Still tokens: LuaTEX scanners

Hans Hagen

1 Introduction

Tokens are the building blocks of the input for TEX and they drive the process of expansion which in turn results in typesetting. If you want to manipulate the input, intercepting tokens is one approach. Other solutions are preprocessing or writing macros that do something with their picked-up arguments. In ConTEXt MkIV we often forget about manipulating the input but manipulate the intermediate typesetting results instead. The advantage is that only at that moment do you know what you're truly dealing with, but a disadvantage is that parsing the so-called node lists is not always efficient and it can even be rather complex, for instance in math. It remains a fact that until LuaTEX version 0.80 ConTEXt hardly used the token interface.

In version 0.80 a new scanner interface was introduced, demonstrated by Taco Hoekwater at the ConTEXt conference 2014. Luigi Scarso and I integrated that code and I added a few more functions. Eventually the team will kick out the old token library and overhaul the input-related code in LuaTEX, because no callback is needed any more (and also because the current code still has traces of multiple Lua instances). This will happen stepwise to give users who use the old mechanism an opportunity to adapt.

Here I will show a bit of the new token scanners and explain how they can be used in ConTEXt. Some of the additional scanners written on top of the built-in ones will probably end up in the generic LuaTEX code that ships with ConTEXt.

2 The TEX scanner

The new token scanner library of LuaTEX provides a way to hook Lua into TEX in a rather natural way. I have to admit that I never had any real demand for such a feature but now that we have it, it is worth exploring.

The TEX scanner roughly provides the following sub-scanners that are used to implement primitives: keyword, token, token list, dimension, glue and integer. Deep down there are specific variants for scanning, for instance, font dimensions and special numbers.

A token is a unit of input, and one or more characters are turned into a token. How a character is interpreted is determined by its current catcode. For instance a backslash is normally tagged as 'escape character' which means that it starts a control sequence: a macro name or primitive. This means that once it is scanned a macro name travels as one token through the system. Take this:

```
\def\foo#1{\scratchcounter=123#1\relax}
```

Here TEX scans `\def` and turns it into a token. This particular token triggers a specific branch in the scanner. First a name is scanned with optionally an argument specification. Then the body is scanned and the macro is stored in memory. Because '`\scratchcounter`', '`\relax`', and '`#1`' are turned into tokens, this body has 7 tokens.

When the macro `\foo` is referenced the body gets expanded which here means that the scanner will scan for an argument first and uses that in the replacement. So, the scanner switches between different states. Sometimes tokens are just collected and stored, in other cases they get expanded immediately into some action.

3 Scanning from Lua

The basic building blocks of the scanner are available at the Lua end, for instance:

```
\directlua{print(token.scan_int())} 123
```

This will print `123` to the console. Or, you can store the number and use it later:

```
\directlua{SavedNumber = token.scan_int())} 123
We saved: \directlua{tex.print(SavedNumber)}
```

The number of scanner functions is (on purpose) limited but you can use them to write additional ones as you can just grab tokens, interpret them and act accordingly.

The `scan_int` function picks up a number. This can also be a counter, a named (math) character or a numeric expression. In TEX, numbers are integers; floating-point is not supported naturally. With `scan_dimen` a dimension is grabbed, where a dimen is either a number (float) followed by a unit, a dimen register or a dimen expression (internally, all become integers). Of course internal quantities are also okay. There are two optional arguments, the first indicating that we accept a filler as unit, while the second indicates that math units are expected. When an integer or dimension is scanned, tokens are expanded till the input is a valid number or dimension. The `scan_glue` function takes one optional argument: a boolean indicating if the units are math.

The `scan_toks` function picks up a (normally) brace-delimited sequence of tokens and (LuaTEX 0.80) returns them as a table of tokens. The function `get_token` returns one (unexpanded) token while `scan_token` returns an expanded one.

Because strings are natural to Lua we also have `scan_string`. This one converts a following brace-delimited sequence of tokens into a proper string.

The function `scan_keyword` looks for the given keyword and when found skips over it and returns `true`. Here is an example of usage:[1]

```
function ScanPair()
  local one = 0
  local two = ""
  while true do
    if token.scan_keyword("one") then
      one = token.scan_int()
    elseif token.scan_keyword("two") then
      two = token.scan_string()
    else
      break
    end
  end
  tex.print("one: ",one,"\\par")
  tex.print("two: ",two,"\\par")
end
```

This can be used as:

```
\directlua{ScanPair()}
```

You can scan for an explicit character (class) with `scan_code`. This function takes a positive number as argument and returns a character or `nil`.

