LuaTEX: A user's perspective

Aditya Mahajan

Abstract

In this article, I explain how to use Lua to write macros in LuaTEX. I give some examples of macros that are complicated in pdfTEX, but can be defined easily using Lua in LuaTEX. These examples include macros that do arithmetic on their arguments, use loops, and parse their arguments.

1 Introduction

TEX is getting a new engine — LuaTEX. As its name suggests, LuaTEX adds Lua, a programming language, to TEX, the typesetter. I cannot overemphasize the significance of being able to program TEX in a high-level programming language. For example, consider a TEX macro that divides two numbers. Such a macro is provided by the `fp` package and also by `pgfmath` library of the `TikZ` package. The following comment is from the `fp` package

```
\def\FP@div#1#2.#3.#4\relax#5.#6.#7\relax{%
 % [...] algorithmic idea (for x>0, y>0)
 %   - %determine \FP@shift  such that
 %      y*10^\FP@shift < 100000000
 %                     <=y*10^(\FP@shift+1)
 %   - %determine \FP@shift' such that
 %      x*10^\FP@shift'< 100000000
 %                     <=x*10^(\FP@shift+1)
 %   - x=x*\FP@shift'
 %   - y=y*\FP@shift
 %   - \FP@shift=\FP@shift-\FP@shift'
 %   - res=0
 %   - while y>0 %fixed-point representation!
 %   -   \FP@times=0
 %   -   while x>y
 %   -     \FP@times=\FP@times+1
 %   -     x=x-y
 %   -   end
 %   -   y=y/10
 %   -   res=10*res+\FP@times/1000000000
 %   - end
 %   - %shift the result according to \FP@shift
```

The `pgfmath` library implements the macro in a similar way, but limits the number of shifts that it does. These macros highlight the state of affairs in writing TEX macros. Even simple things like multiplying two numbers are hard; you either have to work extremely hard to circumvent the programming limitations of TEX, or, more frequently, hope that someone else has done the hard work for you. In LuaTEX, such a function can be written using the `/` operator (I will explain the details later):

```
\def\DIVIDE#1#2{\directlua{tex.print(#1/#2)}}
```

Thus, with LuaTEX ordinary users can write simple macros; and, perhaps more importantly, can read and understand macros written by TEX wizards.

Since the LuaTEX project started it has been actively supported by ConTEXt.[1] These days, the various "How do I write such a macro" questions on the ConTEXt mailing list are answered by a solution that uses Lua. I present a few such examples in this article. I have deliberately avoided examples about fonts and non-Latin languages. There is already quite a bit of documentation about them. In this article, I want to highlight how to use LuaTEX to write macros that require some "flow control": randomized outputs, loops, and parsing.

2 Interaction between TEX and Lua

To a first approximation, the interaction between TEX and Lua is straightforward. When TEX (i.e., the LuaTEX engine) starts, it loads the input file in memory and processes it token by token. When TEX encounters `\directlua`, it stops reading the file in memory, *fully expands the argument of* `\directlua`, and passes the control to a Lua instance. The Lua instance, which runs with a few preloaded libraries, processes the expanded arguments of `\directlua`. This Lua instance has a special output stream which can be accessed using `tex.print(...)`. The function `tex.print(...)` is just like the Lua function `print(...)` except that `tex.print(...)` prints to a "TEX stream" rather than to the standard output. When the Lua instance finishes processing its input, it passes the contents of the "TEX stream" back to TEX.[2] TEX then inserts the contents of the "TEX stream" at the current location of the file that it was reading; expands the contents of the "TEX stream"; and continues. If TEX encounters another `\directlua`, the above process is repeated.

As an exercise, imagine what happens when the following input is processed by LuaTEX.[3]

```
\directlua%
  {tex.print("Depth 1
           \\directlua{tex.print('Depth 2')}")}
```

[1] Not surprising, as two of LuaTEX's main developers — Taco Hoekwater and Hans Hagen — are also the main ConTEXt developers.

