

# The `calc` package

## Infix notation arithmetic in $\text{\LaTeX}^*$

Kresten Krab Thorup, Frank Jensen (and Chris Rowley)

2017/05/25

This file is maintained by the  $\text{\LaTeX}$  Project team.  
Bug reports can be opened (category `tools`) at  
<https://latex-project.org/bugs.html>.

### Abstract

The `calc` package reimplements the  $\text{\LaTeX}$  commands `\setcounter`, `\addtocounter`, `\setlength`, and `\addtolength`. Instead of a simple value, these commands now accept an infix notation expression.

## 1 Introduction

Arithmetic in  $\text{\TeX}$  is done using low-level operations such as `\advance` and `\multiply`. This may be acceptable when developing a macro package, but it is not an acceptable interface for the end-user.

This package introduces proper infix notation arithmetic which is much more familiar to most people. The infix notation is more readable and easier to modify than the alternative: a sequence of assignment and arithmetic instructions. One of the arithmetic instructions (`\divide`) does not even have an equivalent in standard  $\text{\LaTeX}$ .

The infix expressions can be used in arguments to macros (the `calc` package doesn't employ category code changes to achieve its goals).<sup>1</sup>

## 2 Informal description

Standard  $\text{\LaTeX}$  provides the following set of commands to manipulate counters and lengths [2, pages 194 and 216].

`\setcounter{ctr}{num}` sets the value of the counter *ctr* equal to (the value of) *num*. (Fragile)

---

\*We thank Frank Mittelbach for his valuable comments and suggestions which have greatly improved this package.

<sup>1</sup>However, it therefore assumes that the category codes of the special characters, such as `(*/)` in its syntax do not change.

`\addtocounter{ctr}{num}` increments the value of the counter *ctr* by (the value of) *num*. (Fragile)

`\setlength{cmd}{len}` sets the value of the length command *cmd* equal to (the value of) *len*. (Robust)

`\addtolength{cmd}{len}` sets the value of the length command *cmd* equal to its current value plus (the value of) *len*. (Robust)

(The `\setcounter` and `\addtocounter` commands have global effect, while the `\setlength` and `\addtolength` commands obey the normal scoping rules.) In standard L<sup>A</sup>T<sub>E</sub>X, the arguments to these commands must be simple values. The `calc` package extends these commands to accept infix notation expressions, denoting values of appropriate types. Using the `calc` package, *num* is replaced by  $\langle$ integer expression $\rangle$ , and *len* is replaced by  $\langle$ glue expression $\rangle$ . The formal syntax of  $\langle$ integer expression $\rangle$  and  $\langle$ glue expression $\rangle$  is given below.

In addition to these commands to explicitly set a length, many L<sup>A</sup>T<sub>E</sub>X commands take a length argument. After loading this package, most of these commands will accept a  $\langle$ glue expression $\rangle$ . This includes the optional width argument of `\makebox`, the width argument of `\parbox`, `minipage`, and a `tabular` p-column, and many similar constructions. (This package does not redefine any of these commands, but they are defined by default to read their arguments by `\setlength` and so automatically benefit from the enhanced `\setlength` command provided by this package.)

In the following, we shall use standard T<sub>E</sub>X terminology. The correspondence between T<sub>E</sub>X and L<sup>A</sup>T<sub>E</sub>X terminology is as follows: L<sup>A</sup>T<sub>E</sub>X counters correspond to T<sub>E</sub>X's count registers; they hold quantities of type  $\langle$ number $\rangle$ . L<sup>A</sup>T<sub>E</sub>X length commands correspond to T<sub>E</sub>X's dimen (for rigid lengths) and skip (for rubber lengths) registers; they hold quantities of types  $\langle$ dimen $\rangle$  and  $\langle$ glue $\rangle$ , respectively.

T<sub>E</sub>X gives us primitive operations to perform arithmetic on registers as follows:

- addition and subtraction on all types of quantities without restrictions;
- multiplication and division by an *integer* can be performed on a register of any type;
- multiplication by a *real* number (i.e., a number with a fractional part) can be performed on a register of any type, but the stretch and shrink components of a glue quantity are discarded.

The `calc` package uses these T<sub>E</sub>X primitives but provides a more user-friendly notation for expressing the arithmetic.

An expression is formed of numerical quantities (such as explicit constants and L<sup>A</sup>T<sub>E</sub>X counters and length commands) and binary operators (the tokens '+', '-', '\*', and '/' with their usual meaning) using the familiar infix notation; parentheses may be used to override the usual precedences (that multiplication/division have higher precedence than addition/subtraction).

Expressions must be properly typed. This means, e.g., that a dimen expression must be a sum of dimen terms: i.e., you cannot say '2cm+4' but '2cm+4pt' is valid.

In a dimen term, the dimension part must come first; the same holds for glue terms. Also, multiplication and division by non-integer quantities require a special syntax; see below.

Evaluation of subexpressions at the same level of precedence proceeds from left to right. Consider a dimen term such as “4cm\*3\*4”. First, the value of the factor 4cm is assigned to a dimen register, then this register is multiplied by 3 (using `\multiply`), and, finally, the register is multiplied by 4 (again using `\multiply`). This also explains why the dimension part (i.e., the part with the unit designation) must come first; T<sub>E</sub>X simply doesn’t allow untyped constants to be assigned to a dimen register.

The `calc` package also allows multiplication and division by real numbers. However, a special syntax is required: you must use `\real{⟨decimal constant⟩}`<sup>2</sup> or `\ratio{⟨dimen expression⟩}{⟨dimen expression⟩}` to denote a real value to be used for multiplication/division. The first form has the obvious meaning, and the second form denotes the number obtained by dividing the value of the first expression by the value of the second expression.

A later addition to the package (in June 1998) allows an additional method of specifying a factor of type dimen by setting some text (in LR-mode) and measuring its dimensions: these are denoted as follows.

