Tokens in \LaTeX

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 \LaTeX shed light on this?

In a moment we will show examples of how \LaTeX 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!

<table>
<thead>
<tr>
<th>cmd</th>
<th>chr</th>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>letter</td>
<td>72</td>
<td></td>
<td>H</td>
</tr>
<tr>
<td>letter</td>
<td>105</td>
<td></td>
<td>i</td>
</tr>
<tr>
<td>spacer</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>116</td>
<td></td>
<td>t</td>
</tr>
<tr>
<td>letter</td>
<td>104</td>
<td></td>
<td>h</td>
</tr>
<tr>
<td>letter</td>
<td>101</td>
<td></td>
<td>e</td>
</tr>
<tr>
<td>letter</td>
<td>114</td>
<td></td>
<td>r</td>
</tr>
<tr>
<td>letter</td>
<td>101</td>
<td></td>
<td>e</td>
</tr>
<tr>
<td>other_char</td>
<td>33</td>
<td></td>
<td>!</td>
</tr>
</tbody>
</table>

Hi there!

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 \LaTeX \char 33\relax

Hans \& Taco use \LaTeX!

<table>
<thead>
<tr>
<th>cmd</th>
<th>chr</th>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>letter</td>
<td>72</td>
<td></td>
<td>H</td>
</tr>
<tr>
<td>letter</td>
<td>97</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>letter</td>
<td>110</td>
<td></td>
<td>n</td>
</tr>
<tr>
<td>letter</td>
<td>115</td>
<td></td>
<td>s</td>
</tr>
<tr>
<td>spacer</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>char_given</td>
<td>38</td>
<td>1114152</td>
<td>&amp;</td>
</tr>
<tr>
<td>spacer</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>84</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>letter</td>
<td>97</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>letter</td>
<td>99</td>
<td></td>
<td>c</td>
</tr>
<tr>
<td>letter</td>
<td>111</td>
<td></td>
<td>o</td>
</tr>
<tr>
<td>spacer</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>117</td>
<td></td>
<td>u</td>
</tr>
<tr>
<td>letter</td>
<td>115</td>
<td></td>
<td>s</td>
</tr>
<tr>
<td>letter</td>
<td>101</td>
<td></td>
<td>e</td>
</tr>
<tr>
<td>spacer</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>76</td>
<td></td>
<td>L</td>
</tr>
<tr>
<td>letter</td>
<td>117</td>
<td></td>
<td>u</td>
</tr>
<tr>
<td>letter</td>
<td>97</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>call</td>
<td>1554614</td>
<td>1114740</td>
<td>TeX</td>
</tr>
<tr>
<td>char_num</td>
<td>0</td>
<td>1115630</td>
<td>char</td>
</tr>
</tbody>
</table>
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

\begin{verbatim}
Hans
Taco

\begin{verbatim}
cmd  chr  id  name
start_par  0  1141958 noindent
left_brace 123
\endgraf

\begin{verbatim}
cmd  chr  id  name
left_brace 123
letter  72 H
letter  97 a
letter 110 n
letter 115 s
right_brace 125
spacer  32
par_end 1114112 1114870 par
par_end 1114112 1127274 endgraf

\begin{verbatim}
cmd  chr  id  name
letter  98 b
letter 101 e
letter 102 f
letter 110 o
letter 114 r
spacer  32
call 1824889 3226639 inframed
other_char  91 \[
other_char  93 \]
left_brace 123
letter  99 c
letter 109 m
other_char  93 ]
left_brace 123
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 ]
letter  98 b
letter 101 e
letter 102 f
letter 110 o
letter 114 r
right_brace 125
spacer  32
other_char  61 =
other_char  49 1
other_char  48 0
letter  112 p
letter  116 t
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
\end{verbatim}
\end{verbatim}

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).