[2] The output of `tex.print(...)` is buffered and not passed to TEX until the Lua instance has stopped.

[3] In this example, I used two different kinds of quotations to avoid escaping quotes. Escaping quotes inside `\directlua` is tricky. The above was a contrived example; if you ever need to escape quotes, you can use the `\startluacode ... \stopluacode` syntax explained later.

On top of these LuaTEX primitives, ConTEXt provides a higher level interface. There are two ways to call Lua from ConTEXt. The first is a macro `\ctxlua` (read as ConTEXt Lua), which is similar to `\directlua`. (Aside: It is possible to run the Lua instance under different name spaces. `\ctxlua` is the default name space; other name spaces are explained later.) `\ctxlua` is good for calling small snippets of Lua. The argument of `\ctxlua` is parsed under normal TEX catcodes (category codes), so the end of line character has the same catcode as a space. This can lead to surprises. For example, if you try to use a Lua comment, everything after the comment gets ignored.

```
\ctxlua
  {-- A lua comment
   tex.print("This is not printed")}
```

This can be avoided by using a TEX comment instead of a Lua comment. However, working under normal TEX catcodes poses a bigger problem: special TEX characters like &, #, $, {, }, etc., need to be escaped. For example, # has to be escaped with `\string` to be used in `\ctxlua`.

```
\ctxlua
  {local t = {1,2,3,4}
   tex.print("length " .. \string#t)}
```

As the argument of `\ctxlua` is fully expanded, escaping characters can sometimes be tricky. To circumvent this problem, ConTEXt defines a environment called `\startluacode ... \stopluacode`. This sets the catcodes to what one would expect in Lua. Basically only \ has its usual TEX meaning, the catcode of everything else is set to other. So, for all practical purposes, we can forget about catcodes in `\startluacode ... \stopluacode`. The above two examples can be written as

```
\startluacode
  -- A lua comment
   tex.print("This is printed.")
  local t = {1,2,3,4}
   tex.print("length " .. #t)
\stopluacode
```

This environment is meant for moderately sized code snippets. For longer Lua code, it is more convenient to write the code in a separate Lua file and then load it using Lua's `dofile(...)` function.

ConTEXt also provides a Lua function to conveniently write to the TEX stream. The function is called `context(...)` and it is equivalent to `tex.print(string.format(...))`.

Using the above, it is easy to define TEX macros that pass control to Lua, do some processing in Lua, and then pass the result back to TEX. For example, a macro to convert a decimal number to hexadecimal can be written simply, by asking Lua to do the conversion.

```
\def\TOHEX#1{\ctxlua{context("\%X",#1)}}
\TOHEX{35}
```

The percent sign had to be escaped because `\ctxlua` assumes TEX catcodes. Sometimes, escaping arguments can be difficult; instead, it can be easier to define a Lua function inside `\startluacode ... \stopluacode` and call it using `\ctxlua`. For example, a macro that takes a comma separated list of strings and prints a random item can be written as

```
\startluacode
  userdata = userdata or {}
  math.randomseed( os.time() )
  function userdata.random(...)
    context(arg[math.random(1, #arg)])
  end
\stopluacode

\def\CHOOSERANDOM#1%
  {\ctxlua{userdata.random(#1)}}

\CHOOSERANDOM{"one", "two", "three"}
```

I could have written a wrapper so that the function takes a list of words and chooses a random word among them. For an example of such a conversion, see the "sorting a list of tokens" page on the LuaTEX wiki [2].

In the above, I created a name space called `userdata` and defined the function `random` in that name space. Using a name space avoids clashes with the Lua functions defined in LuaTEX and ConTEXt.

