N #1 } { \fp_zero:N #1 }
}

\PrgNewFunction \FpZeroNew { M }
{
  \__fun_do_assignment:Nnn #1 { \fp_gzero_new:N #1 } { \fp_zero_new:N #1 }
}

\PrgNewFunction \FpAdd { M m }
{
  \__fun_do_assignment:Nnn #1 { \fp_gadd:Nn #1 {#2} } { \fp_add:Nn #1 {#2} }
}

\PrgNewFunction \FpSub { M m }
{
  \__fun_do_assignment:Nnn #1 { \fp_gsub:Nn #1 {#2} } { \fp_sub:Nn #1 {#2} }
}

\PrgNewFunction \FpStepInline { m m m n }
{
  \fp_step_inline:nnnn { #1 } { #2 } { #3 } { #4 }
}

\PrgNewFunction \FpStepVariable { m m m M n }
{
  \fp_step_variable:nnnNn { #1 } { #2 } { #3 } #4 { #5 }
}

\PrgNewConditional \FpIfExist { M }
{
  \fp_if_exist:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrgNewConditional \FpCompare { m N m }
{
  \fp_compare:nNnTF {#1} #2 {#3}
  { \Result { \cTrueBool } }
  { \Result { \cFalseBool } }
}

```

14.8 Interfaces for Dimensions (Dim)

```

\cs_set_eq:NN \cMaxDim \c_max_dim
\cs_set_eq:NN \cZeroDim \c_zero_dim

\dim_new:N \lTmPaDim \dim_new:N \lTmPbDim \dim_new:N \lTmPcDim
\dim_new:N \lTmPiDim \dim_new:N \lTmPjDim \dim_new:N \lTmPkDim
\dim_new:N \l@Funx@Dim \dim_new:N \l@Funy@Dim \dim_new:N \l@Funz@Dim

\dim_new:N \gTmPaDim \dim_new:N \gTmPbDim \dim_new:N \gTmPcDim
\dim_new:N \gTmPiDim \dim_new:N \gTmPjDim \dim_new:N \gTmPkDim
\dim_new:N \g@Funx@Dim \dim_new:N \g@Funy@Dim \dim_new:N \g@Funz@Dim

\PrgNewFunction \DimEval { m }

```

```

{
  \Result { \Expand { \dim_eval:n { #1 } } }
}

\PrgNewFunction \DimMathAdd { m m }
{
  \dim_set:Nn \l@Funx@Dim { \dim_eval:n { (#1) + (#2) } }
  \Result { \Value \l@Funx@Dim }
}

\PrgNewFunction \DimMathSub { m m }
{
  \dim_set:Nn \l@Funx@Dim { \dim_eval:n { (#1) - (#2) } }
  \Result { \Value \l@Funx@Dim }
}

\PrgNewFunction \DimMathSign { m }
{
  \Result { \Expand { \dim_sign:n { #1 } } }
}

\PrgNewFunction \DimMathAbs { m }
{
  \Result { \Expand { \dim_abs:n { #1 } } }
}

\PrgNewFunction \DimMathMax { m m }
{
  \Result { \Expand { \dim_max:nn { #1 } { #2 } } }
}

\PrgNewFunction \DimMathMin { m m }
{
  \Result { \Expand { \dim_min:nn { #1 } { #2 } } }
}

\PrgNewFunction \DimMathRatio { m m }
{
  \Result { \Expand { \dim_ratio:nn { #1 } { #2 } } }
}

\PrgNewFunction \DimNew { M } { \dim_new:N #1 }

\PrgNewFunction \DimConst { M m } { \dim_const:Nn #1 {#2} }

\PrgNewFunction \DimUse { M } { \Result { \Value #1 } }

\PrgNewFunction \DimLog { m } { \dim_log:n { #1 } }

\PrgNewFunction \DimVarLog { M } { \dim_log:N #1 }

\PrgNewFunction \DimShow { m } { \dim_show:n { #1 } }

\PrgNewFunction \DimVarShow { M } { \dim_show:N #1 }

```

```

\PrGNewFunction \DimSet { M m }
{
  \__fun_do_assignment:Nnn #1 { \dim_gset:Nn #1 {#2} } { \dim_set:Nn #1 {#2} }
}

