Le package calc
Arithmétique en notation infixe dans LaTeX*

Kresten Krab Thorup, Frank Jensen (and Chris Rowley)

Traduction française par Jean-Pierre Drucbert†

1998/07/07

Résumé

Le package `calc` ré-implémente les commandes LaTeX `\setcounter`, `\addtocounter`, `\setlength` et `\addtolength`. Au lieu d'une simple valeur, ces commandes acceptent aussi désormais une expression en notation infixe.

1 Introduction

L'arithmétique en TeX est effectuée en utilisant des opérations de bas niveau telles que `\advance` et `\multiply`. Ceci peut être acceptable lors du développement d'un package de macros, mais ce n'est pas une interface acceptable pour l'utilisateur final.

Ce package introduit une vraie arithmétique en notation infixe qui est bien plus familière à la plupart des gens. La notation infixe est plus lisible et plus facile à modifier que l'autre : une suite d'affectations et d'instructions arithmétiques. L'une des instructions arithmétique (`\divide`) n'a même pas d'équivalent en LaTeX standard.

Les expressions infixes peuvent être utilisées dans les arguments de macros (le package `calc` n'emploie pas de changements de codes de catégorie pour atteindre ses buts) [1].

2 Description informelle

Le LaTeX standard offre le jeu suivant de commandes pour manipuler compteurs et longueurs [2, pages 194 et 216] :

`\setcounter{`*ctr*`}{`*nombre*`}` force la valeur du compteur *ctr* à (la valeur de) *nombre*. (Fragile)

`\addtocounter{`*ctr*`}{`*nombre*`}` incrémente la valeur du compteur *ctr* de (la valeur de) *nombre*. (Fragile)

`\setlength{`*cmd*`}{`*longueur*`}` force la valeur de la commande longueur *cmd* à (la valeur de) *longueur*. (Robuste)

*Les auteurs du package remercient Frank Mittelbach pour ses utiles commentaires et suggestions qui ont grandement amélioré ce package.

†Dernière mise à jour le 29/01/2000

[1] Cependant, il suppose en conséquence que les codes de catégorie des caractères spéciaux, comme (`*/`) dans sa syntaxe ne changent pas.

`\addtolength{`*cmd*`}{`*longueur*`}` force la valeur de la commande longueur *cmd* à sa valeur courante plus (la valeur de) *longueur*. (Robuste)

(Les commandes `\setcounter` et `\addtocounter` ont un effet global, alors que les commandes `\setlength` et `\addtolength` obéissent aux règles de portée normales.) En LaTeX standard, les arguments de ces commandes doivent être de simples valeurs. Le package `calc` étend ces commandes pour qu'elles acceptent des expressions en notation infixe, représentant des valeurs des types adéquats. En utilisant le package `calc`, *nombre* est remplacé par ⟨expression entière⟩, et *longueur* est remplacé par ⟨expression glu⟩. La syntaxe formelle de ⟨expression entière⟩ et ⟨expression glu⟩ est donnée plus loin.

En plus de ces commandes pour établir explicitement une longueur, de nombreuses commandes LaTeX prennent une longueur comme argument. Après avoir chargé ce package, la plupart de ces commandes accepteront une ⟨expression glu⟩. Ceci inclut l'argument optionnel de largeur de `\makebox`, l'argument largeur de `\parbox`, de `minipage` et d'une `p`-colonne d'un environnement `tabular`, et de nombreuses constructions similaires. (Ce package ne redéfinit aucune de ces commandes, mais elles sont définies par défaut comme lisant leurs arguments par `\setlength` et donc bénéficient automatiquement de la commande `\setlength` améliorée fournie par ce package.)

