Anatomie d'une macro *

Denis Roegel

LORIA
Campus scientifique
BP 239
54506 Vandœuvre-lès-Nancy cedex
roegel@loria.fr

Résumé. Dans cet article, nous faisons un parcours détaillé d'une macro calculant les nombres premiers. Ceci nous donne l'occasion d'illustrer des concepts techniques peu connus de TEX.

Abstract. *In this article, we explain in detail a macro computing the prime numbers. This gives us an opportunity to illustrate little known technical aspects of TEX.*

Dédié à Chrystel Barraband qui a inspiré ces commentaires.

1. Introduction

Une macro TEX est la définition d'une commande en fonction d'autres commandes. À la fois la définition d'une commande et le mode de transmission des paramètres obéissent à des règles précises et simples, mais qui sont d'une part souvent oubliées, d'autre part indispensables à une fine compréhension de TEX.

L'appel d'une macro TEX est un processus très différent de ce qui se produit dans des langages classiques. On peut le rapprocher d'un appel de macro dans le préprocesseur C et on imagine difficilement programmer dans un tel language ! Il n'y a pas de notions de passage par référence ou par valeur. L'appel d'une macro revient simplement à effectuer un remplacement ou une substitution. Mais une macro peut appeler d'autres macros, y compris elle-même ce qui autorise la récursivité.

* Cette version corrige une erreur de l'article publié en décembre 1998 (Cahiers GUTenberg n° 31), découverte à la suite de la traduction anglaise pour TUGboat. D. Roegel, 18 juin 2001.

2. Calcul de nombres premiers

Nous allons nous intéresser au calcul de nombres premiers. Rappelons que $n > 1$ est premier si n n'est divisible par aucun entier autre que lui-même et 1. Si n est de la forme $2m + 1$, il suffit de diviser n par $3,5,7,\ldots,p \leqslant \lfloor\sqrt{n}\rfloor$. En effet, si n est divisible par $p > \lfloor\sqrt{n}\rfloor$, alors n est aussi divisible par $q < \lfloor\sqrt{n}\rfloor$. Les diviseurs p seront testés jusqu'à ce que $p^2 > n$.

3. Macros

L'exemple qui suit, tiré du TEXbook [Knuth, 1984], est d'un niveau avancé mais va permettre d'entrer dans le vif du sujet. La macro `\primes` qui y est définie permet de déterminer les n premiers nombres premiers, à partir de 2. Par exemple, `\primes{30}` donne les 30 premiers nombres premiers. Voici tout d'abord toutes les définitions. Elles vont ensuite être décortiquées :

```
\newif\ifprime \newif\ifunknown

\newcount\n \newcount\p \newcount\d \newcount\a

\def\primes#1{2,~3% #1 est supposé au moins égal à 3
  \n=#1 \advance\n by-2 % encore n à faire
  \p=5 % nombres premiers impairs commençant à p
  \loop\ifnum\n>0 \printifprime\advance\p by2 \repeat}

\def\printp{, % on appellera \printp si p est premier
  \ifnum\n=1 and~\fi % ceci précède la dernière valeur
  \number\p \advance\n by -1 }

\def\printifprime{\testprimality \ifprime\printp\fi}

\def\testprimality{{\d=3 \global\primetrue
  \loop\trialdivision \ifunknown\advance\d by2 \repeat}}

\def\trialdivision{\a=\p \divide\a by\d
  \ifnum\a>\d \unknowntrue\else\unknownfalse\fi
  \multiply\a by\d
  \ifnum\a=\p \global\primefalse\unknownfalse\fi}
```

4. Déclarations

Deux booléens, ou plus précisément deux tests, sont tout d'abord déclarés.

```
\newif\ifprime
```

`\ifprime` est équivalent à `\iftrue` si le booléen « `prime` » est vrai. Ce booléen permettra de voir si un nombre doit être affiché ; ainsi, dans la macro `\printifprime` (on voit en passant que les macros sont définies par `\def`), l'expression `\ifprime\printp\fi` signifie que si `\ifprime` est évalué à `\iftrue` alors `\printp` (c'est-à-dire la fonction qui imprimera le nombre qui nous intéresse, à savoir `\p`) sera exécuté, sinon il ne se passe rien.

```
\newif\ifunknown
```

