OpenType fonts in LuaTeX

Paul Isambert

1 Introduction

As is well-known, LuaTeX can handle standard font formats, notably including OpenType. That's a welcome development because modern font designs use those formats almost exclusively, and whatever the merits of METAFONT, for modern typographic software to stick to it would be suicidal. Lesser known perhaps is that, unlike XeTeX, which opened the way, LuaTeX is completely unable to load such a font if you don't feed it a non-negligible amount of code beforehand. Otherwise it only understands your old TFMs (it actually embeds Type 1 fonts in PDF documents, a behavior inherited from PDFTeX, but only because a mapping exists between the TFM and Type 1 fonts; the latter can't be read directly). The reason is not that LuaTeX isn't so capable after all and you have to rely on some work-around; rather, LuaTeX is consistent with its philosophy (as I see it): it provides tools, not solutions. So you have to do most of the work to make it understand OpenType fonts, and that's no simple work, but in the process you gain freedom.

In this paper I'll try to describe such code. I won't give an entire implementation, and in many places I'll just go with "This or that should be done", because as already mentioned it would be extremely long (and tedious). ConTeXt's fontloader, available for plain TeX and LaTeX as `luaotfload`, is more than 10,000 lines long, and my own code, which doesn't even try to address non-Latin typography, is 2,000 lines. In other words, I'll give a map of the area to the reader, but nothing can replace the actual exploration.

Also, this paper has limitations: first, all my examples will use the Latin alphabet, even though some features would have been better illustrated with other scripts; I apologize to users of other writing systems, but I thought it better not to pretend I was competent in them. This extends to maths, so OpenType maths aren't covered at all; [2,7] should help the interested reader. The omission of maths is even more significant than for non-Latin writing systems, since the latter at least rely on the mechanisms described here, whereas OpenType maths are an entirely distinct area.

Second, I will have nothing to say about AAT fonts. I have never used them, let alone figured out how they work, and anyway that would have pushed the length of this paper beyond reasonable limits. I hereby invite the courageous reader to tackle the issue and write a companion paper.

Third, this article is an introduction to how LuaTeX sees OpenType fonts, not to the OpenType format itself. Of course, the two are closely related, and after reading this, the Microsoft documentation [5] or a general introduction like [1] will look familiar; but there are significant differences too.

Fourth, this explains how LuaTeX sees such fonts *at the present time*. That is bound to evolve, and some of what is said here will become obsolete. Nonetheless, the knowledge gained in OpenType fonts themselves will not, I hope, be wasted.

Fifth, although I hope the reader will feel comfortable with the subject after reading this paper — or, at least, the reader will feel s/he could be comfortable with the subject after reading the paper thoroughly a few times — nothing replaces experimenting with fonts directly. Fortunately, LuaTeX's fontloader is based on George Williams's FontForge, so there exists a GUI counterpart to all the Lua tables we will explore (modulo the previous point). I strongly recommend playing around in FontForge, tweaking fonts to see what changes in LuaTeX, etc.; also reading the FontForge documentation [8].

I will use a single font as illustration. It is a modified version of Philipp H. Poll's Linux Libertine (italic) [6], renamed Test Libertine. The file is available from `tug.org/TUGboat/tb33-1`, along with a `README` listing all the changes made to the original font. In the course of the article, when I write that "Libertine has such and such feature", I always mean the modified Libertine: the feature at stake may not be present in the original. None of the modifications improves Linux Libertine in any way, nor do they have much value by themselves, either typographically or technically; I've added each and every one of them with a single purpose in mind: to serve as an illustration for this paper.

2 The `define_font` callback

Loading an OpenType font requires that we intercept the user's font request and replace LuaTeX's default behavior with code of our own. As usual this is done thanks to a callback: `define_font`. It is called whenever the `\font` control sequence is used,

Author's note: I am not a member of the LuaTeX team, nor should this article be considered an official introduction to fonts in LuaTeX. However, Taco Hoekwater has answered myriad questions, both for this paper and when I was investigating fonts for myself, and many details here would have been obscure or altogether missing if not for his help. All remaining inaccuracies and outright errors are of course mine.

with three arguments: `name`, `size` and `id`. The first two arguments have direct equivalents in the syntax of `\font`:

`\font\myfont=`⟨*name*⟩ `at|scaled` ⟨*size*⟩

where ⟨*name*⟩ is anything between braces or double quotes or a string of non-blank characters. The ⟨*size*⟩ part is passed to the function registered in the callback as follows: if positive, it represents `at` ⟨*size*⟩ in scaled points, e.g. `at 10pt` becomes `655360`. If negative, it represents `scaled` ⟨*size*⟩, e.g. `scaled 500` becomes `-500`. What if no `at` or `scaled` information is given? Then ⟨*size*⟩ is set to `-1000`, as if `scaled 1000` had been specified, which is equivalent to no scaling at all, since TEX scales fonts by a factor of one-thousandth of the given value.

The `id` argument is the numerical representation of the font. Indeed, TEX internally records fonts as numbers, not names. This can be seen in LuaTEX when you query a glyph node's `font` field: it returns a number. For us, it will be useful when applying OpenType features: characters whose `font` is one we've loaded ourselves and containing special features will require our attention. But we'll see that in due time.

The function registered in `define_font` must return a table of the type which LuaTEX understands.[1] To do so, it reads the appropriate font file. I've said earlier that LuaTEX can't load OpenType fonts. That is not exactly true: it can load such a font (otherwise the remark above about its fontloader being based on FontForge wouldn't make sense), it just doesn't know what to do with it. LuaTEX reads an OpenType font file, creates a Lua table with it, and your job is to turn it into another table that the engine can use. In essence, given a table as described in section 4.4.5 of the LuaTEX reference manual [4], you have to produce a table as described in chapter 7 of the same document. Most of this paper deals with such a transformation. (If you truly have nothing else to do, you can also produce the latter table directly from the font file, not relying on LuaTEX's interpretation.)

We'll name our function the same as the callback itself: `define_font`. That means that ultimately something like the following must occur, after our function is properly defined:

```
callback.register("define_font", define_font)
```

Also, since managing fonts is a matter of (sometimes quite complex) tables, it'll help to have a function that prints the contents of a table in a readable fashion. Here it is (all Lua code in this paper is supposed to be written in a `.lua` file, not in `\directlua`, unless catcodes are properly set):

```
local rep, write = string.rep, texio.write_nl
function ExploreTable (tab, offset)
  offset = offset or ""
  for k, v in pairs(tab) do
    local newoffset = offset .. "  "
    k = offset .. k .. " = "
    if type(v) == "table" then
      write(k .. "{")
      ExploreTable(v, newoffset)
      write(newoffset .. "}")
    else
      write(k .. tostring(v))
    end
  end
end
```

This function browses entries in no particular order; however, in this paper, I will often rearrange its output (sometimes adding commas) to impose some organization. Thus the reader shouldn't worry if what s/he gets at home looks slightly different.[2]

3 From names to files

TEX traditionally loads fonts by reference to the filename, e.g.:

```
\font\tenrm=cmr10
```

loads the font contained in `cmr10.tfm`. However, XƎTEX has made popular another way of referring to fonts, namely by their internal names:

```
\font\myfont="Linux Libertine O /I: +smcp"
```

This convenient syntax has been taken over in ConTEXt, thus also in `luaotfload`. The part before the colon denotes a font proper, i.e. a font file: the file containing the italic font of the Linux Libertine O family[3] (i.e. `LinLibertine_RI.otf`, or `fxlri.otf` in TEX Live). After the colon are the tags denoting features to be applied to that font. With such a font call, our function will be executed as:

```
define_font("Linux Libertine O /I: +smcp",
            -1000, 51)
```

(provided we are defining the 51st font, which is the case if this is the first font call after loading plain TEX). Now, if we ask LuaTEX to load a font named "`Linux Libertine O /I: +smcp`", or even, since we're not so naïve, "`Linux Libertine O /I`", it will never find it. Instead, it must be given a filename and nothing else, so you have to link names to files. To do so, we have to open all the font

[1] Actually, a number can also be returned, which will be interpreted as the `id` of another, already defined font. This possibility won't interest us much, of course.

[2] Also, although the resulting table *looks* like a Lua table, it is not (and thus shouldn't be reused as such in Lua code). I leave it as an exercise to the reader to list the differences.

[3] The *O* in the family name is for OpenType.

files available to LuaTeX, as specified by the return value of `kpse.show_path("opentype fonts")`, and check their names. That takes quite a while, so it's not something we want to do on each compilation. Thus we build a database as a Lua table to match names with files, e.g. (Biolinum is the sans-serif companion to Libertine):

```
database = { ...
  ["Linux Libertine O"] = {
    Regular = "LinLibertine_R.otf",
    Italic  = "LinLibertine_RI.otf",
    ... },
  ["Linux Biolinum O"] = {
    Regular = "LinBiolinum_R.otf",
    Italic  = "LinBiolinum_RI.otf",
    ... },
  ... }
```

The first time the fontloader is used, it will create the database and write it to an external file. After that, linking names to files will be fast. Given e.g. `Linux Libertine O /I`, it can easily find that `LinLibertine_RI.otf` is required, since by convention `/I` means italics. To emphasize, this is just a *convention*, and you could decide to use `/Italic` instead, or even to allow the user to specify anything between `I` and `Italic` (then you check whether it is the prefix of a tag). The latter solution is convenient for fonts which have both Bold and Black variants (the `/B` tag is ambiguous). Also, nothing prevents the database from being written as something more readable and/or modifiable. I use a file with a very simple syntax that I can customize by hand, so that I can for instance simplify family names: "Linux Libertine O" becomes simply "Libertine", "Pro" is removed in the names of many Adobe fonts ... or I can lump unrelated files into a single family. An imaginary but familiar example would be:

```
Centaur:
  Regular = "Centaur.otf"
  Italic  = "Arrighi.otf"
```

Finally, the database can be used to retrieve TFM fonts by name too.