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

\begin{verbatim}
cmd  chr  id  name
letter  98 b
letter 101 e
letter 102 f
letter 110 o
letter 114 r
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
other_char  61 =
other_char  49 1
other_char  48 0
letter  112 p
letter  116 t
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
\end{verbatim}
\end{verbatim}

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

\begin{verbatim}
before \dimen2=10pt after \the\dimen2

\begin{verbatim}
cmd  chr  id  name
letter  98 b
letter 101 e
letter 102 f
letter 110 o
letter 114 r
spacer  32
register  1  1117302 dimen
other_char  50 2
\end{verbatim}
\end{verbatim}

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

\begin{verbatim}
before after 10.0pt

\begin{verbatim}
cmd  chr  id  name
letter  98 b
letter 101 e
letter 102 f
letter 110 o
letter 114 r
letter 101 e
spacer  32
register  1  1117302 dimen
other_char  50 2
\end{verbatim}
\end{verbatim}

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

\begin{verbatim}
before after 10.0pt

\begin{verbatim}
cmd  chr  id  name
letter  98 b
letter 101 e
letter 102 f
letter 110 o
letter 114 r
letter 101 e
spacer  32
register  1  1117302 dimen
other_char  50 2
\end{verbatim}
\end{verbatim}

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

\begin{verbatim}
before after 10.0pt

\begin{verbatim}
cmd  chr  id  name
letter  98 b
letter 101 e
letter 102 f
letter 110 o
letter 114 r
letter 101 e
spacer  32
register  1  1117302 dimen
other_char  50 2
\end{verbatim}
\end{verbatim}

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

\begin{verbatim}
before after 10.0pt

\begin{verbatim}
cmd  chr  id  name
letter  98 b
letter 101 e
letter 102 f
letter 110 o
letter 114 r
letter 101 e
spacer  32
register  1  1117302 dimen
other_char  50 2
\end{verbatim}
\end{verbatim}

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

\begin{verbatim}
before after 10.0pt

\begin{verbatim}
cmd  chr  id  name
letter  98 b
letter 101 e
letter 102 f
letter 110 o
letter 114 r
letter 101 e
spacer  32
register  1  1117302 dimen
other_char  50 2
\end{verbatim}
\end{verbatim}

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

\begin{verbatim}
before after 10.0pt

\begin{verbatim}
cmd  chr  id  name
letter  98 b
letter 101 e
letter 102 f
letter 110 o
letter 114 r
letter 101 e
spacer  32
register  1  1117302 dimen
other_char  50 2
\end{verbatim}
\end{verbatim}

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

<table>
<thead>
<tr>
<th>cmd</th>
<th>chr</th>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>letter</td>
<td>97</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>102</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>116</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>101</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>114</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>105</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>110</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>116</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>other_char</td>
<td>40</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>other_char</td>
<td>34</td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>72</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>101</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>108</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>108</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>111</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>spacer</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>87</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>111</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>114</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>108</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>100</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>other_char</td>
<td>33</td>
<td>!</td>
<td></td>
</tr>
<tr>
<td>other_char</td>
<td>34</td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>other_char</td>
<td>41</td>
<td>)</td>
<td></td>
</tr>
<tr>
<td>spacer</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>right_brace</td>
<td>125</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

hm, Hello World!

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

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.

[BA]

<table>
<thead>
<tr>
<th>cmd</th>
<th>chr</th>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>def</td>
<td>0</td>
<td>1114818</td>
<td>def</td>
</tr>
<tr>
<td>undefined_cs</td>
<td>1115536</td>
<td>Test</td>
<td></td>
</tr>
<tr>
<td>mac_param</td>
<td>35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>other_char</td>
<td>49</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>mac_param</td>
<td>35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>other_char</td>
<td>50</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>left_brace</td>
<td>123</td>
<td></td>
<td></td>
</tr>
<tr>
<td>other_char</td>
<td>91</td>
<td>[</td>
<td></td>
</tr>
<tr>
<td>mac_param</td>
<td>35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>other_char</td>
<td>50</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>other_char</td>
<td>93</td>
<td>]</td>
<td></td>
</tr>
<tr>
<td>other_char</td>
<td>91</td>
<td>[</td>
<td></td>
</tr>
<tr>
<td>mac_param</td>
<td>35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>other_char</td>
<td>49</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>other_char</td>
<td>93</td>
<td>]</td>
<td></td>
</tr>
<tr>
<td>right_brace</td>
<td>125</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spacer</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>undefined_cs</td>
<td>1115536</td>
<td>Test</td>
<td></td>
</tr>
<tr>
<td>left_brace</td>
<td>123</td>
<td></td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>65</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>right_brace</td>
<td>125</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:

```plaintext
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:

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

Another example of usage is:

```plaintext
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:

```plaintext
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:

```plaintext
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
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:

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

or under Con\TeX{}'s basic catcode regime:

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

If you like it the hard way:

```plaintext
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 careful reader will notice that reversing the string in \TeX{} takes a bit more trickery, so . . .

Hans Hagen