LuaTEX: What it takes to make a paragraph

Paul Isambert

Introduction

The road that leads from an input to an output document is rather eventful: bytes must be read, interpreted, executed, glyphs must be created, lines must be measured ... With LuaTEX those events can be monitored and their courses can be bent; this happens in callbacks, points in TEX's processing where custom code can be inserted. This paper will look at the callbacks involved from reading an input line to releasing the product of the paragraph builder to a vertical list. The callbacks that we will be looking at are the following:

`process_input_buffer`
How TEX reads each input line.

`token_filter`
How TEX deals with tokens.

`hyphenate`
Where discretionaries are inserted.

`ligaturing`
Where ligatures happen.

`kerning`
Where font kerns are inserted.

`pre_linebreak_filter`
Before the paragraph is built.

`linebreak_filter`
Where the paragraph is built.

`post_linebreak_filter`
After the paragraph is built.

Actually, a few more callbacks are involved, but these are most relevant to paragraph building.

Reading input lines

The `process_input_buffer` callback is executed when TEX needs an input line; the argument passed to the callback is the input line itself, and another line, possibly the same, should be returned. By default, nothing happens, and what TEX reads is what you type in your document.

The code in this paper has been written and tested with the latest build of LuaTEX. The reader probably uses the version released with the latest TEX Live or MikTEX distributions, and differences might occur. More recent versions can be regularly downloaded from TLContrib, a companion repository which hosts material that doesn't make it to TEX Live for whatever reason. Bleeding-edge LuaTEX can also be built from the sources.

A line in this context is actually a Lua string; hence what the callback is supposed to do is string manipulation. Besides, one should remember that this line hasn't been processed at all; for instance, material after a comment sign hasn't been removed, and multiple spaces haven't been reduced to one space; of course, escape characters followed by letters haven't been lumped into control sequences. In other words, the string is exactly the input line.

Can anything useful be done by manipulating input lines? Yes, in fact the `process_input_buffer` callback proves invaluable. Here I'll address two major uses: encoding and verbatim text.

Using any encoding. Unlike its elder brothers, LuaTEX is quite intolerant when it comes to encodings: it accepts UTF-8 and nothing else. Any sequence of bytes that does not denote a valid UTF-8 character makes it complain. Fortunately, ASCII is a subset of UTF-8, thus LuaTEX understands most older documents. For other encodings, however, input lines must be converted to UTF-8 before LuaTEX reads them. One main use of the `process_input_buffer` callback is thus to perform the conversion.

Converting a string involves the following steps (I'll restrict myself to 8-bit encodings here): mapping a byte to the character it denotes, more precisely to its numerical representation in Unicode; then turning that representation into the appropriate sequence of bytes. If the source encoding is Latin-1, the first part of this process is straightforward, because characters in Latin-1 have the same numerical representations as in Unicode. As for the second part, it is automatically done by the `slnunicode` Lua library (included in LuaTEX). Hence, here's some simple code that allows processing of documents encoded in Latin-1.

```
local function convert_char (ch)
  return unicode.utf8.char(string.byte(ch))
end
local function convert (line)
  return string.gsub(line, ".", convert_char)
end
callback.register("process_input_buffer", convert)
```

Each input line is passed to `convert`, which returns a version of that line where each byte has been replaced by one or more bytes to denote the same character in UTF-8. The Lua functions work as follows: `string.gsub` returns its first argument with each occurrence of its second argument replaced with the return value of its third argument (to which each match is passed). Since a dot represents all characters (i.e. all bytes, as far as

Lua is concerned), the entire string is processed piecewise; each character is turned into a numerical value thanks to `string.byte`, and this numerical value is turned back to one or more bytes denoting the same character in UTF-8.

What if the encoding one wants to use isn't Latin-1 but, say, Latin-3 (used to typeset Turkish, Maltese and Esperanto)? Then one has to map the number returned by `string.byte` to the right Unicode value. This is best done with a table in Lua: each cell is indexed by a number m between 0 and 255 and contains a number n such that character c is represented by m in Latin-3 and n in Unicode. For instance (numbers are given in hexadecimal form by prefixing them with `0x`):

```
latin3_table = { [0] = 0x0000, 0x0001, 0x0002,
  ...
  0x00FB, 0x00FC, 0x016D, 0x015D, 0x02D9}
```

This is the beginning and end of a table mapping Latin-3 to Unicode. At the beginning, m and n are equal, because all Latin-x encodings include ASCII. In the end, however, m and n differ. For instance, 'ŭ' is 253 in Latin-3 and 0x016D (365) in Unicode. Note that only index 0 needs to be explicitly specified (because Lua tables starts at 1 by default), all following entries are assigned to the right indexes.

