# LuaTEX: Howling to the moon

Hans Hagen
Pragma ADE, The Netherlands
pragma@wxs.nl

**Abstract**

Occasionally we reach the boundaries of TEX and programming then becomes rather cumbersome. This is partly due to the limitations of the typesetting engine, but more important is that a macro language is not always best suited for the task at hand.

## 1 Some observations

Occasionally we reach the boundaries of TEX and programming then becomes rather cumbersome. This is partly due to the limitations of the typesetting engine, but more important is that a macro language is not always best suited for the task at hand. One problem is for instance that intermediate operations and final results are intermixed (which has a certain charm as well).

```
\def\RepeatMe#1#2%
  {\bgroup
   \count0=#1%
   \loop
     \ifnum\count0>0
       #2%
       \advance\count0 -1
   \repeat
   \egroup}
```

The next call gives *****:

```
\RepeatMe{5}{*}
```

If you want to use this macro to construct a new one holding 5 stars, you're tempted to do something:

```
\edef\ShowMe{\RepeatMe{3}{*}}
```

This will give an error and the reason for this is that `\RepeatMe` is (in TEX-speak) not fully expandable.

In ε-TEX we can define this macro in a fully expandable way:

```
\def\RepeatMe#1#2%
  {\ifnum#1>0 #2\else
     \expandafter\IgnoreMe
   \fi
   \expandafter\RepeatMe\expandafter
   {\the\numexpr#1-1\relax}{#2}}
\def\IgnoreMe#1#2#3#4#5{}
```

Now we can safely say:

```
\edef\ShowMe{\RepeatMe{3}{*}}
```

Or even:

```
\edef\ShowMe{\RepeatMe{3}{\RepeatMe{3}{*}}}
```

Unfortunately, many users will turn away from writing macros as soon as something like `\expandafter` shows up, so this solution is not of much help for novice users, given that they already have noticed that they need to do this kind of expansion in order not to end up in too deep nesting and in order to finish the condition. Also, we only have to add a simple counter that keeps track of the total number to end up with a non-fully expandable macro again.

```
\newcount\NOfRepeats

\def\RepeatMe#1#2%
  {\ifnum#1>0
     \advance\NOfRepeats 1
     #2%
   \else
     \expandafter\IgnoreMe
   \fi
   \expandafter\RepeatMe\expandafter
   {\the\numexpr#1-1\relax}{#2}}
\def\IgnoreMe#1#2#3#4#5{}
```

If we try:

```
\edef\ShowMe{\RepeatMe{2}{*}}
```

We get a macro `\ShowMe` with the meaning:

```
\advance \NOfRepeats 1 *\advance \NOfRepeats
1 *
```

But there is a way out of this! See the following variant:

```
\def\RepeatMe#1#2%
  {\lua
     {for i=1,#1 do
        tex.print("#2")
      end}}
```

Isn't this much more convenient? It even looks readable. Before I explain the `\lua` command, I will make a few more observations.

As said, TEX programming can be cumbersome, but this is no reason to throw away the macro language. There is much macro code around and there are many documents that depend on TEX's longevity. So in order to overcome limitations, we need to talk in terms of extensions, not replacements.

Nowadays TEX is used for more than typesetting documents directly from a source. While manipulating the input, some applications demand more advanced programming features than good old TEX can (conveniently) provide. In many cases, it's not so much the whole machinery that we want to replace. We just want to extend or replace some of the subsystems. In order to provide robust solutions we may also want to have access to the internals. We'd like to add extensions and delegate activities to other programs. Of course there may also be personal motives for extending TEX, for instance because every now and then we need a new challenge.

## 2 Why Lua

I use SCITE as my main editor for documents and programs. This editor is built on top of the SCINTILLA framework. It's a fast, lightweight but powerful program. Being mainly a program editor it lacks some features that you need for text editing and processing, like adjusting text and spell checking. However ... it has a scripting engine that gives access to the user interface and the editing component. This scripting language happens to be LUA.

For a while I was hesitant to use 'yet another programming language', but using the built in LUA machinery made more sense than programming in C++ and trying to keep that up to date with the rest of the program. I was more or less familiar with the SCINTILLA programming interface because I had written the TEX and METAPOST lexers, but in contrast to the lexer code other extensions would involve changes spread more widely over the code base. So, the LUA way we went.