In order to avoid name clashes, ConTEXt also defines independent name spaces of Lua instances. They are

`user`	: a private user instance
`third`	: third party module instance
`module`	: ConTEXt module instance
`isolated`	: an isolated instance

Thus, for example, instead of `\ctxlua` and `\startluacode ... \stopluacode`, the `user` instance can be accessed via the macros `\usercode` and `\startusercode ... \stopusercode`. In instances other than `isolated`, all the Lua functions defined by ConTEXt (but not the inbuilt Lua functions) are stored in a `global` name space. In the `isolated` instance, all Lua functions defined by ConTEXt are hidden and cannot be accessed. Using these instances, we could write the above `\CHOOSERANDOM` macro as follows

```
\startusercode
  math.randomseed( global.os.time() )
```

```
  function random(...)
    global.context(arg[math.random(1, #arg)])
  end
\stopusercode

\def\CHOOSERANDOM#1%
  {\usercode{random(#1)}}
```

Since I defined the function `random` in the `user` instance of Lua, I did not bother to use a separate name space for the function. The Lua functions `os.time`, which is defined by a LuaTEX library, and `context`, which is defined by ConTEXt, needed to be accessed through a `global` name space. On the other hand, the `math.randomseed` function, which is part of Lua, could be accessed as is.

A separate Lua instance also makes debugging slightly easier. With `\ctxlua` the error message starts with

```
! LuaTeX error <main ctx instance>:
```

With `\usercode` the error message starts with

```
! LuaTeX error <private user instance>:
```

This makes it easier to narrow down the source of error.

Normally, it is best to define your Lua functions in the `user` name space. If you are writing a module, then define your Lua functions in the `third` instance and in a name space which is the name of your module. In this article, I will simply use the default Lua instance, but take care to define all my Lua functions in a `userdata` name space.

Now that we have some idea of how to work with LuaTEX, let's look at some examples.

3 Arithmetic without using a abacus

Doing simple arithmetic in TEX can be extremely difficult, as illustrated by the division macro in the introduction. With Lua, simple arithmetic becomes trivial. For example, if you want a macro to find the cosine of an angle (in degrees), you can write

```
\def\COSINE#1%
  {\ctxlua(context(math.cos(#1*2*pi/360))}
```

The built-in `math.cos` function assumes that the argument is specified in radians, so we convert from degrees to radians on the fly. If you want to type the value of π in an article, you can simply say

```
$\pi = \ctxlua{context(math.pi)}$
```

or if you want less precision (notice the percent sign is escaped)

```
$\pi = \ctxlua{context("\%.6f", math.pi)}$
```

4 Loops without worrying about expansion

Loops in TEX are tricky because macro assignments and macro expansion interact in strange ways. For example, suppose we want to typeset a table showing the sum of the roll of two dice and want the output to look like this

(+)	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

The tedious (but faster!) way to achieve this is to simply type the whole table by hand. For example,

```
\bTABLE
  \bTR \bTD $(+)$ \eTD \bTD 1 \eTD ... ... \eTR
  \bTR \bTD   1   \eTD \bTD 2 \eTD ... ... \eTR
  ...   ...   ...   ...   ...   ...
  ...   ...   ...   ...   ...   ...
\eTABLE
```

It is however natural to want to write this table as a loop, and compute the values. A first ConTEXt implementation using the recursion level might be:

```
\bTABLE
 \bTR
  \bTD $(+)$ \eTD
  \dorecurse{6}
   {\bTD \recurselevel \eTD}
  \eTR
\dorecurse{6}
 {\bTR
     \bTD \recurselevel \eTD
     \edef\firstrecurselevel{\recurselevel}
  \dorecurse{6}
  {\bTD
   \the\numexpr\firstrecurselevel+\recurselevel
  \eTD}%
  \eTR}
\eTABLE
```

However, this does not work as expected, yielding all zeros. A natural table stores the contents of all the cells, before typesetting it. But it does not expand the contents of its cell before storing them. So, at the time the table is actually typeset, TEX has already finished the `\dorecurse` and `\recurselevel` is set to 0.

