The calc package: Infix notation arithmetic in LaTeX*

Kresten Krab Thorup   Frank Jensen

1995/04/10

Abstract

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

1 Introduction

Arithmetic in 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 LaTeX.

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

2 Informal description

Standard 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)

\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)

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

(The `\setcounter` and `\addtocounter` commands have global effect, while the `\setlength` and `\addtolength` commands obey the normal scoping rules.) In standard LaTeX, 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 ⟨integer expression⟩, and *len* is replaced by ⟨glue expression⟩. The formal syntax of ⟨integer expression⟩ and ⟨glue expression⟩ is given below.

In the following, we shall use standard TeX terminology. The correspondence between TeX and LaTeX terminology is as follows: LaTeX counters correspond to TeX's count registers; they hold quantities of type ⟨number⟩. LaTeX length commands correspond to TeX's dimen (for rigid lengths) and skip (for rubber lengths) registers; they hold quantities of types ⟨dimen⟩ and ⟨glue⟩, respectively.

TeX 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 TeX primitives but provides a more user-friendly notation for expressing the arithmetic.

An expression is formed of numerical quantitites (such as explicit constants and LaTeX 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⟩`}`[1] 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.

[1]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.

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}}
```

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 defined in Chapter 24 of The TEXbook [1]. We use *type* as a meta-veriable, standing for 'integer', 'dimen', and 'glue'.[2]

⟨*type* expression⟩ ⟶ ⟨*type* term⟩
    | ⟨*type* expression⟩⟨plus or minus⟩⟨*type* term⟩

⟨*type* term⟩ ⟶ ⟨*type* factor⟩
    | ⟨*type* term⟩⟨multiply or divide⟩⟨integer factor⟩
    | ⟨*type* term⟩⟨multiply or divide⟩⟨real number⟩

⟨*type* factor⟩ ⟶ ⟨*type*⟩ | $(_{12}$⟨*type* expression⟩$)_{12}$

⟨integer⟩ ⟶ ⟨number⟩

⟨multiply or divide⟩ ⟶ $*_{12}$ | $/_{12}$

⟨real number⟩ ⟶ `\ratio{`⟨dimen expression⟩`}{`⟨dimen expression⟩`}`
    | `\real{`⟨decimal constant⟩`}`

[2] This version of the `calc` package doesn't support evaluation of muglue expressions.

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: $T_1 + \cdots + T_n$; a term is a product of factors: $F_1 * \cdots * F_m$; a factor is either a simple numeric quantity f (like $\langle$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'.

Our goal is to design a translation scheme that translates X (an expression, a term, or a factor) into a sequence of 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 TeX groups. We shall denote the code sequence corresponding to X by $[\![X]\!]$.

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 TeX-primitive `\aftergroup`.
- "{" denotes the start of a new group, and "}" denotes the end of a group.

Let us consider an expression $T_1+T_2+\cdots+T_n$. Assuming that $[\![T_k]\!]$ $(1 \le k \le n)$ attains the stated goal, the following code clearly attains the stated goal for their sum:

$$\begin{aligned}[\![T_1 + T_2 + \cdots + T_n]\!] \quad \Longrightarrow \quad & \{\,[\![T_1]\!]\,\}\ B \leftarrow A \quad \{\,[\![T_2]\!]\,\}\ B \leftarrow B + A \\ & \ldots \quad \{\,[\![T_n]\!]\,\}\ B \leftarrow B + A \quad A \Leftarrow B\end{aligned}$$

Note the extra level of grouping enclosing each of $[\![T_1]\!]$, $[\![T_2]\!]$, $\ldots$, $[\![T_n]\!]$. 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 $[\![T_1]\!]$ is unnecessary, but it turns out to be simpler if all terms are treated the same way.

The code sequence "$\{\,[\![T_2]\!]\,\}\ B \leftarrow B + A$" can be translated into the following equivalent code sequence: "$\{\hookrightarrow_{[B\leftarrow B+A]} [\![T_2]\!]\,\}$". 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:

$$\begin{aligned}[\![f]\!] \quad &\Longrightarrow \quad A \Leftarrow f \\ [\![(E')]\!] \quad &\Longrightarrow \quad [\![E']\!] \\ [\![F_1 * F_2 * \cdots * F_m]\!] \quad &\Longrightarrow \quad \{\hookrightarrow_{[B\leftarrow A]} [\![F_1]\!]\,\} \quad \{\hookrightarrow_{[B\leftarrow B*A]} [\![F_2]\!]\,\}\end{aligned}$$

$$\begin{array}{rcl} & & \quad\cdots\quad \{\hookrightarrow_{[B\leftarrow B*A]} [\![F_m]\!] \,\} \quad A \Leftarrow B \\ [\![T_1 + T_2 + \cdots + T_n]\!] & \Longrightarrow & \{\hookrightarrow_{[B\leftarrow A]} [\![T_1]\!] \,\} \quad \{\hookrightarrow_{[B\leftarrow B+A]} [\![T_2]\!] \,\} \\ & & \quad\cdots\quad \{\hookrightarrow_{[B\leftarrow B+A]} [\![T_n]\!] \,\} \quad A \Leftarrow B \end{array}$$

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 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 ⟨integer factor⟩, (2) all ⟨numeric⟩s in an expression, except those which are part of an ⟨integer factor⟩, are of the same type as the whole expression, and all ⟨numeric⟩s in an ⟨integer factor⟩ are ⟨number⟩s.

We have to ensure that A and B always have an appropriate type for the ⟨numeric⟩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 ⟨integer factor⟩ 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 TEX. First, we announce the macro package.