\PrGNewFunction \DimSetEq { M M }
{
  \__fun_do_assignment:Nnn #1 { \dim_gset_eq:NN #1 #2 } { \dim_set_eq:NN #1 #2 }
}

\PrGNewFunction \DimZero { M }
{
  \__fun_do_assignment:Nnn #1 { \dim_gzero:N #1 } { \dim_zero:N #1 }
}

\PrGNewFunction \DimZeroNew { M }
{
  \__fun_do_assignment:Nnn #1 { \dim_gzero_new:N #1 } { \dim_zero_new:N #1 }
}

\PrGNewFunction \DimAdd { M m }
{
  \__fun_do_assignment:Nnn #1 { \dim_gadd:Nn #1 {#2} } { \dim_add:Nn #1 {#2} }
}

\PrGNewFunction \DimSub { M m }
{
  \__fun_do_assignment:Nnn #1 { \dim_gsub:Nn #1 {#2} } { \dim_sub:Nn #1 {#2} }
}

\PrGNewFunction \DimStepInline { m m m n }
{
  \dim_step_inline:nnnn { #1 } { #2 } { #3 } { #4 }
}

\PrGNewFunction \DimStepVariable { m m m M n }
{
  \dim_step_variable:nnnNn { #1 } { #2 } { #3 } #4 { #5 }
}

\PrGNewConditional \DimIfExist { M }
{
  \dim_if_exist:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrGNewConditional \DimCompare { m N m }
{
  \dim_compare:nNnTF {#1} #2 {#3}
  { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrGNewFunction \DimCase { m m } { \dim_case:nn {#1} {#2} }
\PrGNewFunction \DimCaseT { m m n } { \dim_case:nnT {#1} {#2} {#3} }
\PrGNewFunction \DimCaseF { m m n } { \dim_case:nnF {#1} {#2} {#3} }
\PrGNewFunction \DimCaseTF { m m n n } { \dim_case:nnTF {#1} {#2} {#3} {#4} }

```

14.9 Interfaces for Sorting Functions (Sort)

```
\cs_set_eq:NN \SortReturnSame \sort_return_same:
\cs_set_eq:NN \SortReturnSwapped \sort_return_swapped:
```

14.10 Interfaces for Comma Separated Lists (Clist)

```
\clist_new:N \lTmпаClist    \clist_new:N \lTmрbClist    \clist_new:N \lTmрcClist
\clist_new:N \lTmрiClist    \clist_new:N \lTmрjClist    \clist_new:N \lTmрkClist
\clist_new:N \l@Funx@Clist  \clist_new:N \l@Funy@Clist  \clist_new:N \l@Funz@Clist
```

```
\clist_new:N \gTmпаClist    \clist_new:N \gTmрbClist    \clist_new:N \gTmрcClist
\clist_new:N \gTmрiClist    \clist_new:N \gTmрjClist    \clist_new:N \gTmрkClist
\clist_new:N \g@Funx@Clist  \clist_new:N \g@Funy@Clist  \clist_new:N \g@Funz@Clist
```

```
\clist_set_eq:NN \cEmptyClist \c_empty_clist
```

```
\PrgNewFunction \ClistNew { M } { \clist_new:N #1 }
```

```
\PrgNewFunction \ClistLog { m } { \clist_log:n { #1 } }
```

```
\PrgNewFunction \ClistVarLog { M } { \clist_log:N #1 }
```

```
\PrgNewFunction \ClistShow { m } { \clist_show:n { #1 } }
```

```
\PrgNewFunction \ClistVarShow { M } { \clist_show:N #1 }
```

```
\PrgNewFunction \ClistVarJoin { M m }
{
  \Expand { \clist_use:Nn #1 { #2 } }
}
```

