Tokens in LuaTEX

Hans Hagen

tokenization

Most TEX users only deal with (keyed in) characters and (produced) output. Some will play with boxes, skips and kerns or maybe even leaders (repeated sequences of the former). Others will be grateful that macro package writers take care of such things.

Macro writers on the other hand deal with properties of characters, like catcodes and a truckload of other codes, with lists made out of boxes, skips, kerns and penalties. But even they cannot look much deeper into TEX's internals. Their deeper understanding comes from reading the TEXbook or even looking at the source code.

When someone enters the magic world of TEX and starts asking around a bit, he or she will at some point get confronted with the concept of tokens. A token is what ends up in TEX after characters have entered its machinery. Sometimes it even seems that one is only considered a qualified macro writer if one can talk the right token–speak. So, what are those magic tokens and how can LuaTEX shed light on this?

In a moment we will show examples of how LuaTEX turns characters into tokens, but when looking at those sequences, you need to keep a few things in mind:

- A sequence of characters that starts with an escape symbol (normally this is the backslash) is looked up in the hash table (which relates those names to meanings) and replaced with its reference. Such a reference is much faster than looking up the sequence each time.
- Characters can have special meanings, for instance a dollar is often used to enter and exit math mode, and a percent symbol starts a comment and hides everything following it on the same line. These meanings are determined by the character's catcode.
- All the characters that will end up actually typeset have catcode letter or other assigned. A sequence of items with catcode letter is considered a word and can potentially become hyphenated.

examples

We will now provide a few examples of how TEX sees your input.

```
Hi there!
```

Hi there!

cmd	chr	id	name
letter	72	H	
letter	105	i	
spacer	32		
letter	116	t	
letter	104	h	
letter	101	e	
letter	114	r	
letter	101	e	
other_char	33	!	

Here we see three kinds of tokens. At this stage a space is still recognizable as such, but later this will become a skip. In our current setup, the exclamation mark is not a letter.

```
Hans \& Taco use Lua\TeX \char 33\relax
```

Hans & Taco use LuaTEX!

cmd	chr		id	name
letter	72	H		
letter	97	a		
letter	110	n		
letter	115	s		
spacer	32			
char_given	38		1114152	&
spacer	32			
letter	84	T		
letter	97	a		
letter	99	c		
letter	111	o		
spacer	32			
letter	117	u		
letter	115	s		
letter	101	e		
spacer	32			
letter	76	L		
letter	117	u		
letter	97	a		
call	1554614		1114740	TeX
char_num	0		1115630	char

```
other_char       51  3
other_char       51  3
relax       1114112     1117492  relax
```

Here we see a few new tokens, a char_given and a call. The first represents a \chardef i.e. a reference to a character slot in a font, and the second one a macro that will expand to the TEX logo. Watch how the space after a control sequence is eaten up. The exclamation mark is a direct reference to character slot 33.

```
\noindent {\bf Hans} \par \hbox{Taco} \endgraf
```

Hans
Taco

```
cmd              chr          id  name
start_par          0     1141958  noindent
left_brace       123
call         1650250     1114412  bf
letter            72  H
letter            97  a
letter           110  n
letter           115  s
right_brace      125
spacer            32
par_end      1114112     1114870  par
make_box         122     1115680  hbox
left_brace       123
letter            84  T
letter            97  a
letter            99  c
letter           111  o
right_brace      125
spacer            32
par_end      1114112     1127274  endgraf
```

As you can see, some primitives and macros that are bound to them (like \endgraf) have an internal representation on top of their name.

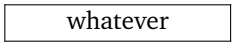
```
before \dimen2=10pt after \the\dimen2
```

before after 10.0pt

```
cmd          chr          id  name
letter        98  b
letter       101  e
letter       102  f
letter       111  o
letter       114  r
letter       101  e
spacer        32
register       1     1117302  dimen
other_char    50  2
other_char    61  =
other_char    49  1
other_char    48  0
letter       112  p
letter       116  t
spacer        32
letter        97  a
letter       102  f
letter       116  t
letter       101  e
letter       114  r
spacer        32
the            0     1114887  the
register       1     1117302  dimen
other_char    50  2
```

As you can see, registers are not explicitly named, one needs the associated register code to determine it's character (a dimension in our case).

```
before \inframed[width=3cm]{whatever} after
```

before  after

```
cmd              chr          id  name
letter            98  b
letter           101  e
letter           102  f
letter           111  o
letter           114  r
letter           101  e
spacer            32
call         1824889     3226639  inframed
other_char        91  [
letter           119  w
letter           105  i
letter           100  d
letter           116  t
letter           104  h
other_char        61  =
other_char        51  3
letter            99  c
letter           109  m
other_char        93  ]
left_brace       123
letter           119  w
letter           104  h
letter            97  a
letter           116  t
letter           101  e
letter           118  v
letter           101  e
letter           114  r
right_brace      125
spacer            32
```

```
letter            97  a
letter           102  f
letter           116  t
letter           101  e
letter           114  r
```

