The TEX–Lua mix

Abstract
An introduction to the combination of TEX and the scripting language Lua.

Introduction

The idea of embedding Lua into TEX originates in some experiments with Lua embedded in the SciTE editor. You can add functionality to this editor by loading Lua scripts. This is accomplished by a library that gives access to the internals of the editing component.

The first integration of Lua in pdfTEX was relatively simple: from TEX one could call out to Lua and from Lua one could print to TEX. My first application was converting math written in a calculator syntax to TEX. Subsequent experiments dealt with MetaPost. At this point integration meant as little as: having some scripting language as an addition to the macro language. But, even in this early stage further possibilities were explored, for instance in manipulating the final output (i.e. the pdf code). The first versions of what by then was already called LuaTEX provided access to some internals, like counter and dimension registers and the dimensions of boxes.

Boosted by the Oriental TEX project, the team started exploring more fundamental possibilities: hooks in the input/output, tokenization, fonts and node lists. This was followed by opening up hyphenation, breaking lines into paragraphs and building ligatures. At that point we not only had access to some internals but also could influence the way TEX operates.

After that, an excursion was made to mplib, which fulfilled a long standing wish for a more natural integration of MetaPost into TEX. At that point we ended up with mixtures of TEX, Lua and MetaPost code.

As of mid-2008 we still need to open up more of TEX, like page building, math, alignments and the backend. Eventually LuaTEX will be nicely split up in components, rewritten in c, and we may even end up with Lua gluing together the components that make up the TEX engine. At that point the interoperation between TEX and Lua may be even richer than it is now.

In the next sections I will discuss some of the ideas behind LuaTEX and the relationship between Lua and TEX and how it presents itself to users. I will not discuss the interface itself, which consists of quite a number of functions (organized in pseudo-libraries) and the mechanisms used to access and replace internals (we call them callbacks).

TEX vs. Lua

TEX is a macro language. Everything boils down to either allowing stepwise expansion or explicitly preventing it. There are no real control features, like loops; tail recursion is a key concept. There are only a few accessible data structures, such as numbers, dimensions, glue, token lists and boxes. What happens inside TEX is controlled by variables, mostly hidden from view, and optimized within the constraints of 30 years ago.

The original idea behind TEX was that an author would write a specific collection of macros for each publication, but increasing popularity among non-programmers quickly resulted in distributed collections of macros, called macro packages. They started small but grew and grew and by now have become pretty large. In these packages there are macros dealing with fonts, structure, page layout, graphic inclusion, etc. There is also code dealing with user interfaces, process control, conversion and much of that code looks out of place: the lack of control features and string manipulation is solved by mimicking other languages, the unavailability of a float datatype is compensated by misusing dimension registers, and you can find provisions to force or inhibit expansion all over the place.

TEX is a powerful typographical programming language but lacks some of the handy features of scripting languages. Handy in the sense that you will need them when you want to go beyond the original purpose of the system. Lua is a powerful scripting language, but knows nothing of typesetting. To some extent it resembles the language that TEX was written in: Pascal. And, since Lua is meant for embedding and extending existing systems, it makes sense to bring Lua into TEX. How do they compare? Let's give some examples.

About the simplest example of using Lua in TEX is the following:

```
\directlua { tex.print(math.sqrt(10)) }
```

This kind of application is probably what most users will want and use, if they use Lua at all. However, we can go further than that.

Loops

In TEX a loop can be implemented as in the plain format (editorial line breaks, but with original comment):

```
\def\loop#1\repeat{\def\body{#1}\iterate}
\def\iterate{\body\let\next\iterate
  \else\let\next\relax\fi\next}
\let\repeat=\fi % this makes \loop..\if..\repeat
                % skippable
```

This is then used as:

```
\newcount \mycounter \mycounter=1
\loop
    ...
    \advance\mycounter 1
    \ifnum\mycounter < 11
\repeat
```

The definition shows a bit how TEX programming works. Of course such definitions can be wrapped in macros, like:

```
\forloop{1}{10}{1}{some action}
```

and this is what often happens in more complex macro packages. In order to use such control loops without side effects, the macro writer needs to take measures to permit, for instance, nested usage and avoid clashes between local variables (counters or macros) and user-defined ones. Above we used a counter in the condition, but in practice expressions will be more complex and this is not that trivial to implement.