Fine: we know what the database should or could look like, but how are we supposed to create it? The answer has already been hinted at above: we browse all directories where OpenType fonts are supposed to be, open all files, retrieve information and store it in a table. In short, we do the following:

```
local open = fontloader.open
database = {}
local storeinfo
function explorefonts (dir)
  if not lfs.isdir(dir) then
    return
  end
  for name in lfs.dir(dir) do
    if name ~= "." and name ~= ".." then
      local file = dir .. "/" .. name
      local attr = lfs.attributes(file)
      if attr then
        if attr.mode == "file" and
         name:lower():match("%.[to]tf$") then
          storeinfo(file, name)
        elseif attr.mode == "directory" then
          explorefonts(file)
        end
      end
    end
  end
end

function storeinfo (file, name)
  local i = open(file)
  if not i then
    return
  end
  local fam, mod
  for _, lang in ipairs(i.names) do
    local n = lang.names
    fam = n.preffamilyname or n.family
    if fam then
      mod = n.prefmodifiers or n.subfamily
      database[fam] = database[fam] or {}
      database[fam][mod or "default"] = name
      break
    end
  end
end
```

Before studying what this code does, a few words on programming details. I won't be necessarily consistent with regard to `local` variables; in production code, most declarations would be local (including `define_font`, since there's no problem with registering a local function in a callback). Here, however, variables will be local if, in a given code snippet, they are used in other functions only; if they are top-level with respect to the current snippet, they will be defined globally. Hence `storeinfo` is local because we won't be using it directly, whereas `explorefonts` is global, as is the `database` table. With respect to the locality of `storeinfo`, the reader might notice that it is declared before `explorefonts` (so it is available to the latter function) but defined (without `local`) after it (so we read its definition only when we know where it is supposed to work); this is a convenient way to organize code in this situation (in other situations, e.g. with two `local` functions calling each other, it is the *only* way). Finally, both functions use:

```
if ⟨something⟩ then
  return
end
⟨code⟩
```

instead of

```
if ⟨something⟩ then
  return
else
  ⟨code⟩
end
```

Both forms achieve the same result; in general, I use the second style, because I find it clearer, but in some cases, as in here with *TUGboat*'s narrow columns, the first style saves precious space (and avoids a lonely `end` whose role might be obscure).

Back to the code itself. We can use it like this:

```
local fontpath = kpse.show_path("opentype fonts")
fontpath = fontpath:match(":")
           and fontpath:explode(":")
           or fontpath:explode(";")
for _, dir in ipairs(fontpath) do
  dir = dir:gsub("^!!", "")
  explorefonts(dir)
end
```

We ask `kpathsea` where OpenType fonts live, and it returns a string of paths separated by colons or semi-colons (the latter case on Windows); we remove exclamation marks that might prefix a path for reasons that won't concern us here, and launch our function `explorefonts`: using the `lfs` (LuaFileSystem) library, we browse the contents of a directory (making sure it *is* a directory, because `kpathsea` stores *possible* paths); for each element, if it is a file and it has the proper `otf` or `ttf` extension, we pass it to our `storeinfo`; if it is a directory, we browse it recursively. The reader may be surprised that we take `ttf` files into account; but there are two breeds of OpenType fonts: OpenType C[ompact] F[ont] F[ormat] files have the `otf` extension, whereas OpenType TrueType files have the `ttf` extension. However, the `kpse` library sees the latter as TrueType fonts (because of the extension), and when searching for a font in that format we need to specify `"truetype fonts"` as the type.

With `storeinfo` comes our real taste of OpenType fonts with LuaTEX. We use the `fontloader.open` function to import a file into a readable format (albeit not terribly readable, as we'll learn later), and if it worked (this might not be the case, e.g. if permission is denied) we retrieve the information we need. Before turning to that, though, I should mention that there exists `fontloader.info`, a function that extracts exactly the information we'll need, and which is much faster than `fontloader.open`. However, `fontloader.info` also regularly gets things wrong, not due to a defect in LuaTEX but because fonts often are badly organized. For instance, in the table returned by `fontloader.info`, the field `familyname` for Robert Slimbach's Minion Pro bold is set to `Minion Pro`, but it is `Minion Pro SmBd` for the semibold version (and the reverse holds for Adobe Caslon, Carol Twombly's adaptation of William Caslon's famous design), as if they belonged to different families. Also, the only information about the 'italic-ness' of a font is the `italicangle` field, which might have a non-zero value even if a font is not italic in the sense of being related to a roman alternative (see for instance calligraphic fonts).[4]

Anyway, we have our font loaded in the table-like (technically: `userdata`) variable `i`, which has a field `names`, an array of subtables with information about the font in different languages, as follows:[5]

```
names = {
  1 = {
    lang  = English (US)
    names = {
      family = Test Libertine
      subfamily = Italic
      fullname = Test Libertine Italic
      postscriptname = TestLibertine
      uniqueid = FontForge 2.0 : Test ...
      version = Version 5.1.1
      designer = Philipp H. Poll
      manufacturer = Philipp H. Poll
      designerurl = http://www. ...
      vendorurl = http://www.tug.org
      licenseurl = http://www.fsf.org/...
      copyright = Test Libertine by ...
      license = GPL ...  } }
  2 = { lang = German German
        names = { subfamily = Kursiv } }
  ...
  }
```

There is often only one such subtable (the first shown here), but sometimes there are several, containing some information in various languages.

The information we're looking for is `family` and `subfamily` or, if they exist, `preffamilyname` and `prefmodifiers`; they correspond to the font name and slash-prefixed tags in a `\font` call with XƎTEX syntax. Some modifiers should be filtered

[4] Actually, it should be possible to do a meaningful analysis of the table returned by `fontloader.info`, but it is just simpler to use `fontloader.open`. It is indeed much slower, but then the database isn't built on every compilation, so it's not so bad if it takes a little time.

Still, there is one case where we *need* `fontloader.info`: TrueType Collection files. This format puts several fonts in one file (with the `ttc` extension, so not taken into account here); `fontloader.info` returns an array of tables similar to the single one it returns for other files; to load a particular font in a collection with `fontloader.open`, we must pass a second argument to the function, the `fontname` found in one of the tables returned by `fontloader.info`.

[5] This shows `names` for Test Libertine, because that's the font we'll be looking at in detail, but since I've modified only one font it doesn't really belong to a family, and the original Linux Libertine is used as an example elsewhere in this discussion of the font database.

out, namely `Regular`, `Book` and others, because they denote the "default" font, i.e. roman with normal weight, and we don't want to have to specify such a font as:

```
\font\myfont="Linux Libertine O /Regular"
```

Also, a modifier isn't always as expected. For instance, William A. Dwiggins's Electra in italics is called `Cursive`; you probably want to normalize that to `Italic`, so it can be called like other italic fonts.

Now, building the database is a simple matter: it will be a table with family names as entries whose values are subtables with ⟨*modifier(s)*⟩ `=` ⟨*filename*⟩ subentries. The database is cached, and when the user specifies:

```
\font\myfont="Linux Libertine O /I : +smcp"
```

we retrieve the value of the `Italic` subentry of the "`Linux Libertine O`" entry and, lo, we are magically directed to the proper file! Admittedly, this requires a bit of string manipulation (left as an exercise to the reader), and we don't know what to do with `+smcp` yet, and in fact we don't even know what to do with the font file itself, but let's shout triumphantly anyway, it's good for morale, and we'll need some because this paper won't get any easier.

4 Basic entries in a font table

The `fontloader.open` function loads a font, but it's not usable by itself; the result should be turned into a table with `fontloader.to_table`, as follows. (The `close` operation simply discards the userdata from which the table is extracted and requires no further comment.)

```
local f = fontloader.open
        ("/your/font/dir/TestLibertine.otf")
fonttable = fontloader.to_table(f)
fontloader.close(f)
```

We shall turn this table into another, as said before. However, all those operations take time, so we'll want to perform them as seldom as possible. That is why the font table should be cached: once a font has been analyzed, relevant information is stored in a file that future compilations will retrieve instead of starting from scratch again. However, the result of some operations can't be cached: e.g. those related to dimensions (sizes of glyphs, etc.) must be performed anew each time the font is loaded, because they depend on the size at which the font is loaded; in the cache file, only "abstract" dimensions are stored, with an arbitrary unit, and they must be converted to fit the real size. Unfortunately, setting kerning pairs (i.e. the — typically negative — amount of space added between pairs of glyphs that don't look good when set next to each other) is one of those operations, and kerning pairs come by the thousands in some fonts (e.g. Minion Pro). Another operation that obviously can't be done in advance and cached is applying features.[6]

Another important remark about size. With TFM fonts, when you call e.g.:

```
\font\myfont=cmr10
```

you're requesting a font with a given size (here 10pt), because that information is part of the font file. But at what size should our Libertine font be loaded? Some fonts have a `design_size` field (expressed in tenths of a point — PostScript point, big point to TeX; we shall ignore that subtlety here); e.g. with `fonttable` above, `fonttable.design_size` returns `110`, meaning 11pt. So we could load the font at 11pt; the problem is that the design size may vary from font to font, so by default you'll be loading Libertine at 11pt and Electra at 12pt; worse still, some fonts don't have a specified design size (the field returns `0`). So it's better to have a default size at which a font will be loaded regardless of its design size; of course the problem vanishes if an `at` ⟨*size*⟩ clause is used in the font call.[7]

Let's get back to loading our font. The operations described in this paper assume we're reading from the original font file, and that nothing is cached (or at best what is cached is the table returned by `fontloader.open/to_table`, i.e. the original font file translated to Lua, so to speak). So we have our original table, which I'll call `fonttable`, and we

[6] One could cache a font at a given size, e.g. 10pt, so that at least when loaded at that size (preferably the most often used, of course), the operations on dimensions are already done. Another option is to cache fully specified fonts, i.e. with size and features applied, for a given job or set of jobs, so that all compilations but the first are faster (under the assumption that the user doesn't change fonts or font specifications on each compilation, of course). Those cache files can then be deleted once the job is done, like auxiliary files in general. Of course the features mentioned here are only those that LuaTeX can handle by itself, as will become clear later.

[7] Actually, an intelligent fontloader, unless instructed otherwise, will try to return the font that best fits the at-size; in many cases there will be one font only that matches the font call, whatever the size; in other cases, though, there will be several, and the right one should be chosen. That happens when a font is drawn at different sizes, as with many Adobe fonts and Latin Modern (by Bogusław Jackowski and Janusz M. Nowacki — and Donald E. Knuth). Of course the font database should reflect the fact that those fonts vary only with respect to size (which shouldn't be thought of as a modifier on a par with those we've been dealing with), something that the database created above didn't do. I won't pursue this matter here, except to mention that not only should `design_size` be taken into account, but also `design_range_bottom` and `design_range_top`, which specifies the (exclusive) lower and (inclusive) upper bounds of the range of sizes for which the font is optimal.

must return another, which I'll call `metrics`. A few fields can be readily set:

```
metrics = {
  name       = fonttable.fontname,
  fullname   = fonttable.fontname .. ⟨id⟩,
  psname     = fonttable.fontname,
  type       = "real",
  filename   = ⟨filename⟩,
  format     = ⟨fonttype⟩,
  embedding  = "subset",
  size       = ⟨size⟩,
  designsize = fonttable.design_size*6553.6
  }
```

First we specify some names: the `name` field is used internally by LuaTEX, e.g. in error messages; `fullname` is suffixed with ⟨*id*⟩ (the third argument to `define_font`) because, in rare cases, fonts with identical names (extracted from the same font file but with different features, e.g. with and without small caps) can cause problems: indeed, if two fonts are sufficiently similar, LuaTEX will merge them in the PDF output; adding ⟨*id*⟩ avoids the merging; as for `psname`, it is relevant to the PDF file. The `type` distinguishes `real` from `virtual` fonts as TEX has always done, but I won't address virtual fonts here.