```
\PrgNewFunction \ClistVarJoinExtended { M m m m }
{
  \Expand { \clist_use:Nnn #1 { #2 } { #3 } { #4 } }
}
```

```
\PrgNewFunction \ClistJoin { m m }
{
  \Expand { \clist_use:nn { #1 } { #2 } }
}
```

```
\PrgNewFunction \ClistJoinExtended { m m m m }
{
  \Expand { \clist_use:nnnn { #1 } { #2 } { #3 } { #4 } }
}
```

```
\PrgNewFunction \ClistConst { M m } { \clist_const:Nn #1 { #2 } }
```

```
\PrgNewFunction \ClistSet { M m }
{
  \__fun_do_assignment:Nnn #1
  { \clist_gset:Nn #1 {#2} } { \clist_set:Nn #1 {#2} }
```

```

}

\PrGNewFunction \ClistSetEq { M M }
{
  \__fun_do_assignment:Nnn #1
  { \clist_gset_eq:NN #1 #2 } { \clist_set_eq:NN #1 #2 }
}

\PrGNewFunction \ClistSetFromSeq { M M }
{
  \__fun_do_assignment:Nnn #1
  { \clist_gset_from_seq:NN #1 #2 } { \clist_set_from_seq:NN #1 #2 }
}

\PrGNewFunction \ClistConcat { M M M }
{
  \__fun_do_assignment:Nnn #1
  { \clist_gconcat:NNN #1 #2 #3 } { \clist_concat:NNN #1 #2 #3 }
}

\PrGNewFunction \ClistClear { M }
{
  \__fun_do_assignment:Nnn #1 { \clist_gclear:N #1 } { \clist_clear:N #1 }
}

\PrGNewFunction \ClistClearNew { M }
{
  \__fun_do_assignment:Nnn #1 { \clist_gclear_new:N #1 } { \clist_clear_new:N #1 }
}

\PrGNewFunction \ClistPutLeft { M m }
{
  \__fun_do_assignment:Nnn #1
  { \clist_gput_left:Nn #1 {#2} } { \clist_put_left:Nn #1 {#2} }
}

\PrGNewFunction \ClistPutRight { M m }
{
  \__fun_do_assignment:Nnn #1
  { \clist_gput_right:Nn #1 {#2} } { \clist_put_right:Nn #1 {#2} }
}

\PrGNewFunction \ClistVarRemoveDuplicates { M }
{
  \__fun_do_assignment:Nnn #1
  { \clist_gremove_duplicates:N #1 } { \clist_remove_duplicates:N #1 }
}

\PrGNewFunction \ClistVarRemoveAll { M m }
{
  \__fun_do_assignment:Nnn #1
  { \clist_gremove_all:Nn #1 {#2} } { \clist_remove_all:Nn #1 {#2} }
}

\PrGNewFunction \ClistVarReverse { M }

```

```

{
  \__fun_do_assignment:Nnn #1 { \clist_greverse:N #1 } { \clist_reverse:N #1 }
}

\PrgNewFunction \ClistVarSort { M m }
{
  \__fun_do_assignment:Nnn #1
  { \clist_gsort:Nn #1 {#2} } { \clist_sort:Nn #1 {#2} }
}

\PrgNewFunction \ClistCount { m } { \Expand { \clist_count:n { #1 } } }

\PrgNewFunction \ClistVarCount { M } { \Expand { \clist_count:N #1 } }

\PrgNewFunction \ClistGet { M M }
{
  \clist_get:NN #1 #2
  \__fun_quark_upgrade_no_value:N #2
}
\PrgNewFunction \ClistGetT { M M n } { \clist_get:NNT #1 #2 {#3} }
\PrgNewFunction \ClistGetF { M M n } { \clist_get:NNF #1 #2 {#3} }
\PrgNewFunction \ClistGetTF { M M n n } { \clist_get:NNTF #1 #2 {#3} {#4} }