As you can see, even when control sequences are collapsed into a reference, we still end up with many tokens, and because each token has three properties (cmd, chr and id) in practice we end up with more memory used after tokenization.

```
compound|-|word
```

compound-word

```
cmd              chr      id  name
letter            99  c
letter           111  o
letter           109  m
letter           112  p
letter           111  o
letter           117  u
letter           110  n
letter           100  d
call         1869296     125  |
other_char        45  -
call         1869296     125  |
letter           119  w
letter           111  o
letter           114  r
letter           100  d
```

This example uses an active character to handle compound words (a ConTEXt feature).

```
hm, \directlua 0 { tex.sprint("Hello World") }
```

hm, Hello World!

```
cmd           chr          id  name
letter        104  h
letter        109  m
other_char     44  ,
spacer         32
convert        23     1166957  directlua
other_char     48  0
spacer         32
left_brace    123
spacer         32
letter        116  t
letter        101  e
letter        120  x
other_char     46  .
letter        115  s
letter        112  p
letter        114  r
letter        105  i
letter        110  n
letter        116  t
other_char     40  (
other_char     34  "
letter         72  H
letter        101  e
letter        108  l
letter        108  l
letter        111  o
spacer         32
letter         87  W
letter        111  o
letter        114  r
letter        108  l
letter        100  d
other_char     33  !
other_char     34  "
other_char     41  )
spacer         32
right_brace   125
```

The previous example shows what happens when we include a bit of lua code ... it is just seen as regular input, but when the string is passed to Lua, only the chr property is passed, so we no longer can distinguish between letters and other characters.

A macro definition converts to tokens as follows.

```
[B][A]
```

```
cmd            chr          id  name
def              0     1114818  def
undefined_cs           1115536  Test
mac_param       35
other_char      49  1
mac_param       35
other_char      50  2
left_brace     123
other_char      91  [
mac_param       35
other_char      50  2
other_char      93  ]
other_char      91  [
mac_param       35
other_char      49  1
other_char      93  ]
right_brace    125
spacer          32
undefined_cs           1115536  Test
left_brace     123
letter          65  A
right_brace    125
```

```
left_brace    123
letter         66  B
right_brace   125
```

As we already mentioned, a token has three properties. More details can be found in the reference manual so we will not go into much detail here. A stupid callback looks like:

```
callback.register('token_filter',token.get_next)
```

In principle you can call `token.get_next` anytime you want to intercept a token. In that case you can feed back tokens into TEX by using a trick like:

```
function tex.printlist(data)
   callback.register('token_filter',function ()
       callback.register('token_filter', nil)
       return data
    end)
end
```

Another example of usage is:

```
callback.register('token_filter', function ()
    local t = token.get_next
    local cmd, chr, id = t[1], t[2], t[3]
    -- do something with cmd, chr, id
    return { cmd, chr, id }
end)
```

There is a whole repertoire of related functions, one is `token.create`, which can be used as:

```
tex.printlist{
    token.create("hbox"),
    token.create(utf.byte("{"),  1),
    token.create(utf.byte("?"), 12),
    token.create(utf.byte("}"),  2),
}
```

This results in: ?

While playing with this we made a few auxiliary functions which permit things like:

```
tex.printlist (
  table.unnest ( {
    tokens.hbox,
    tokens.bgroup,
    tokens.letters("12345"),
    tokens.egroup,
  } ) )
```

Unnesting is needed because the result of the `letters` call is a table, and the `printlist` function wants a flattened table.

The result looks like: 12345

```
cmd          chr          id  name
make_box     122     1115680  hbox
left_brace   123
letter        49  1
letter        50  2
letter        51  3
letter        52  4
letter        53  5
right_brace  125
```

In practice, manipulating tokens or constructing lists of tokens this way is rather cumbersome, but at least we now have some kind of access, if only for illustrative purposes.

```
\hbox{12345\hbox{54321}}
```

can also be done by saying:

```
tex.sprint("\\hbox{12345\\hbox{54321}}")
```

or under ConTEXt's basic catcode regime:

```
tex.sprint(tex.ctxcatcodes,
           "\\hbox{12345\\hbox{54321}}")
```

If you like it the hard way:

```
tex.printlist ( table.unnest ( {
  tokens.hbox,
    tokens.bgroup,
    tokens.letters("12345"),
    tokens.hbox,
      tokens.bgroup,
      tokens.letters(string.reverse("12345")),
      tokens.egroup,
    tokens.egroup
} ) )
```

This method may attract those who dislike the traditional TEX syntax for doing the same thing. Okay, a carefull reader will notice that reversing the string in TEX takes a bit more trickery, so ...

Hans Hagen