Si « `unknown` » est vrai, cela signifie que l'on n'a pas encore de certitude quant au fait que `\p` est composé ou non. On ne sait ni l'un ni l'autre. Initialement, « `unknown` » est donc vrai et `\ifunknown` est un test qui réussit. Si « `unknown` » est faux, on possède une connaissance sur la primalité de `\p`, c'est-à-dire que l'on sait si `\p` est premier ou non.

Ensuite, sont déclarées quelques variables entières utiles par la suite :

- `\newcount\n`
  `\n` est le nombre de nombres premiers qui restent à afficher.
- `\newcount\p`
  `\p` est le nombre courant dont la primalité est testée.
- `\newcount\d`
  `\d` est une variable qui contient les essais de diviseurs successifs de `\p`.
- `\newcount\a`
  `\a` est une variable auxiliaire

5. Macro principale

La macro principale est `\primes`. Elle prend un argument. Lors de la définition de la macro, cet argument porte le nom `#1`. S'il y avait un second argument, ce serait `#2`, etc. (Il n'est pas possible d'avoir—directement—plus de neuf arguments ; indirectement, on peut en avoir autant que l'on veut, y

compris un nombre variable, qui pourrait par exemple être fonction de l'un des arguments.)

```
\def\primes#1{2,~3%
  \n=#1 \advance\n by-2 %
  \p=5 %
  \loop\ifnum\n>0 \printifprime\advance\p by2 \repeat}
```

Cette macro est en fait très simple. Lorsqu'elle est appelée, disons avec comme paramètre d'appel 30, `\primes{30}` est remplacé par le corps de `\primes` (c'est-à-dire le groupe entre accolades qui suit la liste des arguments de `\primes`), dans lequel `#1` est remplacé par les deux caractères 3 et 0. `\primes{30}` devient donc (les espaces sont ignorés en début de ligne) :

```
2,~3%
\n=30 \advance\n by-2 %
\p=5 %
\loop\ifnum\n>0 \printifprime\advance\p by2 \repeat
```

Que se passe-t-il donc maintenant ? On affiche `2,~3` c'est-à-dire 2 suivi d'une virgule, suivi d'une espace insécable (i.e., la ligne ne sera *en aucun cas* coupée après la virgule) ; puis la valeur 30 est donnée à `\n`. Immédiatement, 2 est retranché à `\n`, c'est-à-dire que `\n` contient désormais le nombre de nombres premiers qu'il reste à afficher. Pour simplifier, nous avons supposé que l'on voulait afficher au moins les trois premiers nombres premiers. Il est donc certain que `\n` vaut maintenant au moins 1. C'est aussi pour cela qu'il a été possible de mettre uniquement une virgule entre 2 et 3, car nous savons que 3 n'est pas le dernier nombre affiché. Le dernier nombre affiché doit en effet être précédé de « `and` ». Ainsi, en demandant `\primes{3}`, on veut obtenir « `2, 3, and 5` ».

Nous avons dit que `\p` est le nombre courant dont la primalité doit être testée. Il faut donc initialiser `\p` à 5, puisque c'est le premier nombre suivant 3 dont on ignore s'il est premier ou non.

Le corps de `\primes{30}` s'achève par une boucle :

```
\loop\ifnum\n>0 \printifprime\advance\p by2 \repeat
```

Celle-ci est très simple. C'est une boucle `\loop \repeat`. De manière générale, ces boucles ont la forme

```
\loop texte A \if..   texte B \repeat
```

L'exécution de cette boucle se déroule ainsi : elle commence par `\loop`, le texte A est exécuté, puis le test `\if...` Si ce test réussit, le texte B est exécuté, puis le `\repeat` nous fait revenir au `\loop`. S'il échoue, la boucle est finie.