Now it suffices to modify the `convert_char` function as follows to write in Latin-3:

```
local function convert_char (ch)
  return unicode.utf8.char
    (latin3_table[string.byte(ch)])
end
```

Verbatim text. One of the most arcane areas of TeX is catcode management. This becomes most important when one wants to print verbatim text, i.e. code that TeX should read as characters to be typeset only, with no special characters, and things turn definitely dirty when one wants to typeset a piece of code and execute it too (one generally has to use an external file). With the `process_input_buffer` callback, those limitations vanish: the lines we would normally pass to TeX can be stored and used in various ways afterward. Here's some basic code to do the trick; it involves another LuaTeX feature, catcode tables.

The general plan is as follows: some starting command, say `\Verbatim`, registers a function in the `process_input_buffer`, which stores lines in a table until it is told to unregister itself by way of a special line, e.g. a line containing only `\Endverbatim`. Then the table can be accessed and the lines printed or executed. The Lua side follows. (About the `\noexpand` in `store_lines`: we're assuming this Lua code is read via `\directlua` and not in a separate Lua file; if the latter is the case, then remove the `\noexpand`. It is used here to avoid having `\directlua` expand `\\`.)

```
local verb_table
local function store_lines (str)
  if str == "\noexpand\\Endverbatim" then
    callback.register("process_input_buffer",nil)
  else
    table.insert(verb_table, str)
  end
  return ""
end
function register_verbatim ()
  verb_table = {}
  callback.register("process_input_buffer",
      store_lines)
end
function print_lines (catcode)
  if catcode then
    tex.print(catcode, verb_table)
  else
    tex.print(verb_table)
  end
end
```

The `store_lines` function adds each line to a table, unless the line contains only `\Endverbatim` (a regular expression could also be used to allow more sophisticated end-of-verbatims), in which case it removes itself from the callback; most importantly, it returns an empty string, because if it returned nothing then LuaTeX would proceed as if the callback had never happened and pass the original line. The `register_verbatim` function only resets the table and registers the previous function; it is not `local` because we'll use it in a TeX macro presently. Finally, the `print_lines` uses `tex.print` to make TeX read the lines; a catcode table number can be used, in which case those lines (and only those lines) will be read with the associated catcode regime. Before discussing catcode tables, here are the relevant TeX macros:

```
\def\Verbatim{%
  \directlua{register_verbatim()}%
}
\def\useverbatim{%
  \directlua{print_lines()}%
}
\def\printverbatim{%
  \bgroup\parindent=0pt \tt
  \directlua{print_lines(1)}
  \egroup
}
```

They are reasonably straightforward: `\Verbatim` launches the main Lua function, `\useverbatim` reads the lines, while \printverbatim also reads them but with catcode table 1 and a typewriter font, as is customary to print code. The latter macro could also be launched automatically when store_lines is finished.

What is a catcode table, then? As its name indicates, it is a table that stores catcodes, more precisely the catcodes in use when it was created. It can then be called to switch to those catcodes. To create and use catcode table 1 in the code above, the following (or similar) should be performed:

```
\def\createcatcodes{\bgroup
  \catcode`\\=12 \catcode`\{=12 \catcode`\}=12
  \catcode`\$=12 \catcode`\&=12 \catcode`\^^M=13
  \catcode`\#=12 \catcode`\^=12 \catcode`\_=12
  \catcode`\ =13 \catcode`\~=12 \catcode`\%=12
  \savecatcodetable 1
\egroup}
\createcatcodes
```

The \savecatcodetable primitive saves the current catcodes in the table denoted by the number; in this case it stores the customary verbatim catcodes. Note that a common difficulty of traditional verbatim is avoided here: suppose the user has defined some character as active; then when printing code s/he must make sure that the character is assigned a default (printable) catcode, otherwise it might be executed when it should be typeset. Here this can't happen: the character (supposedly) has a normal catcode, so when table 1 is called it will be treated with that catcode, and not as an active character.

Once defined, a catcode table can be switched with \catcodetable followed by a number, or they can be used in Lua with tex.print and similar functions, as above.