1	0	escape
2	1	begingroup
4	2	endgroup
8	3	mathshift
16	4	alignment
32	5	endofline
64	6	parameter
128	7	superscript
256	8	subscript
512	9	ignore
1024	10	space
2048	11	letter
4096	12	other
8192	13	active
16384	14	comment
32768	15	invalid

So, if you want to grab the character you can say:

```
local c = token.scan_code(2^10 + 2^11 + 2^12)
```

In ConTEXt you can say:

```
local c = tokens.scanners.code(
  tokens.bits.space +
  tokens.bits.letter +
  tokens.bits.other
)
```

When no argument is given, the next character with catcode letter or other is returned (if found).

[1] In LuaTEX 0.80 you should use `newtoken` instead of `token`.

In ConTEXt we use the `tokens` namespace which has additional scanners available. That way we can remain compatible. I can add more scanners when needed, although it is not expected that users will use this mechanism directly.

(new)token	tokens.	arguments
	scanners.boolean	
scan_code	scanners.code	(bits)
scan_dimen	scanners.dimension	(fill,math)
scan_glue	scanners.glue	(math)
scan_int	scanners.integer	
scan_keyword	scanners.keyword	
	scanners.number	
scan_token	scanners.token	
scan_tokens	scanners.tokens	
scan_string	scanners.string	
	scanners.word	
get_token	getters.token	
set_macro	setters.macro	(catcodes,cs, str,global)

All except `get_token` (or its alias `getters.token`) expand tokens in order to satisfy the demands.

Here are some examples of how we can use the scanners. When we would call `Foo` with regular arguments we do this:

```
\def\foo#1{%
  \directlua {
    Foo("whatever","#1",{n = 1})
  }
}
```

but when `Foo` uses the scanners it becomes:

```
\def\foo#1{%
  \directlua{Foo()} {whatever} {#1} n {1}\relax
}
```

In the first case we have a function `Foo` like this:

```
function Foo(what,str,n)
  -- do something with these three parameters
end
```

and in the second variant we have (using the `tokens` namespace):

```
function Foo()
  local what = tokens.scanners.string()
  local str  = tokens.scanners.string()
  local n    = tokens.scanners.keyword("n") and
               tokens.scanners.integer() or 0
  -- do something with these three parameters
end
```

The string scanned is a bit special as the result depends on what is seen. Given the following definition:

```
          \def\bar  {bar}
\unexpanded\def\ubar {ubar}
% that's \protected in e-tex etc.
          \def\foo  {foo-\bar-\ubar}
```

```
\def\wrap {{foo-\bar}}
\def\uwrap{{foo-\ubar}}
```

We get:

`foo`	foo
`foo-\bar`	foo-bar
`foo-\ubar`	foo-\ubar
`foo-\bar`	foo-bar
`foo-\ubar`	foo-ubar
`foo$bar$`	foo*bar*
`\foo`	foo-bar-ubar
`\wrap`	foo-bar
`\uwrap`	foo-\ubar

Because scanners look ahead the following happens: when an open brace is seen (or any character marked as left brace) the scanner picks up tokens and expands them unless they are protected; so, effectively, it scans as if the body of an `\edef` is scanned. However, when the next token is a control sequence it will be expanded first to see if there is a left brace, so there we get the full expansion. In practice this is convenient behaviour because the braced variant permits us to pick up meanings honouring protection. Of course this is all a side effect of how TEX scans.[2]

With the braced variant one can of course use primitives like `\detokenize` and `\unexpanded` (in ConTEXt: `\normalunexpanded`, as we already had this mechanism before it was added to the engine).

4 Considerations

Performance-wise there is not much difference between these methods. With some effort you can make the second approach faster than the first but in practice you will not notice much gain. So, the main motivation for using the scanner is that it provides a more TEX-ified interface. When playing with the initial version of the scanners I did some tests with performance-sensitive ConTEXt calls and the difference was measurable (positive) but deciding if and when to use the scanner approach was not easy. Sometimes embedded Lua code looks better, and sometimes TEX code. Eventually we will end up with a mix. Here are some considerations:

- In both cases there is the overhead of a Lua call.
- In the pure Lua case the whole argument is tokenized by TEX and then converted to a string that gets compiled by Lua and executed.
- When the scan happens in Lua there are extra calls to functions but scanning still happens in TEX; some token to string conversion is avoided and compilation can be more efficient.
- When data comes from external files, parsing with Lua is in most cases more efficient than parsing by TEX.
- A macro package like ConTEXt wraps functionality in macros and is controlled by key/value specifications. There is often no benefit in terms of performance when delegating to the mentioned scanners.

