

# The `calc` package

## Infix notation arithmetic in L<sup>A</sup>T<sub>E</sub>X\*

Kresten Krab Thorup, Frank Jensen (and Chris Rowley)

1998/07/07

### Abstract

The `calc` package reimplements the L<sup>A</sup>T<sub>E</sub>X commands `\setcounter`, `\addtocounter`, `\setlength`, and `\addtolength`. Instead of a simple value, these commands now accept an infix notation expression.

## 1 Introduction

Arithmetic in T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X.

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 L<sup>A</sup>T<sub>E</sub>X 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)

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

`\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 `tbluar` 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;  $\TeX$  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 `\settoheight` etc. on Page 216 of the manual [2].

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

```
\setlength{⟨parskip⟩}{0.68⟨textwd⟩}
```

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}}
```

$\TeX$  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  $\TeX$  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}}
```

---

<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.

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}}
```

### 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 `TEXbook` [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>

$$\begin{aligned} \langle \textit{type} \text{ expression} \rangle &\longrightarrow \langle \textit{type} \text{ term} \rangle \\ &\quad | \langle \textit{type} \text{ expression} \rangle \langle \textit{plus or minus} \rangle \langle \textit{type} \text{ term} \rangle \\ \langle \textit{type} \text{ term} \rangle &\longrightarrow \langle \textit{type} \text{ factor} \rangle \\ &\quad | \langle \textit{type} \text{ term} \rangle \langle \textit{multiply or divide} \rangle \langle \textit{integer factor} \rangle \\ &\quad | \langle \textit{type} \text{ term} \rangle \langle \textit{multiply or divide} \rangle \langle \textit{real number} \rangle \\ \langle \textit{type} \text{ factor} \rangle &\longrightarrow \langle \textit{type} \rangle \quad | \langle \textit{text dimen factor} \rangle \quad | \langle \textit{_{12}type expression} \rangle_{12} \\ \langle \textit{integer} \rangle &\longrightarrow \langle \textit{number} \rangle \\ \langle \textit{text dimen factor} \rangle &\longrightarrow \langle \textit{text dimen command} \rangle \{ \langle \textit{text} \rangle \} \\ \langle \textit{text dimen command} \rangle &\longrightarrow \backslash \textit{widthof} \quad | \quad \backslash \textit{heightof} \quad | \quad \backslash \textit{depthof} \\ \langle \textit{multiply or divide} \rangle &\longrightarrow *_{12} \quad | \quad /_{12} \\ \langle \textit{real number} \rangle &\longrightarrow \backslash \textit{ratio} \{ \langle \textit{dimen expression} \rangle \} \{ \langle \textit{dimen expression} \rangle \} \\ &\quad | \quad \backslash \textit{real} \{ \langle \textit{decimal constant} \rangle \} \end{aligned}$$

Note that during most of the parsing of `calc` expressions, no expansion happens; thus the above syntax must be explicit<sup>4</sup>.

### 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 `<number>`) as described in the `TEXbook`, or a parenthesized expression ( $E'$ ).

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

<sup>4</sup>Two exceptions to this are: the first token is expanded one-level (thus the whole expression can be put into a macro); wherever a `<decimal constant>` or `<type>` is expected.

Since the  $\text{\TeX}$  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’.

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:

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

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  $\backslash\text{begingroup}$  and  $\backslash\text{endgroup}$  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  $\TeX$ . 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.

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  $\TeX$  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   \expandafter\calc@open\expandafter{#4!%
6   \global\calc@A\calc@B\endgroup#3\calc@B}
```

(The `\expandafter` tokens allow the user to use expressions stored one-level deep in a macro as arguments in assignment commands.)

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

```
7 \def\calc@assign@count{\calc@assign@generic\calc@Acount\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@Acount   \newcount\calc@Bcount
11 \newdimen\calc@Adimen   \newdimen\calc@Bdimen
12 \newskip\calc@Askip     \newskip\calc@Bskip
```

---

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

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

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

```
13 \def\setcounter#1#2{\@ifundefined{c@#1}{\@nocounterr{#1}}%
14   {\calc@assign@count{\global\csname c@#1\endcsname}{#2}}
15 \def\addtocounter#1#2{\@ifundefined{c@#1}{\@nocounterr{#1}}%
16   {\calc@assign@count{\global\advance\csname c@#1\endcsname}{#2}}}
17 \DeclareRobustCommand\setlength{\calc@assign@skip}
18 \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 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 many `\expandafters` are needed to efficiently end the nested conditionals so that `\calc@textsize` can process its argument.