LuaTEX uses info in a font to know what glyph to place where; but the PDF file must contain the file to render the glyphs, and that's the meaning of the following entries: the `filename` field must contain the full path to the original font file, i.e. `/your/font/dir/TestLibertine.otf` in our example, so that LuaTEX can embed it. The `format` must be one of `type1`, `type2`, `truetype` and `opentype` (the latter in our case; note that TrueType-based OpenType fonts, i.e. with the `ttf` extension, should use the `truetype` format). Finally, the `embedding` field specifies what the PDF file should contain of the original font file: if the value is `no`, the font won't be embedded at all, and the PDF viewer will try to find it on the disk[8] or, failing that, it will use a default font (but the glyphs will be placed according to what LuaTEX will have read from the original font, so the result, quite obviously, will be a mess); the value `subset` means that only those glyphs that are used in the document are described in the PDF file; finally, `full` means that the PDF document contains the entire font file. What to embed is a matter of size and license; commercial font vendors generally allow subset embedding, which is the best solution anyway, but strictly speaking that should be checked beforehand. (E.g., via the `license` and/or `licenseurl` fields in the `names` table as shown above, and more precisely, the field `fonttable.pfminfo.fstype`, corresponding to the `fsType` entry of the `OS/2` table in the original font file; that is a 16-bit number each bit of which is a boolean. Suffice it to say for us that subset embedding is perfectly ok if the number is 0 or 8.)

[8] For this, `psname` is crucial; if `psname` is missing, LuaTEX will use `fullname` which, when suffixed with ⟨*id*⟩ as is the case here, will not be a name that is findable on disk.

We now turn to the matter of size. The value of ⟨*size*⟩ depends on the font call and the contents of the at-size clause (recall our earlier discussion); if the called font contained an explicit `at` ⟨*size*⟩, then that is the value of the variable; otherwise, it contained a `scaled` ⟨*factor*⟩ clause, perhaps implicit (i.e. no clause was given, which is equivalent to `scaled 1000`); then `-`⟨*factor*⟩ was passed to `define_font`, and we have to agree on a default size to do the scaling. Suppose that default is d and we denote ⟨*factor*⟩ with f; then ⟨*size*⟩ is $-\frac{f\times d}{1000}$. (This obviously yields d if the font was called without `scaled`.) LuaTEX internally treats all dimensions in scaled points: we should never try to pass it `10pt`, and `655360` should be used instead (since there are $65,536$ scaled points per TEX point). The `tex.sp` function can be used: when passed a dimension (as a string), it returns its value in scaled points. In short, a snippet from `define_font` would be (given a default size of 10pt):

```
function define_font (name, size, id)
  ...
  if size < 0 then
    size = size * tex.sp("10pt") / -1000
  end
  ...
end
```

(The observant reader can infer from this code and the discussion above that `size`, if positive, has already been converted to scaled points when passed to the function.)

So, the `size` field is set to the result of that tediously explained computation. It is used not by LuaTEX, which relies on the glyphs' sizes themselves, but by PDF viewers, which will draw glyphs at that size. A mismatch between this value and the real size at which the font was loaded will thus result in a mismatch between the drawn glyphs and the space they occupy (letterspacing can be bluntly implemented this way). As for the `designsize` value, it is used by LuaTEX when reporting information with `\fontname`, for instance, in which case `at` ⟨*size*⟩ will be mentioned if it differs from `designsize` (if the original `fonttable.design_size` is 0, i.e. unspecified, it's better just to set `designsize` to `size`).

Now that we have learned about ⟨*size*⟩ we can turn to another important field of `metrics`: namely `parameters`, a table with seven entries. This number might remind the seasoned TeX user of something with which s/he is familiar: the `\fontdimen` primitive. Indeed, that's where those dimensions are set: `\fontdimen` *n* is the entry at index *n* in `parameters`. However, parameters 1–7 have been given friendlier names: LuaTeX will use entries with those names as keys if they exist, and fall back to the numbered entries otherwise. Here's a brief description of those parameters (math fonts have entries at index 8 and higher, but we won't investigate those here); more complete descriptions can be found in your favorite reference:

1. `slant` Slant per point, for accent positioning.
2. `space` Interword space.
3. `space_stretch` Interword stretch, i.e. the space that can be added to the interword space when a line is justified by stretching.
4. `space_shrink` Interword shrink, i.e. the space that can be subtracted from the interword space when a line is justified by shrinking.
5. `x_height` Value of the unit `ex`.
6. `quad` Value of the unit `em`.
7. `extra_space` Space added when `\spacefactor` $\geq$ 2000.

A few of these parameters can be set from information found in OpenType fonts: `x_height` can be read in `fonttable.pfminfo.os2_xheight` or derived from the height of the letter *x* in the font (a better solution), but that requires a conversion we'll turn to presently; thus the value given below for that parameter is arbitrary. The interword `space` should be the width of the space character, but again we don't know how to retrieve that yet. Finally, `slant` can be derived from `fonttable.italicangle` if the latter is given. Here I have specified interword space and associates as in Computer Modern (10pt):[9]

```
local T, R = math.tan, math.rad
metrics.parameters = {
  slant = 65536/T(R(90 + fonttable.italicangle)),
  space         = ⟨size⟩ / 3,
  space_stretch = ⟨size⟩ / 6,
  space_shrink  = ⟨size⟩ / 9,
  x_height      = 0.4 * ⟨size⟩,
  quad          = ⟨size⟩,
  extra_space   = ⟨size⟩ / 9
  }
```

Those fields could also get their values from the font call; remember that, in keeping with the XeTeX syntax, whatever comes after the colon (if any) will be interpreted as features to be applied to the font (e.g. ligatures, kerning, etc.). We could also allow additional information to be given, so the user could ask for something like this:

```
\font\myfont="Test Libertine /I: stretch=.2;..."
```

to mean that `space_stretch` should be set to a fifth of the loading size (among other features); of course, such a parameter could also be set by hand with `\fontdimen`, but this is a nicer interface.

Before extracting the marrow from OpenType fonts, I shall say that there are many other fields in `fonttable` that we won't explore here because they're not crucial, even though we could make use of them. Interesting information on the font can for instance be found in the `fonttable.pfminfo` table, which lumps together fields from (mostly) the `hhea` and `OS/2` table of the original font file. Similarly, some other fields in `metrics` could be set that we won't consider here.

5 Glyphs, at last!

Now that we've gone through all the preliminary steps we can turn to the crux of the matter: glyphs. In `fonttable`, there is a `glyphs` subtable which contains them all, and we shall use them to populate the `characters` entry in `metrics`.[10] However, entries in the `fonttable.glyphs` table are arbitrary, whereas `metrics.characters` indices are Unicode codepoints; for instance, *a* in Libertine (italic or not) is at index 66 in `fonttable.glyphs`, even though its codepoint is 97, which is also the index where it must appear in `metrics.characters` (unless we're implementing substitutions, but we won't be doing that now).

So, we need to know: a) the characters the font contains and b) the glyph number of each character.

[9] The computation for `slant` is not complicated, but it might not be very readable as expressed here. So here are the steps:

1. `slant` is the horizontal displacement for one point of vertical displacement. Hence it can be expressed as $1 / \tan \alpha$, where α is the angle between the *x*-axis and the font's axis.
2. `fonttable.italicangle` measures the angle between the *y*-axis and the font's axis (which is why it is negative for fonts leaning on the right, as most slanted fonts do); α is thus: $-(90-$`fonttable.italicangle`$)$, i.e. $90+$`fonttable.italicangle`.
3. α is expressed here in degrees, but Lua's `math.tan` function expects radians, hence the use of `math.rad` to do the conversion.
4. The result is expressed in points; we multiply it by 65,536 to convert it to scaled points.

For fonts with proper information for diacritic positioning, `slant` is useless; we'll use OpenType features instead. But setting it correctly does no harm.

[10] My use of the words *character* and *glyph* doesn't reflect the common distinction between an element of a writing system and its representation (so that the character *a* can be represented by various glyphs, e.g. from different fonts); instead I will use *glyph* for an element of `fonttable.glyphs` and *character* for an element of `metrics.characters`.

Fortunately, we have `fonttable.map.map`: it is an array with Unicode codepoints as indices and glyph numbers as values, e.g. `97 = 66`, meaning that the character with Unicode codepoint 97 has its glyph in `fonttable.glyphs[66]`. Let's give a shorter name to this table and use it to look at the glyph *f*:

```
map = fonttable.map.map
ExploreTable(fonttable.glyphs[map[102]])
```

And the result is (somewhat shortened):

```
name    = f
unicode = 102
class   = base
width   = 314
boundingbox = { 1 = -78
                2 = -238
                3 = 523
                4 = 698 }
kerns   = { ... }
lookups = { ... }
anchors = { ... }
```

I have elided the `kerns`, `lookups` and `anchors` table since we aren't able to do much with them for the time being. The first two fields are quite obvious, I suppose; `name` will be of use later, exactly when we'll examine the contents of `kerns` and `lookups`. We'll ignore the `class` field until we start discussing lookups. The `width` field is, unsurprisingly, the width of the glyph. The values in the `boundingbox` table are the position of the glyph's extrema: `1` and `3` are the minimum and maximum x-values respectively, while `2` and `4` are the minimum and maximum y-values; the former are expressed relative to the y-axis (i.e. the left side of the glyph's bounding box proper, where $x = 0$), and the latter relative to the x-axis (the glyph's baseline). Thus `boundingbox[4]` is its height, `-boundingbox[2]`, provided `boundingbox[2]` is negative, i.e. the lowest point is below the baseline, is the glyph's depth, but the glyph's width is `width`, and nothing else! Indeed, a glyph as drawn may be larger than its declared width, and it may extend outside its bounding box, and that's perfectly normal. Figure 1 illustrates that with the glyph we're investigating (the image is from FontForge): the glyph's width is the area between the two vertical lines; the extenders aren't contained between those lines, which means that, for instance, an *i* before the *f* will stand above the *f*'s tail, while an *o* after will stand below its arm: *ifo*. That's welcome behavior, otherwise spurious gaps would occur between letters.