Dans la suite, nous utiliserons la terminologie du TeX standard. La correspondance entre terminologies TeX et LaTeX est la suivante : les compteurs LaTeX correspondent aux registres compteurs de TeX ; ils contiennent des quantités du type ⟨nombre⟩. Les commandes longueurs de LaTeX correspondent aux registres dimensions de TeX « *dimen* » (pour les longueurs rigides) et registres sauts « *skip* » de TeX (pour les longueurs élastiques) ; ils contiennent des quantités de type ⟨dimension⟩ et ⟨glu⟩, respectivement.

TeX nous offre des opérations primitives pour effectuer l'arithmétique sur les registres comme suit :

– addition et soustraction de tous types de quantités sans restriction ;
– multiplication et division par un *entier* peuvent être effectuées sur un registre de tout type ;
– multiplication par un nombre *réel* (c'est-à-dire un nombre avec une partie fractionnaire) peut être effectuée sur un registre de tout type, mais les composantes d'étirement et de contraction d'une quantité glu sont éliminées.

Le package `calc` utilise ces primitives TeX mais offre une notation plus conviviale pour exprimer l'arithmétique.

Une expression est formée de quantités numériques (comme les constantes explicites, les compteurs LaTeX et les commandes longueur) et d'opérateurs binaires (les tokens « `+` », « `-` », « `*` » et « `/` » avec leur signification usuelle) en utilisant la notation infixe familière ; des parenthèses peuvent être utilisées pour outrepasser les règles de priorité usuelles (qui disent que multiplication et division ont une plus forte priorité qu'addition et soustraction).

Les expressions doivent être correctement typées. Ceci signifie, par exemple, qu'une expression dimension doit être la somme de termes dimensions : c'est-à-dire que vous ne pouvez pas dire « `2cm+4` » mais « `2cm+4pt` » est valide.

Dans un terme dimension, la partie dimension doit venir en premier ; ceci est aussi vrai pour les termes glu. De plus, la multiplication et la division par des quantités non entières requièrent une syntaxe spéciale (voir ci-dessous).

L'évaluation des sous-expressions au même niveau de priorité procède de gauche à droite. Considérons un terme dimension tel que « `4cm*3*4` ». D'abord, la valeur du facteur `4cm` est affectée à un registre dimension, puis ce registre est multiplié par 3 (en utilisant `\multiply`) et, enfin, ce registre est multiplié par 4 (en utilisant encore `\multiply`). Ceci explique aussi pourquoi la partie dimension (c'est-à-dire la partie avec la désignation d'unité) doit venir en premier ; TEX ne permet simplement pas que des constantes sans type soient affectées à un registre dimension.

Le package `calc` permet aussi la multiplication et la division par des nombres réels. Cependant une syntaxe spéciale est nécessaire : vous devez utiliser `\real{`⟨constante décimale⟩`}`[2] ou `\ratio{`⟨expression dimension⟩`}{`⟨expression dimension⟩`}` pour représenter une valeur réelle à utiliser dans une multiplication ou division. La première forme a une signification évidente et la seconde représente le nombre obtenu en divisant la valeur de la première expression par la valeur de la seconde expression.

Une addition ultérieure au package (en juin 1998) offre une méthode supplémentaire pour spécifier un facteur de type dimension en composant un certain texte (en mode LR) et mesurant ses dimensions : ce sont les commandes ci-dessous.

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

Ces commandes calculent les tailles naturelles du ⟨texte⟩ exactement de la même manière que les commandes `\settowidth`, etc.

Notez qu'il y a une petite différence dans l'utilisation de ces deux méthodes d'accès au dimensions d'un texte. Après `\settowidth{\txtwd}{Du texte}` vous pouvez utiliser :

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

tandis qu'en utilisant l'accès plus direct à la largeur du texte il faut utiliser la forme longue pour la multiplication, donc :

```
\setlength{\parskip}{\widthof{Du texte} * \real{0.68}}
```

TEX élimine les composantes d'étirement et de compression de la glu lorsque cette glu est multipliée par un nombre réel. Ainsi, par exemple,

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

forcera la séparation entre paragraphes à 4.5pt sans étirement ni compression. (En passant, notez comment les espaces peuvent être utilisés pour améliorer la lisibilité.)