While the PERL, PYTHON, and (my favourite) RUBY languages are heavyweight languages, LUA is a simple but nevertheless powerful language. The first three come with a constantly growing number of libraries and users expect the whole lot to be present, which makes them unsuitable as extension languages: cross platform issues, slow startup, much bigger TEX distributions, etc.

LUA can interface to libraries but its main purpose is to be embedded in an application and present the user with an interface to the parent application. It's made for embedding! It has a small footprint and adds only some 50–100K to the parent binary. One can add additional functionality, for instance support for sockets.

The more I looked into it, the more I liked it, and after positive experiences with integrating METAPOST in ConTEXt, integrating LUA didn't seem that strange to me.

Not being a 'TEX the Program' hacker, I managed to trick Hartmut Henkel (member of the PDFTEX development team) into implementing a `\lua` command. Hartmut and I share a passion for experiments in PDFTEX and he knows how to implement them: I got back a (LINUX) binary within a few days.

After we had this proof of concept, it was time to involve Taco Hoekwater (member of the ConTEXt, METAPOST and other development teams), who knows more about TEX's internals than I ever will. Another few days later we had the first interface to some of TEX's internals.

## 3 A few examples

I will show a few of the examples that I demonstrated at the TUG 2005 conference in Wuhan, China. Keep in mind that an experimental version was used and that the interfaces may change. For instance, it will be possible to activate several LUA engines (`\newlua`) and all variables, the hash table, internal lists, etc., are on the agenda.

For starters, if we say:

```
\lua{a = 1.5 ; b = 1.8 ; c = a*b}
```

we only set some variables, and get nothing in return. However, the next call puts the value of c back into the input stream.

```
\lua{a = 1.5 ; b = 1.8 ; c = a*b ;
     tex.print(c)}
```

The result comes back as a string. This means that we need to process it in the right way in order get something that TEX sees as TEX code. For this reason we define:

```
\def\luatex#1{\scantokens\expandafter
              {\lua{#1}}}
```

The \scantokens primitive has some undesired side effects: it cannot properly handle multiple lines (catcode problem) and acts as a pseudo-file, but Taco is working on a variant.

So, in the next example, we get different results:

```
\lua   {tex.print("$\string\\sqrt{2} = "
                  .. math.sqrt(2) .. "$")}
\luatex{tex.print("$\string\\sqrt{2} = "
                  .. math.sqrt(2) .. "$")}
```

The \lua call returns a verbatim copy, i.e. a dollar shows up as a dollar, while the \luatex call gives a typeset formula, i.e. the dollar sign is seen as the signal to enter math mode.

Next we demonstrate how to use LUA to match strings (using regular expressions):

```
\lua {
  match = {} ;
  match.expression = "" ;
  match.string = "" ;
  match.result = {} ;
  match.start = 0 ;
  match.length = 0 ;
}
\def\matchstring#1#2{\lua {
  match.expression = "#1" ;
  match.string = "#2" ;
  match.result = {} ;
  match.start,match.length,match.result[1],
   match.result[2], match.result[3],
   match.result[4], match.result[5],
   match.result[6], match.result[7],
   match.result[8], match.result[9]
    = string.find(match.string,
                  match.expression);
}}

\def\matchresult#1{\lua {
    tex.print(match.result[#1]) ;
}}
```

When we use the last macro as:

```
\matchstring
  {(\letterpercent d+) (\letterpercent d+)
   (\letterpercent d+)}
  {2005 08 08}
```

the three components are available in:

```
\matchresult{1} \matchresult{2} \matchresult{3}
```

The percent symbol is LUA's escape character, which is why we need \letterpercent. The results can be used in a table like the following:

```
\starttabulate[|l|c|r|]
\NC year  \NC \matchresult{1}
          \NC \number \matchresult{1} \NC \NR
\NC month \NC \matchresult{2}
          \NC \number \matchresult{2} \NC \NR
\NC day   \NC \matchresult{3}
          \NC \number \matchresult{3} \NC \NR
\stoptabulate
```