Another consideration is that when using macros, parameters are often passed between `{}`:

```
\def\foo#1#2#3%
  {...}
\foo {a}{123}{b}
```

and suddenly changing that to

```
\def\foo{\directlua{Foo()}}
```

and using that as:

```
\foo {a} {b} n 123
```

means that `123` will fail. So, eventually you will end up with something:

```
\def\myfakeprimitive{\directlua{Foo()}}
\def\foo#1#2#3{\myfakeprimitive {#1} {#2} n #3 }
```

and:

```
\foo {a} {b} {123}
```

So in the end you don't gain much here apart from the fact that the fake primitive can be made more clever and accept optional arguments. But such new features are often hidden for the user who uses higher-level wrappers.

When you code in pure TEX and want to grab a number directly you need to test for the braced case; when you use the Lua scanner method you still need to test for braces. The scanners are consistent with the way TEX works. Of course you can write helpers that do some checking for braces in Lua, so there are no real limitations, but it adds some overhead (and maybe also confusion).

One way to speed up the call is to use the `\luafunction` primitive in combinations with predefined functions and although both mechanisms can benefit from this, the scanner approach gets more out of that as this method cannot be used with regular function calls that get arguments. In (rather low level) Lua it looks like this:

```
luafunctions[1] = function()
```

[2] This lookahead expansion can sometimes give unexpected side effects because often TEX pushes back a token when a condition is not met. For instance when it scans a number, scanning stops when no digits are seen but the scanner has to look at the next (expanded) token in order to come to that conclusion. In the process it will, for instance, expand conditionals. This means that intermediate catcode changes will not be effective (or applied) to already-seen tokens that were pushed back into the input. This also happens with, for instance, `\futurelet`.

```
  local a token.scan_string()
  local n token.scan_int()
  local b token.scan_string()
  -- whatever --
end
```

And in TEX:

```
\luafunction1 {a} 123 {b}
```

This can of course be wrapped as:

```
\def\myprimitive{\luafunction1 }
```

5 Applications

The question now pops up: where can this be used? Can you really make new primitives? The answer is yes. You can write code that exclusively stays on the Lua side but you can also do some magic and then print back something to TEX. Here we use the basic token interface, not ConTEXt:

```
\directlua {
local token = newtoken or token
function ColoredRule()
  local w, h, d, c, t
  while true do
    if token.scan_keyword("width") then
      w = token.scan_dimen()
    elseif token.scan_keyword("height") then
      h = token.scan_dimen()
    elseif token.scan_keyword("depth") then
      d = token.scan_dimen()
    elseif token.scan_keyword("color") then
      c = token.scan_string()
    elseif token.scan_keyword("type") then
      t = token.scan_string()
    else
      break
    end
  end
  if c then
    tex.sprint("\\color[",c,"]{"); end
  if t == "vertical" then
    tex.sprint("\\vrule")
  else
    tex.sprint("\\hrule")
  end
  if w then
    tex.sprint("width ",w,"sp"); end
  if h then
    tex.sprint("height ",h,"sp"); end
  if d then
    tex.sprint("depth ",d,"sp"); end
  if c then
    tex.sprint("\\relax}"); end
end
}
```

This can be given a TEX interface like:

```
\def\myhrule{\directlua{ColoredRule()}
             type {horizontal} }
\def\myvrule{\directlua{ColoredRule()}
             type {vertical} }
```

And then used as:

```
\myhrule width \hsize height 1cm color {darkred}
```

giving (grayscaled for *TUGboat* on paper, sorry):

Of course ConTEXt users can use the following commands to color an otherwise-black rule (likewise):

```
\blackrule[width=\hsize,height=1cm,
           color=darkgreen]
```

The official ConTEXt way to define such a new command is the following. The conversion back to verbose dimensions is needed because we pass back to TEX.

```
\startluacode
local myrule = tokens.compile {
  {
    { "width",  "dimension", "todimen" },
    { "height", "dimension", "todimen" },
    { "depth",  "dimension", "todimen" },
    { "color",  "string" },
    { "type",   "string" },
  }
}

interfaces.scanners.ColoredRule = function()
  local t = myrule()
  context.blackrule {
    color  = t.color,
    width  = t.width,
    height = t.height,
    depth  = t.depth,
  }
end
\stopluacode
```