Lorsque TEX effectue de l'arithmétique sur des entiers, toutes les parties fractionnaires des résultats sont perdues. Par exemple,

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

affectera la valeur 3 au compteur `x`, la valeur 4 à `y` et la valeur 5 à `z`. Cette troncature s'applique aussi aux résultats *intermédiaires* dans le calcul séquentiel d'une expression composite ; donc la commande suivante

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

[2]En fait, au lieu de ⟨constante décimale⟩, la forme plus générale ⟨signes optionnels⟩⟨facteur⟩ peut être utilisée. Cependant ceci n'apporte aucune puissance expressive supplémentaire au langage des expressions infixes.

affectera 6 à `x`.

Comme exemple d'utilisation de `\ratio`, considérons le problème de l'agrandissement d'une figure pour qu'elle occupe toute la largeur (c'est-à-dire `\textwidth`) du corps d'une page. En supposant que les dimensions d'origine de la figure sont données par des variables longueur (dimensions) `\Xsize` et `\Ysize`. La hauteur de la figure agrandie peut alors s'exprimer par

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

2.1 Syntaxe formelle

La syntaxe est décrite par le jeu de règles suivant. Notez que les définitions de ⟨nombre⟩, ⟨dimension⟩, ⟨glu⟩, ⟨constante décimal⟩ et ⟨plus ou minus⟩ sont comme dans le chapitre 24 du *TEXbook* [1] ; et ⟨texte⟩ est du matériel en mode LR, comme dans le manuel [2]. Nous utilisons *type* comme une méta-variable, représentant « entier », « dimension » et « glu »[3].

⟨*type* expression⟩ ⟶ ⟨terme *type*⟩
  | ⟨expression *type*⟩⟨plus ou minus⟩⟨terme *type*⟩

⟨terme *type*⟩ ⟶ ⟨facteur *type*⟩
  | ⟨terme *type*⟩⟨multiply ou divide⟩⟨facteur entier⟩
  | ⟨terme *type*⟩⟨multiply ou divide⟩⟨nombre réel⟩

⟨facteur *type*⟩ ⟶ ⟨*type*⟩ | ⟨facteur dimension de texte⟩
  | $(_{12}$⟨expression *type*⟩$)_{12}$

⟨entier⟩ ⟶ ⟨nombre⟩

⟨facteur dimension de texte⟩ ⟶ ⟨commande dimension de texte⟩`{`⟨texte⟩`}`

⟨commande dimension de texte⟩ ⟶ `\widthof` | `\heightof` | `\depthof`

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

⟨nombre réel⟩ ⟶ `\ratio{`⟨expression dimension⟩`}{`⟨expression dimension⟩`}`
  | `\real{`⟨constante décimale⟩`}`

Notez que durant la plus grande partie de l'analyse des expressions `calc`, aucune expansion n'a lieu ; donc la syntaxe ci-dessus doit être explicite[4].

2.2 Le schéma d'évaluation

Dans cette section, dans un souci de simplicité nous ne considérerons que des expressions contenant les opérateurs « + » (addition) et « $*$ » (multiplication). Il est trivial d'ajouter la soustraction et la division.

Une expression E est une somme de termes : $T_1 + \cdots + T_n$; un terme est un produit de facteurs : $F_1 * \cdots * F_m$; un facteur est soit une simple quantité numérique f (comme ⟨nombre⟩ tel que décrit dans le *TEXbook*), ou une expression parenthésée (E').

[3] Cette version du package `calc` ne supporte pas l'évaluation des expressions de glu mathématique.

[4] Il y a deux exceptions à ceci : le premier token est expansé sur un niveau (donc l'expression entière peut être mise dans une macro) ; chaque fois qu'une ⟨constante décimale⟩ ou un ⟨type⟩ est attendu.

Puisque le moteur TeX ne peut exécuter les opérations arithmétiques que d'une manière analogue au code machine, il nous faut trouver un moyen pour traduire la notation infixe en son « jeu d'instructions ».