`\widthof{⟨text⟩}   \heightof{⟨text⟩}   \depthof{⟨text⟩}`

These calculate the natural sizes of the `⟨text⟩` in exactly the same way as is done for the commands `\settowidth` etc. on Page 216 of the manual [2]. In August 2005 the package was further extended to provide the command

`\totalheightof{⟨text⟩}`

This command does exactly what you’d expect from its name. Additionally the package also provides the command

`\settototalheight{⟨cmd⟩}{⟨text⟩}`

Note that there is a small difference in the usage of these two methods of accessing text dimensions. After `\settowidth{⟨txtwd⟩}{Some text}` you can use:

`\setlength{⟨parskip⟩}{0.68⟨txtwd⟩}`

whereas using the more direct access to the width of the text requires the longer form for multiplication, thus:

`\setlength{⟨parskip⟩}{\widthof{Some text} * \real{0.68}}`

T<sub>E</sub>X discards the stretch and shrink components of glue when glue is multiplied by a real number. So, for example,

`\setlength{⟨parskip⟩}{3pt plus 3pt * \real{1.5}}`

will set the paragraph separation to 4.5pt with no stretch or shrink. Incidentally, note how spaces can be used to enhance readability. When T<sub>E</sub>X is scanning for a `⟨number⟩` etc. it is common to terminate the scanning with a space token or by inserting `\relax`. As of version 4.3 `calc` allows `\relax` tokens to appear in places where they would usually be used for terminating T<sub>E</sub>X’s scanning. In short this is just before any of `+-*/` or at the end of the expression being evaluated.

---

<sup>2</sup>Actually, instead of `⟨decimal constant⟩`, the more general `⟨optional signs⟩⟨factor⟩` can be used. However, that doesn’t add any extra expressive power to the language of infix expressions.

When  $\text{\TeX}$  performs arithmetic on integers, any fractional part of the results are discarded. For example,

```
\setcounter{x}{7/2}
\setcounter{y}{3*\real{1.6}}
\setcounter{z}{3*\real{1.7}}
```

will assign the value 3 to the counter `x`, the value 4 to `y`, and the value 5 to `z`. This truncation also applies to *intermediate* results in the sequential computation of a composite expression; thus, the following command

```
\setcounter{x}{3 * \real{1.6} * \real{1.7}}
```

will assign 6 to `x`.

As an example of the use of `\ratio`, consider the problem of scaling a figure to occupy the full width (i.e., `\textwidth`) of the body of a page. Assume that the original dimensions of the figure are given by the `dimen` (length) variables, `\Xsize` and `\Ysize`. The height of the scaled figure can then be expressed by

```
\setlength{\newYsize}{\Ysize*\ratio{\textwidth}{\Xsize}}
```

Another new feature introduced in August 2005 was max and min operations with associated macros

```
\maxof{<type expression>}{<type expression>}
\minof{<type expression>}{<type expression>}
```

When *type* is either *glue* or *dimen* these macros are allowed only as part of addition or subtraction but when *type* is *integer* they can also be used when multiplying and dividing. In the latter case they follow the same syntax rules as `\ratio` and `\real` which means they must come after the `*` or the `/`. Thus

```
\setcounter{x}{3*\maxof{4+5}{3*4}+\minof{2*\real{1.6}}{5-1}}
```

will assign  $3 \times \max(9, 12) + \min(3, 4) = 39$  to `x`. Similarly

```
\setlength{\parindent}{%
\minof{3pt}{\parskip}*\real{1.5}*\maxof{2*\real{1.6}}{2-1}}
```

will assign  $\min(13.5\text{pt}, 4.5\text{\parskip})$  to `\parindent`

### 3 Formal syntax

The syntax is described by the following set of rules. Note that the definitions of *number*, *dimen*, *glue*, *decimal constant*, and *plus or minus* are as in Chapter 24 of The  $\text{\TeX}$ book [1]; and *text* is LR-mode material, as in the manual [2]. We use *type* as a meta-variable, standing for ‘integer’, ‘dimen’, and ‘glue’.<sup>3</sup>

$$\begin{array}{lcl} \langle \textit{type} \text{ expression} \rangle & \longrightarrow & \langle \textit{type} \text{ term} \rangle \quad | \quad \langle \textit{type} \text{ expression} \rangle \\ & & \langle \textit{plus or minus} \rangle \langle \textit{type} \text{ term} \rangle \\ \\ \langle \textit{type} \text{ term} \rangle & \longrightarrow & \langle \textit{type} \text{ term} \rangle \langle \textit{type} \text{ scan stop} \rangle \quad | \quad \langle \textit{type} \text{ factor} \rangle \\ & & | \quad \langle \textit{type} \text{ term} \rangle \langle \textit{multiply or divide} \rangle \langle \textit{integer} \rangle \quad | \quad \langle \textit{type} \text{ term} \rangle \end{array}$$


---

<sup>3</sup>This version of the `calc` package doesn’t support evaluation of *muglue* expressions.

