Anatomy of a macro

Denis Roegel

Abstract

In this article, we explain in detail a TEX macro for computing prime numbers. This gives us an opportunity to illustrate technical aspects often ignored by TEX beginners.

This article is dedicated to Chrystel Barraband for whom the first version was written in 1993.

This article is a translation, with corrections, of the article "Anatomie d'une macro" published in the *Cahiers GUTenberg*, number 31, December 1998, pages 19–27. Reprinted with permission.

Introduction

A TEX macro can be seen as the definition of a command by other commands. Both the definition of a command and the way arguments are passed obey rules which are both precise and simple, but which are often overlooked, though indispensable to a good understanding of TEX.

Moreover, the call of a TEX macro is a very different process from what happens in classical languages. It is similar to a macro call in the C preprocessor and it is hard to imagine programming with such a language! A macro call merely entails a replacement or a substitution, but it can also call other macros, including itself, which allows recursion.

Computing prime numbers

We will focus on the computation of prime numbers. $n > 1$ is prime if n is divisible only by itself and 1. If

n is odd, it is sufficient to divide n by $3, 5, 7, \ldots, p \leq \lfloor\sqrt{n}\rfloor$. For, if n can be divided by $p > \lfloor\sqrt{n}\rfloor$, then n can also be divided by $q < \lfloor\sqrt{n}\rfloor$. The divisors p will be tried until $p^2 > n$.

Macros

The following example, from *The TEXbook* (Knuth, 1984), is of an advanced level but will allow us to go straight to the heart of the matter. The macro `\primes` makes it possible to determine the first n prime numbers, starting with 2. For instance, `\primes{30}` returns the first 30 prime numbers. Here are all the definitions.[1] We will then analyze them in detail:

```
\newif\ifprime \newif\ifunknown
\newcount\n \newcount\p
\newcount\d \newcount\a
\def\primes#1{2,~3% assume that #1>2
  \n=#1 \advance\n by-2 % n more to go
  \p=5 % odd primes starting with p
  \loop\ifnum\n>0 \printifprime
       \advance\p by2 \repeat}
\def\printp{, % invoked if p is prime
  \ifnum\n=1 and~\fi
  \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}
```

Declarations

First, we declare two booleans, or more precisely two tests.

```
\newif\ifprime
```

`\ifprime` is equivalent to `\iftrue` if "`prime`" is true. This boolean will make it possible to see if a number must be printed; thus, in `\printifprime`, the expression `\ifprime\printp\fi` means that if `\ifprime` is evaluated to `\iftrue`, then `\printp` (that is, the macro that will print the number of interest to us, namely `\p`) will be executed, otherwise nothing will happen.

```
\newif\ifunknown
```

"`unknown`" will be true if we are not yet sure whether `\p` is composed or not. Neither is known. Initially, "`unknown`" is thus true and the `\ifunknown` test succeeds. If "`unknown`" is false, we have knowledge about `\p`'s primality, that is, we know if `\p` is prime or not.

Next the code defines a few integer variables useful in what follows:

- `\newcount\n`
  `\n` is the number of prime numbers that remain to be printed.
- `\newcount\p`
  `\p` is the current number for which primality is tested.
- `\newcount\d`
  `\d` is a variable containing the sequence of trials of divisors of `\p`.
- `\newcount\a`
  `\a` is an auxiliary variable.

Main macro

The main macro is `\primes`. It takes an argument. When the macro is defined, this argument has the name `#1`. If there were a second argument, it would be `#2`, etc. (It is not possible to have — directly — more than nine arguments; indirectly however, one can have as many arguments as one wants, including a variable number, which could for instance be a function of one of the arguments.)