\PrgNewFunction \ClistPop { M M }
{
  \__fun_do_assignment:Nnn #1
  { \clist_gpop:NN #1 #2 } { \clist_pop:NN #1 #2 }
  \__fun_quark_upgrade_no_value:N #2
}
\PrgNewFunction \ClistPopT { M M n }
{
  \__fun_do_assignment:Nnn #1
  { \clist_gpop:NNT #1 #2 {#3} } { \clist_pop:NNT #1 #2 {#3} }
}
\PrgNewFunction \ClistPopF { M M n }
{
  \__fun_do_assignment:Nnn #1
  { \clist_gpop:NNF #1 #2 {#3} } { \clist_pop:NNF #1 #2 {#3} }
}
\PrgNewFunction \ClistPopTF { M M n n }
{
  \__fun_do_assignment:Nnn #1
  { \clist_gpop:NNTF #1 #2 {#3} {#4} } { \clist_pop:NNTF #1 #2 {#3} {#4} }
}

\PrgNewFunction \ClistPush { M m }
{
  \__fun_do_assignment:Nnn #1
  { \clist_gpush:Nn #1 {#2} } { \clist_push:Nn #1 {#2} }
}

\PrgNewFunction \ClistItem { m m } { \Expand { \clist_item:nn {#1} {#2} } }

\PrgNewFunction \ClistVarItem { M m } { \Expand { \clist_item:Nn #1 {#2} } }

\PrgNewFunction \ClistRandItem { m } { \Expand { \clist_rand_item:n {#1} } }

```

```

\PrGNewFunction \ClistVarRandItem { M } { \Expand { \clist_rand_item:N #1 } }

\PrGNewFunction \ClistMapInline { m n }
{
  \clist_map_inline:nn {#1} {#2}
}

\PrGNewFunction \ClistVarMapInline { M n }
{
  \clist_map_inline:Nn #1 {#2}
}

\PrGNewFunction \ClistMapVariable { m M n }
{
  \clist_map_variable:nNn {#1} #2 {#3}
}

\PrGNewFunction \ClistVarMapVariable { M M n }
{
  \clist_map_variable:NNn #1 #2 {#3}
}

\cs_set_eq:NN \ClistMapBreak \clist_map_break:

\PrGNewConditional \ClistIfExist { M }
{
  \clist_if_exist:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrGNewConditional \ClistIfEmpty { m }
{
  \clist_if_empty:nTF {#1} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrGNewConditional \ClistVarIfEmpty { M }
{
  \clist_if_empty:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrGNewConditional \ClistIfIn { m m }
{
  \clist_if_in:nnTF {#1} {#2}
  { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrGNewConditional \ClistVarIfIn { M m }
{
  \clist_if_in:NnTF #1 {#2}
  { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

```

14.11 Interfaces for Sequences and Stacks (Seq)

```

\seq_new:N \lTmpaSeq   \seq_new:N \lTmpbSeq   \seq_new:N \lTmpcSeq

```



```

\seq_new:N \lTmpiSeq \seq_new:N \lTmpjSeq \seq_new:N \lTmpkSeq
\seq_new:N \l@Funx@Seq \seq_new:N \l@Funy@Seq \seq_new:N \l@Funz@Seq

\seq_new:N \gTmptaSeq \seq_new:N \gTmpbSeq \seq_new:N \gTmpcSeq
\seq_new:N \gTmpiSeq \seq_new:N \gTmpjSeq \seq_new:N \gTmpkSeq
\seq_new:N \g@Funx@Seq \seq_new:N \g@Funy@Seq \seq_new:N \g@Funz@Seq