$\langle \text{multiply or divide} \rangle \langle \text{real number} \rangle \quad | \quad \langle \text{type term} \rangle$   
 $\langle \text{multiply or divide} \rangle \quad \langle \text{max or min integer} \rangle$   
 $\langle \text{type scan stop} \rangle \quad \longrightarrow \quad \langle \text{empty} \rangle \quad | \quad \langle \text{optional space} \rangle \quad | \quad \backslash \text{relax}$   
 $\langle \text{type factor} \rangle \quad \longrightarrow \quad \langle \text{type} \rangle \quad | \quad \langle \text{text dimen factor} \rangle$   
 $\quad | \quad \langle \text{max or min type} \rangle \quad | \quad (_{12} \langle \text{type expression} \rangle )_{12}$   
 $\langle \text{integer} \rangle \quad \longrightarrow \quad \langle \text{number} \rangle$   
 $\langle \text{max or min type} \rangle \quad \longrightarrow \quad \langle \text{max or min command} \rangle \quad \{$   
 $\quad \langle \text{type expression} \rangle \quad \} \quad \{ \langle \text{type expression} \rangle \quad \}$   
 $\langle \text{max or min command} \rangle \quad \longrightarrow \quad \backslash \text{maxof} \quad | \quad \backslash \text{minof}$   
 $\langle \text{text dimen factor} \rangle \quad \longrightarrow \quad \langle \text{text dimen command} \rangle \{ \langle \text{text} \rangle \}$   
 $\langle \text{text dimen command} \rangle \quad \longrightarrow \quad \backslash \text{widthof} \quad | \quad \backslash \text{heightof}$   
 $\quad | \quad \backslash \text{depthof} \quad | \quad \backslash \text{totalheightof}$   
 $\langle \text{multiply or divide} \rangle \quad \longrightarrow \quad *_{12} \quad | \quad /_{12}$   
 $\langle \text{real number} \rangle \quad \longrightarrow \quad \backslash \text{ratio} \{ \langle \text{dimen expression} \rangle \} \{ \langle \text{dimen expression} \rangle$   
 $\quad \} \quad | \quad \backslash \text{real} \{ \langle \text{optional signs} \rangle \langle \text{decimal constant} \rangle \}$   
 $\langle \text{plus or minus} \rangle \quad \longrightarrow \quad +_{12} \quad | \quad -_{12}$   
 $\langle \text{decimal constant} \rangle \quad \longrightarrow \quad ._{12} \quad | \quad ,_{12} \quad | \quad \langle \text{digit} \rangle$   
 $\quad \langle \text{decimal constant} \rangle \quad | \quad \langle \text{decimal constant} \rangle \langle \text{digit} \rangle$   
 $\langle \text{digit} \rangle \quad \longrightarrow \quad 0_{12} \quad | \quad 1_{12} \quad | \quad 2_{12} \quad | \quad 3_{12} \quad | \quad 4_{12} \quad | \quad 5_{12}$   
 $\quad | \quad 6_{12} \quad | \quad 7_{12} \quad | \quad 8_{12} \quad | \quad 9_{12}$   
 $\langle \text{optional signs} \rangle \quad \longrightarrow \quad \langle \text{optional spaces} \rangle \quad | \quad \langle \text{optional signs} \rangle$   
 $\quad \langle \text{plus or minus} \rangle \langle \text{optional spaces} \rangle$

Relying heavily on TeX to do the underlying assignments, it is only natural for `calc` to simulate TeX's parsing machinery for these quantities. Therefore it a) imposes the same restrictions on the catcode of syntax characters as TeX and b) tries to expand its argument fully. a) means that implicit characters for the tokens  $*_{12}$ ,  $/_{12}$ ,  $(_{12}$ , and  $)_{12}$  will not work<sup>4</sup> but because of b), the expansion should allow you to use macros that expand to explicit syntax characters.

## 4 The evaluation scheme

In this section, we shall for simplicity consider only expressions containing ‘+’ (addition) and ‘\*’ (multiplication) operators. It is trivial to add subtraction and division.

An expression  $E$  is a sum of terms:  $T_1 + \dots + T_n$ ; a term is a product of factors:  $F_1 * \dots * F_m$ ; a factor is either a simple numeric quantity  $f$  (like  $\langle \text{number} \rangle$  as described in the TeXbook), or a parenthesized expression ( $E'$ ).

Since the TeX engine can only execute arithmetic operations in a machine-code like manner, we have to find a way to translate the infix notation into this ‘instruction set’.

<sup>4</sup>eTeX also assumes these catcodes when parsing a `\numexpr`, `\dimexpr`, `\glueexpr`, or `\muglueexpr` and does not allow implicit characters.

Our goal is to design a translation scheme that translates  $X$  (an expression, a term, or a factor) into a sequence of  $\text{\TeX}$  instructions that does the following [Invariance Property]: correctly evaluates  $X$ , leaves the result in a global register  $A$  (using a global assignment), and does not perform global assignments to the scratch register  $B$ ; moreover, the code sequence must be balanced with respect to  $\text{\TeX}$  groups. We shall denote the code sequence corresponding to  $X$  by  $\llbracket X \rrbracket$ .

In the replacement code specified below, we use the following conventions:

- $A$  and  $B$  denote registers; all assignments to  $A$  will be global, and all assignments to  $B$  will be local.
- “ $\Leftarrow$ ” means global assignment to the register on the lhs.
- “ $\leftarrow$ ” means local assignment to the register on the lhs.
- “ $\hookrightarrow_{[C]}$ ” means “save the code  $C$  until the current group (scope) ends, then execute it.” This corresponds to the  $\text{\TeX}$ -primitive `\aftergroup`.
- “{” denotes the start of a new group, and “}” denotes the end of a group.

Let us consider an expression  $T_1 + T_2 + \dots + T_n$ . Assuming that  $\llbracket T_k \rrbracket$  ( $1 \leq k \leq n$ ) attains the stated goal, the following code clearly attains the stated goal for their sum:

$$\begin{aligned} \llbracket T_1 + T_2 + \dots + T_n \rrbracket \implies & \{ \llbracket T_1 \rrbracket \} B \leftarrow A \quad \{ \llbracket T_2 \rrbracket \} B \leftarrow B + A \\ & \dots \quad \{ \llbracket T_n \rrbracket \} B \leftarrow B + A \quad A \Leftarrow B \end{aligned}$$

Note the extra level of grouping enclosing each of  $\llbracket T_1 \rrbracket$ ,  $\llbracket T_2 \rrbracket$ ,  $\dots$ ,  $\llbracket T_n \rrbracket$ . This will ensure that register  $B$ , used to compute the sum of the terms, is not clobbered by the intermediate computations of the individual terms. Actually, the group enclosing  $\llbracket T_1 \rrbracket$  is unnecessary, but it turns out to be simpler if all terms are treated the same way.