Now, the only fields LuaTEX *requires* for a character are `width`, `height`, `depth` and `index`, the latter being the glyph's index in `fonttable.glyphs`. In fact, LuaTEX is directly interested in the first three fields only: they are the basic data it requires

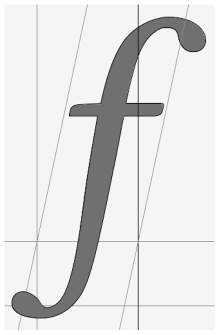


Fig. 1: *f* in Libertine Italic

to do its job properly. On the other hand, `index` is used to denote the glyph in the PDF file.[11] So, that's it, we can transfer glyphs in `fonttable` to characters in `metrics`! Oh, no, we can't: we know *f*'s width is 271, but 271 *what*? Scaled points? No, that would be too easy.[12] The unit in which dimensions are expressed in font files is a relative unit, which makes sense since the font may be loaded at whatever size. The value of that unit is recorded in `fonttable.units_per_em`, generally $1,000$ or $2,048$, but the real value is of little importance: what counts is that, given ⟨*size*⟩ as computed above, we are able to derive a (this time absolute) unit for interpreting glyph dimensions. Since by definition ⟨*size*⟩ is one em, then the value of the unit is obviously $\frac{s}{u}$ with $s = \langle \mathit{size} \rangle$ and $u =$ `fonttable.units_per_em`. Let's record it in a variable:

```
unit = ⟨size⟩ / fonttable.units_per_em
```

and here we go, let's translate `fonttable`'s glyphs into `metrics`'s characters (recall how `map` was defined above):

```
for ch, idx in pairs(map) do
  local glyph = fonttable.glyphs[idx]
  metrics.characters[ch] = {
    index  = idx,
    width  = glyph.width * unit,
    height = glyph.boundingbox[4] * unit,
    depth  = glyph.boundingbox[2] < 0 and
             (-glyph.boundingbox[2] * unit) or 0 }
end
```

This time, that's it, we're done! The `define_font` function can return `metrics` and LuaTEX will be able to use it. No kidding: extracting the glyphs is child's play by comparison.

[11] Indeed, *TeX* will be written ⟨*003500460039*⟩ in the PDF file, instructing the viewer to fetch glyphs at position `0x35`, `0x46` and `0x39` in the current font (assuming the current font is Libertine). Actually, things are a bit more complicated than that, but we definitely don't want to dwell on PDF.

[12] Not to mention that a glyph with a width of 271 scaled points would be a billboard for atoms but not exactly for us.

6 Some easy-to-implement niceties

Okay, well, we're done, but let's face it: our font isn't that exciting. If this is all we can do with OpenType, that's rather disappointing. The font isn't even being respected, since kerning information has been ignored.

Of course, we will be doing much, much more. But we'll postpone that as long as there are simpler areas to investigate. For instance, the letter *f* wasn't randomly selected to illustrate the previous section: I chose it because it is the letter that requires italic correction *par excellence*. To wit: "*arf*" must be uttered by an uneducated dog, whereas "*arf*" is from a dog with manners. As the reader certainly knows, it's the difference between

```
``{\it arf}''  and  ``{\it arf\/}''
```

where `\/` denotes italic correction, a small amount of space to be added after the letter. The problem is that italic correction was born with TFM format and didn't prosper. In other words, there is no such thing in OpenType fonts. However, we can mimic it: we'll define italic correction as the difference between a glyph's rightmost point and its width. Given that LuaTEX stores the italic correction in a character table's `italic` field, we can thus enhance the `char` table defined above as follows (where `glyph` is as before):

```
char.italic = (glyph.boundingbox[3] - glyph.width)
              * unit
```

The results might be more or less felicitous, since italic correction was meant to be specified for each glyph by the designer, not automatically computed, but I find this much better than no italic correction at all.[13]

Other things can be easily implemented, this time properly, because they're just the Lua version of existing operations. For instance, the `extend` field in `metrics` corresponds to the `ExtendFont` keyword in a PDFTEX map file: it lets you stretch or shrink the glyphs. The value ranges between $-5,000$ and $5,000$; glyphs are then horizontally distorted by a thousandth of the given value (so that with $1,000$ the font is untouched); a negative value reverses the glyphs. The glyphs' widths are not actually modified; extension takes place in the PDF. In other words, LuaTEX sees and positions them with their original size, thus proper extension should also modify the glyphs' horizontal dimensions (something not possible in PDFTEX).

The `slant` entry in `metrics`[14] corresponds to the `SlantFont` keyword in a map file and allows creation of artificially obliqued versions of a font (or artificially upright versions of a slanted font). It ranges between $-2,000$ and $2,000$ with 0 meaning no slant (other than the font's native slant, of course). The calculation is as follows: given a value of s for `slant`, the resulting angle between the y-axis and the (modified) font's axis is $-\arctan\frac{s}{1000}$ (under the assumption than the font is originally unslanted; otherwise, add that to the font's angle). For instance, if `slant` is $1,000$, then the font will make an angle of $-45°$ (or $45°$ clockwise) with the y-axis. Again, LuaTEX doesn't take that value into account when positioning glyphs, and artificial slant should be paired with increased italic correction.

Let's stop disfiguring fonts with electronic surgery and turn instead to something definitely useful: character protrusion and font expansion (a.k.a. HZ for Hermann Zapf). Both have TEX interfaces, but we can define them here at once when loading the font: instead of using `\lpcode` and `\rpcode` for protrusion and `\pdffontexpand` and `\efcode` for expansion, we can specify those values in Lua. For each character, we can set the `left_protruding` and `right_protruding` fields, which correspond to `\lpcode` and `\rpcode` respectively (the values are thousandths of ems).[15] As for expansion, the `\pdffontexpand` primitive is reflected as follows in Lua. To the statement

```
\pdffontexpand\myfont
  ⟨stretch⟩ ⟨shrink⟩ ⟨step⟩ [autoexpand]
```

corresponds to the following settings in `metrics`:

```
metrics.stretch = ⟨stretch⟩
metrics.shrink  = ⟨shrink⟩
metrics.step    = ⟨step⟩
metrics.auto_expand = ⟨boolean⟩
```

where specifying `true` for ⟨*boolean*⟩ is equivalent to using the `autoexpand` keyword (which is highly recommended, otherwise you need several versions of the same font). Finally, each character can have a field `expansion_factor` corresponding to `\efcode`.

As noted above, both protrusion and expansion can be set in TEX;[16] however, the Lua way can be

[13] One could also use an external file to store italic corrections for a given set of glyphs from a given font, and retrieve the values and apply them when the font is loaded. This might seem like overkill, but it's no more tedious than checking kerning pairs. Also, LuaTEX is of little use if not for such subtleties.

[14] Not to be confused with the previously-discussed entry of the same name in `metrics.parameters`.

[15] Character protrusion in LuaTEX is still buggy as I write this (with v0.71); negative protrusion isn't properly obeyed.

[16] Also, one must set the parameters `\pdfprotrudechars` and `\pdfadjustspacing` to 1 or 2 if protrusion and expansion are to be performed, or use the Lua formulations `tex.pdfadjustspacing` and `tex.pdfprotrudechars`. Perhaps even set

the basis for a nice interface in the XƎTEX-like syntax of the font call, e.g.:

```
\font\myfont="Test Libertine /I:
  expansion = 30 20 5;
  expansion_factor = ⟨whatever⟩"
```

where ⟨*whatever*⟩, as the name indicates, could point to an external file or previously-defined Lua table or, well, whatever contains the individual glyph expansion factors.

Before turning to the implementation of advanced typographic features,[17] we can have an easy first taste of such operations. The studious reader who has followed this paper with a computer next to him/her and experimented with each step will have seen some strange behavior: the line of code

```
``I'm going to be interrupted---''
```

comes out as

``*I'm going to be interrupted---*''

instead of

"I'm going to be interrupted—"

The reason for this apparent misbehavior is that OpenType fonts don't follow a well-known convention in TEX: namely, that TFM fonts have (single) quotation marks in lieu of the "grave accent" and "apostrophe" characters (hence ' instead of `); that two quotation marks in a row are replaced by double quotation marks (hence " instead of '', but the latter can't be seen here because of the previous point); that two hyphens in a row form an en-dash (-- becomes –); and that an en dash followed by a hyphen turns into an em dash (–- becomes —).[18]

The quotation marks could be easily substituted for the grave accent and apostrophe:

```
metrics.characters[96] = metrics.characters[8216]
metrics.characters[39] = metrics.characters[8217]
```

This is not a good idea, however! It upsets the character/glyph correspondence, and that should never be done lightly (i.e. without keeping track of whatever substitution has been performed).

On the other hand, the "TEX ligatures" can instead be implemented as follows. Characters in `metrics` may have a `ligatures` table which is organized like this:

```
ligatures = {
  [⟨nextidx1⟩] = { char=⟨ligidx1⟩, type=⟨type⟩ },
  [⟨nextidx2⟩] = { char=⟨ligidx2⟩, type=⟨type⟩ },
  ... }
```

where each ⟨*nextidx n*⟩ is the index of a character in `metrics` and ⟨*ligidx n*⟩ is the character being substituted if the current character and ⟨*nextidx n*⟩ are found next to the other. As for ⟨*type*⟩, it is either a number or a string denoting the result of the ligature: either the new character or the new character with one or the other or both of the old characters; ⟨*type*⟩ also specifies where TEX should resume scanning. We will content ourselves with type 0.

So, given that the codepoint of hyphen is 45, en-dash is 8,211 and em-dash is 8,212, we can create our convenient ligatures like this:

```
metrics.characters[45].ligatures = {
  [45] = { char = 8211, type = 0 } }
metrics.characters[8211].ligatures = {
  [45] = { char = 8212, type = 0 } }
```

Although we could start implementing f-ligatures and the like this way also, we'd rather do things properly. So let's turn to the reason why OpenType fonts exist in the first place.

* * *

As we now dive into the core of OpenType fonts, the reader should recall the warning issued in the introduction to this paper: LuaTEX's fontloader will evolve, the organization of `fonttable` will change; thus we should observe one field in particular, not mentioned before: `table_version`. Here I describe a `fonttable` whose version is 0.3.

7 A first look at features

It's simpler to approach features with a real example than with an abstract definition. If we explore again the table for *f* (102) in Libertine,[19] but this time retain the `lookups`, `kerns` and `anchors` tables only, and in each only the first entry, we see:

```
lookups = {
  ss_l_0_s = {
    1 = { type = substitution
          specification = { variant = f.short }
        } }
  ...  }
```

them automatically when loading a font with protrusion and/or expansion enabled.

[17] I mean *advanced* from a software point of view; substitutions and positioning are of course as old as writing.

[18] There are a few other substitutions/ligatures, e.g. `!`` gives ¡. Those substitutions/ligatures aren't exactly that, if we define "substitution" as the replacement of a glyph with another denoting the same character (here a character is replaced with another character: the grave accent and the left quote aren't the same thing) and "ligature" as the merging of two or more glyphs into a new one for æsthetic reasons (hyphens aren't supposed to happen in a row, and even if they would, there's no typographic convention to the effect that they should be merged into an en-dash). Rather, these are convenient abbreviations.