\seq_set_eq:NN \cEmptySeq \c_empty_seq

\PrgNewFunction \SeqNew { M } { \seq_new:N #1 }

\PrgNewFunction \SeqVarLog { M } { \seq_log:N #1 }

\PrgNewFunction \SeqVarShow { M } { \seq_show:N #1 }

\PrgNewFunction \SeqVarJoin { M m }
{
  \Expand { \seq_use:Nn #1 { #2 } }
}

\PrgNewFunction \SeqVarJoinExtended { M m m m }
{
  \Expand { \seq_use:Nnn #1 { #2 } { #3 } { #4 } }
}

\PrgNewFunction \SeqJoin { m m }
{
  \Expand { \seq_use:nn { #1 } { #2 } }
}

\PrgNewFunction \SeqJoinExtended { m m m m }
{
  \Expand { \seq_use:nnnn { #1 } { #2 } { #3 } { #4 } }
}

\PrgNewFunction \SeqConstFromClist { M m } { \seq_const_from_clist:Nn #1 { #2 } }

\PrgNewFunction \SeqSetFromClist { M m }
{
  \__fun_do_assignment:Nnn #1
  { \seq_gset_from_clist:Nn #1 {#2} } { \seq_set_from_clist:Nn #1 {#2} }
}

\PrgNewFunction \SeqSetEq { M M }
{
  \__fun_do_assignment:Nnn #1
  { \seq_gset_eq:NN #1 #2 } { \seq_set_eq:NN #1 #2 }
}

\PrgNewFunction \SeqSetSplit { M m m }
{
  \__fun_do_assignment:Nnn #1
  { \seq_gset_split:Nnn #1 {#2} {#3} } { \seq_set_split:Nnn #1 {#2} {#3} }
}

```

```

\PrNewFunction \SeqConcat { M M M }
{
  \__fun_do_assignment:Nnn #1
  { \seq_gconcat:NNN #1 #2 #3 } { \seq_concat:NNN #1 #2 #3 }
}

\PrNewFunction \SeqClear { M }
{
  \__fun_do_assignment:Nnn #1 { \seq_gclear:N #1 } { \seq_clear:N #1 }
}

\PrNewFunction \SeqClearNew { M }
{
  \__fun_do_assignment:Nnn #1 { \seq_gclear_new:N #1 } { \seq_clear_new:N #1 }
}

\PrNewFunction \SeqPutLeft { M m }
{
  \__fun_do_assignment:Nnn #1
  { \seq_gput_left:Nn #1 {#2} } { \seq_put_left:Nn #1 {#2} }
}

\PrNewFunction \SeqPutRight { M m }
{
  \__fun_do_assignment:Nnn #1
  { \seq_gput_right:Nn #1 {#2} } { \seq_put_right:Nn #1 {#2} }
}

\PrNewFunction \SeqVarRemoveDuplicates { M }
{
  \__fun_do_assignment:Nnn #1
  { \seq_gremove_duplicates:N #1 } { \seq_remove_duplicates:N #1 }
}

\PrNewFunction \SeqVarRemoveAll { M m }
{
  \__fun_do_assignment:Nnn #1
  { \seq_gremove_all:Nn #1 {#2} } { \seq_remove_all:Nn #1 {#2} }
}

\PrNewFunction \SeqVarReverse { M }
{
  \__fun_do_assignment:Nnn #1 { \seq_greverse:N #1 } { \seq_reverse:N #1 }
}

\PrNewFunction \SeqVarSort { M m }
{
  \__fun_do_assignment:Nnn #1
  { \seq_gsort:Nn #1 {#2} } { \seq_sort:Nn #1 {#2} }
}

\PrNewFunction \SeqVarCount { M } { \Expand { \seq_count:N #1 } }

\PrNewFunction \SeqGet { M M }
{

```

```

    \seq_get:NN #1 #2
    \__fun_quark_upgrade_no_value:N #2
}
\PrgNewFunction \SeqGetT { M M n } { \seq_get:NNT #1 #2 {#3} }
\PrgNewFunction \SeqGetF { M M n } { \seq_get:NNF #1 #2 {#3} }
\PrgNewFunction \SeqGetTF { M M n n } { \seq_get:NNTF #1 #2 {#3} {#4} }

