Representation of macro parameters

Hans Hagen

When TEX reads input it either does something directly, like setting a register, loading a font, turning a character into a glyph node, packaging a box, or it sort of collects tokens and stores them somehow, in a macro (definition), in a token register, or someplace temporary to inject them into the input later. Here we'll be discussing macros, which have a special token list containing the preamble defining the arguments and a body doing the real work. For instance when you say:

```
\def\foo#1#2{#1 + #2 + #1 + #2}
```

the macro `\foo` is stored in such a way that it knows how to pick up the two arguments and when expanding the body, it will inject the collected arguments each time a reference like `#1` or `#2` is seen. In fact, quite often, TEX pushes a list of tokens (like an argument) in the input stream and then detours in taking tokens from that list. Because TEX does all its memory management itself the price of all that copying is not that high, although during a long and more complex run the individual tokens that make the forward linked list of tokens get scattered in token memory and memory access is still the bottleneck in processing.

A somewhat simplified view of how a macro like this gets stored is the following:

```
hash entry "foo" with property "macro call" =>

    match (# property stored)
    match (# property stored)
    end of match

    match reference 1
    other character +
    match reference 2
    other character +
    match reference 1
    other character +
    match reference 2
```

When a macro gets expanded, the scanner first collects all the passed arguments and then pushes those (in this case two) token lists on the parameter stack. Keep in mind that due to nesting many kinds of stacks play a role. When the body gets expanded and a reference is seen, the argument that it refers to gets injected into the input, so imagine that we have this definition:

```
\foo#1#2{\ifdim\dimen0=0pt #1\else #2\fi}
```

and we say:

```
\foo{yes}{no}
```

then it's as if we had typed:

```
\ifdim\dimen0=0pt yes\else no\fi
```

So, you'd better not have something in the arguments that messes up the condition parser! From the perspective of an expansion machine it all makes sense. But it also means that when arguments are not used, they still get parsed and stored. Imagine using this one:

```
\def\foo#1{\iffalse#1\oof#1\oof#1\oof#1\oof#1\fi}
```

When TEX sees that the condition is false it will enter a fast scanning mode where it only looks at condition related tokens, so even if `\oof` is not defined this will work ok:

```
\foo{!}
```

But when we say this:

```
\foo{\else}
```

it will bark! This is because each `#1` reference will be resolved, so we effectively have (line breaks in the following are editorial)

```
\def\foo#1{\iffalse\else\oof\else\oof\else\oof
                                   \else\oof\else\fi}
```

which is not good. On the other hand, since no expansion takes place in quick parsing mode, this will work:

```
\def\oof{\else}
\foo\oof
```

which to TEX looks like:

```
\def\foo#1{\iffalse\oof\oof\oof\oof\oof\oof\oof
                                          \oof\oof\fi}
```

So, a reference to an argument effectively is just a replacement. As long as you keep that in mind, and realize that while TEX is skipping 'if' branches nothing gets expanded, you're okay.

Most users will associate the `#` character with macro arguments or preambles in low level alignments, but since most macro packages provide a higher level set of table macros the latter is less well known. But, as often with characters in TEX, you can do magic things:

```
\catcode`?=\catcode`#
\def\foo #1#2?3{?1?2?3}
  \meaning\foo\space=>\foo{1}{2}{3}\par
\def\foo ?1#2?3{?1?2?3}
  \meaning\foo\space=>\foo{1}{2}{3}\par
\def\foo ?1?2#3{?1?2?3}
  \meaning\foo\space=>\foo{1}{2}{3}\par
```

Here the question mark also indicates a macro argument. However, when expanded we see this as result:

```
macro:#1#2?3->?1?2?3 =>123
macro:?1#2?3->?1?2?3 =>123
macro:?1?2#3->#1#2#3 =>123
```

The last used argument signal character (officially called a match character, here we have two that fit that category, `#` and `?`) is used in the serialization! Now, there is an interesting aspect here. When TEX stores the preamble, as in our first example:

```
    match (# property stored)
    match (# property stored)
    end of match
```

the property is stored, so in the later example we get:

```
    match (# property stored)
    match (# property stored)
    match (? property stored)
    end of match
```

But in the macro body the number is stored instead, because we need it as reference to the parameter, so when that bit gets serialized TEX (or more accurately: LuaTEX, which is what we're using here) doesn't know what specific signal was used. When the preamble is serialized it does keep track of the last so-called match character. This is why we see this inconsistency in rendering.

A simple solution would be to store the used signal for the match argument, which probably only takes a few lines of extra code (using a nine integer array instead of a single integer), and use that instead. I'm willing to see that as a bug in LuaTEX but when I ran into it I was playing with something else: adding the ability to prevent storing unused arguments. But the resulting confusion can make one wonder why we do not always serialize the match character as `#`.