With:

```
\unprotect \let\myrule\scan_ColoredRule \protect
```

and

```
\myrule width \textwidth height 1cm
        color {darkblue} \relax
```

we get:

There are many ways to use the scanners and each has its charm. We will look at some alternatives from the perspective of performance. The timings are more meant as relative measures than absolute

ones. After all it depends on the hardware. We assume the following shortcuts:

```
local scannumber  = tokens.scanners.number
local scankeyword = tokens.scanners.keyword
local scanword    = tokens.scanners.word
```

We will scan for four different keys and values. The number is scanned using a helper `scannumber` that scans for a number that is acceptable for Lua. Thus, `1.23` is valid, as are `0x1234` and `12.12E4`.

```
function getmatrix()
  local sx, sy = 1, 1
  local rx, ry = 0, 0
  while true do
    if scankeyword("sx") then
      sx = scannumber()
    elseif scankeyword("sy") then
      sy = scannumber()
    elseif scankeyword("rx") then
      rx = scannumber()
    elseif scankeyword("ry") then
      ry = scannumber()
    else
      break
    end
  end
  -- action --
end
```

Scanning the following specification 100000 times takes 1.00 seconds:

```
sx 1.23 sy 4.5 rx 1.23 ry 4.5
```

The "tight" case (no spaces) takes 0.94 seconds:

```
sx1.23 sy4.5 rx1.23 ry4.5
```

We can compare this to scanning without keywords. In that case there have to be exactly four arguments. These have to be given in the right order which is no big deal as often such helpers are encapsulated in a user-friendly macro.

```
function getmatrix()
  local sx, sy = scannumber(), scannumber()
  local rx, ry = scannumber(), scannumber()
  -- action --
end
```

As expected, this is more efficient than the previous examples. It takes 0.80 seconds to scan this 100000 times:

```
1.23 4.5 1.23 4.5
```

A third alternative is the following:

```
function getmatrix()
  local sx, sy = 1, 1
  local rx, ry = 0, 0
  while true do
    local kw = scanword()
    if kw == "sx" then
      sx = scannumber()
    elseif kw == "sy" then
      sy = scannumber()
    elseif kw == "rx" then
      rx = scannumber()
    elseif kw == "ry" then
      ry = scannumber()
    else
      break
    end
  end
  -- action --
end
```

Here we scan for a keyword and assign a number to the right variable. This one call happens to be less efficient than calling `scan_keyword` 10 times $(4 + 3 + 2 + 1)$ for the explicit scan. This run takes 1.11 seconds for the next line. The spaces are really needed as words can be anything that has no space.[3]

```
sx 1.23 sy 4.5 rx 1.23 ry 4.5
```

Of course these numbers need to be compared to a baseline of no scanning (i.e. the overhead of a Lua call which here amounts to 0.10 seconds. This brings us to the following table.

keyword checks	0.9 sec
no keywords	0.7 sec
word checks	1.0 sec

The differences are not that impressive given the number of calls. Even in a complex document the overhead of scanning can be negligible compared to the actions involved in typesetting the document. In fact, there will always be some kind of scanning for such macros so we're talking about even less impact. So you can just use the method you like most. In practice, the extra overhead of using keywords in combination with explicit checks (the first case) is rather convenient.

If you don't want to have many tests you can do something like this:

```
local keys = {
  sx = scannumber,
  sy = scannumber,
  rx = scannumber,
  ry = scannumber,
}

function getmatrix()
  local values = { }
  while true do
    for key, scan in next, keys do
      if scankeyword(key) then
        values[key] = scan()
```

[3] Hard-coding the word scan in a C code helper makes little sense, as different macro packages can have different assumptions about what a word is. And we don't extend LuaTEX for specific macro packages.

```
      else
        break
      end
    end
  end
  -- action --
end
```

This is still quite fast although one now has to access the values in a table. Working with specifications like this is clean anyway so in ConTEXt we have a way to abstract the previous definition.

```
local specification = tokens.compile {
  {
    { "sx", "number" },
    { "sy", "number" },
    { "rx", "number" },
    { "ry", "number" },
  },
}

function getmatrix()
  local values = specification()
  -- action using values.sx etc --
end
```

Although one can make complex definitions this way, the question remains if it is a better approach than passing Lua tables. The standard ConTEXt way for controlling features is:

```
\getmatrix[sx=1.2,sy=3.4]
```