```
⟨*package⟩
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{calc}[\filedate\space\fileversion]
\typeout{Package: `calc' \fileversion\space <\filedate> (KKT and FJ)}
```

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 ⟨integer variable⟩.

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 parentheseis 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`.

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

(The `\expandafter` tokens allow the user to use expressions stored in macros as arguments in assignment commands.)

`\calc@assign@count`
`\calc@assign@dimen`
`\calc@assign@skip`

We need three instances of the `\calc@assign@generic` macro, corresponding to the types ⟨integer⟩, ⟨dimen⟩, and ⟨glue⟩.

```
\def\calc@assign@count{\calc@assign@generic\calc@Acount\calc@Bcount}
\def\calc@assign@dimen{\calc@assign@generic\calc@Adimen\calc@Bdimen}
\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.

```
\newcount\calc@Acount    \newcount\calc@Bcount
\newdimen\calc@Adimen    \newdimen\calc@Bdimen
\newskip\calc@Askip      \newskip\calc@Bskip
```

5.3 The LaTeX interface

As promised, we redefine the following standard LaTeX commands: `\setcounter`, `\addtocounter`, `\setlength`, and `\addtolength`.

```
\def\setcounter#1#2{\@ifundefined{c@#1}{\@nocounterr{#1}}%
   {\calc@assign@count{\global\csname c@#1\endcsname}{#2}}}
\def\addtocounter#1#2{\@ifundefined{c@#1}{\@nocounterr{#1}}%
   {\calc@assign@count{\global\advance\csname c@#1\endcsname}{#2}}}
```

```
\DeclareRobustCommand\setlength{\calc@assign@skip}
\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`

`\calc@pre@scan` reads the initial part (until some ⟨numeric⟩ is seen) of expressions; only left parentheses are allowed here, everything else is taken to be a ⟨numeric⟩ of some sort; this allows unary '`+`' and unary '`-`' to be treated correctly.

```
\def\calc@pre@scan#1{%
  \ifx(#1%
      \let\calc@next\calc@open
  \else
      \let\calc@next\calc@numeric
  \fi
  \calc@next#1}
```

`\calc@open` is used when there is a left parenthesis right ahead. This parenthesis is replaced by TEX 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.

```
\def\calc@open({\begingroup\aftergroup\calc@initB
  \begingroup\aftergroup\calc@initB
  \calc@pre@scan}
\def\calc@initB{\calc@B\calc@A}
```

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

```
\def\calc@numeric{\afterassignment\calc@post@scan \global\calc@A}
```

`\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. Depending on our findings, we call a suitable macro to generate the corresponding 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 ⟨numeric⟩. 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.