Donc, dans le cas de `\primes{30}`, cela revient à exécuter

```
\printifprime\advance\p by2
```

tant que `\n` est strictement positif, c'est-à-dire, tant qu'il reste des nombres premiers à afficher. Pour que ceci ait le résultat attendu, il faut bien entendu décrémenter la valeur de `\n`. Cela se fait à chaque affichage au sein de l'appel de `\printifprime`.

Par conséquent, s'il reste au moins un nombre à afficher, `\printifprime` est appelé et imprimera `\p` si `\p` est premier. Quel que soit le résultat, on passe ensuite au nombre impair suivant, avec `\advance\p by2`.

6. Affichage

La macro d'affichage est elle aussi très simple. C'est `\printp`.

```
\def\printp{, %
    \ifnum\n=1 and~\fi %
    \number\p \advance\n by -1 }
```

Cette macro n'est appelée que lorsque `\p` est premier (cf. son appel dans `\printifprime`). Dans tous les cas, cette macro sans argument s'expanse en

```
, %
\ifnum\n=1 and~\fi %
\number\p \advance\n by -1
```

c'est-à-dire une virgule, suivie d'un « `and` » si `\n` vaut 1 (donc dans le cas où le nombre qui va être affiché est le dernier), suivi de `\p` (la fonction `\number` analogue à `\the` pour convertir une variable en une suite de caractères affichables est utilisée) ; enfin `\n` est décrémenté de 1, comme annoncé, ce qui permet un déroulement normal de la boucle `\loop...\repeat` dans la macro `\primes`.

La macro `\printifprime` est appelée depuis la macro `\primes`. Elle appelle la fonction calculant la primalité de `\p` ce qui détermine s'il faut ou non afficher `\p`.

```
\def\printifprime{\testprimality \ifprime\printp\fi}
```

Comme on le devine, la macro `\testprimality` a pour effet de donner une valeur « `true` » ou « `false` » au booléen « `prime` » ou, si l'on préfère, de faire réussir ou échouer le test `\ifprime`.

7. Test de primalité

La macro testant la primalité de `\p` utilise l'algorithme classique consistant à faire des essais de division, et à s'arrêter lorsque le diviseur testé dépasse la racine carrée de `\p`.

```
\def\testprimality{{\d=3 \global\primetrue
  \loop\trialdivision \ifunknown\advance\d by2 \repeat}}
```

Cette macro est plus complexe car elle fait intervenir un « groupe » supplémentaire, matérialisé par les accolades. Par conséquent, lorsque la macro `\testprimality` est expansée, on se retrouve avec

```
{\d=3 \global\primetrue
 \loop\trialdivision \ifunknown\advance\d by2 \repeat}
```

c'est-à-dire que ce qui va se passer entre les accolades sera — sauf instruction contraire — local à ce groupe. Cela n'était pas le cas dans les expansions vues précédemment.

Ignorons dans un premier temps ce groupe. Que fait-on ? La valeur 3 est tout d'abord donnée à `\d`. `\d` est le diviseur testé. Successivement vont être testés 3, 5, 7, etc., et cela se poursuivra tant qu'on ne saura pas avec certitude si `\p` est premier ou composé. Dès que l'on saura si `\p` est premier ou composé, le booléen « `unknown` » deviendra faux et le test `\ifunknown` échouera.

Reprenons : on commence avec `\d=3` ; par défaut, `\p` est considéré premier, c'est-à-dire que l'on donne la valeur « `true` » au booléen `\prime`. Cela se fait normalement avec

```
\primetrue
```

mais dans notre cas, cela serait insuffisant. En effet, à l'issue de l'exécution du groupe

```
{\d=3 \primetrue
 \loop\trialdivision \ifunknown\advance\d by2 \repeat}
```

toutes les variables reprennent leur ancienne valeur, car les affectations sont *locales*. Or, le booléen `prime` est nécessaire lorsque l'on fait le test `\ifprime...`

dans `\printifprime`. Il faut donc *transcender* le groupe et forcer l'affectation à être globale. Cela s'obtient avec

```
\global\primetrue
```

Le reste est alors clair : on essaie de diviser `\p` par `\d`, ce qui fait l'objet de la macro `\trialdivision`. Si rien n'a été appris de plus, c'est-à-dire si « `unknown` » est toujours « `true` », on passe au diviseur suivant avec l'instruction `\advance\d by2`. Tôt ou tard le processus s'arrête, comme l'indique la définition de `\trialdivision`.