The solution is to place `\expandafter` at the correct location(s) to coax TEX into expanding

the \recurselevel macro before the natural table stores the cell contents. The difficult part is figuring out the exact location of \expandafters. Here is a solution that works:

```
\bTABLE
 \bTR
  \bTD $(+)$ \eTD
  \dorecurse{6}
   {\expandafter \bTD \recurselevel \eTD}
  \eTR
\dorecurse{6}
{\bTR
 \edef\firstrecurselevel{\recurselevel}
 \expandafter\bTD \recurselevel \eTD
 \dorecurse{6}
 {\expandafter\bTD
  \the\numexpr\firstrecurselevel+\recurselevel
  \relax
  \eTD}
 \eTR}
\eTABLE
```

We only needed to add three \expandafters to make the naive loop work. Nevertheless, finding the right location pf \expandafter can be frustrating, especially for a non-expert.

By contrast, in LuaTEX writing loops is easy. Once a Lua instance starts, TEX does not see anything until the Lua instance exits. So, we can write the loop in Lua, and simply print the values that we would have typed to the TEX stream. When the control is passed to TEX, TEX sees the input as if we had typed it by hand. Consequently, macro expansion is no longer an issue. For example, we can get the above table by:

```
\startluacode
context.bTABLE()
  context.bTR()
    context.bTD() context("$(+)$") context.eTD()
    for j=1,6 do
      context.bTD() context(j) context.eTD()
    end
  context.eTR()
  for i=1,6 do
    context.bTR()
    context.bTD() context(i) context.eTD()
    for j=1,6 do
      context.bTD() context(i+j) context.eTD()
    end
    context.eTR()
  end
context.eTABLE()
\stopluacode
```

The Lua functions such as context.bTABLE() and context.bTR() are just abbreviations for running context ("\\bTABLE"), context("\\bTR"), etc. See the ConTEXt Lua document manual for more details about such functions [3]. The rest of the code is a simple nested for-loop that computes the sum of two dice. We do not need to worry about macro expansion at all!

5 Parsing input without exploding your head

In order to get around the weird rules of macro expansion, writing a parser in TEX involves a lot of macro jugglery and catcode trickery. It is a black art, one of the biggest mysteries of TEX for ordinary users.

As an example, let's consider typesetting chemical molecules in TEX. Normally, molecules should be typeset in text mode rather than math mode. For example, $H_2SO_4^-$, can be input as H\low{2}SO\lohi{4}{--}. Typing so much markup can be cumbersome. Ideally, we want a macro such that we type \molecule{H_2SO_4^-} and the macro translates this into H\low{2}SO\lohi{4}{--}. Such a macro can be written in TEX as follows.

```
\newbox\chemlowbox
\def\chemlow#1%
  {\setbox\chemlowbox
          \hbox{{\switchtobodyfont[small]#1}}}

\def\chemhigh#1%
  {\ifvoid\chemlowbox
      \high{{\switchtobodyfont[small]#1}}%
   \else
      \lohi{\box\chemlowbox}
           {{\switchtobodyfont[small]#1}}
   \fi}

\def\finishchem%
  {\ifvoid\chemlowbox\else
      \low{\box\chemlowbox}
   \fi}

\unexpanded\def\molecule%
  {\bgroup
   \catcode`\_=\active \uccode`\~=`\_
       \uppercase{\let~\chemlow}%
   \catcode`\^=\active \uccode`\~=`\^
       \uppercase{\let~\chemhigh}%
   \dostepwiserecurse {65}{90}{1}
   {\catcode \recurselevel = \active
    \uccode`\~=\recurselevel
        \uppercase{\edef~{\noexpand\finishchem
    \rawcharacter{\recurselevel}}}}%
    \catcode`\-=\active \uccode`\~=`\-
        \uppercase{\def~{--}}%
    \domolecule }%

\def\domolecule#1{#1\finishchem\egroup}
```

This monstrosity is a typical TEX parser. Ap-

propriate characters need to be made active; occasionally, `\lccode` and `\uccode` need to be set; signaling tricks are needed (for instance, checking if `\chemlowbox` is void); and then magic happens (or so it seems to a flabbergasted user). More sophisticated parsers involve creating finite state automata, which look even more monstrous.