It was then that I noticed that the preamble stored the match tokens and not the number and that TEX in fact assumes that no mixture is used. And, after prototyping that in itself trivial change I decided that in order to properly serialize this new feature it also made sense to always serialize the match token as `#`. I simply prefer consistency over confusion and so I caught two flies in one stroke. The new feature is indicated with a `\#0` parameter:

```
\bgroup
\catcode`?=\catcode`#
\def\foo ?1?0?3{?1?2?3}
  \meaning\foo\space=>\foo{1}{2}{3}\crlf
\def\foo ?1#0?3{?1?2?3}
  \meaning\foo\space=>\foo{1}{2}{3}\crlf
\def\foo #1#2?3{?1?2?3}
  \meaning\foo\space=>\foo{1}{2}{3}\crlf
\def\foo ?1#2?3{?1?2?3}
  \meaning\foo\space=>\foo{1}{2}{3}\crlf
\def\foo ?1?2#3{?1?2?3}
  \meaning\foo\space=>\foo{1}{2}{3}\crlf
\egroup
```

shows us:

```
macro:#1#0#3->#1#2#3 =>13
macro:#1#0#3->#1#2#3 =>13
macro:#1#2#3->#1#2#3 =>123
macro:#1#2#3->#1#2#3 =>123
macro:#1#2#3->#1#2#3 =>123
```

So, what is the rationale behind this new `#0` variant? Quite often you don't want to do something with an argument at all. This happens when a macro acts upon for instance a first argument and then expands another macro that follows up but only deals with one of many arguments and discards the rest. Then it makes no sense to store unused arguments. Keep in mind that in order to use it more than once an argument does need to be stored, because the parser only looks forward. In principle there could be some optimization in case the tokens come from macros but we leave that for now. So, when we don't need an argument, we can avoid storing it and just skip over it. Consider the following:

```
\def\foo#1{\ifnum#1=1 \expandafter\fooone
          \else\expandafter\footwo\fi}
\def\fooone#1#0{#1}
\def\footwo#0#2{#2}
\foo{1}{yes}{no}
\foo{0}{yes}{no}
```

We get:
yes no

Just for the record, tracing of a macro shows that indeed there is no argument stored:

```
\def\foo#1#0#3{....}
\foo{11}{22}{33}
\foo #1#0#3->....
#1<-11
#2<-
#3<-33
```

Now, you can argue, what is the benefit of not storing tokens? As mentioned above, the TeX engines do their own memory management.[1] This has large benefits in performance especially when one keeps in mind that tokens get allocated and are recycled constantly (take only lookahead and push back).

However, even if this means that storing a couple of unused arguments doesn't put much of a dent in performance, it does mean that a token sits somewhere in memory and that this bit of memory needs to get accessed. Again, this is no big deal on a computer where a TeX job can take one core and basically is the only process fighting for CPU cache usage. But less memory access might be more relevant in a scenario of multiple virtual machines running on the same hardware or multiple TeX processes on one machine. I didn't carefully measure that so I might be wrong here. Anyway, it's always good to avoid moving around data when there is no need for it.

[1] An added benefit is that dumping and undumping is relatively efficient too.

Just to temper expectations with respect to performance, here are some examples:

```
\catcode`!=9 % ignore this character
\firstoftwoarguments
  {!!!!!!!!!!!!!!!!!!!}{!!!!!!!!!!!!!!!!!!!}
\secondoftwoarguments
  {!!!!!!!!!!!!!!!!!!!}{!!!!!!!!!!!!!!!!!!!}
\secondoffourarguments
  {!!!!!!!!!!!!!!!!!!!}{!!!!!!!!!!!!!!!!!!!}
  {!!!!!!!!!!!!!!!!!!!}{!!!!!!!!!!!!!!!!!!!}
```

In ConTeXt we define these macros as follows:

```
\def\firstoftwoarguments      #1#2{#1}
\def\secondoftwoarguments     #1#2{#2}
\def\secondoffourarguments#1#2#3#4{#2}
```

The performance of 2 million expansions is the following (probably half or less on a more modern machine):

macro	total	step
`\firstoftwoarguments`	0.245	0.000000123
`\secondoftwoarguments`	0.251	0.000000126
`\secondoffourarguments`	0.390	0.000000195

But we could use this instead:

```
\def\firstoftwoarguments      #1#0{#1}
\def\secondoftwoarguments     #0#2{#2}
\def\secondoffourarguments#0#2#0#0{#2}
```

which gives:

macro	total	step
`\firstoftwoarguments`	0.229	0.000000115
`\secondoftwoarguments`	0.236	0.000000118
`\secondoffourarguments`	0.323	0.000000162

So, no impressive differences, especially when one considers that when that many expansions happen in a run, getting the document itself rendered plus expanding real arguments (not something defined to be ignored) will take way more time compared to this. I always test an extension like this on the test suite[2] as well as the LuaMetaTeX manual (which takes about 11 seconds) and although one can notice a little gain, it makes more sense not to play music on the same machine as we run the TeX job, if gaining milliseconds is that important. But, as said, it's more about unnecessary memory access than about CPU cycles.