\PrgNewFunction \SeqPop { M M }
{
    \__fun_do_assignment:Nnn #1
    { \seq_gpop:NN #1 #2 } { \seq_pop:NN #1 #2 }
    \__fun_quark_upgrade_no_value:N #2
}
\PrgNewFunction \SeqPopT { M M n }
{
    \__fun_do_assignment:Nnn #1
    { \seq_gpop:NNT #1 #2 {#3} } { \seq_pop:NNT #1 #2 {#3} }
}
\PrgNewFunction \SeqPopF { M M n }
{
    \__fun_do_assignment:Nnn #1
    { \seq_gpop:NNF #1 #2 {#3} } { \seq_pop:NNF #1 #2 {#3} }
}
\PrgNewFunction \SeqPopTF { M M n n }
{
    \__fun_do_assignment:Nnn #1
    { \seq_gpop:NNTF #1 #2 {#3} {#4} } { \seq_pop:NNTF #1 #2 {#3} {#4} }
}

\PrgNewFunction \SeqPush { M m }
{
    \__fun_do_assignment:Nnn #1
    { \seq_gpush:Nn #1 {#2} } { \seq_push:Nn #1 {#2} }
}

\PrgNewFunction \SeqGetLeft { M M }
{
    \seq_get_left:NN #1 #2
    \__fun_quark_upgrade_no_value:N #2
}
\PrgNewFunction \SeqGetLeftT { M M n } { \seq_get_left:NNT #1 #2 {#3} }
\PrgNewFunction \SeqGetLeftF { M M n } { \seq_get_left:NNF #1 #2 {#3} }
\PrgNewFunction \SeqGetLeftTF { M M n n } { \seq_get_left:NNTF #1 #2 {#3} {#4} }

\PrgNewFunction \SeqGetRight { M M }
{
    \seq_get_right:NN #1 #2
    \__fun_quark_upgrade_no_value:N #2
}
\PrgNewFunction \SeqGetRightT { M M n } { \seq_get_right:NNT #1 #2 {#3} }
\PrgNewFunction \SeqGetRightF { M M n } { \seq_get_right:NNF #1 #2 {#3} }
\PrgNewFunction \SeqGetRightTF { M M n n } { \seq_get_right:NNTF #1 #2 {#3} {#4} }

\PrgNewFunction \SeqPopLeft { M M }
{
    \__fun_do_assignment:Nnn #1
    { \seq_gpop_left:NN #1 #2 } { \seq_pop_left:NN #1 #2 }
}

```

```

    \__fun_quark_upgrade_no_value:N #2
  }
\PrgNewFunction \SeqPopLeftT { M M n }
{
  \__fun_do_assignment:Nnn #1
  { \seq_gpop_left:NNT #1 #2 {#3} } { \seq_pop_left:NNT #1 #2 {#3} }
}
\PrgNewFunction \SeqPopLeftF { M M n }
{
  \__fun_do_assignment:Nnn #1
  { \seq_gpop_left:NNF #1 #2 {#3} } { \seq_pop_left:NNF #1 #2 {#3} }
}
\PrgNewFunction \SeqPopLeftTF { M M n n }
{
  \__fun_do_assignment:Nnn #1
  { \seq_gpop_left:NNTF #1 #2 {#3} {#4} }
  { \seq_pop_left:NNTF #1 #2 {#3} {#4} }
}