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

When the `\primes` macro is called, for instance with 30, `\primes{30}` is replaced by the body of `\primes` (that is, the group between braces which follows the list of `\primes`' formal arguments), in which `#1` is replaced by the two characters 3 and 0. `\primes{30}` hence becomes (we have removed spaces at the beginning of the lines, because they are ignored by TEX):

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

What happens now? We print "`2,~3`", that is, 2 followed by a comma, followed by an unbreakable space (i.e., the line will *in no case* be split after the comma); then 30 is assigned to `\n`. Immediately, 2 is

[1] The code was slightly reformatted to fit in the columns.

subtracted from \n, and \n then contains the number of primes that remain to be printed. To keep it simple, we have assumed that at least the three first primes must be displayed. Therefore, we are sure that \n is at least equal to 1. This is also why it was possible to put a comma between 2 and 3, because we know that 3 is not the last number to be printed. We want the last number printed to be preceded by "and". Hence, when we ask \primes{3}, we want to obtain "2, 3, and 5". It should also be noticed that the "%" after "3" is essential to prevent insertion of a spurious space. "3" will be followed by a comma when \printp is called. The "%" after the second and third lines are not really needed since TEX gobbles all spaces after explicit numbers; these "%" signs appear only as remnants of comments.

We said that \p is the current number whose primality must be tested. We must therefore initialize \p to 5, since it is the first odd number after 3 (which we don't bother to check if it is prime or not).

The body of \primes{30} ends with a loop:

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

It is a \loop/\repeat loop. In general, these loops have the form

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

This loop executes as follows: it starts with \loop, the A text is executed, then the \if... test. If this test succeeds, the B text is executed, then \repeat makes us return to \loop. If the test fails, the loop is over.

Hence, in the case of \primes{30}, it amounts to execute

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

as long as \n is strictly positive, that is, as long as prime numbers remain to be printed. In order for this to produce the expected result, it is of course necessary to decrement the value of \n. This is done every time a number is printed with the call to \printifprime.

As a consequence, if at least one number remains to be printed, \printifprime will be called and will print \p if \p is prime. Whatever the result, we pass then to the next odd number with \advance\p by2.

Printing

The prime numbers are printed with \printp:

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

This macro is called only when \p is prime (see its call in \printifprime). In any case, this macro has no arguments and gets expanded into

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

that is a comma and a space, followed by "and " if \n equals 1 (in the case where the number to be printed is the last one), followed by \p (the \number function is analogous to \the and converts a variable into a sequence of printable characters); finally, \n is decremented by 1, as announced, and this allows a normal unfolding of the \loop...\repeat loop in the \primes macro.

The macro \printifprime is called by \primes. It calls the function computing the primality of \p and this determines if \p must be printed or not.

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

As one can guess, the \testprimality macro sets the "prime" boolean to "true" or "false," or if one prefers, it makes the \ifprime test succeed or fail.

Primality test

The macro testing \p's primality uses the classical algorithm where divisions are tried by numbers smaller than \p's square root.

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

This macro is more complex because it involves an additional "group," shown here by the braces. Therefore, when \testprimality is expanded, we are left with

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

meaning that what happens between the braces will be — when not otherwise specified — local to that group. This was not the case in the expansions seen previously.

Let us first ignore the group. What are we doing? 3 is first assigned to \d where \d is the divisor being tested. We will test 3, 5, 7, etc., in succession, and this will go on as long as it is not known for certain whether \p is prime or not. As soon as we know if \p is prime or composed, the "unknown" boolean will become false and the \ifunknown test will fail.

Now, let us look at this again: we start with \d=3; the default is to consider \p prime, hence the

"true" value is given to the "prime" boolean. This is normally done with

```
\primetrue
```

but in our case, it would not be sufficient. Indeed, at the end of

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

all variables take again their former value, because the assignments are *local* to the group. But the "prime" boolean is used when the `\ifprime...` test is being done in `\printifprime`, which is called after `\testprimality`. The group must therefore be *transcended* and the assignment is coerced to be global. This is obtained with

```
\global\primetrue
```

The remainder is then obvious: an attempt is made to divide `\p` by `\d`, and this is the purpose of `\trialdivision`. If nothing more has been discovered, that is, if "unknown" is still "true", the value of the trial divisor is set to the next value with `\advance\d by2`. Sooner or later this process stops, as shown by the `\trialdivision` definition.