The code sequence “ $\{ \llbracket T_2 \rrbracket \} B \leftarrow B + A$ ” can be translated into the following equivalent code sequence: “ $\{ \hookrightarrow_{[B \leftarrow B + A]} \llbracket T_2 \rrbracket \}$ ”. This observation turns out to be the key to the implementation: The “ $\hookrightarrow_{[B \leftarrow B + A]}$ ” is generated *before*  $T_2$  is translated, at the same time as the ‘+’ operator between  $T_1$  and  $T_2$  is seen.

Now, the specification of the translation scheme is straightforward:

$$\begin{aligned} \llbracket f \rrbracket & \implies A \Leftarrow f \\ \llbracket (E') \rrbracket & \implies \llbracket E' \rrbracket \\ \llbracket T_1 + T_2 + \dots + T_n \rrbracket & \implies \{ \hookrightarrow_{[B \leftarrow A]} \llbracket T_1 \rrbracket \} \quad \{ \hookrightarrow_{[B \leftarrow B + A]} \llbracket T_2 \rrbracket \} \\ & \quad \dots \quad \{ \hookrightarrow_{[B \leftarrow B + A]} \llbracket T_n \rrbracket \} \quad A \Leftarrow B \\ \llbracket F_1 * F_2 * \dots * F_m \rrbracket & \implies \{ \hookrightarrow_{[B \leftarrow A]} \llbracket F_1 \rrbracket \} \quad \{ \hookrightarrow_{[B \leftarrow B * A]} \llbracket F_2 \rrbracket \} \\ & \quad \dots \quad \{ \hookrightarrow_{[B \leftarrow B * A]} \llbracket F_m \rrbracket \} \quad A \Leftarrow B \end{aligned}$$

By structural induction, it is easily seen that the stated property is attained.

By inspection of this translation scheme, we see that we have to generate the following code:

- we must generate “ $\{ \hookrightarrow_{[B \leftarrow A]} \{ \hookrightarrow_{[B \leftarrow A]} \}$ ” at the left border of an expression (i.e., for each left parenthesis and the implicit left parenthesis at the beginning of the whole expression);

- we must generate “ $\}A \leftarrow B\}A \leftarrow B$ ” at the right border of an expression (i.e., each right parenthesis and the implicit right parenthesis at the end of the full expression);
- ‘\*’ is replaced by “ $\}\{\hookrightarrow[B \leftarrow B * A]$ ”;
- ‘+’ is replaced by “ $\}A \leftarrow B\}\{\hookrightarrow[B \leftarrow B + A]\{\hookrightarrow[B \leftarrow A]$ ”;
- when we see (expect) a numeric quantity, we insert the assignment code “ $A \leftarrow$ ” in front of the quantity and let  $\text{\TeX}$  parse it.

## 5 Implementation

For brevity define

$$\langle \text{numeric} \rangle \longrightarrow \langle \text{number} \rangle \mid \langle \text{dimen} \rangle \mid \langle \text{glue} \rangle \mid \langle \text{muglue} \rangle$$

So far we have ignored the question of how to determine the type of register to be used in the code. However, it is easy to see that (1) ‘\*’ always initiates an  $\langle \text{integer factor} \rangle$ , (2) all  $\langle \text{numeric} \rangle$ s in an expression, except those which are part of an  $\langle \text{integer factor} \rangle$ , are of the same type as the whole expression, and all  $\langle \text{numeric} \rangle$ s in an  $\langle \text{integer factor} \rangle$  are  $\langle \text{number} \rangle$ s.

We have to ensure that  $A$  and  $B$  always have an appropriate type for the  $\langle \text{numeric} \rangle$ s they manipulate. We can achieve this by having an instance of  $A$  and  $B$  for each type. Initially,  $A$  and  $B$  refer to registers of the proper type for the whole expression. When an  $\langle \text{integer factor} \rangle$  is expected, we must change  $A$  and  $B$  to refer to integer type registers. We can accomplish this by including instructions to change the type of  $A$  and  $B$  to integer type as part of the replacement code for ‘\*’; if we append such instructions to the replacement code described above, we also ensure that the type-change is local (provided that the type-changing instructions only have local effect). However, note that the instance of  $A$  referred to in  $\hookrightarrow[B \leftarrow B * A]$  is the integer instance of  $A$ .

We shall use `\begingroup` and `\endgroup` for the open-group and close-group characters. This avoids problems with spacing in math (as pointed out to us by Frank Mittelbach).

### 5.1 Getting started

Now we have enough insight to do the actual implementation in  $\text{\TeX}$ . First, we announce the macro package.<sup>5</sup>

```
1 \*package
2 %\NeedsTeXFormat{LaTeX2e}
3 %\ProvidesPackage{calc}[\filedate\space\fileversion]
```

### 5.2 Assignment macros

`\calc@assign@generic` The `\calc@assign@generic` macro takes four arguments: (1 and 2) the registers to be used for global and local manipulations, respectively; (3) the lvalue part; (4) the expression to be evaluated.

---

<sup>5</sup>Code moved to top of file

The third argument (the lvalue) will be used as a prefix to a register that contains the value of the specified expression (the fourth argument).

In general, an lvalue is anything that may be followed by a variable of the appropriate type. As an example, `\linepenalty` and `\global\advance\linepenalty` may both be followed by an  $\langle$ integer variable $\rangle$ .

The macros described below refer to the registers by the names `\calc@A` and `\calc@B`; this is accomplished by `\let`-assignments.