The original definition of the iterator can be written a bit more efficiently:

```
\def\iterate
  {\body \expandafter\iterate \fi}
```

And indeed, in macro packages you will find many such expansion control primitives being used, which does not make reading macros easier.

Now, get me right, this does not make TEX less powerful, it's just that the language is focused on typesetting and not on general purpose programming, and in principle users can do without that: documents can be preprocessed using another language, and document specific styles can be used.

We have to keep in mind that TEX was written in a time when resources in terms of memory and cpu cycles were far less abundant than they are now. The 255 registers per class and (about) 3000 hash slots in original TEX were more than enough for typesetting a book, but in huge collections of macros they are not all that much. For that reason many macro packages use obscure names to hide their private registers from users and instead of allocating new ones with meaningful names, existing ones are shared. It is therefore not completely fair to compare TEX code with Lua code: in Lua we have plenty of memory and the only limitations are those imposed by modern computers.

In Lua, a loop looks like this:

```
for i=1,10 do
    ...
end
```

But while in the TEX example, the content directly ends up in the input stream, in Lua we need to do that explicitly, so in fact we will have:

```
for i=1,10 do
    tex.print("...")
end
```

And, in order to execute this code snippet, in LuaTEX we will do:

```
\directlua 0 {
    for i=1,10 do
        tex.print("...")
    end
}
```

So, eventually we will end up with more code than just Lua code, but still the loop itself looks quite readable and more complex loops are possible:

```
\directlua 0 {
    local t, n = { }, 0
    while true do
        local r = math.random(1,10)
        if not t[r] then
            t[r], n = true, n+1
            tex.print(r)
            if n == 10 then break end
        end
    end
}
```

This will typeset the numbers 1 to 10 in randomized order. Implementing a random number generator in pure TEX takes a fair amount of code and keeping track of already defined numbers in macros can be done with macros, but neither of these is very efficient.

Basic typesetting

I already stressed that TEX is a typographical programming language and as such some things in TEX are easier than in Lua, given some access to internals:

```
\setbox0=\hbox{x}\the\wd0
```

In Lua we can do this as follows:

```
\directlua 0 {
    local n = node.new('glyph')
    n.font = font.current()
    n.char = string.byte('x')
    tex.box[0] = node.hpack(n)
    tex.print(tex.wd[0]/65536 .. "pt")
}
```

One pitfall here is that TEX rounds off numbers differently than Lua. Both implementations can be wrapped in a macro resp. function:

```
\def\measured#1%
   {\setbox0=\hbox{#1}\the\wd0\relax}
```

Now we get:

```
\measured{x}
```

The same macro using Lua looks as follows:

```
\directlua 0 {
    function measure(chr)
        local n = node.new('glyph')
        n.font = font.current()
        n.char = string.byte(chr)
        tex.box[0] = node.hpack(n)
        tex.print(tex.wd[0]/65536 .. "pt")
    end
}
\def\measured#1{\directlua0{measure("#1")}}
```

In both cases, special tricks are needed if you want to pass for instance a # character to the TEX implementation, or a " to Lua; namely, using \# in the first case, and Lua's ``long strings" marked with double square brackets in the second.

This example is somewhat misleading. Imagine that we want to pass more than one character. The TEX variant is already suited for that, but the Lua function will now look like:

```
\directlua 0 {
    function measure(str)
        if str == "" then
            tex.print("0pt")
        else
            local head, tail = nil, nil
            for chr in str:gmatch(".") do
                local n = node.new('glyph')
                n.font = font.current()
                n.char = string.byte(chr)
                if not head then
                    head = n
                else
                    tail.next = n
                end
                tail = n
            end
            tex.box[0] = node.hpack(head)
            tex.print(tex.wd[0]/65536 .. "pt")
        end
    end
}
```

And still it's not okay, since TEX inserts kerns between characters (depending on the font) and glue between words, and doing all of this in Lua takes more code. So, it will be clear that although we will use Lua to implement advanced features, TEX itself still has quite a lot of work to do.