This extension is downward compatible and its overhead can be neglected. Okay, the serialization

[2] Currently some 1600 files that take 24 minutes plus or minus 30 seconds to process on a high end 2013 laptop. The 260 page manual with lots of tables, verbatim and MetaPost images takes around 11 seconds. A few milliseconds more or less don't really show here. I only time these runs because I want to make sure that there are no dramatic consequences.

now always uses # but it was inconsistent before, so I'm willing to sacrifice that (and I'm pretty sure no ConTEXt user cares or will even notice). Also, it's only in LuaMetaTEX (for now) so that other macro packages don't suffer from this patch. The few cases where ConTEXt can benefit from it are easy to isolate for MkIV and LMTX so we can support LuaTEX and LuaMetaTEX.

I mentioned LuaTEX and how it serializes, but for the record, let's see how pdfTEX, which is very close to original TEX in terms of source code, does it. If we have this input:

```
\catcode`D=\catcode`#
\catcode`O=\catcode`#
\catcode`N=\catcode`#
\catcode`-=\catcode`#
\catcode`K=\catcode`#
\catcode`N=\catcode`#
\catcode`U=\catcode`#
\catcode`T=\catcode`#
\catcode`H=\catcode`#
\def\dek D1O2N3-4K5N6U7T8H9{#1#2#3 #4#5#6#7#8#9}
{\meaning\dek \tracingall \dek don{}knuth}
```

The meaning gets typeset as (again, line break is editorial):

```
macro:D1O2N3-4K5N6U7T8H9->H1H2H3 H4H5H6H7
H8H9don knuth
```

while the tracing reports:

```
\dek D1O2N3-4K5N6U7T8H9->H1H2H3 H4H5H6H7H8H9
D1<-d
O2<-o
N3<-n
-4<-
K5<-k
N6<-n
U7<-u
T8<-t
H9<-h
```

The reason for the difference, as mentioned, is that the tracing uses the template and therefore uses the stored match token, while the meaning uses the reference match tokens that carry the number and at that time has no access to the original match token. Keeping track of that for the sake of tracing would not make sense anyway. So, traditional TEX, which is what pdfTEX is very close to, uses the last used match token, the H. Maybe this example can convince you that dropping that bit of log related compatibility is not that much of a problem. I just tell myself that I turned an unwanted side effect into a new feature.

A few side notes

The fact that characters can be given a special meaning is one of the charming properties of TEX. Take these two cases:

```
\bgroup\catcode`\&=5 &\egroup
\bgroup\catcode`\!=5 !\egroup
```

In both lines there is now an alignment character used outside an alignment. And, in both cases the error message is similar:

```
! Misplaced alignment tab character &
! Misplaced alignment tab character !
```

So, indeed the right character is shown in the message. But, as soon as you ask for help, there is a difference: in the first case the help is specific for a tab character, but in the second case a more generic explanation is given. Just try it.

The reason is an explicit check for the ampersand being used as tab character. Such is the charm of TEX. I'll probably opt for a trivial change to be consistent here, although in ConTEXt the ampersand is just an ampersand so no user will notice.

There are a few more places where, although in principle any character can serve any purpose, there are hard coded assumptions, like $ being used for math, so a missing dollar is reported, even if math started with another character being used to enter math mode. This makes sense because there is no urgent need to keep track of what specific character was used for entering math mode. An even stronger argument could be that TEXies expect dollars to be used for that purpose. Of course this works fine:

```
\catcode`€=\catcode`$
€ \sqrt{x^3} €
```

But when we forget an € we get messages like:

```
! Missing $ inserted
```

or more generic:

```
! Extra }, or forgotten $
```

which is definitely a confirmation of “America first”. Of course we can compromise in display math because this is quite okay:

```
\catcode`€=\catcode`$
$€ \sqrt{x^3} €$
```

unless of course we forget the last dollar in which case we are told that

```
! Display math should end with $$
```

so no matter what, the dollar wins. Given how ugly the Euro sign looks I can live with this, although I always wonder what character would have been taken if TEX was developed in another country.

◇ Hans Hagen
http://pragma-ade.com