```
19 \def\calc@pre@scan#1{%
20   \ifx(#1%
21     \expandafter\calc@open
22   \else
23     \ifx\widthof#1%
24       \expandafter\expandafter\expandafter\calc@textsize
25     \else
26       \calc@numeric% no \expandafter needed for this one.
27     \fi
28   \fi
29   #1}
```

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

```
30 \def\calc@open({\begingroup\aftergroup\calc@initB
31   \begingroup\aftergroup\calc@initB
```

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

```

32 \calc@pre@scan}
33 \def\calc@initB{\calc@B\calc@A}
\calc@numeric assigns the following value to \calc@A and then transfers control
to \calc@post@scan.
34 \def\calc@numeric{\afterassignment\calc@post@scan \global\calc@A}

```

`\widthof` 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 `\calc@pre@scan` finds any of them. They are first defined using `\newcommand` so that they appear to be normal user commands to a L<sup>A</sup>T<sub>E</sub>X user<sup>8</sup>.

```

35 \newcommand\widthof{}
36 \let\widthof\ignorespaces
37 \newcommand\heightof{}
38 \let\heightof\ignorespaces
39 \newcommand\depthof{}
40 \let\depthof\ignorespaces

```

`\calc@textsize` The presence of the above three 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>9</sup>.

Within the text argument we must restore the normal meanings of the three user-level commands since arbitrary material can appear in here, including further uses of `calc`.

```

41 \def\calc@textsize #1#2{%
42 \begingroup
43 \let\widthof\wd
44 \let\heightof\ht
45 \let\depthof\dp
46 \@settodim #1%
47 {\global\calc@A}%
48 {%
49 \let\widthof\ignorespaces
50 \let\heightof\ignorespaces
51 \let\depthof\ignorespaces
52 #2}%
53 \endgroup
54 \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, and the end-of-expression mark (`'!`) is allowed<sup>10</sup>. Depending on our findings, we call a suitable macro to generate the corresponding T<sub>E</sub>X code (except when we detect the end-of-expression marker: then scanning ends, and control is returned to `\calc@assign@generic`).

<sup>7</sup>If this level of safety is not needed then the code can be speeded up: CAR.

<sup>8</sup>Is this necessary, CAR?

<sup>9</sup>It is based on suggestions by Donald Arsenau and David Carlisle.

<sup>10</sup>Is ! a good choice, CAR?

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.

```

55 \def\calc@post@scan#1{%
56   \ifx#!\let\calc@next\endgroup \else
57   \ifx#\let\calc@next\calc@add \else
58   \ifx#-\let\calc@next\calc@subtract \else
59   \ifx#\let\calc@next\calc@multiply \else
60   \ifx#/\let\calc@next\calc@dividex \else
61   \ifx#)\let\calc@next\calc@close \else \calc@error#1%
62   \fi
63   \fi
64   \fi
65   \fi
66   \fi
67   \fi
68   \calc@next}

```

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

```

69 \def\calc@add{\calc@generic@add\calc@addAtoB}
70 \def\calc@subtract{\calc@generic@add\calc@subtractAfromB}
71 \def\calc@generic@add#1{\endgroup\global\calc@A\calc@B\endgroup
72  \begingroup\aftergroup#1\begingroup\aftergroup\calc@initB
73  \calc@pre@scan}
74 \def\calc@addAtoB{\advance\calc@B\calc@A}
75 \def\calc@subtractAfromB{\advance\calc@B-\calc@A}

```

`\real` The multiplicative operators, ‘\*’ and ‘/’, may be followed by a `\real` or a `\ratio` token. Those control sequences 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>11</sup> (by some other package: dangerous but possible!); this will also make them appear to be undefined to a  $\text{\LaTeX}$  user (also possibly dangerous).

```

76 \ifx\real\@undefined\let\real\relax\fi
77 \ifx\ratio\@undefined\let\ratio\relax\fi

```

In order to test for them, we define these two<sup>12</sup>.

```

78 \def\calc@ratio@x{\ratio}
79 \def\calc@real@x{\real}

```

```

80 \def\calc@multiply#1{\def\calc@tmp{#1}%
81  \ifx\calc@tmp\calc@ratio@x \let\calc@next\calc@ratio@multiply \else

```

<sup>11</sup>Suggested code from David Carlisle.