Typesetting stylistic variations

In the following examples we show code, but it is not of production quality. It just demonstrates a new way of dealing with text in TEX.

Occasionally a design demands that at some place the first character of each word should be uppercase, or that the first word of a paragraph should be in small caps, or that each first line of a paragraph has to be in dark blue. When using traditional TEX the user then has to fall back on parsing the data stream, and preferably you should then start such a sentence with a command that can pick up the text. For accentless languages like English this is quite doable but as soon as commands (for instance dealing with accents) enter the stream this process becomes quite hairy.

The next code shows how ConTEXt MkII defines the `\Word` and `\Words` macros that capitalize the first characters of a word or words. The spaces are really important here because they signal the end of a word.

```
\def\doWord#1%
  {\bgroup\the\everyuppercase\uppercase{#1}%
   \egroup}

\def\Word#1%
  {\doWord#1}
```

```
\def\doprocesswords#1 #2\od
  {\doifsomething{#1}{\processword{#1} % space!
  \doprocesswords#2 \od}}

\def\processwords#1%
  {\doprocesswords#1 \od\unskip}

\let\processword\relax

\def\Words
  {\let\processword\Word \processwords}
```

The code here is not that complex. We split off each word and feed it to a macro that picks up the first token (hopefully a character) which is then fed into the \uppercase primitive. This assumes that for each character a corresponding uppercase variant is defined using the \uccode primitive. Exceptions can be dealt with by assigning relevant code to the token register \everyuppercase. However, such macros are far from robust. What happens if the text is generated and not input as is? What happens with commands in the stream that do something with the following tokens?

A Lua-based solution could look as follows:

```
\def\Words#1{\directlua 0
for s in unicode.utf8.gmatch("#1", "([^ ])") do
   tex.sprint(string.upper(
                        s:sub(1,1)) .. s:sub(2))
 end
}
```

But there is no real advantage here, apart from the fact that less code is needed. We still operate on the input and therefore we need to look to a different kind of solution: operating on the node list.

```
function CapitalizeWords(head)
  local done = false
  local glyph = node.id("glyph")
  for start in node.traverse_id(glyph,head) do
    local prev, next = start.prev, start.next
    if prev and prev.id == kern
       and prev.subtype == 0 then
      prev = prev.prev
    end
    if next and next.id == kern
       and next.subtype == 0 then
      next = next.next
    end
    if (not prev or prev.id ~= glyph)
       and next and next.id == glyph then
      done = upper(start)
    end
  end
  return head, done
end
```

A node list is a forward-linked list. With a helper function in the node library we can loop over such lists. Instead of traversing we can use a regular while loop, but it is probably less efficient in this case. But how to apply this function to the relevant part of the input? In LuaTEX there are several callbacks that operate on the horizontal lists and we can use one of them to plug in this function. However, in that case the function is applied to probably more text than we want.

The solution for this is to assign attributes to the range of text which a function is intended to take care of. These attributes (there can be many) travel with the nodes. This is also a reason why such code normally is not written by end users, but by macro package writers: they need to provide the frameworks where you can plug in code. In ConTEXt we have several such mechanisms and therefore in MkIV this function looks (slightly simplified) as follows:

```
function cases.process(namespace,attribute,head)
  local done, actions = false, cases.actions
  for start in node.traverse_id(glyph,head) do
    local attr = has_attribute(start,attribute)
    if attr and attr > 0 then
      unset_attribute(start,attribute)
      local action = actions[attr]
      if action then
        local _, ok = action(start)
        done = done and ok
      end
    end
  end
  return head, done
end
```

Here we check attributes (these are set on the TEX side) and we have all kind of actions that can be applied, depending on the value of the attribute. Here the function that does the actual uppercasing is defined somewhere else. The cases table provides us a namespace; such namespaces need to be coordinated by macro package writers.