[19] From now on when exploring a particular character, I'll mention the character itself or its name followed by the Unicode codepoint between parentheses; the resulting table is thus produced with
`ExploreTable(fonttable.glyphs[map[⟨codepoint⟩]])`.

```
kerns = {
  1 = { char   = o
        off    = -11
        lookup = { 1 = pp_l_2_g_0,
                   2 = pp_l_2_k_1 } }
  ...  }
anchors = {
  basechar = {
    Anchor-3 = { x = 136, y = 215,
                 lig_index = 0 }
    ... } }
```

This tells us a few things: first, *f* should be replaced by `f.short` (*f* becomes *f*) if lookup `ss_l_0_s` is active. Also, the distance between *f* and *o* should be decreased by 11 units (with units defined as above) if `pp_l_2_g_0` is active.[20] Finally, anchors are specified in case a mark is placed relative to the glyph.

One absolutely crucial point: the `f.short` and `o` glyphs aren't denoted by their Unicode codepoints, nor by their positions in `fonttable.glyphs`, but by their names. When we first looked at *f* we could see that it had a `name` field. This is the case for all glyphs (even if sometimes the name is as uninformative as `uni0358`), and all typographic operations denote glyphs in that way. On the other hand, in `metrics` names are useless and only codepoints matter. That means that we badly need a table linking names to codepoints; here it is:

```
name_to_unicode = {}
for ch, gl in pairs(map) do
  name_to_unicode[fonttable.glyph[gl].name] = ch
end
```

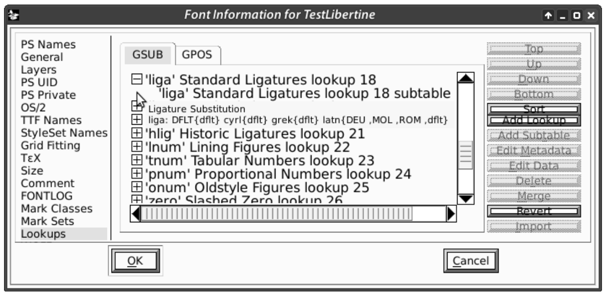
Now, `name_to_unicode["f.short"]` nicely returns 57, 568,[21] and the reader who has followed the previous instructions to load Libertine can check that `\char57568` indeed yields *f*. And if we were to rush into the kerning of *f*, we could do:

```
metrics.characters.f.kerns = {
  [name_to_unicode.o] = -11 * units }
```

But let's not do that. Instead, let's conscientiously study how features and lookups work, and first of all, how to know whether a lookup is active or not.

8 Lookups, tags, scripts and languages

A lookup is activated if the right combination of script, language and tag obtains; again, a concrete example will be easier to understand. Substitution lookups are stored in `fonttable.gsub`, which is an array where each entry is a table describing a feature.[22] If we explore entry 19 in Libertine's `gsub`, we get the following:[23]



```
⊟'liga' Standard Ligatures lookup 18
    'liga' Standard Ligatures lookup 18 subtable
  Ligature Substitution
⊞ liga: DFLT{dflt} cyrl{dflt} grek{dflt} latn{DEU ,MOL ,ROM ,dflt}
⊞'hlig' Historic Ligatures lookup 21
⊞'lnum' Lining Figures lookup 22
⊞'tnum' Tabular Numbers lookup 23
⊞'pnum' Proportional Numbers lookup 24
⊞'onum' Oldstyle Figures lookup 25
⊞'zero' Slashed Zero lookup 26
```

Fig. 2: `ls_l_18` as seen by FontForge.

```
name      = ls_l_18
type      = gsub_ligature
flags     = { }
subtables = { 1 = { name = ls_l_18_s } }
features  = {
 1 = {
  tag = liga
  scripts = {
   1 = { script = DFLT, langs = { 1 = dflt } }
   2 = { script = cyrl, langs = { 1 = dflt } }
   3 = { script = grek, langs = { 1 = dflt } }
   4 = { script = latn,
         langs = { 1 = "DEU ", 2 = "MOL ",
                   3 = "ROM ", 4 = dflt } } } } }
```

(Compare this to Figure 2, which shows the same lookup as seen by FontForge.) Let's focus on the `features` field, which is what we are interested in. It is an array of tables (here there is only one), each containing a possible combination of tag, scripts and languages activating the lookup. We can see that some ligatures (since we're looking at such a feature, as the `type` indicates) will be activated if a) `liga` is on and b) no script and language is specified (i.e. both are defaults) or the script is Cyrillic and the language is unspecified, and so on and so forth. (In the rest of this paper, I will often write that "this tag activates that lookup"; the reader should mentally add: "if script and language do not say otherwise".) There might be several such subtables in `features`, one per tag, since a feature might be implemented

[20] The second entry `pp_l_2_k_1` must be ignored; in fact, it shouldn't be there at all and the `lookup` field should be a string, not a table.

[21] What, the reader might exclaim, a glyph variant has a place in Unicode? Not really, no: 57, 568, i.e. 0xE0E0, is in the "private use area", where font designers are free to put whatever they wish, and which generally contains alternate forms. Nevertheless, Unicode does contain some variants, such as the f-ligatures.

[22] The reader may have noticed that I use the word *feature* somewhat freely, indeed eschewing any technical definition, denoting vaguely but conveniently a typographic manipulation on the font.

[23] The table entry is at index 19 even though the lookup has 18 in its name (and lookups are indeed numbered consecutively), because lookup numbering starts at 0, whereas Lua tables prefer 1 as the first index (table length and the `ipairs` function are sensitive to that). So, whenever I mention lookup `ab_c_`x, the reader should look at index $x + 1$ in the `gsub` or `gpos` tables.

by several tags: for instance, in Minion, "oldstyle" numbers (e.g. 123 instead of 123) are activated by the `onum` tag, but also by `smcp` (small caps) and `c2sc` (capitals to small caps, i.e. turning *uppercase* letters to small caps, whereas `smcp` affects lowercase letters), because those kinds of numbers are deemed mandatory with small caps by the designer. In such a case of multiple subtables, only one tag/script/language combination suffices for the lookup to be activated. It might also happen that no `features` table is present (for instance, the first two lookups in Libertine); that is the case for lookups that aren't tied to a tag but instead are called by other (contextual) lookups. We shall come back to that later.

One important yet easily missed property of language tags is that they are four-character strings (which is why I added unusual quotes around a few strings in the table), the last character being a space (except for `dflt`). On the other hand, you're not going to demand that users specify the language they want to use with a space at the end, so you have to add it yourself.

That tags interact with scripts and languages to activate (or not) lookups is an important feature in OpenType, allowing for a flexible interface. For instance, the `locl` tag can be activated in all circumstances, to activate variants only for the specified scripts and languages: given n sets of localized forms, there is thus no need for n tags. Another well-known example is the *fi* ligature, which shouldn't be used in Turkish, since this language distinguishes between i and $ı$, and the i's dot is generally gobbled in the ligature.[24,25] Consequently, `liga` often points to two lookups: one with the *(f)fi* ligature(s) only, in which Turkish is excluded, and another implementing all other standard ligatures. That is exactly how the lookup shown here works: the reader may have noticed that for the Latin script, `TRK` is missing; and indeed that lookup controls the *(f)fi* ligature(s).[26]

One might think that tags are only pointers to lookups, which implement a feature, so that tags can be somewhat overlooked as simply being a user-friendly way to activate lookups, furthered by the fact that tags can be freely made up by font designers whereas there is only a finite number of lookup types. Although that is true in general, in some cases tags have some additional content. For one thing, some tags should be active by default (e.g. `liga`, standard ligatures) whereas others are optional (e.g. `hlig`, historical ligatures); this means that the former should be applied unless the user specifies otherwise.[27] Another, more important way in which tags have semantics is when the lookup they point to doesn't suffice to implement a feature properly. For instance, the `init`, `medi` and `fina` tags denote the initial, medial and final letterforms in (mostly) Arabic scripts; the feature is implemented by a simple substitution, not a contextual substitution (on which, see below). This means that if the user selects `init`, a given variant should be used for letters at the beginning of a word only, even though the lookup implementing it doesn't mention contexts; in other words, manipulations shouldn't rely on the lookup type only and should keep in mind the tag that activated it. A list of registered tags, along with their meaning, is available at [5].

Finally, we must note that the user's choice of script and/or language should be obeyed if and only if the font knows about them. Indeed, fonts do not specify all possible scripts and languages, but only those which make a difference; for instance, French isn't mentioned anywhere in Libertine, but this doesn't mean that no features should be applied if the font is loaded with `FRA` as the value of `lang` (and, of course, `latn` as the value of `script`). Instead, it means that the font knows nothing of that language and will treat it like the `dflt` language in the `latn` script. How do we know which scripts and languages a font is designed to handle? It is given nowhere in `fonttable`; instead, we must scan all lookups in `gsub` and `gpos` and collect all scripts and languages (actually, `gsub` and `gpos` should be treated independently of each other, but that often makes little difference). Only then can the user's request be interpreted correctly.

[24] Dotless $ı$ notes the close back unrounded vowel, which means it's at the back of the throat (like the Castle of Aaaaarrgh), the mouth is only slightly open (unlike said Castle), and the lips are at rest; you can pronounce it more or less as follows: hold "oooooo" as in *boot* and stop pouting.

[25] The *fi* ligature does not always gobble the i's dot; for instance it doesn't in Kris Holmes and Charles Bigelow's Lucida; accordingly, the ligature is applied even in Turkish.

[26] Another way to avoid the ligature in Turkish is illustrated in Minion: it has only one set of ligatures (containing *fi*) defined for all languages; however, it also has a special substitution activated by `locl` only if the language is Turkish (and similar), which replaces i with an identical glyph whose only difference is precisely not forming *fi* when preceded by *f*.

[27] What counts as a standard ligature (or any other feature) is up to the font designer; for instance, some fonts do not have the *fi* or *ff* ligatures (for the latter, see e.g. Adrian Frutiger's Frutiger), because they don't need it, whereas some less common ligatures are sometimes required: Libertine, for instance, has *Th*, *ch* and others.

9 Applying lookups

When several lookups are active, the order in which they are applied is crucial; for instance, if both `smcp` (small caps) and `liga` (ligatures) are active, the former should generally be executed before the latter, otherwise the sequence $f + i$ will be replaced with the ligature *fi*, which has no counterpart in small caps: and that's DEfiNITELY ugly.[28]

Fortunately, the order in which lookups should be applied is easily read from the font file and doesn't depend on any additional knowledge, as the previous example might imply. It works as follows (here "lookup" means "active lookup"): substitution lookups, i.e. those in the `gsub` table, are applied before the positioning lookups in `gpos`; in each category, lookups are applied in the order in which they are declared, i.e. according to their indices in `fonttable.gsub/gpos`. A lookup is applied to all glyphs in a node list before moving on to the next lookup (although "all glyphs" will be explicated below). Applying a lookup to one or more glyph(s) means that each subtable in the lookup is tried until one matches or no subtable remains (in which case the lookup simply does not apply).