As usual, we have set space and end-of-line to active characters in our table 1; we should then define them accordingly, although there's nothing new here:

```
\def\Space{ }
\bgroup
\catcode`\^^M=13\gdef^^M{\quitvmode\par}%
\catcode`\ = 13\gdef {\quitvmode\Space}%
\egroup
```

Now, after

```
\Verbatim
\def\luatex{%
  Lua\kern-.01em\TeX
  }%
\Endverbatim
```

one can use \printverbatim to typeset the code and \useverbatim to define \luatex to LuaTeX. The approach can be refined: for instance, here each new verbatim text erases the preceding one, but one could assign the stored material to tables accessible with a name, and \printverbatim and \useverbatim could take an argument to refer to a specific piece of code; other catcode tables could also be used, with both macros (and not only \printverbatim). Also, when typesetting, the lines could be interspersed with macros obeying the normal catcode regime (thanks to successive calls to tex.print, or rather tex.sprint, which processes its material as if it were in the middle of a line), and the text could be acted on.

Reading tokens

Now our line has been processed, and TeX must read its contents. What is read might actually be quite different from what the previous callback has returned, because some familiar operations have also taken place: material after a comment sign has been discarded, end-of-line characters have been turned to space, blank lines to \par, escape characters and letters have been lumped into control sequences, multiple spaces have been reduced to one ... What TeX reads are tokens, and what tokens are read is decided by the token_filter callback.

Nothing is passed to the callback: it must fetch the next token and pass it (or not) to TeX. To do so, the token.get_next function is available, which, as its name indicates, gets the next token from the input (either the source document or resulting from macro expansion).

In LuaTeX, a token is represented as a table with three entries containing numbers: entry 1 is the command code, which roughly tells TeX what to do. For instance, letters have command code 11 (not coincidentally equivalent to their catcode), whereas a { has command code 1: TeX is supposed to behave differently in each case. Most other command codes (there are 138 of them for the moment) denote primitives (the curious reader can take a look at the first lines of the luatoken.w file in the LuaTeX source). Entry 2 is the command modifier: it distinguishes tokens with the same entry 1: for letters and 'others', the command modifier is the character code; if the token is a command, it specifies its behavior: for instance, all conditionals have the same entry 1 but differ in entry 2. Finally, entry 3 points into the equivalence table for commands, and is 0 otherwise.

To illustrate the token_filter callback, let's address an old issue in TeX: verbatim text as argument to a command. It is, traditionally, impossible, at least without higher-order wizardry (less so with ε-TeX). It is also actually impossible with LuaTeX, for the reasons mentioned in the first paragraph of this section: commented material has already been discarded, multiple spaces have been reduced, etc. However, for short snippets, our pseudo-verbatim will be quite useful and easy. Let's restate the problem. Suppose we want to be able to write something like:

```
... some fascinating code%
\footnote*{That is \verb"\def\luatex{Lua\TeX}".}
```

i.e. we want verbatim code to appear in a footnote. This can't be done by traditional means, because `\footnote` scans its argument, including the code, and fixes catcodes; hence `\def` is a control sequence and cannot be turned back to four characters. The code below doesn't change that state of affairs; instead it examines and manipulates tokens in the `token_filter` callback. Here's the TeX side (which uses `"..."` instead of the more verbose `\verb"..."`); it simply opens a group, switches to a typewriter font, and registers our Lua function in the callback:

```
\catcode`\"=13
\def"{\bgroup\tt
  \directlua{callback.register("token_filter",
                               verbatim)}%
}
```

And now the Lua side:

```
function verbatim ()
  local t = token.get_next()
  if t[1] > 0 and t[1] < 13 then
    if t[2] == 34 then
      callback.register("token_filter", nil)
      return token.create("egroup")
    else
      local cat = (t[2] == 32 and 10 or 12)
      return {cat, t[2], t[3]}
    end
  else
    return {token.create("string"), t}
  end
end
```

It reads as follows: first we fetch the next token. If it isn't a command, i.e. if its command code is between 1 and 12, then it may be the closing double quote, with character code 34; in this case, we unregister the function and pass to TeX a token created on the fly with `token.create`, a function that produces a token from (among others) a string: here we simply generate `\egroup`. If the character isn't a double quote, we return it but change its command code (i.e. its catcode) to 12 (or 10 if it is a space), thus turning specials to simple characters (letters also lose their original catcode, but that is harmless). We return our token as a table with the three entries mentioned above for the token representation. Finally, if the token is a command, we return a table representing a list of tokens which TeX will read one after the other: the first is `\string`, the second is the original token.

If the reader experiments with the code, s/he might discover that the double quote is actually seen twice: first, when it is active (hence, a command), and prefixed with `\string`; then as the result of the latter operation. Only then does it shut off the processing of tokens.