This approach means that the macro code looks completely different; in pseudo code:

```
\def\Words#1{{<setattribute><cases>
              <somevalue>#1}}
```

Or alternatively:

```
\def\StartWords {\begingroup
      <setattribute><cases><somevalue>}
```

```
\def\StopWords {\endgroup}
```

Because starting a paragraph with a group can have unwanted side effects (such as \everypar being expanded inside a group) a variant is:

```
\def\StartWords
    {<setattribute><cases><somevalue>}
\def\StopWords {<resetattribute><cases>}
```

So, what happens here is that the user sets an attribute using some high level command, and at some point during the transformation of the input into node lists, some action takes place. At that point commands, expansion and the like can no longer interfere.

In addition to some infrastructure, macro packages need to carry some knowledge, just as with the \uccode used in \uppercase. The upper function in the first example looks as follows:

```
local function upper(start)
  local data, char = characters.data, start.char
  if data[char] then
    local uc = data[char].uccode
    if uc and
       fonts.tfm.id[start.font].characters[uc]
    then
      start.char = uc
      return true
    end
  end
  return false
end
```

Such code is really macro package dependent: LuaTEX provides only the means, not the solutions. In ConTEXt we have collected information about characters in a data table in the characters namespace. There we have stored the uppercase codes (uccode). The fonts table, again ConTEXt specific, keeps track of all defined fonts and before we change the case, we make sure that this character is present in the font. Here id is the number by which LuaTEX keeps track of the used fonts. Each glyph node carries such a reference.

In this example, eventually we end up with more code than in TEX, but the solution is much more robust. Just imagine what would happen when in the TEX solution we would have:

```
\Words{\framed[offset=3pt]{hello world}}
```

It simply does not work. On the other hand, the Lua code never sees TEX commands, it only sees the two words represented by glyph nodes and separated by glue.

Of course, there is a danger when we start opening TEX's core features. Currently macro packages know what to expect, they know what TEX can and cannot do, and macro writers have exploited every corner of TEX, even the darkest ones. while the dirty tricks in The TEX-book had an educational purpose, those of users sometimes have obscene traits. If we just stick to the trickery introduced for parsing input, converting this into that, doing some calculations, and the like, it will be clear that Lua is more than welcome. It may hurt to throw away thousands of lines of impressive code and replace it by a few lines of Lua but that's the price the user pays for abusing TEX. Eventually ConTEXt MkIV will be a decent mix of Lua and TEX code, and hopefully the solutions programmed in those languages are as clean as possible.

Of course we can discuss until eternity whether Lua is the best choice. Taco, Hartmut and I are pretty confident that it is, and in the couple of years that we have been working on LuaTEX nothing has proved us wrong yet. One can fantasize about concepts, only to find out that they are impossible to implement or hard to agree on; we just go ahead using trial and error. We can talk over and over how opening up should be done, which is what the team does in a nicely closed and efficient loop, but at some points decisions have to be made. Nothing is perfect, neither is LuaTEX, but most users won't notice it as long as it extends TEX's life and makes usage more convenient.

Groups

Users of TEX and MetaPost will have noticed that both languages have their own grouping (scope) model. In TEX grouping is focused on content: by grouping the macro writer (or author) can limit the scope to a specific part of the text or have certain macros live within their own world.

```
.1. \bgroup .2. \egroup .1.
```

Everything done at 2 is local unless explicitly told otherwise. This means that users can write (and share) macros with a small chance of clashes. In MetaPost grouping is available too, but variables explicitly need to be saved.

```
.1. begingroup; save p; path p; .2. endgroup .1.
```

After using MetaPost for a while this feels quite natural because an enforced local scope demands multiple return values which is not part of the macro language. Actually, this is another fundamental difference between the languages: MetaPost has (a kind of) functions, which TEX lacks. In MetaPost you can write

```
draw origin
    for i=1 upto 10: ..(i,sin(i)) endfor;
```

but also:

```
draw some(0) for i=1 upto 10: ..some(i) endfor;
```

with

```
vardef some (expr i) =
    if i > 4 : i = i - 4 fi ;
    (i,sin(i))
enddef ;
```

The condition and assignment in no way interfere with the loop where this function is called, as long as some value is returned (a pair in this case).

In TEX things work differently. Take this:

```
\count0=1
\message{\advance\count0 by 1 \the\count0}
\the\count0
```

The terminal will show:

```
\advance \count 0 by 1 1
```

At the end the counter still has the value 1. There are quite a few situations like this, for instance when data such as a table of contents has to be written to a file. You cannot write macros where such calculations are done, hidden away, and only the result is seen.

