LuaTeX as seen by a novice

Dirk Laurie

Written on 10–11 November 2010, my first two days with LuaTeX
Some blunders removed on 16 November

1 What is LuaTeX?

This question has several answers. Until LuaTeX reaches version 1.0, the answers may in fact still change. The ones below apply to LuaTeX 0.50, as supplied in the 2009 release of TeX Live.

1.1 A drop-in replacement for pdfTeX

You can use `luatex` instead of `pdftex` and `lualatex` instead of `pdflatex`.

That is to say, if you take any LaTeX document `mydoc.tex`, and issue the command:

```
lualatex mydoc
```

you should get a PDF file that looks almost exactly like the PDF file you get from `pdflatex mydoc`.

This does not mean that the two PDF files will be equivalent to a more discerning reader than the human eye: even something as basic as a PDF-to-text converter might produce text files that are not identical.

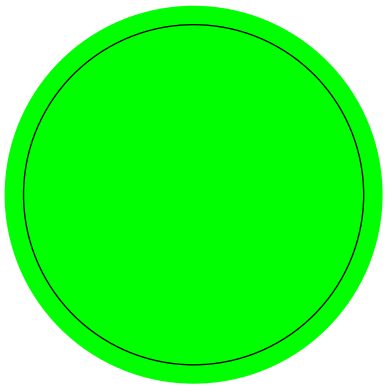
1.1.1 Unicode

LuaLaTeX supports Unicode input, but not in exactly the same way as LaTeX. In particular, you should say `\usepackage[utf8]{luainputenc}` instead of `\usepackage[utf8]{inputenc}`.

Some sources I have seen say you can simply write your file in UTF-8, and any non-ASCII characters will be picked up by `lualatex` with no further effort on your part. It didn't work that way on my system. Apparently the fonts I use are too old to have UTF-8 encodings.

1.2 More than pdfTEX

Most add-on packages that work with TEX or LATEX should work with LuaTEX. However, there are some packages that work only with LuaTEX. Foremost among these is `luamplib`, which allows METAPOST drawings directly into your document, thus:



```
\begin{mplibcode}
beginfig(1);
fill fullcircle scaled 5cm
  withcolor green;
draw fullcircle scaled 4.5cm;
endfig;
\end{mplibcode}
```

There is also a font selection scheme that is much more human-friendly than the one currently used, but I didn't try that.

1.3 A standalone Lua interpreter

The same executable is invoked whether you call it `luatex`, `lualatex` or `texlua`, but the last of the three expects your source file to contain only Lua code. Thus, even if you don't have any other Lua installed on you system, you can still happily write application programs in Lua.

1.3.1 So what is Lua?

Lua is a programming language made in Brazil and named "moon" in Portuguese.

It is rather like Pascal in the sense that it has a small number of reserved words, 21 of them, almost all of reassuringly familiar appearance to Pascal programmers.

```
nil false true and or not
function local break return end
if then elseif else
while for in do repeat until
```

It has six datatypes that everybody needs:

```
nil  boolean  number  string  table  function
```

and two more for advanced programmers.

It uses ASCII special characters in much the way one expects, and everything else is done with function calls. An experienced computer user reading Lua code written by a considerate programmer will understand most of it, for example:

```
function gcd (x,y)
    if x<0 then x = -x end
    if y<0 then y = -y end
    if y<x then x,y = y,x end
    while x<y do
        if x==0 then return y end
        x,y = math.mod(y,x), x
    end
end

print(gcd(10,15))
```

Now if your input file `luademo.lua` contained only the above lines, the command `texlua luademo.lua` would print `5` and exit.

As it stands, Lua is not really object-oriented, though it may look that way. There are no classes, no inheritance, and no polymorphism in the language itself. However, it has do-it-yourself object-oriented programming — tables behave somewhat like objects and can borrow attributes from a prototype. In fact, this attitude of providing functionality in a well-designed library rather than defining it as part of the language is very typical of Lua.

The above Lua code looks clean, but since Lua syntax has no need of statement separators, a malicious programmer might have written the same program as

```
function gcd (x,y) if x<0 then x = -x end if y<0 then y =
-y end if y<x then x,y = y,x end while x<y do if x==
0 then return y end x, y = math.mod(y,x), x end end print(
gcd(10,15))
```

That's enough about Lua here. You will need to know a lot more to get something useful from it. Read the excellent book *Programming in Lua* by the main architect of Lua, Roberto Ierusalimschy, the first edition of which is available online at `http://www.lua.org/pil`.

1.4 Using Lua inside TEX

We all love TEX, but it is not a general-purpose programming language. The moment your document regularly contains stuff that requires a bit of work before it can be described in terms of text and boxes, doing it in TEX is a challenge. The CTAN is full of monumental examples of programmers rising to that challenge, providing packages for things like music typesetting and crosswords, and doing almost everything in TEX itself.

How much more could these giants on whose shoulders we stand have achieved if the data shuffling and computations could have been done in a powerful scripting language embedded in TEX!