L'explication du groupe supplémentaire peut maintenant être donnée. En fait, si ce groupe n'est pas introduit, l'expansion de `\primes{30}` conduit à

```
...
\loop\ifnum\n>0 \printifprime\advance\p by2 \repeat
```

qui s'expanse en

```
\loop\ifnum\n>0 \testprimality ...\repeat
```

qui s'expanse en

```
\loop\ifnum\n>0 \d=3 \primetrue
   \loop\trialdivision \ifunknown\advance\d by2 \repeat
                               ...\repeat
```

ce qui va conduire TEX à associer le premier `\loop` avec le premier `\repeat`, ce qui est erronné.

L'introduction du groupe élimine donc une interférence possible entre les deux boucles.

L'explication du groupe supplémentaire peut maintenant être donnée. En fait, si ce groupe n'est pas introduit, l'expansion de `\primes{30}` conduit à

```
...
\loop\ifnum\n>0 \printifprime
  \advance\p by2 \repeat
```

Plain TEX définit `\loop` de la manière suivante :

```
\def\loop#1\repeat{\def\body{#1}\iterate}
\def\iterate{\body\let\next\iterate
  \else\let\next\relax\fi \next}
```

Par conséquent, le texte initial est expansé en

```
\def\body{\ifnum\n>0 \printifprime
  \advance\p by2 }\iterate
```

La construction `\loop...\repeat` devient donc

```
\ifnum\n>0 \printifprime\advance\p by2
           \let\next\iterate
\else \let\next\relax\fi \next
```

Si `\n` > 0, ceci conduit à

```
\printifprime ...
\let\next\iterate \next
```

et donc à

```
\testprimality ...
\let\next\iterate \next
```

et à

```
... \loop\trialdivision
    \ifunknown\advance\d by2 \repeat ...
\let\next\iterate \next
```

Maintenant, `\iterate` va appeler `\body`, mais la définition de `\body` appelée sera celle définie par la seconde boucle (interne) `\loop`, et ce sera le chaos ! Ceci explique pourquoi un groupe a été introduit. Le groupe sépare bien la définition interne de `\body` de la structure `\loop` externe, et chaque appel de `\iterate` produit ainsi le résultat approprié.

8. Essais de divisions

La dernière macro est celle qui fait réellement la division de `\p` par `\d`. Une variable intermédiaire `\a` est utilisée.

```
\def\trialdivision{\a=\p \divide\a by\d
  \ifnum\a>\d \unknowntrue\else\unknownfalse\fi
  \multiply\a by\d
  \ifnum\a=\p \global\primefalse\unknownfalse\fi}
```

`\p` est recopié dans `\a`, puis `\a` est divisé par `\d`. Ceci donne en `\a` la *partie entière* du quotient $\frac{\p}{\d}$. Deux cas sont alors à considérer :

1. si `\a` $>$ `\d` , c'est-à-dire si `\d` est inférieur à la racine carrée de `\p`, alors on se trouve encore dans l'inconnu. Il se peut très bien que `\d` divise `\p`, ou qu'il y ait un autre diviseur de `\p` supérieur à `\d` et inférieur à

la racine carrée de `\p` encore à découvrir. La valeur « `true` » est donc donnée au booléen « `unknown` », en écrivant : `\unknowntrue`.