The nice thing about the way Lua is presented to the user is that it permits the following:

```
\count0=1
\message{\directlua0{%
  tex.count[0] = tex.count[0] + 1}%
  \the\count0}
\the\count0
```

This will report 2 to the terminal and typeset a 2 in the document. Of course this does not solve everything, but it is a step forward. Also, compared to TEX and MetaPost, grouping is done differently: there is a `local` prefix that makes variables (and functions are variables too) local in modules, functions, conditions, loops, etc. The Lua code in this article contains such locals.

An example: XML

In practice most users will use a macro package and so, if a user sees TEX, he or she sees a user interface, not the code behind it. As such, they will also not encounter the code written in Lua that handles, for instance, fonts or node list manipulations. If a user sees Lua, it will most probably be in processing actual data. Therefore, in this section I will give an example of two ways to deal with xml: one more suitable for traditional TEX, and one inspired by Lua. It demonstrates how the availability of Lua can result in different solutions for the same problem.

MkII: stream-based processing

In ConTEXt MkII, the version that deals with pdfTEX and XETEX, we use a stream-based xml parser, written in TEX. Each < and & triggers a macro that then parses the tag and/or entity. This method is quite efficient in terms of memory but the associated code is not simple because it has to deal with attributes, namespaces and nesting.

The user interface is not that complex, but involves quite a few commands. Take for instance the following xml snippet:

```
<document>
    <section>
        <title>Whatever</title>
        <p>some text</p>
        <p>some more</p>
    </section>
</document>
```

When using ConTEXt commands, we can imagine the following definitions:

```
\defineXMLenvironment[document]
          {\starttext} {\stoptext}
\defineXMLargument   [title]
          {\section}
\defineXMLenvironment[p]
          {\ignorespaces}{\par}
```

When attributes have to be dealt with, for instance a reference to this section, things quickly start looking more complex. Also, users need to know what definitions to use in situations like this:

```
<table>
  <tr><td>first</td> ... <td>last</td></tr>
  <tr><td>left</td>  ... <td>right</td></tr>
</table>
```

Here we cannot be sure that a cell does not contain a nested table, which is why we need to define the mapping as follows:

```
\defineXMLnested[table]{\bTABLE} {\eTABLE}
\defineXMLnested[tr]   {\bTR}    {\eTR}
```

```
\defineXMLnested[td]   {\bTD}    {\eTD}
```

The `\defineXMLnested` macro is rather messy because it has to collect snippets and keep track of the nesting level, but users don't see that code, they just need to know when to use what macro. Once it works, it keeps working.

Unfortunately mappings from source to style are never that simple in real life. We usually need to collect, filter and relocate data. Of course this can be done before feeding the source to TEX, but MkII provides a few mechanisms for that too. For instance, to reverse the order you can do this:

```
<article>
    <title>Whatever</title>
    <author>Someone</author>
    <p>some text</p>
</article>

\defineXMLenvironment[article]
    {\defineXMLsave[author]}
    {\blank author: \XMLflush{author}}
```

This will save the content of the `author` element and flush it when the end tag `article` is seen. So, given previous definitions, we will get the title, some text and then the author. You may argue that instead we should use for instance xslt but even then a mapping is needed from the xml to TEX, and it's a matter of taste where the burden is put.

Because ConTEXt also wants to support standards like MathML, there are some more mechanisms but these are hidden from the user. And although these do a good job in most cases, the code associated with the solutions has never been satisfying.

Supporting xml this way is doable, and ConTEXt has used this method for many years in fairly complex situations. However, now that we have Lua available, it is possible to see if some things can be done more simply (or differently).

MkIV: tree-based processing

After some experimenting I decided to write a full blown xml parser in Lua, but contrary to the stream-based approach, this time the whole tree is loaded in memory. Although this uses more memory than a streaming solution, in practice the difference is not significant because often in MkII we also needed to store whole chunks.