As discovered in Section 4, we have to generate code as if the expression is parenthesized. As described below, `\calc@open` is the macro that replaces a left parenthesis by its corresponding T<sub>E</sub>X code sequence. When the scanning process sees the exclamation point, it generates an `\endgroup` and stops. As we recall from Section 4, the correct expansion of a right parenthesis is “ $\}A \Leftarrow B\}A \Leftarrow B$ ”. The remaining tokens of this expansion are inserted explicitly, except that the last assignment has been replaced by the lvalue part (i.e., argument #3 of `\calc@assign@generic`) followed by `\calc@B`.

```

4 \def\calc@assign@generic#1#2#3#4{\let\calc@A#1\let\calc@B#2%
5   \calc@open{#4!%
6   \global\calc@A\calc@B\endgroup#3\calc@B}

```

\calc@assign@count We need three instances of the \calc@assign@generic macro, corresponding to \calc@assign@dimen the types  $\langle\text{integer}\rangle$ ,  $\langle\text{dimen}\rangle$ , and  $\langle\text{glue}\rangle$ .

```
\calc@assign@skip 7 \def\calc@assign@count{\calc@assign@generic\calc@Account\calc@Bcount}
8 \def\calc@assign@dimen{\calc@assign@generic\calc@Adimen\calc@Bdimen}
9 \def\calc@assign@skip{\calc@assign@generic\calc@Askip\calc@Bskip}
```

These macros each refer to two registers, one to be used globally and one to be used locally. We must allocate these registers.

```

10 \newcount\calc@Account      \newcount\calc@Bcount
11 \newdimen\calc@Adimen       \newdimen\calc@Bdimen
12 \newskip\calc@Askip         \newskip\calc@Bskip

```

### 5.3 The L<sup>A</sup>T<sub>E</sub>X interface

`\setcounter` As promised, we redefine the following standard L<sup>A</sup>T<sub>E</sub>X commands: `\setcounter`,  
`\addtocounter` `\addtocounter`, `\setlength`, and `\addtolength`.

```

\steptocounter 13 \def\setcounter#1#2{\@ifundefined{c@#1}{\@nocounterr{#1}}%
\setlength 14 {\calc@assign@count{\global\curname c@#1\endcurname}{#2}}
\addtolength 15 \def\addtocounter#1#2{\@ifundefined{c@#1}{\@nocounterr{#1}}%
16 {\calc@assign@count{\global\advance\curname c@#1\endcurname}{#2}}}%

```

We also fix `\stepcounter` to not go through the whole `calc` process.

```

17 \def\stepcounter#1{\@ifundefined {c#1}%
18 {\@nocounterr {#1}}%
19 {\global\advance\csname c#1\endcsname \@ne
20 \begingroup
21 \let\elt\stpelet\csname cl#1\endcsname
22 \endgroup}}%

```

If the `amstext` package is loaded we must add the `\iffirstchoice@` switch as well. We patch the commands this way since it's good practice when we know how many arguments they take.



```

23 \@ifpackageloaded{amstext}{%
24   \expandafter\def\expandafter\stepcounter
25     \expandafter#\expandafter1\expandafter{%
26     \expandafter\iffirstchoice@\stepcounter{#1}\fi
27   }
28   \expandafter\def\expandafter\addtocounter
29     \expandafter#\expandafter1\expandafter#\expandafter2\expandafter{%
30     \expandafter\iffirstchoice@\addtocounter{#1}{#2}\fi
31   }
32 }{}

33 \DeclareRobustCommand\setlength{\calc@assign@skip}
34 \DeclareRobustCommand\addtolength[1]{\calc@assign@skip{\advance#1}}

```

(`\setlength` and `\addtolength` are robust according to [2].)

## 5.4 The scanner

We evaluate expressions by explicit scanning of characters. We do not rely on active characters for this.

The scanner consists of two parts, `\calc@pre@scan` and `\calc@post@scan`; `\calc@pre@scan` consumes left parentheses, and `\calc@post@scan` consumes binary operator, `\real`, `\ratio`, and right parenthesis tokens.

`\calc@pre@scan` Note that this is called at least once on every use of `calc` processing, even when `\@calc@pre@scan` none of the extended syntax is present; it therefore needs to be made very efficient.

It reads the initial part of expressions, until some `<text dimen factor>` or `<numeric>` is seen; in fact, anything not explicitly recognized here is taken to be a `<numeric>` of some sort as this allows unary `+` and unary `-` to be treated easily and correctly<sup>6</sup> but means that anything illegal will simply generate a T<sub>E</sub>X-level error, often a reasonably comprehensible one!

The `\romannumeral-‘\a` part is a little trick which forces expansion in case `#1` is a normal macro, something that occurs from time to time. A conditional test inside will possibly leave a trailing `\fi` but this remnant is removed later when `\calc@post@scan` performs the same trick.

The many `\expandafters` are needed to efficiently end the nested conditionals so that `\calc@textsize` and `\calc@maxmin@addsub` can process their argument.

```

35 \def\calc@pre@scan#1{%
36   \expandafter\@calc@pre@scan\romannumeral-‘\a#1}
37 \def\@calc@pre@scan#1{%
38   \ifx(#1%
39     \expandafter\calc@open
40   \else
41     \ifx\widthof#1%
42       \expandafter\expandafter\expandafter\calc@textsize
43     \else
44       \ifx\maxof#1%
45         \expandafter\expandafter\expandafter\expandafter
46         \expandafter\expandafter\expandafter\calc@maxmin@addsub
47       \else
48         \calc@numeric% no \expandafter needed for this one.

```

---

<sup>6</sup>In the few contexts where signs are allowed: this could, I think, be extended (CAR).

`\calc@open` `\calc@open` is used when there is a left parenthesis right ahead. This parenthesis `\calc@initB` is replaced by T<sub>E</sub>X code corresponding to the code sequence “ $\{\hookrightarrow_{[B\leftarrow A]}\{\hookrightarrow_{[B\leftarrow A]}\}$ ” derived in Section 4. Finally, `\calc@pre@scan` is called again.