The additional group in `\testprimality` can now be explained. If the group is not introduced, the expansion of `\primes{30}` leads to

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

Plain TEX defines `\loop` as follows:

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

Therefore, the initial text is expanded into

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

Hence, the `\loop...\repeat` construct becomes

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

If $\n > 0$, this leads to

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

and hence to

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

and to

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

Now, `\iterate` will call `\body`, but the `\body` definition called will be the one defined by the second (inner) `\loop`, and chaos will follow! This explain why a group has been introduced. The group keeps the inner `\body` definition away from the outer `\loop` construct, hence each `\iterate` call produces the appropriate result.

Division trials

The last macro is where the actual division of `\p` by `\d` is made. An auxiliary variable `\a` is used.

```
\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` is copied into `\a`, then `\a` is divided by `\d`. This puts into `\a` the *integer part* of $\frac{\p}{\d}$. Two cases must then be considered:

1. if $\a > \d$, that is, if `\d` is smaller than the square root of `\p`, we are still in unknown territory. `\d` may be a divisor of `\p`, or there might be another divisor of `\p` larger than `\d` and smaller than the square root of `\p` root. The "unknown" boolean is therefore set to "true" with `\unknowntrue`.
2. if $\a \leq \d$, we assume that we know, or at least, that we will know momentarily. We write therefore `\unknownfalse`.

In order to be sure, we must check if there is a remainder to `\p`'s division by `\d`, or rather to `\a`'s division by `\d`: `\a` is therefore multiplied by `\d`:

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

If `\p` is found again, it means that `\d` is one of `\p`'s divisors. In that case, `\p` is of course not prime and the "prime" boolean is set to false with `\primefalse`. Since `\trialdivision` is actually located in the group surrounding the body of the `\testprimality` macro, and since the "prime" is needed outside `\testprimality`, the group must once again be transcended and the "prime" assignment must be forced to be global. Hence:

```
\global\primefalse
```

Finally, in the case where `\d` divides `\p`, we set `\unknownfalse`, which has as sole effect of causing the loop to end:

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

that is, no other divisor is tested. One can observe that there is no `\global` in front of `\unknownfalse`, because `\ifunknown` is used within and not outside the group.

If `\p` is not found again after the multiplication, it means that `\d` is not a divisor of `\p`. At that time, we had

- either `\a` $\leq$ `\d`, and therefore `\a` $<$ `\d` (otherwise `\p` would have been found after the multiplication), and hence `\unknownfalse`, therefore the loop

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

  stops and since this happens in the context

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

  where "`prime`" had been set to true, we conclude naturally that, no divisor having been found up to `\p`'s square root, `\p` is prime.

  Therefore, at the end of `\testprimality`'s call, `\ifprime` succeeds and `\p` is printed.
- or `\a` $>$ `\d`: in that case, we know nothing more, `\unknowntrue`, and the next divisor must be tried.

Conclusion

This ends the explanation of these macros, apart from a few subtleties which were not mentioned.

It takes TEX a lot of time to do complex operations such as the ones described. In order to execute`\primes{30}`, TEX spends more time than it needs on average to typeset a whole page with plain TEX. `\trialdivision` is expanded 132 times. With `\primes{1000}` there are 41331 expansions and with `\primes{10000}` there are 1441624 expansions.

It should be stressed that the previous macros are given in *The TEXbook* (Knuth, 1984, pp. 218–219), with the following lines as the only explanation:

> 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.

TEX's programming language is quite peculiar and we gave only a glimpse of it. The interested reader should dive into TEX's "bible", namely Donald Knuth's *TEXbook* (Knuth, 1984).

Acknowledgments

I would like to thank an anonymous referee for noticing an important error in the French version of the article.

References

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

◇ Denis Roegel
LORIA
Campus scientifique
BP 239
54506 Vandœuvre-lès-Nancy cedex
FRANCE
roegel@loria.fr
http://www.loria.fr/~roegel/