Notre but est de concevoir un schéma de traduction qui traduise X (une expression, un terme ou un facteur) en une séquence d'instructions TeX qui fasse les choses suivantes [Propriété d'Invariance] : évaluer correctement X, laisser le résultat dans un registre global A (en utilisant une affectation globale) et ne pas faire d'affections globales au registre de travail B ; de plus, la séquence de code doit être équilibrée par rapport aux groupes TeX. Nous noterons la séquence de code correspondant à X par $[\![X]\!]$.

Dans le code de remplacement spécifié ci-dessous, nous utilisons les conventions suivantes :

- A et B représentent des registres ; toutes les affectation à A seront globales, et toutes les affectations à B seront locales.
- « $\Leftarrow$ » signifie affectation globale au registre en partie gauche.
- « $\leftarrow$ » signifie affectation locale au registre en partie gauche.
- « $\hookrightarrow_{[C]}$ » signifie « sauvegarder le C jusqu'à ce que le groupe (portée) courant se termine, et l'exécuter alors ». Ceci correspond à la primitive TeX `\aftergroup`.
- « $\{$ » représente le départ d'un nouveau groupe et « $\}$ » représente la fin d'un groupe.

Considérons une expression $T_1 + T_2 + \cdots + T_n$. En supposant que $[\![T_k]\!]$ $(1 \leq k \leq n)$ atteint le but établi, le code suivant atteint manifestement le but établi pour leur somme :

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

Notez le niveau supplémentaire de groupement entourant chacun des $[\![T_1]\!]$, $[\![T_2]\!]$, …, $[\![T_n]\!]$. Ceci garantira que le registre B, utilisé pour calculer la somme des termes, n'est pas écrasé par les calculs intermédiaires des termes individuels. En fait, le groupe entourant $[\![T_1]\!]$ n'est pas nécessaire, mais ils se trouve qu'il est plus simple de traiter tous les termes de la même façon.

La séquence de code « $\{\,[\![T_2]\!]\,\}\ B \leftarrow B + A$ » peut être traduite par la séquence de code équivalente suivante : « $\{\hookrightarrow_{[B \leftarrow B+A]} [\![T_2]\!]\,\}$ ». Cette observation se trouve être la clé pour l'implémentation : le « $\hookrightarrow_{[B \leftarrow B+A]}$ » est engendré *avant* que T_2 soit traduit, au même moment que l'opérateur « + » entre T_1 et T_2 est vu.

Maintenant, la spécification du schéma de traduction est évidente :

$$\begin{aligned}[\![f]\!] \quad &\Longrightarrow \quad A \Leftarrow f \\ [\![(E')]\!] \quad &\Longrightarrow \quad [\![E']\!] \\ [\![F_1 * F_2 * \cdots * F_m]\!] \quad &\Longrightarrow \quad \{\hookrightarrow_{[B \leftarrow A]} [\![F_1]\!]\,\} \quad \{\hookrightarrow_{[B \leftarrow B*A]} [\![F_2]\!]\,\} \\ &\qquad\qquad \ldots \quad \{\hookrightarrow_{[B \leftarrow B*A]} [\![F_m]\!]\,\} \quad A \Leftarrow B \\ [\![T_1 + T_2 + \cdots + T_n]\!] \quad &\Longrightarrow \quad \{\hookrightarrow_{[B \leftarrow A]} [\![T_1]\!]\,\} \quad \{\hookrightarrow_{[B \leftarrow B+A]} [\![T_2]\!]\,\} \\ &\qquad\qquad \ldots \quad \{\hookrightarrow_{[B \leftarrow B+A]} [\![T_n]\!]\,\} \quad A \Leftarrow B\end{aligned}$$

Par induction structurelle, on voit aisément que la propriété voulue est atteinte.