These subtables are found (not so surprisingly) in the `subtables` entry. They generally contain only a single `name`, rather than what the subtable actually does: the latter is to be found in the glyph tables, where that information is associated with `name` (recall what was said above about the substitution of `f.short` for *f* and the kerning with *o*; we said that the substitution should occur if `ss_l_0_s` is active: `ss_l_0_s` is the `name` of a subtable in the `ss_l_0` lookup).

To sum up: the user specifies a script (or we use `DFLT`), a language (or `dflt`), and some tags (to which we add those activated by default). This combination activates lookups in `gsub` and `gpos`; with the names of those lookups' subtables, we retrieve what should be done by browsing the rest of the font (mostly, the glyphs); then, well, we do it. What should be done and how to do it is explained in the two sections below about lookup types.

For instance, let's suppose the user has selected `liga`, `kern` (kerning) and `mark` (mark positioning) in Libertine, and no script or language is specified. Further suppose that the input node list is:

fie̒re che̒vre

a French phrase (*fière chèvre*) meaning "proud goat" and interesting only to the extent that it lets us test our lookup-application skills. Here, ̀ is the character whose codepoint is 768, i.e. the "combining grave accent", not the "grave accent" at position 96. So, we begin with the `gsub` lookups, and `liga` activates two of them: `ls_l_18` (the one shown above) and `ls_l_19`; `ls_l_18` is applied to the beginning of the string, i.e. we test whether its only subtable matches, and since that subtable is supposed to turn $f + i$ into *fi*, indeed it does match: the first two glyphs are replaced by a single one representing the ligature; `ls_l_18` then moves to the third glyph,[29] obviously it doesn't match, so it moves to the fourth, which doesn't match either, and so on until the list has been scanned to the end. Enter `ls_l_19`: it is in charge of all remaining ligatures, and it matches on `ch` (see footnote 27) and only there.

Next come the `gpos` features; `kern` activates lookup 2, `pp_l_2`, and `mark` lookup 3, `mb_l_3`. The former is applied first, and has two subtables: thus, on each glyph, if the first matches, the second isn't applied, even though it might well have matched too. The first subtable deals with individual glyph pairs; the second with more general but less precise kerning tables (more details on this below). One could say that the second subtable defines default kerning, while the first deals with special cases. The order of their application thus makes sense.

So, for each glyph, `pp_l_2` will add some kerning if necessary. An interesting situation arises with our node list: there should be some kerning between *e* and *v*; however, the intervening accent, which the next lookup will properly place on the preceding letter, hides the fact that the two glyphs are next to each other. Or does it? No, it doesn't, for the precise reason that `pp_l_2` was instructed to ignore marks, and as we'll see later this accent belongs to that category; thus, when looking for an $e + v$ sequence, the lookup will operate as if the mark were not there. Finally, `mb_l_3` is applied, which has no less than seven subtables, one for each of the possible places where diacritics might be positioned relative to a base mark. In our case the fourth subtable will do the job, since it defines an anchor common to both *e* and the accent.

Applying lookups doesn't seem too complicated,

[28] Hermann Zapf's Palatino does have an *fi* (and others) to FI lookup. But since `smcp` is applied before `liga` anyway, it is useful only when the ligature has been entered by hand, e.g. as `\char"FB01`. The substitution is also in Test Libertine.

[29] Since *fi* has been processed, it can't be used as input any more for the current lookup. If the font defined a ligature between *fi* and *e*, then it should be implemented in a subsequent lookup, but the font is more likely to define a three-component ligature ($f + i + e$), which `ls_l_18` would have tried before *fi* because it is longer. More details in the subsection on ligatures below.

and indeed in many cases it isn't. Sometimes things are more difficult, though. For instance, given the node list "c¸h", we want it to form *çh*, i.e. we want the ligature with the cedilla on the first letter.[30] The ligaturing lookup is clever and ignores marks, so the ligature is properly formed; the mark positioning lookup is clever too and knows how to set a mark relative to one component of a ligature (e.g. a mark on *c* won't be placed like a mark on *h*). But the problem is that once the ligature is formed, and the mark has been put after it so that the positioning lookup can find it, how do we know where in the ligature the cedilla is to be positioned? Before ligaturing, it was clear: the cedilla was to be positioned on the first glyph; afterwards, that information was lost. Thus, you are in charge of passing this information from the ligaturing lookup to the positioning lookup; attributes can be used for this.

I've mentioned lookups ignoring marks. Indeed, glyphs in an OpenType font belong to one of four classes: base, mark, ligature, and component (of a ligature); glyphs also may have no class. Lookups can be instructed to ignore some of these classes (except the "component" class), in which case, when looking for possible input, they act as if glyphs of those classes were not present, so that *ABC* is seen as *AC* if *B* belongs to an ignored class. The class of a glyph is stored in the `class` entry of the glyph's table in `fonttable`, as we saw when inspecting *f* above.

A lookup's ignoring of some glyph classes is indicated in its `flags` subtable; for instance, `ls_l_19` in `gsub` has:

```
flags = { ignorecombiningmarks = true }
```

Thus that lookup, responsible for forming some ligatures, is blind to mark glyphs. Indeed, as we've seen, in "c¸h" the ligature will be created as expected. Besides `ignorecombiningmarks`, there are `ignorebaseglyphs` and `ignoreligatures` to ignore other classes. A lookup may also have another entry, `mark_class`, a string holding a mark class's name; in that case, the lookup ignores all marks except those in that class. To know which mark belongs to which class, we can inspect `fonttable.mark_classes`:

```
MarkClass-1 = gravecomb acutecomb uni0302
MarkClass-2 = cedilla dotbelowcomb uni0325
```

Thus, if `flags` above was:

```
flags = { mark_class = MarkClass-1 }
```

then the lookup would ignore all marks except the grave, acute and circumflex accents.

A lookup can have one last flag, `r2l` ("right to left"), which we will discuss briefly along with cursive positioning, since it is significant only with that lookup type.

10 Implementing lookups

There are several kinds of lookups, each specifying a particular implementation (modulo the remark above about tags that must also be taken into account). Most of them cannot be handled by LuaTeX by itself, meaning that we can't simply return a `metrics` table containing the information for performing the necessary operations. What LuaTeX implements natively is: non-contextual kerning, non-contextual two-glyph ligatures (although we'll later see a trick to create ligatures involving more than two glyphs), and non-contextual substitution (e.g. *f* in small caps; actually LuaTeX doesn't substitute anything, we simply return `metrics` with small-cap *f* at index 102). Other features must be done by hand in one of the pre-paragraph-building callbacks, i.e. `hyphenate`, `ligaturing`, `kerning` or `pre_linebreak_filter`[31] (see [3] for an introduction to what those callbacks do by default).

But even what LuaTeX can do is better done by hand; suppose that you have a contextual substitution (which LuaTeX can't handle) and a non-contextual substitution (which you can ask LuaTeX to perform by itself simply by returning `metrics` with the replacement glyphs at the position of the glyphs they replace); and further suppose that the former should happen before the latter (because it has a lower index in `fonttable.gsub`). You can never get that right, because LuaTeX will map characters to glyph nodes well before any of the above-mentioned callbacks are executed; in effect, the second substitution will have already occurred when the first is performed, thus reversing the desired order. Nonetheless, with e.g. the Latin script, non-contextual kerning and ligatures, and simple, across-the-board substitutions (like small caps), LuaTeX can be trusted, and you don't have to do anything by hand — in particular, you don't have to worry about hyphenation, a thorny issue as we'll see below; that's why I'll describe how some lookups can

[30] A *ç* can occur before an *h* in Manx, spoken (infrequently nowadays) on the Isle of Man. Nicolas Beauzée, a French grammarian of the XVIIIth century (often with modern insights) and contributor to Diderot and d'Alembert's *Encyclopédie*, suggested *çh* also be used in French orthography.

[31] The `linebreak_filter` (the paragraph builder proper) and `post_linebreak_filter` callbacks can be considered too for features related to justification, but I will not pursue that complex subject here. In addition, if anything is done in `pre_linebreak_filter`, it should be done in `hpack_filter` too, since the latter, but not the former, is called when material is processed in a horizontal box.

be implemented without extra work, even though I'll also give the bigger, callback-based picture.

In what follows, when explaining implementations with those callbacks, I will rely on the following assumption: that it is possible, given a font `id` (remember the third argument to `define_font`), to determine what should be done for each character in that font. It means that when a font is loaded, features to be implemented via callbacks should be stored somewhere. Also, applying lookups makes sense only for a sequence of glyphs sharing the same font; thus a paragraph made of

```
\myfont This is {\myotherfont very} special
```

would be processed in three independent steps, as if there were three paragraphs. A simple solution is to break the node list into sublists each with glyphs from the same font.

OpenType in general assumes that a text-processing application takes an input string, maps the characters to glyph positions in the font, and then manipulates those glyphs according to active lookups. the first part (characters to glyphs) is done automatically by LuaTEX, provided that at index m in `metrics.characters` there is the glyph at position n in `fonttable.glyphs`, with n being the value of entry m in `fonttable.map.map`. We've already done that, but there is an exception: TEX doesn't map a space character to a glyph, but creates a glue node instead.[32] Also, the list we'll be working on is interspersed with nodes which have absolutely no relation to characters, like penalties, whatsits, etc., so that we don't deal with a simple string of characters as assumed by the OpenType model. Finally, some of those non-glyph nodes are discretionaries, which require special treatment.

We solve the first problem by considering a glue node to be a character whose glyph is `fonttable.map.map[32]` (since 32 is the space's codepoint); this might seem like overkill, but it is not: lookups do take spaces into account, and Libertine has kerning pairs with space as either the first or the second member. Compare:

How Terrible!
How Terrible!

In the first line the space has its default width, whereas in the second line (negative) kern is added after w and before T (while that kerning is not applied between other letters). In Minion, space is subject in some circumstances to substitution (being replaced with a narrower space glyph).[33] However, some lookups (or rather, tags, such as `init` above) require that a "word" be a well-defined entity, because position in the word is important; in that case, spaces should be considered boundaries.

The second problem, non-glyph nodes, is solved just by ignoring those nodes. Thus, in this node list:
⟨*glyph node 1*⟩ ⟨*non-glyph node*⟩ ⟨*glyph node 2*⟩
a function asked to retrieve the glyph following the first one should return the third node. Things might get a little bit more complicated, but one shouldn't worry too much; special circumstances may simply demand special solutions.