2. si `\a` $\leqslant$ `\d`, on considère que l'on sait, ou en tout cas que l'on va savoir tout de suite. On écrit donc : `\unknownfalse`

Pour lever les derniers doutes, il faut voir s'il y a un reste à la division de `\p` par `\d`, ou plutôt de `\a` par `\d` : `\a` est donc multiplié par `\d` :

```
\multiply\a by\d
\ifnum\a=\p \global\primefalse\unknownfalse\fi
```

Si `\p` est retrouvé, c'est que `\d` est un diviseur de `\p`. Dans ce cas, `\p` n'est évidemment pas premier et la valeur « `false` » est donnée au booléen « `prime` », avec `\primefalse`. Comme `\trialdivision` apparaît en fait dans le groupe entourant le corps de `\testprimality`, et que l'on a besoin du booléen « `prime` » en dehors de `\testprimality`, il faut à nouveau transcender ce groupe et forcer l'affectation de « `prime` » à être globale. D'où l'écriture :

```
\global\primefalse
```

Enfin, dans le cas présent où `\d` est un diviseur de `\p`, on pose évidemment `\unknownfalse` ce qui a pour unique effet de mettre fin à la boucle

```
\loop\trialdivision \ifunknown\advance\d by2 \repeat
```

c'est-à-dire que l'on cesse de tester un autre diviseur. On peut noter qu'il n'y a pas de `\global` devant `\unknownfalse`, puisque `\ifunknown` est utilisé dans le groupe en question, et non en dehors.

Si `\p` n'est pas retrouvé après la multiplication, c'est que `\d` n'est pas un diviseur de `\p`. À ce moment là, on avait

– soit `\a` $\leqslant$ `\d`, (cas 2 plus haut) et donc `\a` $<$ `\d` (sinon `\p` aurait été retrouvé après la multiplication), et donc `\unknownfalse`, d'où arrêt de la boucle

  ```
  \loop\trialdivision \ifunknown\advance\d by2 \repeat
  ```

  et puisque ceci se place dans le contexte

  ```
  \d=3 \global\primetrue
  \loop\trialdivision \ifunknown\advance\d by2 \repeat
  ```

où « `prime` » avait été initialisé à « `true` », on conclut naturellement que n'ayant pas trouvé de diviseur jusqu'à la racine de `\p`, `\p` est premier.
Donc, à l'issue de l'appel de `\testprimality`, `\ifprime` réussit et `\p` est affiché.

– soit `\a` $>$ `\d` : dans ce cas, on reste dans l'inconnu, `\unknowntrue`, et il faut passer au prochain diviseur.

9. Conclusion

Ceci achève d'expliquer l'essentiel à connaître pour comprendre ces macros, à l'exception de quelques subtilités qui ont été passées sous silence.

Il faut savoir que lorsque TEX effectue des opérations complexes comme celles qui viennent d'être décrites, il passe beaucoup de temps. Pour exécuter `\primes{30}`, TEX passe plus de temps qu'il ne lui faut en moyenne pour mettre toute une page en forme avec plain TEX. `\trialdivision` subit dans ce cas 132 expansions. Avec `\primes{1000}` il y a 41331 expansions et avec `\primes{10000}` 1441624 expansions.

À titre d'information, les macros de l'exemple précédent sont données dans le TEXbook [Knuth, 1984], mais avec pour seules explications les lignes suivantes, dans lesquelles j'ai déjà puisé :

> The computation is fairly straightforward, except that it involves a loop inside a loop; therefore `\testprimality` introduces an extra set of braces, to keep the inner loop control from interfering with the outer loop. The braces make it necessary to say "`\global`" when `\ifprime` is being set true or false. TEX spent more time constructing that sentence than it usually spends on an entire page; the `\trialdivision` macro was expanded 132 times.

Le langage de programmation de TEX est comme on le voit assez particulier et nous n'en avons donné ici qu'un aperçu. Le lecteur qui voudrait en savoir davantage est invité à se plonger dans la « bible » de TEX, à savoir le TEXbook [Knuth, 1984], de Donald Knuth.

Bibliographie

[Knuth, 1984] Donald E. Knuth. The TEXbook. Addison-Wesley, Reading, MA, USA, 1984.