<sup>12</sup>May not need the extra names, CAR?

```

82     \ifx\calc@tmp\calc@real@x \let\calc@next\calc@real@multiply \else
83         \let\calc@next\calc@multiply
84     \fi
85 \fi
86 \calc@next#1}
87 \def\calc@dividex#1{\def\calc@tmp{#1}%
88     \ifx\calc@tmp\calc@ratio@x \let\calc@next\calc@ratio@divide \else
89         \ifx\calc@tmp\calc@real@x \let\calc@next\calc@real@divide \else
90             \let\calc@next\calc@divide
91         \fi
92     \fi
93     \calc@next#1}

```

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.

```

94 \def\calc@multiply{\calc@generic@multiply\calc@multiplyBbyA}
95 \def\calc@divide{\calc@generic@multiply\calc@divideBbyA}
96 \def\calc@generic@multiply#1{\endgroup\beginingroup
97     \let\calc@A\calc@Account \let\calc@B\calc@Bcount
98     \aftergroup#1\calc@pre@scan}
99 \def\calc@multiplyBbyA{\multiply\calc@B\calc@Account}
100 \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` 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.

```

101 \def\calc@close
102     {\endgroup\global\calc@A\calc@B
103     \endgroup\global\calc@A\calc@B
104     \calc@post@scan}

```

## 5.5 Calculating a ratio

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.

```
105 \def\calc@ratio@multiply\ratio{\calc@ratio@evaluate}
106 \def\calc@ratio@divide\ratio#1#2{\calc@ratio@evaluate{#2}{#1}}
```

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

```
107 \let\calc@numerator=\calc@Bcount
108 \newcount\calc@denominator
```

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.

```
109 \def\calc@ratio@evaluate#1#2{%
110   \endgroup\begingroup
111     \calc@assign@dimen\calc@numerator{#1}%
112     \calc@assign@dimen\calc@denominator{#2}%
```

Here we calculate the ratio. First, we check for negative numerator and/or denominator; note that  $\TeX$  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.

```
113   \gdef\calc@the@ratio{}%
114   \ifnum\calc@numerator<0 \calc@numerator-\calc@numerator
115     \gdef\calc@the@ratio{-}%
116   \fi
117   \ifnum\calc@denominator<0 \calc@denominator-\calc@denominator
118     \xdef\calc@the@ratio{\calc@the@ratio-}%
119   \fi
120   \calc@Acount\calc@numerator
121   \divide\calc@Acount\calc@denominator
122   \xdef\calc@the@ratio{\calc@the@ratio\number\calc@Acount.}%
```

Now we generate the digits after the decimal point, one at a time. When  $\TeX$  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.

```
123     \calc@next@digit \calc@next@digit \calc@next@digit
124     \calc@next@digit \calc@next@digit \calc@next@digit
125   \endgroup
```

Now we have the ratio represented (as the expansion of the global macro `\calc@the@ratio`) in the syntax `\langle decimal constant \rangle` [1, page 270]. This is fed to `\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 `\aftergroup` mechanism in this case.

```
126   \calc@multiply@by@real\calc@the@ratio
127   \begingroup
128   \calc@post@scan}
```

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

```

129 \def\calc@next@digit{%
130     \multiply\calc@Acount\calc@denominator
131     \advance\calc@numerator -\calc@Acount
132     \multiply\calc@numerator 10
133     \calc@Acount\calc@numerator
134     \divide\calc@Acount\calc@denominator
135     \xdef\calc@the@ratio{\calc@the@ratio\number\calc@Acount}}

```

In the following code, it is important that we first assign the result to a `dimen` register. Otherwise, `TeX` won't allow us to multiply with a real number.

```

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

```

(Note that this code wouldn't work if `\calc@B` were a `muglue` register. This is the real reason why the `calc` package doesn't support `muglue` expressions. To support `muglue` expressions in full, the `\calc@multiply@by@real` macro must use a `muglue` register instead of `\calc@Bdimen` when `\calc@B` is a `muglue` register; otherwise, a `dimen` register should be used. Since integer expressions can appear as part of a `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

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.

```

137 \def\calc@real@multiply\real#1{\endgroup
138     \calc@multiply@by@real{#1}\begingroup
139     \calc@post@scan}
140 \def\calc@real@divide\real#1{\calc@ratio@evaluate{1pt}{#1pt}}

```

## 6 Reporting errors

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`.

```

141 \def\calc@error#1{%
142     \PackageError{calc}%
143     {'#1' invalid at this point}%
144     {I expected to see one of: + - * / )}}
145 </package>

```

## References

- [1] D. E. KNUTH. *The TeXbook* (Computers & Typesetting Volume A). Addison-Wesley, Reading, Massachusetts, 1986.

- [2] L. LAMPORT. *L<sup>A</sup>T<sub>E</sub>X, A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, Second edition 1994/1985.