The third problem is harder, because we definitely can't ignore discretionary nodes. Suppose you have the string `X{a}{b}{c}Y`, where `X` and `Y` can be anything, `a` is the `pre` field of a discretionary (i.e. the part typeset before the break if hyphenation occurs at this point; typically, a hyphen), `b` is the `post` field (the part typeset at the beginning of the following line if hyphenation occurs; typically empty) and `c` the `replace` field (what is typeset if the hyphenation point is not chosen; also typically empty). (See [3] for a slightly more detailed description of discretionary nodes.) You may have lookups acting on the strings `Xa`, `bY` and `XcY`, so that each lookup should be applied three times, and the resulting material should be `{a'}{b'}{c'}`, where `a'`, `b'` and `c'` are respectively `Xa`, `bY` and `XcY` after lookups have been applied. Then any common part at the beginning of `a'` and `c'` can be moved out of and before the discretionary; likewise, common material at the end of `b'` and `c'` can be moved after. Consider the following extremely hypothetical, not to

[32] Of course TEX does this only because space has catcode 10; give it catcode 12, and it will be treated as a glyph. But you don't want to do that, do you?

[33] What shall we do in this case? We'll see below that we implement substitution between glyph nodes; shall we replace the glue node here with a glyph node? If the replacement is a real glyph, then yes, we can't do otherwise; but if it is another space glyph, we will adjust the original glue node's width (stretch and shrink are of course irrelevant to OpenType and should be dealt with as best seen fit). A space glyph can be spotted quite easily: all its `boundingbox` values are 0. This does not prevent it from having a positive `width` field, though, since a glyph's width does not depend on its shape.

Another problem is that not all glue nodes come from space characters (as I write this, LuaTEX does not distinguish between glue from a space character and glue from an `\hskip`), and kerns should also be considered as space characters. But lookups on space glyphs crucially depend on the space's shape, i.e. its width; such lookups are generally kerning pairs (even Minion's case is kerning in disguise) meant to yield a homogenous space between all pairs of letters — especially important in flowery script fonts — but what if two letters are already separated by an important glue or kern? Should additional kerning be performed? In other words, how to solve the mismatch between what the user wants and what the font requires? I have no answer to that.

mention extremely dumb, example:

```
V{a-}{T}{o}e
```

This gives `Voe` if no hyphenation occurs, and `Va-Te` otherwise. Provided it is typeset with Computer Modern, there should be some kerning between `V` and `a` or `o`, `T` and `e`, and `o` and `e`, i.e. using ⟨*k*⟩ to denote a kern node, without hyphenation we get V⟨*k*⟩o⟨*k*⟩e, and with hyphenation V⟨*k*⟩a-T⟨*k*⟩e. So our string first becomes:

```
{Va-}{Te}{Voe}
```

Then, after applying the kerning lookup:

{V⟨*k*⟩a-}{T⟨*k*⟩e}{V⟨*k*⟩o⟨*k*⟩e}

And now, moving out common parts:

V{⟨*k*⟩a-}{T⟨*k*⟩}{⟨*k*⟩o⟨*k*⟩}e

The kerns have *not* been moved, because they're different in each part.[34]

Now, this is all easier said than done (and it wasn't that easy to say). And it still doesn't solve some problems, for one, what `X` and `Y` actually are depends on the lookup under investigation; e.g. a typical ligature would set `X` as `f` and `Y` as `fi` in `of{-}{}{}fice`, resulting in o{f-}{⟨*fi*⟩}{⟨*ffi*⟩}ce. Worse, this does not even begin to consider what happens if a lookup spans several discretionaries (as in `of{-}{}{}f{-}{}{}ice`), in which case LuaTEX becomes really perverted. We'll leave the issue at that — and as an exercise to the most courageous readers only; suffice it to say here that you might have to "flatten" a few discretionaries, i.e. simply remove them and put their `replace` fields in their place.

11 Lookup types: substitutions

There are two main families of lookups: those related to substitutions, and those related to positioning. The conditions for the former, as illustrated in the preceding section (i.e. the tag/script/language combination), are found in the `fonttable.gsub` table, and those for the latter in `fonttable.gpos`. What a lookup *does* is generally found in the affected glyph(s), with some exceptions. I will review both lookup types: first substitutions, then positioning.

Single substitution. This kind of lookup replaces one glyph with another glyph: e.g. small caps, "old-style" numerals, etc. As an example, Libertine replaces Ş with Ș[35] if the `locl` feature is on and the language is Romanian or Moldavian (and the script, of course, is Latin):

```
name = ss_latn_l_7
type = gsub_single
flags = { }
subtables = { 1 = { name = ss_latn_l_7_s } }
features = {
  1 = { tag = locl
        scripts = { 1 = {
                      script = latn
                      langs  = { 1 = MOL
                                 2 = ROM } } } } }
```

And indeed if we look at some selected portion of the `lookups` table in Ş (350), we'll see:

```
ss_latn_l_7_s = {
  1 = { type = substitution
        specification = {
          variant = Scommaaccent } } }
```

In both tables, the `type` tells us we're dealing with a single substitution; the `specification` subtable will occur in many different types of lookups, but the entries it contains will be different. Here it's quite straightforward: the `variant` entry points to the replacement.

Implementing a single substitution is easy: you just change the `char` field of the node under investigation, and its dimensions (`width`, `height` and `depth`) follow suit. If we want to let LuaTEX do that by itself, though, we can do the following when loading the font:

```
metrics.characters[350] = metrics.characters[536]
name_to_unicode["Scommaaccent"] = 350
```

But this is only the tip of the iceberg. Suppose for instance that we're implementing small caps this way: at position 104, letter *h*, we'll put small-cap *h*. Now, both *T* and *c* form a ligature with *h* in Libertine; *c* will be replaced with its small-cap counterpart, so the ligature will vanish; but *T* (under the assumption that ligatures are handled by LuaTEX too) will be instructed to form a ligature with character 104 (since `metrics` is only interested in codepoints), the result being: *This*. To avoid that, each time something refers to *h* in the original `metrics`, it should be made to refer to small-cap *h* instead. That is not particularly difficult if `name_to_unicode` is properly kept up to date. But single substitutions aren't so innocent anyway; beside the ordering problem mentioned above, single substitutions can be difficult because, as already said, they may requiring analyzing the context, as is the case with `init` and associates, which replaces glyphs in some positions

[34] The user can check that the last code snippet is what LuaTEX returns when left to its own devices by scanning the node list in the `pre_linebreak_filter` callback for a paragraph made of `V\discretionary{a-}{T}{o}e`.

[35] Denoting a voiceless postalveolar fricative commonly used to *shush* people. Put your tongue against the back of your upper teeth; move it a bit toward your palate, you'll feel a ridge there; move it again: that's your postalveolar region (the ridge itself is the alveolar region). A fricative is a consonant whose sound is the air going through an opening so narrow that it generates turbulence.

only. Also, I've said that a tag may be associated with more than one lookup; what I've not said is that those lookups can very well belong to different types, and they do not even need to be all substitutions or positioning; for instance, `onum` in Libertine points to a single substitution (see `ss_l_25`) but also to a positioning lookup in `gpos` (`sp_l_1`) that lowers mathematical operators. So you can leave the substitution to LuaTEX, but you'll still apply the second lookup yourself.

Ligatures. A ligature is the replacement of two or more glyphs with a single one. In TEX, such a replacement is implemented as information in the first character in the ligature, more precisely in its `ligatures` table; for instance *f* (102) in Don Knuth's Computer Modern has the following:[36]

```
ligatures = {
  102 = { char = 11, type = 0 }
  105 = { char = 12, type = 0 }
  108 = { char = 13, type = 0 } }
```

We're looking at the *ff*, *fi* and *fl* ligatures respectively; if TEX does ligaturing by itself, whenever a node with `char` 102 (*f*) precedes a node with `char` 105 (*i*), it will replace them with a node whose `char` is 12, and at position 12 in `cmr10` is *fi*. The `type` we shall ignore.[37]

But where are the *ffi* and *ffl* ligatures? They are stored in the *ff* character (11). Indeed, for TEX, those two ligatures are formed by *ff + i/l*, not by *f + f + i/l*,[38] because a ligature always involves two characters. OpenType fonts, on the other hand, have no upper limit on the number of components of a ligature, and even though *ffi* might be (and sometimes is) described as a ligature between *ff* and *i*, it can also (and generally is) implemented as involving three glyphs. So what? Can't we just write a few lines of code so that *f + f + i = ffi* in `fonttable` is split into *f + f = ff* and *ff + i = ffi* in `metrics`? We can do that; more generally, we could turn a ligature L with n components into $n-1$ ligatures with two components as follows:

$l_1 = c_1 + c_2$
$l_2 = l_1 + c_3$
...
$l_{n-1} = l_{n-2} + c_n = L$

The problem is: what if l_i doesn't exist as an independent glyph? The *ffi* ligature is misleading, because there does exist an *ff* ligature. But that is not always the case, quite the contrary. As an example, the `frac` tag in Libertine defines ½ as *1 + / + 2*, and there is obviously no *1/* glyph. If we proceed carefully, we could define "phantom" characters such as *1/*, whose only purpose in life is to form a ligature with the subsequent character; but then we'll run into trouble if there is no such character, as in:

```
An enumeration:
1/ First ...;
2/ then ...;
```

So the method might work for standard ligatures in a Latin script, where the ligatures with three components are the ones described above, but apart from that it's better to use callbacks.

But let's get back to `fonttable` for a while; I've already shown a ligature lookup above (`ls_l_18`). The only thing we haven't remarked upon yet is the `type`, which is, not surprisingly, `gsub_ligature`. Ligatures themselves are to be found in the replacement glyphs, not in the first glyph as in TEX; thus *f + f + i* is stored in *ffi* (64, 259):

```
name    = f_f_i
unicode = 64259
class   = ligature
width   = 819
boundingbox = { ... }
lookups = {
  1 = { type = lcaret
        specification = { 1 = 266, 2 = 538 } } }
  ls_l_18 = {
    1 = { type = ligature
          specification = {
            char = f_f_i,
            components = f f i } } }
```

The `class` says `ligature` but it may very well be absent; glyph classes aren't mandatory. Hence, what informs us unambiguously that we're dealing with a ligature is the `ls_l_18_s` lookup and its `type`. As with single substitutions, the relevant information is stored in `specification`, and what we're really looking for is `components`, a string of glyph

[36] We are seeing the equivalent of `metrics` for `cmr10` here. Recall that our job is to turn `fonttable` into `metrics` and return it to LuaTEX; but all fonts are implemented with such a table, even if they were loaded automatically, i.e. nothing was registered in `define_font` and a TFM font was used. The table is returned by the `font.getfont` function, which takes as its single argument a font id. Font id's themselves are given by `font.id`, which is passed the control sequence (without the escape character) associated with the font. So what is shown here is `font.getfont(font.id("tenrm")).characters[102].ligatures` (with plain TEX, where `cmr10` was loaded with `\font\tenrm=cmr10`).