We can rewrite the LUA code in a more efficient way (less code in the main macro):

```
\lua {
  function match.find(expr, str)
   match.expression = expr ;
   match.string = str ;
   match.result = {} ;
   match.start, match.length, match.result[1],
   match.result[2], match.result[3],
   match.result[4], match.result[5],
   match.result[6], match.result[7],
   match.result[8], match.result[9]
     = string.find(match.string,
                   match.expression) ;
 end
}

\def\matchstring#1#2{\lua {
    match.find("#1","#2") ;
} }
```

```
\def\matchresult#1{\lua {
    tex.print(match.result[#1]) ;
} }
```

The usage is the same. In the previous example we used `\number` to get rid of leading zeros, but we can also adapt the expression like this:

```
\matchstring
  {0*(\letterpercent d+)
   0*(\letterpercent d+)
   0*(\letterpercent d+)}
  {2005 08 08}
```

The next example takes some close reading, but it demonstrates how to mix TEX and LUA. Let's start with defining some tables, a very LUAish but powerful data structure (the more one reads about LUA, the more one realizes that it is pretty well designed).

```
\lua {
    document = { }
    document.dimens = { }
}
```

Next we store the widths of a bunch of characters in the dimens subtable. (We could have used ε-TEX's char dimension primitives instead.)

```
\dostepwiserecurse{`a}{`z}{1} {
 \setbox\tempbox\hbox{\char\recurselevel}
 \lua {
  document.dimens[\recurselevel]
   = tex.wd[\number\tempbox]
 }
}
```

Now we can calculate the average widths of the characters. Of course this particular case can be done in pure TEX quite conveniently, but one can envision more tricky calculations.

```
\lua {
  local total, n = 0, 0
  for d in pairs(document.dimens) do
    total, n
      = total + document.dimens[d], n + 1
  end
  if n>0 then
    document.mean = total/n
  else
    document.mean = 0
  end
}
```

We leave it to your imagination to envision what the next code will produce (this article is not produced using LUATEX):

```
\mathematics {
 \lua { tex.dimen[0] = document.mean }
 \withoutpt \the\dimen0  =
 \lua { tex.print(document.mean/65536) }
 \approx
 \lua { tex.print(
         math.ceil(document.mean/65536)) }
}
```

Did you notice how we have access to TEX's dimensions? We can read and write them — when a string is assigned, the usual dimensions are interpreted. In a similar fashion we have access to counters.

```
\bgroup
 \count0=10 \count2=30
 \scratchcounter =
   \lua { tex.print((tex.count[0]
                     + tex.count[2])/2) }
 \number\scratchcounter
\egroup
```

Now keep in mind, we're not piping data from TEX into some external program and reading it back in again. Here LUA is an integral part of TEX!

Token registers can be accessed as well:

```
\toks0 = {interesting}
\lua {
  tex.toks[0]
    = string.gsub(tex.toks[0],
                  "(.)", " (\letterpercent1)
")
}
\the\toks0
```

This will return the individual characters of 'interesting' surrounded by parentheses. LUA has sufficient functionality for string manipulations.

Can you envision what this next bit of code does? Again we use a token register but this time we also rescan the result because it contains a control sequence. (The quote is from Hermann Zapf.)

```
\toks0 = {
 Coming back to the use of typefaces in
 electronic publishing: many of the new
 typographers receive their knowledge and
 information about the rules of typography
 from books, from computer magazines or the
 instruction manuals which they get with
 the purchase of a PC or software. There
 is not so much basic instruction, as of
 now, as there was in the old days,
 showing the differences between good and
 bad typographic design. Many people are
 just fascinated by their PC's tricks,
 and think that a widely-praised program,
 called up on the screen, will make
 everything automatic from now on.
}

\lua {
 str = tex.toks[0]
 str = string.gsub(str,"\letterpercent w+",
        function(w)
         if str.len(w) > 4 then
          return "\string\\color[red]{"
                 .. w .. "}"
         else
          return "\string\\color[green]{"
                 .. w .. "}"
         end
        end
       )
 tex.toks[0] = str
}

\scantokens\expandafter{\the\toks0}
```

Words longer than 4 characters will become red, while short words become green. The `..` operator concatenates strings. Of course a solution with more complex input will demand a different approach, and in the end we need the ability to process TEX's internal lists.