Loading xml files in memory is very fast and once it is done we can have access to the elements in a way similar to xpath. We can selectively pipe data to TEX and manipulate content using TEX or Lua. In most cases this is faster than the stream-based method. An interesting fact is that we can do this without linking to existing xml libraries, and as a result we are pretty independent.

So how does this look from the perspective of the user? Say that we have the simple article definition stored in `demo.xml`.

```
<?xml version ='1.0'?>
<article>
    <title>Whatever</title>
    <author>Someone</author>
    <p>some text</p>
</article>
```

This time we associate so-called setups with the elements. Each element can have its own setup, and we can use expressions to assign them. Here we have just one such setup:

```
\startxmlsetups xml:document
   \xmlsetsetup{main}{article}{xml:article}
\stopxmlsetups
```

When loading the document it will automatically be associated with the tag `main`. The previous rule associates the setup `xml:article` with the `article` element in tree `main`. We register this setup so that it will be applied to the document after loading:

```
\xmlregistersetup{xml:document}
```

and the document itself is processed with (the empty braces are an optional setup argument):

```
\xmlprocessfile{main}{demo.xml}{}
```

The setup `xml:article` can look as follows:

```
\startxmlsetups xml:article
    \section{\xmltext{#1}{/title}}
    \xmlall{#1}{!(title|author)}
    \blank author: \xmltext{#1}{/author}
\stopxmlsetups
```

Here `#1` refers to the current node in the xml tree, in this case the root element, `article`. The second argument of `\xmltext` and `\xmlall` is a path expression, comparable to xpath: `/title` means: the `title` element anchored to the current root (`#1`), and `!(title|author)` is the negation of (complement to) `title` or `author`. Such expressions can be more complex than the one above, for instance:

```
\xmlfirst{#1}{/one/(alpha|beta)/two/text()}
```

which returns the content of the first element that satisfies one of the paths (nested tree):

```
/one/alpha/two
/one/beta/two
```

There is a whole bunch of commands like `\xmltext` that filter content and pipe it into TEX. These are calling Lua functions. This article is no manual, so we will not discuss them here. However, it is important to realize that we have to associate setups (consider them free formatted macros) with at least one element in order to get started. Also, xml inclusions have to be dealt with before assigning the setups. These are simple one-line commands. You can also assign defaults to elements, which saves some work.

Because we can use Lua to access the tree and manipulate content, we can now implement parts of xml handling in Lua. An example of this is dealing with so-called Cals tables. This is done in approximately 150 lines of Lua code, loaded at runtime in a module. This time the association uses functions instead of setups and those functions will pipe data back to TEX. In the module you will find:

```
\startxmlsetups xml:cals:process
    \xmlsetfunction {\xmldocument} {cals:table}
                    {lxml.cals.table}
\stopxmlsetups

\xmlregistersetup{xml:cals:process}
\xmlregisterns{cals}{cals}
```

These commands tell MkIV that elements with a namespace specification that contains `cals` will be remapped to the internal namespace `cals` and the setup associates a function with this internal namespace.

By now it will be clear that from the perspective of the user Lua is hardly visible. Sure, he or she can deduce that deep down some magic takes place, especially when you run into more complex expressions like this (the `@` denotes an attribute):

```
\xmlsetsetup
  {main}
  {item[@type='mpctext' or @type='mrtext']}
  {questions:multiple:text}
```

Such expressions resemble xpath, but can go much further, just by adding more functions to the library.

```
item[position() > 2 and position() < 5
    and text() == 'ok']
item[position() > 2 and position() < 5
    and text() == upper('ok')]
item[@n=='03' or @n=='08']
item[number(@n)>2 and number(@n)<6]
item[find(text(),'ALSO')]
```

Just to give you an idea, in the module that implements the parser you will find definitions that match the function calls in the above expressions.

```
xml.functions.find   = string.find
xml.functions.upper  = string.upper
xml.functions.number = tonumber
```

So much for the different approaches. It's up to the user what method to use: stream-based MkII, tree-based MkIV, or a mixture.

TEX–Lua in conversation

The main reason for taking xml as an example of mixing TEX and Lua is in that it can be a bit mind-boggling if you start thinking of what happens behind the scenes. Say that we have

```
<?xml version ='1.0'?>
<article>
    <title>Whatever</title>
    <author>Someone</author>
    <p>some <b>bold</b> text</p>
</article>
```

and we use the setup shown before with `article`.