\PrgNewFunction \SeqPopRight { M M }
{
  \__fun_do_assignment:Nnn #1
  { \seq_gpop_right:NN #1 #2 } { \seq_pop_right:NN #1 #2 }
  \__fun_quark_upgrade_no_value:N #2
}
\PrgNewFunction \SeqPopRightT { M M n }
{
  \__fun_do_assignment:Nnn #1
  { \seq_gpop_right:NNT #1 #2 {#3} } { \seq_pop_right:NNT #1 #2 {#3} }
}
\PrgNewFunction \SeqPopRightF { M M n }
{
  \__fun_do_assignment:Nnn #1
  { \seq_gpop_right:NNF #1 #2 {#3} } { \seq_pop_right:NNF #1 #2 {#3} }
}
\PrgNewFunction \SeqPopRightTF { M M n n }
{
  \__fun_do_assignment:Nnn #1
  { \seq_gpop_right:NNTF #1 #2 {#3} {#4} }
  { \seq_pop_right:NNTF #1 #2 {#3} {#4} }
}

\PrgNewFunction \SeqVarItem { M m } { \Expand { \seq_item:Nn #1 {#2} } }

\PrgNewFunction \SeqVarRandItem { M } { \Expand { \seq_rand_item:N #1 } }

\PrgNewFunction \SeqVarMapInline { M n }
{
  \seq_map_inline:Nn #1 {#2}
}

\PrgNewFunction \SeqVarMapVariable { M M n }
{
  \seq_map_variable:NNn #1 #2 {#3}
}

\cs_set_eq:NN \SeqMapBreak \seq_map_break:

```

```

\PrGNewConditional \SeqIfExist { M }
{
  \seq_if_exist:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrGNewConditional \SeqVarIfEmpty { M }
{
  \seq_if_empty:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrGNewConditional \SeqVarIfIn { M m }
{
  \seq_if_in:NnTF #1 {#2}
  { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

```

14.12 Interfaces for Property Lists (Prop)

```

\prop_new:N \lTmPaProp   \prop_new:N \lTmPbProp   \prop_new:N \lTmPcProp
\prop_new:N \lTmPiProp   \prop_new:N \lTmPjProp   \prop_new:N \lTmPkProp
\prop_new:N \l@Funx@Prop \prop_new:N \l@Funy@Prop \prop_new:N \l@Funz@Prop

\prop_new:N \gTmPaProp   \prop_new:N \gTmPbProp   \prop_new:N \gTmPcProp
\prop_new:N \gTmPiProp   \prop_new:N \gTmPjProp   \prop_new:N \gTmPkProp
\prop_new:N \g@Funx@Prop \prop_new:N \g@Funy@Prop \prop_new:N \g@Funz@Prop

\prop_set_eq:NN \cEmptyProp \c_empty_prop

\PrGNewFunction \PropNew { M } { \prop_new:N #1 }

\PrGNewFunction \PropVarLog { M } { \prop_log:N #1 }

\PrGNewFunction \PropVarShow { M } { \prop_show:N #1 }

\PrGNewFunction \PropConstFromKeyval { M m }
{ \prop_const_from_keyval:Nn #1 { #2 } }

\PrGNewFunction \PropSetFromKeyval { M m }
{
  \__fun_do_assignment:Nnn #1
  { \prop_gset_from_keyval:Nn #1 {#2} } { \prop_set_from_keyval:Nn #1 {#2} }
}

\PrGNewFunction \PropSetEq { M M }
{
  \__fun_do_assignment:Nnn #1
  { \prop_gset_eq:NN #1 #2 } { \prop_set_eq:NN #1 #2 }
}

\PrGNewFunction \PropClear { M }
{
  \__fun_do_assignment:Nnn #1 { \prop_gclear:N #1 } { \prop_clear:N #1 }
}

```

```

\PrgNewFunction \PropClearNew { M }
{
  \__fun_do_assignment:Nnn #1 { \prop_gclear_new:N #1 } { \prop_clear_new:N #1 }
}

\PrgNewFunction \PropConcat { M M M }
{
  \__fun_do_assignment:Nnn #1
  { \prop_gconcat:NNN #1 #2 #3 } { \prop_concat:NNN #1 #2 #3 }
}