Inserting discretionaries

Now TeX has read and interpreted tokens. Among the things which have happened, we will now be interested in the following: the nodes that TeX has created and concatenated into a horizontal list. This is where typesetting proper begins. The `hyphenate` callback receives the list of nodes that is the raw material with which the paragraph will be built; it is meant to insert hyphenation points, which it does by default if no function is registered.

In this callback and others, it is instructive to know what nodes are passed, so here's a convenient function that takes a list of nodes and prints their `id` fields to the terminal and log (what number denotes what type of node is explained in chapter 8 of the LuaTeX reference manual), unless the node is a glyph node (`id` 37, but better to get the right number with `node.id`), in which case it directly prints the character:

```
local GLYF = node.id("glyph")
function show_nodes (head)
  local nodes = ""
  for item in node.traverse(head) do
    local i = item.id
    if i == GLYF then
      i = unicode.utf8.char(item.char)
    end
    nodes = nodes .. i .. " "
  end
  texio.write_nl(nodes)
end
```

Let's register it at once in the `hyphenate` callback:

```
callback.register("hyphenate", show_nodes)
```

No hyphenation point will be inserted for the moment, we'll take care of that later.

Now suppose we're at the beginning of some kind of postmodern minimalist novel. It starts with a terse paragraph containing exactly two words:

Your office.

What list of nodes does the `hyphenate` callback receive? Our `show_nodes` function tells us:

$\boxed{50}\ \boxed{8}\ \boxed{0}\ \boxed{Y}\ \boxed{o}\ \boxed{u}\ \boxed{r}\ \boxed{10}\ \boxed{O}\ \boxed{f}\ \boxed{f}\ \boxed{i}\ \boxed{c}\ \boxed{e}\ \boxed{.}\ \boxed{10}$

First comes a `temp` node; it is there for technical reasons and is of little interest. The node with `id` 8 is a whatsit, and if we asked we'd learn its subtype is 6, so it is a `local_par` whatsit and contains, among other things, the paragraph's direction of writing. The third node is a horizontal list, i.e. an hbox; its subtype (3) indicates that it is the indentation box, and if we queried its width we would be returned the value of `\parindent` (when the paragraph was started) in scaled points (to be divided by 65,536 to yield a value in TEX points).

The nodes representing characters have many fields, among them `char` (a number), which our `show_nodes` function uses to print something a little more telling than an `id` number, `width`, `height` and `depth` (numbers too, expressing dimensions in scaled points), and `font` (yet another number: fonts are internally represented by numbers). Their `subtype` field will be of interest later.

Finally, the nodes with `id` 10 are glues, i.e. the space between the two words and the space that comes from the paragraph's end of line (which wouldn't be there if the last character was immediately followed by `\par` or a comment sign). Their specifications can be accessed via subfields to their `spec` fields (because a glue's specs constitute a node by themselves).

Now, what can be done in this callback? Well, first and foremost, insert hyphenation points into our list of nodes as LuaTEX would have done by itself, had we left the callback empty. The `lang.hyphenate` function does this:

```
callback.register("hyphenate",
  function (head, tail)
    lang.hyphenate(head)
    show_nodes(head)
  end)
```

There is no need to return the list, because LuaTEX takes care of it in this callback, as is also the case with the `ligaturing` and `kerning` callbacks. Also, those three callbacks take two arguments: `head` and `tail`, respectively the first and last nodes of the list to be processed. The tail can generally be ignored.

Now we we can see what hyphenation produces:

$\boxed{50}\ \boxed{8}\ \boxed{0}\ \boxed{Y}\ \boxed{o}\ \boxed{u}\ \boxed{r}\ \boxed{10}\ \boxed{o}\ \boxed{f}\ \boxed{7}\ \boxed{f}\ \boxed{i}\ \boxed{c}\ \boxed{e}\ \boxed{.}\ \boxed{10}$

As expected, a discretionary has been inserted with `id` 7; it is a discretionary node, with `pre`, `post` and `replace` fields, which are equivalent to the first, second and third arguments of a `\discretionary` command: the `pre` is the list of nodes to be inserted before the line break, the `post` is the list of nodes to be inserted after the line break, and the `replace` is the list of nodes to be inserted if the hyphenation point isn't chosen. In our case, the `pre` field contains a list with only one node, a hyphen character, and the other fields are empty.

A final word on hyphenation. The exceptions loaded in `\hyphenation` can now contain the equivalent of `\discretionary`, by inserting `{pre}{post}{replace}` sequences; German users (and probably users of many other languages) will be delighted to know that they no longer need to take special care of *backen* in their document; a declaration such as the following suffices:

```
\hyphenation{ba{k-}{}{c}ken}
```

Also, with a hyphen as the first and third arguments, compound words can be hyphenated properly.

