Keyword scanning

Hans Hagen

Some primitives in TEX can take one or more optional keywords and/or keywords followed by one or more values. In traditional TEX it concerns a handful of primitives, in pdfTEX there are plenty of backend-related primitives, LuaTEX introduced optional keywords to some math constructs and attributes to boxes, and LuaMetaTEX adds some more too. The keyword scanner in TEX is rather special. Keywords are used in cases like:

```
\hbox spread 10cm {...}
\advance\scratchcounter by 10
\vrule width 3cm height 1ex
```

Sometimes there are multiple keywords, as with rules, in which case you can imagine a case like:

```
\vrule width 3cm depth 1ex width 10cm depth 0ex
      height 1ex\relax
```

Here we add a `\relax` to end the scanning. If we don't do that and the rule specification is followed by arbitrary (read: unpredictable) text, the next word might be a valid keyword and when followed by a dimension (unlikely) it will happily be read as a directive, or when not followed by a dimension an error message will show up. Sometimes the scanning is more restricted, as with glue where the optional `plus` and `minus` are to come in that order, but when missing, again a word from the text can be picked up if one doesn't explicitly end with a `\relax` or some other token.

```
\scratchskip = 10pt plus 10pt minus 10pt % okay
\scratchskip = 10pt plus 10pt            % okay
\scratchskip = 10pt minus 10pt           % okay
\scratchskip = 10pt plus whatever        % error
  % typesets "plus 10pt":
\scratchskip = 10pt minus 10pt plus 10pt
```

The scanner is case insensitive, so the following specifications are all valid:

```
\hbox To 10cm {To}
\hbox TO 10cm {TO}
\hbox tO 10cm {tO}
\hbox to 10cm {to}
```

It happens that keywords are always simple English words so the engine uses a cheap check deep down, just offsetting to uppercase, but of course that will not work for arbitrary UTF-8 (as used in LuaTEX) and it's also unrelated to the upper- and lowercase codes as TEX knows them.

The above lines scan for the keyword `to` and after that for a dimension. While keyword scanning is case tolerant, dimension scanning is period tolerant:

```
\hbox to 10cm   {10cm}
\hbox to 10.0cm {10.0cm}
\hbox to .0cm   {.0cm}
\hbox to .cm    {.cm}
\hbox to 10.cm  {10.cm}
```

These are all valid and according to the specification; even the single period is okay, although it looks funny. It would not be hard to intercept that but I guess that when TEX was written anything that could harm performance was taken into account. One can even argue for cases like:

```
\hbox to \first.\second cm {.cm}
```

Here `\first` and/or `\second` can be empty. Most users won't notice these side effects of scanning numbers anyway.

Pushing back tokens

The reason for writing up any discussion of keywords is the following. Optional keyword scanning is kind of costly, not so much now, but more so decades ago (which led to some interesting optimizations, as we'll see). For instance, in the first line below, there is no keyword. The scanner sees a `1` and it not being a keyword, pushes that character back in the input.

```
\advance\scratchcounter 10
\advance\scratchcounter by 10
```

In the case of:

```
\scratchskip 10pt plux
```

it has to push back the four scanned tokens `plux`. Now, in the engine there are lots of cases where lookahead happens and when a condition is not satisfied, the just-read token is pushed back. Incidentally, when picking up the next token triggered some expansion, it's not the original next token that gets pushed back, but the first token seen after the expansion. Pushing back tokens is not that inefficient, although it involves allocating a token and pushing and popping input stacks (we're talking of a mix of reading from file, token memory, Lua prints, etc.) but it always takes a little time and memory. In LuaTEX there are more keywords for boxes, and there we have loops too: in a box specification one or more optional attributes are scanned before the optional `to` or `spread`, so again there can be push back when no more `attr` are seen.

```
\hbox attr 1 98 attr 2 99 to 1cm{...}
```

In LuaMetaTEX there is even more optional keyword scanning, but we leave that for now and just show one example:

```
\hbox spread 10em {\hss
 \hbox orientation 0 yoffset 1mm to 2em  {up}\hss
 \hbox                           to 2em{here}\hss
 \hbox orientation 0 xoffset-1mm to 2em{down}\hss
}
```

Although one cannot mess too much with these low-level scanners there was room for some optimization, so the penalty we pay for more keyword scanning in LuaMetaTEX is not that high. (I try to compensate when adding features that have a possible performance hit with some gain elsewhere.)

It will be no surprise that there can be interesting side effects to keyword scanning. For instance, using the two character keyword `by` in an `\advance` can be more efficient because nothing needs to be pushed back. The same is true for the sometimes optional equal:

```
\scratchskip = 10pt
```

Similar impacts on efficiency can be found in the way the end of a number is seen, basically anything not resolving to a number (or digit). (For these, assume a following token will terminate the number if needed; we're focusing on the spaces here.)

```
\scratchcounter 10% space not seen, ends \cs
\scratchcounter =10% no push back of optional =
\scratchcounter = 10% extra optional space gobble
\scratchcounter = 10 % efficient end of scanning
\scratchcounter = 10\relax % maybe less efficient
```

In the above examples scanning the number involves: skipping over spaces, checking for an optional equal, skipping over spaces, scanning for a sign, checking for an optional octal or hexadecimal trigger (single or double quote character), scanning the number till a non-digit is seen. In the case of dimensions there is fraction scanning as well as unit scanning too.