We don't yet have access to the `lccode` table but we can already efficiently define vectors. Currently TEX lacks a way to quickly initialize switches, which is a pity because nowadays PDFTEX provides a lot of them.

```
\lua {
  lccodes = { }
}

\dorecurse{255} {
  \lua { lccodes[\recurselevel]
          = \number\lccode\recurselevel ;
       }
}

\luatex {
  for i in pairs(lccodes) do
    tex.print("\string\\lccode" .. i
              .. "=" .. lccodes[i] .. " ")
  end
}
```

In the last example we return to the kind of example that we started with: fully expandable definitions.

```
\lua {
 interface = {}
 interface.noftests = 0
 function interface.oneoftwo(result)
  interface.noftests = interface.noftests+1
  if result then
    tex.print("firstoftwoarguments")
  else
    tex.print("secondoftwoarguments")
   end
 end
}
```

This function returns a given string, depending on the boolean (condition) fed into it. The two possible strings correspond to macro names (ConTEXt has a bunch of those):

```
\long\def\firstoftwoarguments #1#2{#1}
\long\def\secondoftwoarguments#1#2{#2}
```

Now imagine the following definitions:

```
\def\DoIfElse#1#2{%
  \csname\lua{%
      interface.oneoftwo("#1"=="#2")
  }\endcsname
}
```

We feed the string comparison into the function, which returns a string, which in turn is expanded into a control sequence. Without any chance of interference we also keep track of the number of tests. We can ask for this number with:

```
\def\NOfTests
  {\lua{tex.print(interface.noftests)}}
```

The following code shows how to use the test. They all return (typeset) OK.

```
\def\xxx{yyy} \def\yyy{yyy}

\DoIfElse{one}{one}{OK}{NO}
\DoIfElse{two}{one}{NO}{OK}
\DoIfElse\xxx \yyy {OK}{NO}
```

What goes in gets expanded, but when it enters LUA it has been turned into a harmless sequence of characters:

```
\DoIfElse
  {this is an \hbox to \hsize{real} mess}
  {and \rm this is even worse}
  {NO}{OK}
```

This test can be used in an \edef and fed back into LUA if needed.

```
\edef\zzz{this
          \DoIfElse{a}{b}{or}{and}
          that}

\lua{tex.print("\zzz")}
\lua{tex.print("this
                \DoIfElse{a}{a}{or}{and}
                that")}
```

These examples demonstrate that we can comfortably mix TEX and LUA and use whichever suits the problem best. Also keep in mind that nesting LUA calls is no problem either:

```
\lua{tex.print("this
                \lua{tex.print("---")}
                that")}
```

Of course we need additional trickery. For instance we need a way to escape characters before they enter LUA, or the following will fail:

```
\DoIfElse{one "two" three}
         {one 'two' three}
         {OK}{NO}
```

Instead we need something like this:

```
\DoIfElse
  {\luaesc{one "two" three}}
  {\luaesc{one 'two' three}}
  {OK}{NO}
```

which will comfortably escape the sensitive characters. So, there is some work left to do.

## 4 The potential

Since we recognized the potential we decided to continue this effort and the lean and mean LUATEX team was there: Hartmut Henkel, Taco Hoekwater & Hans Hagen. We will discuss, meet, develop and finally present a working and stable version, probably around EuroTEX and/or TUG 2006. The binary will start as a fork of PDFTEX, and in the end perhaps replace PDFTEX since its author likes the idea.

In the process we consider removing some of the fuzzy parts of PDFTEX; good candidates for removal are enctex and mltex, among others, especially if we can preprocess the input stream. We may as well replace/extend some of the subsystems (such as font handling for which we need to go to OPENTYPE anyway) and/or prototype new subsystems. We will provide an interface to plugins; for instance, the paragraph building components that Karel Skoupý is building and has demonstrated at conferences.

In the end we will provide a playground for all kinds of features; for instance, language/script specific font handlers and paragraph builders for languages such as Chinese. Of course my personal intent is to replace (or extend) some macro-defined parts of ConTEXt by alternatives in LUA (think of ConTEXt 4). We can even add functionality that until now was beyond reason to implement. And this is just the start ...

My thanks to Hartmut Henkel and Taco Hoekwater for their collaboration on this project and this paper, and Karl Berry for editorial help.