`\calc@numeric` assigns the following value to `\calc@A` and then transfers control to `\calc@post@scan`.

|   |   |
|---|---|
| $\backslash\text{widthof}$<br>$\backslash\text{heightof}$<br>$\backslash\text{depthof}$<br>$\backslash\text{totalheightof}$ | <p>These do not need any particular definition when they are scanned so, for efficiency and robustness, we make them all equivalent to the same harmless (I hope) unexpandable command.<sup>7</sup> Thus the test in <code>\@calc@pre@scan</code> finds any of them.</p> <p>As we have to check for these commands explicitly we must ensure that our definition wins. Using <code>\newcommand</code> gives an error when loading <code>calc</code> and may be mildly surprising. This should be a little more informative.</p> |
|---|---|

`\calc@textsize` The presence of the above four commands invokes this code, where we must distinguish them from each other. This implementation is somewhat optimized by using low-level code from the commands `\settowidth`, etc.<sup>8</sup>

```

75 \def\calc@textsize #1#2{%
76   \begingroup
77   \let\widthof\wd

```

<sup>8</sup>It is based on suggestions by Donald Arseneau and David Carlisle.

```

78 \let\heightof\ht
79 \let\depthof\dp
80 \def\totalheightof{\ht\dp}%

```

We must expand the argument one level if it's `\totalheightof` and it doesn't hurt the other three.

```

81 \expandafter\@settodim\expandafter{#1}%
82 {\global\calc@A}%
83 {%
84 \let\widthof\ignorespaces
85 \let\heightof\ignorespaces
86 \let\depthof\ignorespaces
87 \let\totalheightof\ignorespaces
88 #2}%
89 \endgroup
90 \calc@post@scan}

```

`\calc@post@scan` The macro `\calc@post@scan` is called right after a value has been read. At this point, a binary operator, a sequence of right parentheses, an optional `\relax`, and the end-of-expression mark (`'!`) is allowed.<sup>9</sup> Depending on our findings, we call a suitable macro to generate the corresponding  $\text{\TeX}$  code (except when we detect the end-of-expression marker: then scanning ends, and control is returned to `\calc@assign@generic`).

This macro may be optimized by selecting a different order of `\ifx`-tests. The test for `'!` (end-of-expression) is placed first as it will always be performed: this is the only test to be performed if the expression consists of a single  $\langle$ numeric $\rangle$ . This ensures that documents that do not use the extra expressive power provided by the `calc` package only suffer a minimum slowdown in processing time.

```

91 \def\calc@post@scan#1{%
92 \expandafter\@calc@post@scan\romannumeral-'\a#1}
93 \def\@calc@post@scan#1{%
94 \ifx#1!\let\calc@next\endgroup \else
95 \ifx#1+\let\calc@next\calc@add \else
96 \ifx#1-\let\calc@next\calc@subtract \else
97 \ifx#1*\let\calc@next\calc@multiply \else
98 \ifx#1/\let\calc@next\calc@dividex \else
99 \ifx#1)\let\calc@next\calc@close \else
100 \ifx#1\relax\let\calc@next\calc@post@scan \else
101 \def\calc@next{\calc@error#1}%
102 \fi
103 \fi
104 \fi
105 \fi
106 \fi
107 \fi
108 \fi
109 \calc@next}

```

`\calc@add` The replacement code for the binary operators `+` and `-` follow a common pattern; `\calc@subtract` the only difference is the token that is stored away by `\aftergroup`. After this `\calc@generic@add` replacement code, control is transferred to `\calc@pre@scan`.

`\calc@addAtoB`

`\calc@subtractAfromB` <sup>9</sup>Is ! a good choice, CAR?

```

110 \def\calc@add{\calc@generic@add\calc@addAtoB}
111 \def\calc@subtract{\calc@generic@add\calc@subtractAfromB}
112 \def\calc@generic@add#1{\endgroup\global\calc@A\calc@B\endgroup
113   \begingroup\aftergroup#1\begingroup\aftergroup\calc@initB
114   \calc@pre@scan}
115 \def\calc@addAtoB{\advance\calc@B\calc@A}
116 \def\calc@subtractAfromB{\advance\calc@B-\calc@A}

```

\real The multiplicative operators, ‘\*’ and ‘/’, may be followed by a \real, \ratio, \ratio \minof, or \maxof token. The last two of these control sequences are defined by calc as they are needed by the scanner for addition or subtraction while the \calc@ratio@x first two are not defined (at least not by the calc package); this, unfortunately, leaves them highly non-robust. We therefore equate them to \relax but only if they have not already been defined<sup>10</sup> (by some other package: dangerous but possible!); this will also make them appear to be undefined to a L<sup>A</sup>T<sub>E</sub>X user (also possibly dangerous).

```

117 \ifx\real@undefined\let\real\relax\fi
118 \ifx\ratio@undefined\let\ratio\relax\fi

```

In order to test for \real or \ratio, we define these two.<sup>11</sup>

```

119 \def\calc@ratio@x{\ratio}
120 \def\calc@real@x{\real}

```

\calc@multiplyx Test which operator followed \* or /. If none followed it’s just a standard multiplication or division.

```

121 \def\calc@multiplyx#1{\def\calc@tmp{#1}%
122   \ifx\calc@tmp\calc@ratio@x \let\calc@next\calc@ratio@multiply \else
123   \ifx\calc@tmp\calc@real@x \let\calc@next\calc@real@multiply \else
124   \ifx\maxof#1\let\calc@next\calc@maxmin@multiply \else
125   \let\calc@next\calc@multiply
126   \fi
127   \fi
128   \fi
129   \calc@next#1}
130 \def\calc@dividex#1{\def\calc@tmp{#1}%
131   \ifx\calc@tmp\calc@ratio@x \let\calc@next\calc@ratio@divide \else
132   \ifx\calc@tmp\calc@real@x \let\calc@next\calc@real@divide \else
133   \ifx\maxof#1\let\calc@next\calc@maxmin@divide \else
134   \let\calc@next\calc@divide
135   \fi
136   \fi
137   \fi
138   \calc@next#1}

```

\calc@multiply The binary operators ‘\*’ and ‘/’ also insert code as determined above. Moreover, the meaning of \calc@A and \calc@B is changed as factors following a multiplication and division operator always have integer type; the original meaning of these macros will be restored when the factor has been read and evaluated.

```

\calc@multiplyBbyA 139 \def\calc@multiply{\calc@generic@multiply\calc@multiplyBbyA}
\calc@divideBbyA 140 \def\calc@divide{\calc@generic@multiply\calc@divideBbyA}
141 \def\calc@generic@multiply#1{\endgroup\begingroup

```

<sup>10</sup>Suggested code from David Carlisle.