Ligatures

As its name indicates, the `ligaturing` callback is supposed to insert ligatures (this happens by itself if no function is registered). If we used the `show_nodes` function here, we'd see no difference from the latest output, because that callback immediately follows `hyphenate`. But we can register our function after ligatures have been inserted with the `node.ligaturing` function (again, no return value):

```
callback.register("ligaturing",
  function (head, tail)
    node.ligaturing(head)
    show_nodes(head)
  end)
```

And this returns:

$\boxed{50}\ \boxed{8}\ \boxed{0}\ \boxed{Y}\ \boxed{o}\ \boxed{u}\ \boxed{r}\ \boxed{10}\ \boxed{o}\ \boxed{7}\ \boxed{c}\ \boxed{e}\ \boxed{.}\ \boxed{10}$

Did something go wrong? Why is *office* thus mangled? Simply because there is an interaction between hyphenation and ligaturing. If the hyphenation point is chosen, then the result is `of-<fi>ce`, where `<fi>` represents a ligature; if the hyphenation point isn't chosen, then we end up with `o<ffi>ce`, i.e. another ligature; in other words, what ligature is chosen depends on hyphenation. Thus the discretionary node has `f-` in its `pre` field, `<fi>` in `post` and `<ffi>` in `replace`.

Ligature nodes are glyph nodes with `subtype` 2, whereas normal glyphs have `subtype` 1; as such, they have a special field, `components`, which points to a node list made of the individual glyphs that make up the ligature. For instance, the components of an `<ffi>` ligature are `<ff>` and `i`, and the components of `<ff>` are `f` and `f`. Ligatures can thus be decomposed when necessary.

How does LuaTEX (either as the default behavior of the `ligaturing` callback or as the `node.ligaturing` function) know what sequence of glyph nodes should result in a ligature? The information is encoded in the font: LuaTeX looks up the `ligatures` table associated (if any) with each character, and if the following character is included in that table, then a ligature is created. For instance, for `f` in Computer Modern Roman, the `ligatures` table has a cell at index 105, that is `i`, which points to character 12, which contains the `<fi>` ligature. Thus, LuaTeX knows nothing about ligatures involving more than two glyphs. Even the `<ffi>` ligature is a ligature between `<ff>` and `i`.

However, fonts, especially of the OpenType breed, sometimes define ligatures with more than two glyphs; for instance, the input `3/4` is supposed to produce something like ¾ (a single glyph). One can choose, when creating the font from the OpenType file, to create a phantom ligature `<3/>` and make ¾ a ligature between `<3/>` and `4`; then LuaTeX can handle it automatically. It is more elegant and less error-prone, though, to deal with such ligatures by hand, so to speak: register a function in the `ligaturing` callback which, given a string of nodes, creates a ligature. It is also slower.

Also in this callback, such things as contextual substitutions should take place. For instance, initial and final forms of a glyph, be it in Arabic or in some flourished Latin font, should be handled here. In theory, that is quite easy: check the context of a node, i.e. the surrounding nodes; if it matches the context for a given substitution, then apply it. For instance, if our example paragraph were typeset in Minion (shipped gratis with Adobe Reader) with the `ss02` feature on, the *r* of *Your* and the *e* of *office* would be replaced by their final variants, because the contexts match: *r* is followed by a glue and *e* is followed by a stop (technically, they're not followed by glyphs inhibiting the substitution, that is, glyphs denoting a letter). In practice, however, things are more complicated, if only because you have to read such contextual substitutions from the font file.

However, we can perform a very simple type of contextual substitution. Code used to load a font in LuaTeX generally applies the `trep` feature (inspired by XƎTeX), so that the grave and single quote characters are replaced with left and right quotes; but one might want to be lazier still and use `"` everywhere; then the proper quotation mark should be substituted, depending on where the double quote occur.

Here's some simple code to implement this rule for such substitutions: if `"` is found, replace it with ' if the node immediately preceding (if any) is a glyph and its character isn't a left parenthesis; otherwise, replace it with ‘. (Here and elsewhere, I use `not (x == y)` where `x ~= y` would be simpler, but `~` would be expanded in `\directlua`, and `x \noexpand~= y` isn't so simple anymore.)

```
local GLYF = node.id("glyph")
callback.register("ligaturing",
  function (head)
    for glyph in node.traverse_id(GLYF, head) do
      if glyph.char == 34 then
        if glyph.prev and glyph.prev.id == GLYF
           and not (glyph.prev.char == 40) then
          glyph.char = 39
        else
          glyph.char = 96
        end
      end
    end
    node.ligaturing(head)
  end)
```

Note that we still apply `node.ligaturing`. Now one can use `"word"` to print ‘word’ and thus rediscover the thrill of modern word processors.