With LuaTEX, things are different. LuaTEX includes a general parser based on PEG (parsing expression grammar) called `lpeg` [4]. This makes writing parsers in TEX much more comprehensible. For example, the above `\molecule` macro can be written as

```
\startluacode
userdata = userdata or {}

local lowercase   = lpeg.R("az")
local uppercase   = lpeg.R("AZ")
local backslash   = lpeg.P("\\")
local csname      = backslash * lpeg.P(1)
                  * (1-backslash)^0
local plus        = lpeg.P("+") / "\\textplus "
local minus       = lpeg.P("-") / "\\textminus "
local digit       = lpeg.R("09")
local sign        = plus + minus
local cardinal    = digit^1
local integer     = sign^0 * cardinal
local leftbrace   = lpeg.P("{")
local rightbrace  = lpeg.P("}")
local nobrace     = 1 - (leftbrace + rightbrace)
local nested      = lpeg.P {leftbrace
                  * (csname + sign + nobrace
                  + lpeg.V(1))^0   * rightbrace}
local any         = lpeg.P(1)

local subscript   = lpeg.P("_")
local superscript = lpeg.P("^")
local somescript  = subscript + superscript

local content     = lpeg.Cs(csname + nested
                          + sign + any)

local lowhigh     = lpeg.Cc("\\lohi{%s}{%s}")
                    * subscript   * content
                    * superscript * content
                    / string.format
local highlow     = lpeg.Cc("\\hilo{%s}{%s}")
                    * superscript * content
                    * subscript   * content
                    / string.format
local low         = lpeg.Cc("\\low{%s}")
                    * subscript   * content
                    / string.format
local high        = lpeg.Cc("\\high{%s}")
                    * superscript * content
                    / string.format
local justtext    = (1 - somescript)^1
local parser      = lpeg.Cs((csname + lowhigh
                          + highlow + low
                          + high + sign + any)^0)

userdata.moleculeparser = parser

function userdata.molecule(str)
    return parser:match(str)
end
\stopluacode

\def\molecule#1%
  {\ctxlua{userdata.molecule("#1")}}
```

This is more verbose than the TEX solution, but is easier to read and write. With a proper parser, I do not have to use tricks to check if either one or both _ and ^ are present. More importantly, anyone (once they know the Lpeg syntax) can read the parser and easily understand what it does. This is in contrast to the implementation based on TEX macro jugglery which require you to implement a TEX interpreter in your head to understand.

6 Conclusion

LuaTEX is removing many TEX barriers: using system fonts, reading and writing Unicode files, typesetting non-Latin languages, among others. However, the biggest feature of LuaTEX is the ability to use a high-level programming language to program TEX. This can potentially lower the learning curve for programming TEX.

In this article, I have mentioned only one aspect of programming TEX: macros that manipulate their input and output some text to the main TEX stream. Many other kinds of manipulations are possible: LuaTEX provides access to TEX boxes, token lists, dimensions, glues, catcodes, direction parameters, math parameters, etc. The details can be found in the LuaTEX manual [1].

7 References

[1] LuaTEX reference manual, http://www.luatex.org/documentation.html

[2] Sorting a list of tokens, in the Joy of LuaTEX. http://luatex.bluwiki.com/go/Sort_a_token_list

[3] Hans Hagen, "CLD: ConTEXt Lua document", http://www.pragma-ade.com/general/manuals/cld-mkiv.pdf

[4] Lpeg: Parsing Expression Grammars for Lua, http://www.inf.puc-rio.br/~roberto/lpeg/lpeg.html

◇ Aditya Mahajan
adityam (at) umich dot edu