So it doesn't matter much if deep down we see:

```
\def\getmatrix[#1]{%
 \getparameters[@@matrix][sx=1,sy=1,
                          rx=1,ry=1,#1]%
   \domatrix
     \@@matrixsx
     \@@matrixsy
     \@@matrixrx
     \@@matrixry
   \relax}
```

or:

```
\def\getmatrix[#1]{%
 \getparameters[@@matrix][sx=1,sy=1,
                          rx=1,ry=1,#1]%
   \domatrix
     sx \@@matrixsx
     sy \@@matrixsy
     rx \@@matrixrx
     ry \@@matrixry
   \relax}
```

In the second variant (with keywords) can be a scanner like we defined before:

```
\def\domatrix#1#2#3#4%
  {\directlua{getmatrix()}}
```

but also:

```
\def\domatrix#1#2#3#4%
  {\directlua{getmatrix(#1,#2,#3,#4)}}
```

given:

```
function getmatrix(sx,sy,rx,ry)
    -- action using sx etc --
end
```

or maybe nicer:

```
\def\domatrix#1#2#3#4%
  {\directlua{domatrix{
     sx = #1, sy = #2,
     rx = #3, ry = #4
   }}}
```

assuming:

```
function getmatrix(values)
    -- action using values.sx etc --
end
```

If you go for speed the scanner variant without keywords is the most efficient one. For readability the scanner variant with keywords or the last shown example where a table is passed is better. For flexibility the table variant is best as it makes no assumptions about the scanner — the token scanner can quit on unknown keys, unless that is intercepted of course. But as mentioned before, even the advantage of the fast one should not be overestimated. When you trace usage it can be that the (in this case matrix) macro is called only a few thousand times and that doesn't really add up. Of course many different sped-up calls can make a difference but then one really needs to optimize consistently the whole code base and that can conflict with readability. The token library presents us with a nice chicken–egg problem but nevertheless is fun to play with.

6 Assigning meanings

The token library also provides a way to create tokens and access properties but that interface can change with upcoming versions when the old library is replaced by the new one and the input handling is cleaned up. One experimental function is worth mentioning:

```
token.set_macro("foo","the meaning of bar")
```

This will turn the given string into tokens that get assigned to `\foo`. Here are some alternative calls:

```
set_macro("foo")
≡ \def \foo {}
set_macro("foo", "meaning")
≡ \def \foo {meaning}
set_macro("foo", "meaning", "global")
≡ \gdef \foo {meaning}
```

The conversion to tokens happens under the current catcode regime. You can enforce a different regime by passing a number of an allocated catcode

table as the first argument, as with `tex.print`. As we mentioned performance before, setting at the Lua end like this:

```
token.set_macro("foo","meaning")
```

is about two times as fast as:

```
tex.sprint("\\def\\foo{meaning}")
```

or (with slightly more overhead) in ConTEXt terms:

```
context("\\def\\foo{meaning}")
```

The next variant is actually slower (even when we alias `setvalue`):

```
context.setvalue("foo","meaning")
```

but although 0.4 versus 0.8 seconds looks like a lot on a TEX run I need a million calls to see such a difference, and a million macro definitions during a run is a lot. The different assignments involved in, for instance, 3000 entries in a bibliography (with an average of 5 assignments per entry) can hardly be measured as we're talking about milliseconds. So again, it's mostly a matter of convenience when using this function, not a necessity.

7 Conclusion

For sure we will see usage of the new scanner code in ConTEXt, but to what extent remains to be seen. The performance gain is not impressive enough to justify many changes to the code but as the low-level interfacing can sometimes become a bit cleaner it will be used in specific places, even if we sacrifice some speed (which then probably will be compensated for by a little gain elsewhere).

The scanners will probably never be used by users directly simply because there are no such low level interfaces in ConTEXt and because manipulating input is easier in Lua. Even deep down in the internals of ConTEXt we will use wrappers and additional helpers around the scanner code. Of course there is the fun-factor and playing with these scanners is fun indeed. The macro setters have as their main benefit that using them can be nicer in the Lua source, and of course setting a macro this way is also conceptually cleaner (just like we can set registers).

Of course there are some challenges left, like determining if we are scanning input of already converted tokens (for instance in a macro body or token list expansion). Once we can properly feed back tokens we can also look ahead like `\futurelet` does. But for that to happen we will first clean up the LuaTEX input scanner code and error handler.

◇ Hans Hagen
Pragma ADE
http://pragma-ade.com
http://luatex.org