Inserting kerns

Analogous to `ligaturing`, `kerning` is supposed to insert font kerns, and again this happens by itself if no function is registered. Furthermore, nothing happens between the `ligaturing` and `kerning` callbacks, so again, using `show_nodes` would be uninformative. The equivalent function in this case is `node.kerning`, so we can do:

```
callback.register("kerning",
  function (head, tail)
    node.kerning(head)
    show_nodes(head)
  end)
```

And the result is:

$\boxed{50}\ \boxed{8}\ \boxed{0}\ \boxed{Y}\ \boxed{11}\ \boxed{o}\ \boxed{u}\ \boxed{r}\ \boxed{10}\ \boxed{o}\ \boxed{7}\ \boxed{c}\ \boxed{e}\ \boxed{.}\ \boxed{10}$

What has changed is that a node of `id` 11, i.e. a kern, has been inserted, because an *o* after a *Y* looks better if it is moved closer to it. Compare ‘Yo’ and ‘Yo’. Such kerns are specified in the fonts, like ligatures, which make sense, since kerns, like ligatures, depends on the glyphs' design.

Like contextual ligatures and substitutions, there is contextual positioning. Kerns are encoded in the font (as internalized by LuaTeX) like ligatures, i.e. glyphs have a `kerns` table indexed with character numbers and dimensions (in scaled points) as values; hence kerning is automatic with glyph pairs only, and contextual positioning should be made by hand. For instance, in ‘A.V.’, a (negative) kern should be inserted between the first stop and the *V*; however, this should happen only when the stop is preceded by an *A*; if the first letter were *T* (i.e. 'T.V.'), the kern is much less desirable (or at least a different amount of kerning should be used).

Finally, some hand-made kerning is to take place here too. For instance, French typographic rules require that a thin space be inserted before some punctuation marks (not so thin, actually, which sometimes verges on the ugly, but one is free to adopt Robert Bringhurst's 'Channel Island compromise'), but there are some exceptions: for instance, although a question mark should be preceded by such a space, the rule obviously doesn't apply if the mark follows an opening parenthesis (?) or another question mark. Technically, this could be encoded in the font; in practice, it is much easier to handle in the `kerning` callback. If one chooses to do so, one should make sure all the newly inserted kerns are of `subtype` 1, because kerns of `subtype` 0 are font kerns and might be reset if the paragraph is built with font expansion.

One last callback before building the paragraph

The previous callbacks apply no matter whether we're building a paragraph or creating an `\hbox`. The ones we'll see now are used only in the first case, i.e. (TeXnically speaking) when a vertical command is encountered in unrestricted horizontal mode. The first of those is `pre_linebreak_filter`, and if we use `show_nodes` on the list it is given, then we notice that something has happened between the `kerning` callback and now:

$$\boxed{8}\,\boxed{0}\,\boxed{\text{Y}}\,\boxed{11}\,\boxed{\text{o}}\,\boxed{\text{u}}\,\boxed{\text{r}}\,\boxed{10}\,\boxed{\text{o}}\,\boxed{7}\,\boxed{\text{c}}\,\boxed{\text{e}}\,\boxed{.}\,\boxed{12}\,\boxed{10}$$

First, the temporary node at the beginning of the list has been removed; it is not needed anymore, but from now on we should always return the list. Hence, the `show_nodes` call would have been embedded in:

```
callback.register("pre_linebreak_filter",
  function (head)
    show_nodes(head)
    return head
  end)
```

Second, a new node has been inserted, with `id` 12: that is a penalty. If we queried its `penalty` field, it'd return 10,000. Where does the infinite penalty come from? The reader might know that, when preparing to build a paragraph, TeX removes a last space (i.e. the last glue node) of the horizontal list and replaces it with a glue whose value is `\parfillskip`, and prefixes the latter with an infinite penalty so no line break can occur. That is what has happened here: the last node is a glue (`id` 10), but not the same as before, as its `subtype` (15) would indicate: it is the `\parfillskip` glue.

Nothing in particular is supposed to happen in the `pre_linebreak_filter` callback, and TeX does nothing by default. The callback is used for special effects before the list is broken into lines; its arguments are the head of the list to be processed and a string indicating in which circumstances the paragraph is being built; relevant values for the latter are an empty string (we're in the main vertical list), `vbox`, `vtop` and `insert`.