<sup>11</sup>May not need the extra names, CAR?

```

142 \let\calc@A\calc@Account \let\calc@B\calc@Bcount
143 \aftergroup#1\calc@pre@scan}
144 \def\calc@multiplyBbyA{\multiply\calc@B\calc@Account}
145 \def\calc@divideBbyA{\divide\calc@B\calc@Account}

```

Since the value to use in the multiplication/division operation is stored in the `\calc@Account` register, the `\calc@multiplyBbyA` and `\calc@divideBbyA` macros use this register.

`\calc@close` `\calc@close` generates code for a right parenthesis (which was derived to be “ $A \leftarrow B$ ” $A \leftarrow B$ ” in Section 4). After this code, the control is returned to `\calc@post@scan` in order to look for another right parenthesis or a binary operator.

```

146 \def\calc@close
147   {\endgroup\global\calc@A\calc@B
148   \endgroup\global\calc@A\calc@B
149   \calc@post@scan}

```

## 5.5 Calculating a ratio

`\calc@ratio@multiply` When `\calc@post@scan` encounters a `\ratio` control sequence, it hands control to one of the macros `\calc@ratio@multiply` or `\calc@ratio@divide`, depending on the preceding character. Those macros both forward the control to the macro `\calc@ratio@evaluate`, which performs two steps: (1) it calculates the ratio, which is saved in the global macro token `\calc@the@ratio`; (2) it makes sure that the value of `\calc@B` will be multiplied by the ratio as soon as the current group ends.

The following macros call `\calc@ratio@evaluate` which multiplies `\calc@B` by the ratio, but `\calc@ratio@divide` flips the arguments so that the ‘opposite’ fraction is actually evaluated.

```

150 \def\calc@ratio@multiply\ratio{\calc@ratio@evaluate}
151 \def\calc@ratio@divide\ratio#1#2{\calc@ratio@evaluate{#2}{#1}}

```

`\calc@Ccount` We shall need two registers for temporary usage in the calculations. We can save `\calc@numerator` one register since we can reuse `\calc@Bcount`.

```

\calc@denominator 152 \newcount\calc@Ccount
153 \let\calc@numerator=\calc@Bcount
154 \let\calc@denominator=\calc@Ccount

```

`\calc@ratio@evaluate` Here is the macro that handles the actual evaluation of ratios. The procedure is this: First, the two expressions are evaluated and coerced to integers. The whole procedure is enclosed in a group to be able to use the registers `\calc@numerator` and `\calc@denominator` for temporary manipulations.

```

155 \def\calc@ratio@evaluate#1#2{%
156   \endgroup\begin{group}
157     \calc@assign@dimen\calc@numerator{#1}%
158     \calc@assign@dimen\calc@denominator{#2}%

```

Here we calculate the ratio. First, we check for negative numerator and/or denominator; note that  $\text{\TeX}$  interprets two minus signs the same as a plus sign. Then, we calculate the integer part. The minus sign(s), the integer part, and a decimal point, form the initial expansion of the `\calc@the@ratio` macro.

```

159 \gdef\calc@the@ratio{}%
160 \ifnum\calc@numerator<0 \calc@numerator-\calc@numerator
161 \gdef\calc@the@ratio{-}%
162 \fi
163 \ifnum\calc@denominator<0 \calc@denominator-\calc@denominator
164 \xdef\calc@the@ratio{\calc@the@ratio}%
165 \fi
166 \calc@Acount\calc@numerator
167 \divide\calc@Acount\calc@denominator
168 \xdef\calc@the@ratio{\calc@the@ratio\number\calc@Acount.}%

```

Now we generate the digits after the decimal point, one at a time. When  $\text{\TeX}$  scans these digits (in the actual multiplication operation), it forms a fixed-point number with 16 bits for the fractional part. We hope that six digits is sufficient, even though the last digit may not be rounded correctly.

```

169 \calc@next@digit \calc@next@digit \calc@next@digit
170 \calc@next@digit \calc@next@digit \calc@next@digit
171 \endgroup

```

Now we have the ratio represented (as the expansion of the global macro  $\backslash\text{calc@the@ratio}$ ) in the syntax  $\langle\text{decimal constant}\rangle$  [1, page 270]. This is fed to  $\backslash\text{calc@multiply@by@real}$  that will perform the actual multiplication. It is important that the multiplication takes place at the correct grouping level so that the correct instance of the  $B$  register will be used. Also note that we do not need the  $\backslash\text{aftergroup}$  mechanism in this case.

```

172 \calc@multiply@by@real\calc@the@ratio
173 \begingroup
174 \calc@post@scan}

```

The  $\backslash\text{begingroup}$  inserted before the  $\backslash\text{calc@post@scan}$  will be matched by the  $\backslash\text{endgroup}$  generated as part of the replacement of a subsequent binary operator or right parenthesis.