```
\def\calc@post@scan#1{%
  \ifx#1!\let\calc@next\endgroup \else
   \ifx#1+\let\calc@next\calc@add \else
    \ifx#1-\let\calc@next\calc@subtract \else
     \ifx#1*\let\calc@next\calc@multiplyx \else
      \ifx#1/\let\calc@next\calc@dividex \else
       \ifx#1)\let\calc@next\calc@close \else \calc@error#1%
```

```
        \fi
       \fi
      \fi
     \fi
    \fi
   \fi
   \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`.

```
\def\calc@add{\calc@generic@add\calc@addAtoB}
\def\calc@subtract{\calc@generic@add\calc@subtractAfromB}
\def\calc@generic@add#1{\endgroup\global\calc@A\calc@B\endgroup
   \begingroup\aftergroup#1\begingroup\aftergroup\calc@initB
   \calc@pre@scan}
\def\calc@addAtoB{\advance\calc@B\calc@A}
\def\calc@subtractAfromB{\advance\calc@B-\calc@A}
```

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).

```
\def\calc@multiplyx#1{\def\calc@tmp{#1}%
   \ifx\calc@tmp\calc@ratio@x \let\calc@next\calc@ratio@multiply \else
      \ifx\calc@tmp\calc@real@x \let\calc@next\calc@real@multiply \else
         \let\calc@next\calc@multiply
      \fi
   \fi
   \calc@next#1}
\def\calc@dividex#1{\def\calc@tmp{#1}%
   \ifx\calc@tmp\calc@ratio@x \let\calc@next\calc@ratio@divide \else
      \ifx\calc@tmp\calc@real@x \let\calc@next\calc@real@divide \else
         \let\calc@next\calc@divide
      \fi
   \fi
   \calc@next#1}
\def\calc@ratio@x{\ratio}
\def\calc@real@x{\real}
```

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.

```
\def\calc@multiply{\calc@generic@multiply\calc@multiplyBbyA}
\def\calc@divide{\calc@generic@multiply\calc@divideBbyA}
\def\calc@generic@multiply#1{\endgroup\begingroup
   \let\calc@A\calc@Acount \let\calc@B\calc@Bcount
   \aftergroup#1\calc@pre@scan}
\def\calc@multiplyBbyA{\multiply\calc@B\calc@Acount}
\def\calc@divideBbyA{\divide\calc@B\calc@Acount}
```

Since the value to use in the multiplication/division operation is stored in the `\calc@Acount` 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.

```
\def\calc@close
   {\endgroup\global\calc@A\calc@B
    \endgroup\global\calc@A\calc@B
    \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.

```
\def\calc@ratio@multiply\ratio{\calc@ratio@evaluate}
\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`.

```
\let\calc@numerator=\calc@Bcount
\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.

```
\def\calc@ratio@evaluate#1#2{%
   \endgroup\begingroup
      \calc@assign@dimen\calc@numerator{#1}%
      \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.

```
      \gdef\calc@the@ratio{}%
      \ifnum\calc@numerator<0 \calc@numerator-\calc@numerator
         \gdef\calc@the@ratio{-}%
      \fi
      \ifnum\calc@denominator<0 \calc@denominator-\calc@denominator
         \xdef\calc@the@ratio{\calc@the@ratio-}%
      \fi
      \calc@Acount\calc@numerator
      \divide\calc@Acount\calc@denominator
      \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.

```
        \calc@next@digit \calc@next@digit \calc@next@digit
        \calc@next@digit \calc@next@digit \calc@next@digit
    \endgroup
```

Now we have the ratio represented (as the expansion of the global macro `\calc@the@ratio`) in the syntax ⟨decimal constant⟩ [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.

```
    \calc@multiply@by@real\calc@the@ratio
    \begingroup
    \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.

```
\def\calc@next@digit{%
        \multiply\calc@Acount\calc@denominator
        \advance\calc@numerator -\calc@Acount
        \multiply\calc@numerator 10
        \calc@Acount\calc@numerator
        \divide\calc@Acount\calc@denominator
        \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.

```
\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.

```
\def\calc@real@multiply\real#1{\endgroup
    \calc@multiply@by@real{#1}\begingroup
    \calc@post@scan}
\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.

```
\def\calc@error#1{%
  \errhelp{Calc error: I expected to see one of: + - * / )}%
  \errmessage{Invalid character `#1' in arithmetic expression}}
⟨/package⟩
```

References

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

[2] L. Lamport. *LaTeX, A Document Preparation System.* Addison-Wesley, Reading, Massachusetts, Second edition 1994/1985.