In any case, the equal is optional and kind of a keyword. Having an equal can be more efficient then not having one, again due to push back in case of no equal being seen, In the process spaces have been skipped, so add to the overhead the scanning for optional spaces. In LuaMetaTEX all that has been optimized a bit. By the way, in dimension scanning `pt` is actually a keyword and as there are several dimensions possible quite some push back can happen there, but we scan for the most likely candidates first.

Catcode surprises

All that said, we're now ready for a surprise. The keyword scanner gets a string that it will test for, say, `to` in case of a box specification. It then will fetch tokens from whatever provides the input. A token encodes a so-called command and a character and can be related to a control sequence. For instance, the character `t` becomes a letter command with related value 116. So, we have three properties: the command code, the character code and the control sequence code. Now, instead of checking if the command code is a letter or other character (two checks) a fast check happens for the control sequence code being zero. If that is the case, the character code is compared. In practice that works out well because the characters that make up a keyword are in the range 65–90 and 97–122, and all other character codes are either below that (the ones that relate to primitives where the character code is actually a subcommand of a limited range) or much larger numbers that, for instance, indicate an entry in some array, where the first useful index is above the mentioned ranges.

The surprise is in the fact that there is no checking for letters or other characters, so this is why the following code will work too:[1]

```
\catcode`O= 1 \hbox tO 10cm {...}% { begingroup
\catcode`O= 2 \hbox tO 10cm {...}% } endgroup
\catcode`O= 3 \hbox tO 10cm {...}% $ mathshift
\catcode`O= 4 \hbox tO 10cm {...}% & alignment
\catcode`O= 6 \hbox tO 10cm {...}% # parameter
\catcode`O= 7 \hbox tO 10cm {...}% ^ superscript
\catcode`O= 8 \hbox tO 10cm {...}% _ subscript
\catcode`O=11 \hbox tO 10cm {...}%   letter
\catcode`O=12 \hbox tO 10cm {...}%   other
```

In the first line, if we changed the catcode of `T` (instead of `O`), it gives an error because TEX sees a begin group character (category code 1) and starts the group, but as a second character in a keyword (`O`) it's okay because TEX will not look at the category code.

Of course only the cases `11` and `12` make sense in practice. Messing with the category codes of regular letters this way will definitely give problems with processing normal text. In a case like:

```
{\catcode `o=3 \hbox to 10cm {oeps}} % \hb
{\catcode `O=3 \hbox to 10cm {Oeps}} % {$eps}
```

we have several issues: the primitive control sequence `\hbox` has an `o` so TEX will stop after `\hb` which can be undefined or a valid macro and what happens next is hard to predict. Using uppercase will work but then the content of the box is bad because there the `O` enters math. Now consider:

```
{\catcode `O=3 \hbox tO 10cm {Oeps Oeps}}
  % {$eps $eps}
```

This will work because there are now two `O`'s in the box, so we have balanced inline math triggers. But how does one explain that to a user? (Who probably doesn't understand where an error message comes from in the first place.) Anyway, this kind of tolerance is still not pretty, so in LuaMetaTEX we now check for the command code and stick to letters

[1] No longer in LuaMetaTEX where we do a bit more robust check.

and other characters. On today's machines (and even on my by now ancient workhorse) the performance hit can be neglected.

In fact, by intercepting the weird cases we also avoid an unnecessary case check when we fall through the zero control sequence test. Of course that also means that the above mentioned category code trickery doesn't work any more: only letters and other characters are now valid in keyword scanning. Now, it can be that some macro programmer actually used those side effects but apart from some macro hacker being hurt because no longer mastering those details can be showed off, it is users that we care more for, don't we?

Current performance

To be sure, the abovementioned performance of keyword and equal scanning is not that relevant in practice. But for the record, here are some timings on a laptop with a i7-3849QM processor using MinGW binaries on a 64-bit Windows 10 system. The times are the averages of five times a million such assignments and advancements.

one million times	terminal	LMTX	LuaTEX
`\advance\scratchctr 1`	space	0.068	0.085
`\advance\scratchctr 1`	`\relax`	0.135	0.149
`\advance\scratchctr by 1`	space	0.087	0.099
`\advance\scratchctr by 1`	`\relax`	0.155	0.161
`\scratchctr 1`	space	0.057	0.096
`\scratchctr 1`	`\relax`	0.125	0.151
`\scratchctr=1`	space	0.063	0.080
`\scratchctr=1`	`\relax`	0.131	0.138

We differentiate here between using a space as terminal or a `\relax`. The latter is a bit less efficient because more code is involved in resolving the meaning of the control sequence (which eventually boils down to nothing) but nevertheless, these are not timings that one can lose sleep over, especially when the rest of a decent TEX run is taken into account. And yes, LuaMetaTEX (LMTX) is a bit faster here than LuaTEX, but I would be disappointed if that weren't the case.

◇ Hans Hagen
http://pragma-ade.com