Par inspection de schéma de traduction, nous voyons que nous avons à engendrer le code suivant :

- nous devons engendrer « $\{\hookrightarrow_{[B\leftarrow A]}\{\hookrightarrow_{[B\leftarrow A]}$ » au bord gauche d'une expression (c'est-à-dire pour chaque parenthèse gauche et à la parenthèse gauche implicite au début de l'expression complète) ;
- nous devons engendrer « $\}A \Leftarrow B\}A \Leftarrow B$ » au bord droit d'une expression (c'est-à-dire chaque parenthèse droite et la parenthèse droite implicite à la fin de l'expression complète) ;
- « `*` » est remplacé par « $\}\{\hookrightarrow_{[B\leftarrow B*A]}$ » ;
- « `+` » est remplacé par « $\}A \Leftarrow B\}\{\hookrightarrow_{[B\leftarrow B+A]}\{\hookrightarrow_{[B\leftarrow A]}$ » ;
- lorsque nous voyons (attendons) une quantité numérique, nous insérons le code d'affectation « $A \Leftarrow$ » devant cette quantité et laissons TEX l'analyser.

3 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).

3.1 Getting started

Now we have enough insight to do the actual implementation in TEX. First, we announce the macro package[5].

```
⟨*package⟩
%\NeedsTeXFormat{LaTeX2e}
%\ProvidesPackage{calc}[\filedate\space\fileversion]
```

3.2 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.

[5] 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 ⟨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 2.2, 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 2.2, 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 one-level deep in a macro 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
```

3.3 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].)

3.4 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[6] but means that anything illegal will simply generate a TeX-level error, often a reasonably comprehensible one!

The many `\expandafter`s are needed to efficiently end the nested conditionals so that `\calc@textsize` can process its argument.

```
\def\calc@pre@scan#1{%
   \ifx(#1%
       \expandafter\calc@open
   \else
        \ifx\widthof#1%
            \expandafter\expandafter\expandafter\calc@textsize
        \else
            \calc@numeric% no \expandafter needed for this one.
        \fi
   \fi
   #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 2.2. 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}
```

`\widthof`
`\heightof`
`\depthof`

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[7]. 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 LaTeX user[8].

```
\newcommand\widthof{}
\let\widthof\ignorespaces
\newcommand\heightof{}
\let\heightof\ignorespaces
\newcommand\depthof{}
\let\depthof\ignorespaces
```

[6] In the few contexts where signs are allowed: this could, I think, be extended (CAR).

[7] If this level of safety is not needed then the code can be speeded up: CAR.

[8] Is this necessary, CAR?

\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[9].

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.

```
\def\calc@textsize #1#2{%
  \begingroup
    \let\widthof\wd
    \let\heightof\ht
    \let\depthof\dp
    \@settodim #1%
      {\global\calc@A}%
      {%
       \let\widthof\ignorespaces
       \let\heightof\ignorespaces
       \let\depthof\ignorespaces
       #2}%
  \endgroup
  \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[10]. 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`.

[9] It is based on suggestions by Donald Arsenau and David Carlisle.
[10] Is `!` a good choice, CAR?

```
\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}
```

\real
\ratio

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[11] (by some other package: dangerous but possible!); this will also make them appear to be undefined to a LaTeX user (also possibly dangerous).

```
\ifx\real\@undefined\let\real\relax\fi
\ifx\ratio\@undefined\let\ratio\relax\fi
```

In order to test for them, we define these two[12].

```
\def\calc@ratio@x{\ratio}
\def\calc@real@x{\real}

\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}
```

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.

[11] Suggested code from David Carlisle.
[12] May not need the extra names, CAR?

`\calc@close` generates code for a right parenthesis (which was derived to be “$\}A \Leftarrow B\}A \Leftarrow B$” in Section 2.2). 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}
```

3.5 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.)

3.6 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}}
```

4 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{%
  \PackageError{calc}%
    {`#1' invalid at this point}%
    {I expected to see one of: + - * / )}}
⟨/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.