$\backslash\text{calc@next@digit}$

```

175 \def\calc@next@digit{%
176 \multiply\calc@Acount\calc@denominator
177 \advance\calc@numerator -\calc@Acount
178 \multiply\calc@numerator 10
179 \calc@Acount\calc@numerator
180 \divide\calc@Acount\calc@denominator
181 \xdef\calc@the@ratio{\calc@the@ratio\number\calc@Acount}}

```

$\backslash\text{calc@multiply@by@real}$  In the following code, it is important that we first assign the result to a  $\text{dimen}$  register. Otherwise,  $\text{\TeX}$  won't allow us to multiply with a real number.

```

182 \def\calc@multiply@by@real#1{\calc@Bdimen #1\calc@B \calc@B\calc@Bdimen}

```

(Note that this code wouldn't work if  $\backslash\text{calc@B}$  were a  $\text{muglue}$  register. This is the real reason why the  $\text{calc}$  package doesn't support  $\text{muglue}$  expressions. To support  $\text{muglue}$  expressions in full, the  $\backslash\text{calc@multiply@by@real}$  macro must use a  $\text{muglue}$  register instead of  $\backslash\text{calc@Bdimen}$  when  $\backslash\text{calc@B}$  is a  $\text{muglue}$  register; otherwise, a  $\text{dimen}$  register should be used. Since integer expressions can appear as part of a  $\text{muglue}$  expression, it would be necessary to determine the correct register to use each time a multiplication is made.)

## 5.6 Multiplication by real numbers

`\calc@real@multiply` This is similar to the `\calc@ratio@evaluate` macro above, except that it is considerably simplified since we don't need to calculate the factor explicitly.

```

183 \def\calc@real@multiply\real#1{\endgroup
184   \calc@multiply@by@real{#1}\begingroup
185   \calc@post@scan}
186 \def\calc@real@divide\real#1{\calc@ratio@evaluate{1pt}{#1pt}}
```

## 5.7 max and min operations

`\maxof` With version 4.2, the max and min operators were added to `calc`. The user functions for them are `\maxof` and `\minof` respectively. These macros are internally similar to `\widthof` etc. in that they are unexpandable and easily recognizable by the scanner.

```

187 \let\maxof\@@italiccorr
188 \let\minof\@@italiccorr
```

`\calc@Cskip` The max and min operations take two arguments so we need an extra `<skip>` register. We also add a switch for determining when to perform a `<skip>` or a `<count>` assignment.

```

189 \newskip\calc@Cskip
190 \newif\ifcalc@count@
```

`\calc@maxmin@addsub` When doing addition or subtraction with a max or min operator, we first check if `\calc@A` is a `<count>` register or not and then set the switch. Then call the real function which sets `\calc@A` to the desired value and continue as usual with `\calc@post@scan`.

```

191 \def\calc@maxmin@addsub#1#2#3{\begingroup
192   \ifx\calc@A\calc@Acount%
193     \calc@count@true
194   \else
195     \calc@count@false
196   \fi
197   \calc@maxmin@generic#1{#2}{#3}%
198   \endgroup
199   \calc@post@scan
200 }
```

Check the switch and do either `<count>` or `<skip>` assignments. Note that `\maxof` and `\minof` are not set to `>` and `<` until after the assignments, which ensures we can nest them without problems. Then set `\calc@A` to the correct one.

```

201 \def\calc@maxmin@generic#1#2#3{%
202   \begingroup
203   \ifcalc@count@
204     \calc@assign@count\calc@Ccount{#2}%
205     \calc@assign@count\calc@Bcount{#3}%
206     \def\minof{<}\def\maxof{>}%
207     \global\calc@A\ifnum\calc@Ccount#1\calc@Bcount
208       \calc@Ccount\else\calc@Bcount\fi
209   \else
210     \calc@assign@skip\calc@Cskip{#2}%
211     \calc@assign@skip\calc@Bskip{#3}%
212   \fi
213 }
```

```

212     \def\minof{<}\def\maxof{>}%
213     \global\calc@A\ifdim\calc@Cskip#1\calc@Bskip
214         \calc@Cskip\else\calc@Bskip\fi
215     \fi
216 \endgroup
217 }

```

`\calc@maxmin@divmul` When doing division or multiplication we must be using `<count>` registers so we  
`\calc@maxmin@multiply` set the switch. Other than that it is almost business as usual when multiplying or  
`\calc@maxmin@divide` dividing. #1 is the instruction to either multiply or divide `\calc@B` by `\calc@A`,  
#2 is either `\maxof` or `\minof` which is waiting in the input stream and #3 and #4  
are the calc expressions. We end it all as usual by calling `\calc@post@scan`.

```

218 \def\calc@maxmin@divmul#1#2#3#4{%
219     \endgroup\begin group
220     \calc@count@true
221     \aftergroup#1%
222     \calc@maxmin@generic#2{#3}{#4}%
223     \endgroup\begin group
224     \calc@post@scan
225 }

```

The two functions called when seeing a `*` or a `/`.

```

226 \def\calc@maxmin@multiply{\calc@maxmin@divmul\calc@multiplyBbyA}
227 \def\calc@maxmin@divide  {\calc@maxmin@divmul\calc@divideBbyA}

```

## 6 Reporting errors

`\calc@error` If `\calc@post@scan` reads a character that is not one of `'+'`, `'-'`, `'*'`, `'/'`, or `)'`, an  
error has occurred, and this is reported to the user. Violations in the syntax of  
`<numeric>`s will be detected and reported by `TEX`.

```

228 \def\calc@error#1{%
229     \PackageError{calc}%
230     {'#1' invalid at this point}%
231     {I expected to see one of: + - * / )}}

```

## 7 Other additions

`\@settodim` The kernel macro `\@settodim` is changed so that it runs through a list containing  
`\setttotalheight` `\ht`, `\wd`, and `\dp` and then advance the length one step at a time. We just have  
to use a scratch register in case the user decides to put in a `\global` prefix on the  
length register. A search on the internet confirmed that some people do that kind  
of thing.

```

232 \def\@settodim#1#2#3{%
233     \setbox\@tempboxa\hbox{#{#3}}%
234     \dimen@ii=\z@
235     \@tf@r\reserved@a #1\do{%
236     \advance\dimen@ii\reserved@a\@tempboxa}%
237     #2=\dimen@ii
238     \setbox\@tempboxa\box\voidb@x}

```

Now the user level macro is straightforward.

```

239 \def\setttotalheight{\@settodim{\ht\dp}}

```



```
240 </package>
```