At some point, we are done with defining setups and load the document. The first thing that happens is that the list of manipulations is applied: file inclusions are processed first, setups and functions are assigned next, maybe some elements are deleted or added, etc. When that is done we serialize the tree to TEX, starting with the root element. When piping data to TEX we use the current catcode regime; linebreaks and spaces are honored as usual.

Each element can have a function (command) associated and when this is the case, control is given to that function. In our case the root element has such a command, one that will trigger a setup. And so, instead of piping content to TEX, a function is called that lets TEX expand the macro that deals with this setup.

However, that setup itself calls Lua code that filters the title and feeds it into the `\section` command, next it flushes everything except the title and author, which again involves calling Lua. Last it flushes the author. The nested sequence of events is as follows:

lua: Load the document and apply setups and the like.

lua: Serialize the `article` element, but since there is an associated setup, tell TEX to expand that one instead.
- tex: Execute the setup, first expand the `\section` macro, but its argument is a call to Lua.
  - lua: Filter `title` from the subtree under `article`, print the content to TEX and return control to TEX.
- tex: Tell Lua to filter the paragraphs i.e. skip `title` and `author`; since the `b` element has no associated setup (or whatever) it is just serialized.
  - lua: Filter the requested elements and return control to TEX.
- tex: Ask Lua to filter `author`.
  - lua: Pipe `author`'s content to TEX.
- tex: We're done.

lua: We're done.

This is a very simple case. In my daily work I am dealing with rather extensive and complex educational documents where in one source there is text, math, graphics, all kind of fancy stuff, questions and answers in several categories and of different kinds, to be reshuffled or not, omitted or combined. So there we are talking about many more levels of TEX calling Lua and Lua piping to TEX, etc. To stay in TEX speak: we're dealing with one big ongoing nested expansion (because Lua calls expand), and you can imagine that this somewhat stresses TEX's input stack, but so far I have not encountered any problems.

Final remarks

Here I discuss several possible applications of Lua in TEX. I didn't mention yet that because LuaTEX contains a scripting engine plus some extra libraries, it can also be used purely for that. This means that support programs can now be written in Lua and that we need no longer depend on other scripting engines being present on the system. Consider this a bonus.

Usage in TEX can be categorized in four ways:

1. Users can use Lua for generating data, do all kind of data manipulations, maybe read data from file, etc. The only link with TEX is the print function.
2. Users can use information provided by TEX and use this when making decisions. An example is collecting data in boxes and use Lua to do calculations with the dimensions. Another example is a converter from MetaPost output to pdf literals. No real knowledge of TEX's internals is needed. The MkIV xml functionality discussed before demonstrates this: it's mostly data processing and piping to TEX. Other examples are dealing with buffers, defining character mappings, and handling error messages, verbatim ... the list is long.
3. Users can extend TEX's core functionality. An example is support for OpenType fonts: LuaTEX itself does not support this format directly, but provides ways to feed TEX with the relevant information. Support for OpenType features demands manipulating node lists. Knowledge of internals is a requirement. Advanced spacing and language specific features are made possible by node list manipulations and attributes. The alternative `\Words` macro is an example of this.
4. Users can replace existing TEX functionality. In MkIV there are numerous examples of this, for instance all file io is written in Lua, including reading from zip files and remote locations. Loading and defining fonts is also under Lua control. At some point MkIV will provide dedicated splitters for multicolumn typesetting and probably also better display spacing and display math splitting.

The boundaries between these categories are not set in stone. For instance, support for image inclusion and mplib in ConTEXt MkIV sits between categories 3 and 4. Categories 3 and 4, and probably also 2, are normally the domain of macro package writers and more advanced users who contribute to macro packages. Because a macro package has to provide some stability it is not a good idea to let users mess around with all those internals, due to potential interference. On the other hand, normally users operate on top of a kernel using some kind of api, and history has proved that macro packages are stable enough for this.

Sometime around 2010 the team expects LuaTEX to be feature complete and stable. By that time I can probably provide a more detailed categorization.

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