That is precisely what LuaTEX is all about. You give Lua something that is known to TEX, and allow Lua to work on it and give the result back to TEX.

For example, suppose I want to prettyprint that program written by a malicious programmer. I could do it by hand (probably making a mistake or two), I could get some program on my operating system to do it and cut-and-paste the result into my document, or I could do the work right here using LuaTEX.

Thus particular job is a little tricky, and even the rather rudimentary prettyprinter that I wrote in Lua is about 60 lines long, so I have saved the Lua code in the file `prettyprint.lua`.

All you need to know about it is that there is a function called `pretty` that accepts a text string containing a Lua program and produces another text string containing the program as LATEX code. I found that `tex.print` only works properly if I take care that the LATEX code contains no newline characters.

```
function gcd(x, y)
    if x < 0 then x = -x end
    if y < 0 then y = -y end
    if y < x then x, y = y, x end
    while x < y do
        if x == 0 then return y end
        x, y = math.mod(y, x), x
        end
    end
print(gcd(10, 15))
```

The command to achieve that, is:

```
\directlua{dofile('prettyprint.lua');
tex.print(pretty(
[[function gcd (x,y) if x<0 then x = -x end if y<0 then y =
-y end if y<x then x,y = y,x end while x<y do if x==
0 then return y end x, y = math.mod(y,x), x end end print(
gcd(10,15))]]
))}
```

You only need to `dofile` once per Lua file. Later calls to `\directlua` know the new functions. You could also say `tex.sprint`. I have been unable to figure out the difference between the two.

1.5 Do I need to read the LuaTEX Reference Manual?

The short answer is: probably not.

As long as you think of yourself as a novice, my advice is: rather read *Programming in Lua*. You can get very far knowing only `\directlua` and `tex.print`, but you won't get off the start line if you don't know much more Lua than I have told you about.

Once you are well on the way to becoming an expert, you will want to dip into the Reference Manual every now and then. That's what reference manuals are for.

I'll have to stop writing here, since I am in danger of not being enough of a novice any more. Scraping an acquaintance with LuaTEX has given me a most enjoyable couple of days!

2 Code of `prettyprint.lua`

This code is just enough to cope with the example in the text. A real prettyprinter should do some syntax analysis, not just rely on search-and-replace techniques, and especially not need a user-supplied identifier list.

Five days later, when removing the blunders from the rest of the article, I have not revised this code (despite strong temptation to show off my already greatly improved Lua skills) so that it remains an example of what a novice programmer wrote. The learning curve for Lua is very gentle.

Of course, for your own new programs, imitate the style of *Programming in Lua*, not this!

```
function pretty(a)
-- Replace existing whitespace by single blanks
    a = string.gsub(a,'%s+',' ')..' '
-- Put a newline after each 'end'
    a = string.gsub(a,'end ','end\n')
-- And before each 'if'
    a = string.gsub(a,' if ',' \nif ')
-- Last character must be newline
    a=string.sub(a,1,-2)..'\n'
-- Newline before any 'end' on a line not started by 'if'
    local b=''
    while a~='' do
        i, j = string.find(a,'\n')
        if string.sub(a,1,3) ~= 'if ' and
                string.sub(a,i-3,i-1) == 'end'
            then b = b..string.sub(a,1,i-4)..'\nend\n'
        else b = b..string.sub(a,1,i)
            end
        if j==nil then break end
        a = string.sub(a,j+1,-1)
        end
-- Remove blank lines
    b = string.gsub(b,'\n\n','\n')
-- Indent lines
    a = b; b=''; local indent=0
    while a~='' do
        i, j = string.find(a,'\n')
        b = b..prettyline(indent,string.sub(a,1,i))
        if string.sub(a,1,3) == 'if '
            or string.sub(a,1,6) == 'while '
            or string.sub(a,1,9) == 'function '
            then indent = indent + 1
            end
        if string.sub(a,i-3,i-1) == 'end' then
            indent = indent - 1 end
        a = string.sub(a,j+1,-1)
        end
    return '\\begin{align*}'..b..'\\end{align*}'
    end
```

```
-- Note that even though prettyline is used above, it is not
-- necessary to define it before use.

function prettyline(indent,line)
-- \textbf for Lua keywords
    for n,word in pairs({'function','if','then','end','while',
        'do','return'}) do
        line = string.gsub(line,word,
            string.format('\\;\\textbf{%s}\\;',word))
        end
-- \textrm for multi-character identifiers
    for n,word in pairs({'print','math','mod','gcd'}) do
        line = string.gsub(line,word,
            string.format('\\textrm{%s}',word))
        end
-- there may be two keywords together
    line = string.gsub(line,'%s*\\;%s*\\;','\\;')
-- or keywords at the start of a line
    if string.sub(line,1,2)=='\\;'
        then line = string.sub(line,3,-1) end
    return '&'..string.rep('\\qquad ',indent)..
        string.sub(line,1,-2)..'\\\\'
    end
```