At last, building the paragraph!

Finally we must build the paragraph. To do so we use the `linebreak_filter` callback; by default, paragraph building is automatic (fortunately), but if we register a function in the callback we should break lines by ourselves. Well, more or less: as usual, there is a function, `tex.linebreak`, which does exactly that

The callback receives two arguments: a list of nodes and a boolean; the latter is `true` if we're building the part of a larger paragraph before a math display, otherwise it is false. Now, given the list of nodes, one must return another list of an entirely different nature: it should be made of horizontal boxes (lines of text), glues (interline glues), penalties (e.g. widow and club penalties), perhaps inserts or `\vadjust`-ed material, etc. As just mentioned, the `tex.linebreak` function does all this; it can also take an optional argument, a table with TeX parameters as keys (for instance `hsize`, `tolerance`, `widowpenalty`), so paragraphs can easily be built with special values.

As an example of paragraph building, let's address the issue of setting a paragraph's first line in small caps, as is often done for the first paragraph of a chapter. We're using LuaTeX, so we don't want any dirty trick, and we want TeX to build the best paragraph (i.e. we don't want to simply mess with space in the first line), which includes the possibility that the first line is hyphenated. The code below is just a sketch, but it gives an overview of the approach. First things first, we need a font, and we need a macro to declare that the next paragraph should be treated in a special way:

```
\font\firstlinefont=cmcsc10
\def\firstparagraph{\directlua{
 callback.register("hyphenate", false)
 callback.register("ligaturing", false)
 callback.register("kerning", false)
 callback.register("linebreak_filter",
  function (head, is_display)
    local par, prevdepth, prevgraf =
      check_par(head)
    tex.nest[tex.nest.ptr].prevdepth=prevdepth
    tex.nest[tex.nest.ptr].prevgraf=prevgraf
    callback.register("hyphenate", nil)
    callback.register("ligaturing", nil)
    callback.register("kerning", nil)
    callback.register("linebreak_filter", nil)
    return par
  end)}}
```

First, we deactivate the first three node-processing callbacks by registering `false`, because we want to keep the original list of nodes with only the replacements that occur before the `pre_linebreak_filter` callback; we'll do hyphenating, ligaturing and kerning by hand. In practice, it would be preferable to retrieve the functions (if any) that might be registered in those callbacks and use those, because there might be more than what is done by default. We could do that with `callback.find` but won't bother here.

Next, we register `linebreak_filter` function that calls `check_par`; the latter will return a paragraph and the new value for `prevdepth` and `prevgraf`, so we can set the values for the current nesting level (the list we're in) by hand (it isn't done automatically). Finally, we return the callbacks to their default behavior by registering `nil`.

Before turning to the main `check_par` function, here's a subfunction that it uses to do the job we've prevented LuaTEX from doing by itself: insert hyphenation points, ligatures and kerns, and then build the paragraph. There's no need to set `head` to the return value of `lang.hyphenate`, since no new head can be produced (no hyphenation point can be inserted at the beginning of the list), and anyway `lang.hyphenate` returns a boolean indicating success or failure. Besides the paragraph itself, `tex.linebreak` also returns a table with the values of `prevdepth` and `prevgraf` (and also `looseness` and `demerits`). The last line of the code retrieves the inner numerical representation of the font we've chosen for the first line.

```
local function do_par (head)
  lang.hyphenate(head)
  head = node.ligaturing(head)
  head = node.kerning(head)
  local p, i = tex.linebreak(head)
  return p, i.prevdepth, i.prevgraf
end
local firstlinefont = font.id("firstlinefont")
```

Now we can turn to the big one, called by `linebreak_filter`. First, it builds a tentative paragraph; it works on a copy of the original list because we don't want to spoil it with hyphenation points that might be removed later. Then it finds the first line of the paragraph (the head of the paragraph list might be a glue, or `\vadjust-pre`'d material).

```
local HLIST = node.id("hlist")
local GLYF  = node.id("glyph")
local KERN  = node.id("kern")
function check_par (head)
  local par =  node.copy_list(head)
  par, prevdepth, prevgraf = do_par(par)
  local line = par
  while not (line.id == HLIST) do
    line = line.next
  end
```