\PrgNewFunction \PropPut { M m m }
{
  \__fun_do_assignment:Nnn #1
  { \prop_gput:Nnn #1 {#2} {#3} } { \prop_put:Nnn #1 {#2} {#3} }
}

\PrgNewFunction \PropPutIfNew { M m m }
{
  \__fun_do_assignment:Nnn #1
  { \prop_gput_if_new:Nnn #1 {#2} {#3} } { \prop_put_if_new:Nnn #1 {#2} {#3} }
}

\PrgNewFunction \PropPutFromKeyval { M m }
{
  \__fun_do_assignment:Nnn #1
  { \prop_gput_from_keyval:Nn #1 {#2} } { \prop_put_from_keyval:Nn #1 {#2} }
}

\PrgNewFunction \PropVarRemove { M m }
{
  \__fun_do_assignment:Nnn #1
  { \prop_gremove:Nn #1 {#2} } { \prop_remove:Nn #1 {#2} }
}

\PrgNewFunction \PropVarCount { M } { \Expand { \prop_count:N #1 } }

\PrgNewFunction \PropVarItem { M m } { \Expand { \prop_item:Nn #1 {#2} } }

\PrgNewFunction \PropToKeyval { M } { \Expand { \prop_to_keyval:N #1 } }

\PrgNewFunction \PropGet { M m M }
{
  \prop_get:NnN #1 {#2} #3
  \__fun_quark_upgrade_no_value:N #3
}

\PrgNewFunction \PropGetT { M m M n } { \prop_get:NnNT #1 {#2} #3 {#4} }
\PrgNewFunction \PropGetF { M m M n } { \prop_get:NnNF #1 {#2} #3 {#4} }
\PrgNewFunction \PropGetTF { M m M n n } { \prop_get:NnNTF #1 {#2} #3 {#4} {#5} }

\PrgNewFunction \PropPop { M m M }
{
  \__fun_do_assignment:Nnn #1
  { \prop_gpop:NnN #1 {#2} #3 } { \prop_pop:NnN #1 {#2} #3 }
  \__fun_quark_upgrade_no_value:N #3
}

```

```

}
\PrgNewFunction \PropPopT { M m M n }
{
  \__fun_do_assignment:Nnn #1
  { \prop_gpop:NnNT #1 {#2} #3 {#4} } { \prop_pop:NnNT #1 {#2} #3 {#4} }
}
\PrgNewFunction \PropPopF { M m M n }
{
  \__fun_do_assignment:Nnn #1
  { \prop_gpop:NnNF #1 {#2} #3 {#4} } { \prop_pop:NnNF #1 {#2} #3 {#4} }
}
\PrgNewFunction \PropPopTF { M m M n n }
{
  \__fun_do_assignment:Nnn #1
  { \prop_gpop:NnNTF #1 {#2} #3 {#4} {#5} }
  { \prop_pop:NnNTF #1 {#2} #3 {#4} {#5} }
}

\PrgNewFunction \PropVarMapInline { M n }
{
  \prop_map_inline:Nn #1 {#2}
}

\cs_set_eq:NN \PropMapBreak \prop_map_break:

\PrgNewConditional \PropIfExist { M }
{
  \prop_if_exist:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrgNewConditional \PropVarIfEmpty { M }
{
  \prop_if_empty:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

\PrgNewConditional \PropVarIfIn { M m }
{
  \prop_if_in:NnTF #1 {#2}
  { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

```

14.13 Interfaces for Quarks (Quark)

```

\quark_new:N \qNoValue

\cs_new_protected:Npn \__fun_quark_upgrade_no_value:N #1
{
  \quark_if_no_value:NT #1 { \tl_set_eq:NN #1 \qNoValue }
}

\PrgNewConditional \QuarkVarIfNoValue { M }
{
  \tl_if_eq:NNTF \qNoValue #1
  { \Result { \cTrueBool } } { \Result { \cFalseBool } }
}

```