Next, in that first line, we check whether all glyphs have the right font; as soon as we find one which isn't typeset in small caps (our `firstlinefont`), we review all the glyphs in the original list until we find the first one that isn't typeset in small caps, and we change its font as we want it. The reader can perhaps see where this is headed: we'll rebuild the paragraph as often as necessary, each time turning one more glyph of the original horizontal list to a small capital, until all the glyphs in the first line are small caps; that is also why we must reinsert hyphenation points, ligatures and kerns each time: fonts have changed, so the typesetting process must be relaunched from the start.*

```
  local again
  for item in node.traverse_id(GLYF, line.head)
  do if not (item.font == firstlinefont) then
    again = true
    for glyph in node.traverse_id(GLYF, head)
    do if not (glyph.font == firstlinefont) then
      glyph.font = firstlinefont
      break
    end; end
    break
  end; end
```

If we must typeset `again`, free the paragraph from TEX's memory and start again with the modified head:

```
  if again then
    node.flush_list(par)
    return check_par(head)
```

* The user might wonder what `line.head` stands for in the second line; that is the same thing as `line.list`, i.e. it gets the contents of a list (its first node). Since LuaTEX v.0.65, `list` has been replaced with `head` for reasons not so clearly explained in my previous paper (see *TUGboat* 31:3); `list` should remain (deprecated) until around v.0.8.

Otherwise (our first line is good, all glyphs are small caps), there's one more thing to check; suppose the last character we turned to small capital was x. By definition, x is at the end of the first line before its font is changed; but is it still the case after the change? Not necessarily: TEX may very well have decided that, given x's new dimensions, it should be better to break before — and perhaps not immediately before x but a couple glyphs before. So perhaps we ended up with small capitals in the second line. They must be removed, but how? Turn them back to lowercase and build the paragraph again? No, definitely not, we'd be stuck in a loop (lowercase in the first line, small caps in the second line, and again ...). The solution adopted here is to turn those glyphs to the original font (say `\tenrm`) and keep them where they are:

```
else
  local secondline = line.next
  while secondline
        and not (secondline.id == HLIST) do
    secondline = secondline.next
  end
  if secondline then
    local list = secondline.head
    for item in node.traverse_id(GLYF,list)
    do if item.font == firstlinefont then
        item.font = font.id("tenrm")
      else
        break
    end; end
```

Now, what if those first glyphs in the second line were *f* and *i*; in small caps they presumably did not form a ligature, but now? We should reapply ligatures. And what about kerning? We should remove all font kerns (they have `subtype` 0) and also reapply kerning. Finally we should repack the line to its original width, so that glues are stretched or shrunken to the appropriate values. That is not optimal, but such cases where small caps end up in the second line are very rare.

The last lines delete the original list and return the paragraph with the associated parameters.

```
    list = node.ligaturing(list)
    for kern in node.traverse_id(KERN, list)
    do if kern.subtype == 0 then
        node.remove(list, kern)
    end; end
    list = node.kerning(list)
    secondline.head = node.hpack(
           list, secondline.width, "exactly")
  end
  node.flush_list(head)
  return par, prevdepth, prevgraf
end
end
```

The reader may have spotted more than one flaw in this code. A full solution would have greatly exceeded the limits of this already quite long article. So it is left as an exercise: work out a solution that doesn't rely on the assumption that no functions are registered in the other callbacks, for instance. Or give an alternative way to cope with small capitals in the second line (rebuild the paragraph from that line on?).

Handling the paragraph

The `post_linebreak_filter` callback is very calm after all we've just been through: nothing happens by default. It is passed what `linebreak_filter` returns as its first argument, i.e. a list of horizontal lists, penalties, glues, and perhaps interline material (e.g. inserts). It also receives a second argument, a string as with the `pre_linebreak_filter` callback. In my previous paper, I gave examples of what can be done here, for instance underlining. I won't give another example, but the reader looking for practise could try to adapt to LuaTEX Victor Eijkhout's code in section 5.9.6 of *TEX by Topic*.

The callback should return a paragraph, possibly the same as the one it was passed. That paragraph is then appended to the surrounded vertical list, and what follows is the job of the page builder. Our exploration ends here.

Conclusion

Most of the operations we have reviewed aren't new in TEX: LuaTEX simply gives access to them. Since the very beginning, TEX has read lines and tokens and built lists of nodes (although the hyphenating/ ligaturing pass has changed a lot in LuaTEX); that is its job. Control over the typesetting process is what makes TEX so good, perhaps better than any other typography software; LuaTEX brings that control one step further and allows manipulating of the very atoms that make digital typography: characters and glyphs, and a few other technical bells and whistles. In view of the freedom that has been gained, I sometimes tend to find TEX82 and its offspring a bit dictatorial, in retrospect.

◇ Paul Isambert
Université de la Sorbonne Nouvelle
France
`zappathustra (at) free dot fr`