[37] But let's explain briefly: it specifies what the output of the ligature should be. 0 means that only the ligature glyph is returned, but one of the original nodes, or both, could also be retained (e.g. *f + i* would produce *fii*); `type` also determines where TEX should continue its ligaturing.

[38] I said in note 29 that once the initial *fi* has been formed, it can't be reused to form another ligature with the following character. I was then talking about the rules for OpenType lookups. TEX, on the other hand, does restart ligaturing at the newly formed ligature itself (unless `type` says otherwise), and *ff* is available to form a ligature with what follows.

names separated by space (glyph names themselves can never contain space).

While we're at it, the first subtable in `lookups` is not really a lookup; its index isn't a lookup name (a string) but a simple number, and it specifies where the caret should be placed to highlight individual components in the ligature. Since PDF has no support for this anyway, we shall ignore it.

Implementing ligatures in LuaTeX can be done as follows: we store all (active) ligatures into categories headed by the first component. Then, when scanning a node list, we check whether any ligature begins with the character of the node under investigation, and if so, whether the following nodes also match the other components. In doing so, it is crucial that ligatures be ordered by length (i.e. number of components) and that longer ligatures be tried before shorter one; otherwise, $f + f + i$ might be turned into *ff* + i and stop there.

Identifying a string of nodes might require skipping over other nodes, if only because the lookups are instructed to ignore some glyph classes; the ligaturing lookup `ls_l_19_s`, for instance, should ignore marks, as discussed above. Once component nodes are properly identified, there are two ways to proceed. The easy way is: delete all component nodes except the first, whose `char` field you set to the ligature's codepoint, as in a single substitution. The best way is (though it makes a difference only when TeX reports information): remove all component nodes, replacing the first with a new glyph node with subtype 2 (ligature); arrange the removed nodes in a proper list (i.e. node i has `prev` set to node $i-1$ and `next` to node $i+1$, except the first node's `prev` field and the last node's `next` are both `nil`), and set the new node's `components` field to the first removed node. For instance, given three component nodes `n1`, `n2` and `n3` in list `head`:

```
local lig = node.new("glyph", 2)
lig.font, lig.attr = n1.font, n1.attr
lig.lang, lig.uchyph = n1.lang, n1.uchyph
lig.char = ⟨ligature codepoint⟩
head = node.insert_before(head, n1, lig)
node.remove(head, n1)
node.remove(head, n2)
node.remove(head, n3)
n1.prev, n1.next = nil, n2
n2.prev, n2.next = n1, n3
n3.prev, n3.next = n2, nil
lig.components   = n1
```

We set the ligature node's `font` (of course), `attr` (for attributes), `lang` and `uchyph` (both for hyphenation) fields to the original node's values, because they store important information.

If nodes were skipped, they will now occur after the ligature node. If those nodes were mark glyphs, then they should be marked (no pun intended) with an attribute so that they can later be placed on the correct component of the ligature (a subject we address more thoroughly below).

Multiple substitution. This type of lookup is the inverse of the previous one: one glyph is replaced with several glyphs.[39] In the `gsub` table, this lookup has `type` set to `gsub_multiple`; in the affected glyph, `type` is `multiple`, and `specification` has a single `components` entry similar to that of a ligature, i.e. a list of glyphs separated by spaces.

What should be done on the TeX side is just the reverse of the previous lookup type; since there is nothing particularly instructive in that, I leave it as an exercise to the reader. The *f*-ligatures in Libertine all have multiple substitution when `smcp` is on (see `ms_l_12`).

Alternate substitution. This maps a single glyph to one or more variants (often, but not always, used in other features); for instance, *R* (82) in Libertine is mapped to a stylistic alternative (otherwise tied to the `ss02` tag) and a small capital (from `c2sc`). In `gsub`, the lookup has type `gsub_alternate`; in the glyph, `type` is `alternate` and `specification` contains a single `components` entry, a string with one or more glyph names separated by space.

This kind of lookup is typically activated with the `aalt` (*Access All Alternates*) tag, through which the user may choose a variant for a glyph, even though the conditions required for this variant to occur aren't met (e.g. you want the medial form of a glyph even though you're not in the middle of a word). A graphical interface with a drop-down list is obviously better fitted than TeX to do that, although one can easily design a `\usevariant{`⟨*number*⟩`}{`⟨*glyph*⟩`}` command.

But alternate substitutions can be put to better effect with another tag, `rand`, for selecting a glyph variant at random. Given the `components` list, one can decide which variant to use via Lua's `math.random` function. Once this is done, the replacement is identical to a single substitution. This feature can be found in the Punk Nova font by Hans Hagen and Taco Hoekwater (after Don Knuth).

[39] I've been telling a white lie: "ligatures" can take a single glyph as input, and "multiple substitutions" can output a single glyph too. In both case, they're equivalent to single substitutions. So those lookup types are better described as respectively "n glyph(s) to 1 glyph" and "1 glyph to n glyph(s)" replacements, with $n \geq 1$. That doesn't change the import of what is described in the main text. See the slash (47) in Libertine for an example of a "one-glyph ligature".

Contextual substitution. Those last couple of lookup types weren't very exciting. But now, fasten your seat-belt, we're in for the real thing: contextual substitution, and chaining contextual substitution, and finally reverse chaining contextual single substitution, all that possibly expressed in three different formats ... Alas, the terribly-named reverse chaining contextual single substitution isn't supported in LuaTEX (or virtually anywhere else), so we won't be studying it.[40]

Up to now, we've seen lookups identifying input and replacing it with some output. Contextual lookups also identify input, but then they call other lookups on parts of that input to do the substitutions. For instance, given input `ABCD`, `A` may be turned to a small cap, `BC` may form a ligature, whereas nothing happens to `D` but its being there is still crucial to identify the proper input, i.e. otherwise the former two substitutions wouldn't have been performed.

As alluded to above, contextual lookups may be expressed in three formats. For contextual substitutions proper, we'll see the glyph-based format; in the subsection below about chaining contextual substitution, class-based and coverage-based formats will be investigated; but all three formats can be used with both types of lookup.

Libertine has a tag called `gtex` (for *Greek TEX*) which turns the input sequence `TeX` into $\tau\varepsilon\chi$ (with lowercase *tau* and *chi* so they won't be confused with T and X). Of course, you don't want to change every T, e and X to tau, epsilon and chi respectively; rather, you want the substitution to be performed only when the three letters are ordered as in `TeX`; in other words, you want a contextual substitution.

If we look into `cs_l_4` — the lookups associated with `gtex` — in `fonttable.gsub`, we won't learn anything we don't already know, except that it has type `gsub_context`. While we're at it, we can retrieve the name of its only subtable: `cs_l_4_s`. We won't find what this subtable does among the glyphs, as with the previous lookup types; instead, contextual lookups live in their own table, `fonttable.lookups`, where they are indexed by name. So, let's look at `fonttable.lookups.cs_l_4_s`:

```
type   = contextsub
format = glyphs
rules  = {
  1 = { glyphs  = { names = T e X }
        lookups = { 1 = ss_l_2
                    2 = ss_l_2
                    3 = ss_l_2 } } }
```

The important information is contained in the `rules` subtable; it describes the context and what to do with it. But in order to understand that subtable, the value of `format` is crucial: it tells us how the context will be defined. The `rules` table is made of subtables at consecutive indices, each defining a context and what to do with it, somewhat like subtables in a lookup of the types seen before: the first subtable that matches wins the prize.

The context itself is defined in `glyphs`, which contains at least a `names` subtable, and if it is a chaining contextual substitution, perhaps also `back` and `fore` fields. Those three fields are strings made of glyph names that denote an input sequence (space is used to delimit names only), somewhat like the `components` entry of a ligature substitution. Here, then (it goes without saying but let's say it anyway), the relevant input string is `TeX`.

The `lookups` table lists what should be done to the successful input sequence. It is not a simple array; instead, the indices correspond to positions in the input sequence. Here the table reads: apply lookup `ss_l_2` to the glyph at position 1; then apply it again at position 2 and 3. If we explore `ss_l_2` in `fonttable.gsub`, we'll see that it's a single substitution, and in the tables for T (84), e (101) and X (88), we'll find that it maps those glyphs to their Greek counterparts. In other words, `ss_l_2` is executed like any other lookup, except that, each time it is called, it inspects only one position in the node list, not the entire list.[41]

A crucial point is that each index in `lookups` identifies a position in the input sequence *after* the previous lookup has been applied. In our example, this doesn't make any difference, but suppose a contextual lookup is designed to match against `ABC`, and suppose the first lookup it calls (at index 1) is a ligature, so that `AB` is turned into `X`; then the sequence is now `XC`, and if a second lookup is called to act on `C`, it will have index 2, not 3, even though `C` originally was at position 3.

Also, the `lookups` table isn't mandatory: a contextual lookup may very well identify a valid input sequence and do nothing with it. Below we'll see a

[40] The main difference between reverse [...] substitution and the others is that it processes the node list starting at the end; e.g. given `ABC`, first it deals with `C`, then `B`, then `A`. This has nothing to do with the direction of writing, though it was meant for Arabic calligraphy, where a glyph variant may be determined by the shape of the following glyph (as in the Nasta‘līq style), so that the latter should be set before the former. Nonetheless, none of the Arabic fonts I've seen use reverse [...] substitution.

[41] More accurately, the lookup inspects input from one position only, even though it might need to inspect several glyphs, as in a ligature.

detailed example of that, but here we can say that it can be used to prevent a subsequent subtable from matching: for instance, if you want to define context `ABX` as `AB` followed by any glyph but, say, `z`, then you can define a contextual lookup with a first subtable matching `ABz` and doing nothing, and the second simply as `AB`, i.e. matching independently of what follows.

How shall we implement a contextual substitution? Identifying the input can be done as for a ligature; then each lookup is applied as usual, albeit on a particular node (according to the lookup's index), which can be ensured simply by counting glyph nodes. The only subtlety is that lookups should be applied in order, i.e. lookup at index m should be executed before lookup at index n, with $m < n$. In our example, since indices in `lookups` are consecutive, a simple `ipairs` will respect the ordering; however, there might not be a lookup at each index, and `lookups` could have entries at indices `1`, `2` and `4`, for instance, in which case `ipairs` will stop at `2` whereas `pairs` won't iterate in any particular order. Thus one should create another table reflecting the original order but also retaining the positions:

```
local t = {}
for i, l in pairs(lookups) do
  table.insert(t, {position = i, lookup = l})
end
table.sort(t, function (a,b)
               return a.position < b.position end)
```

Now we can traverse `t` with `ipairs` and apply the lookups in order.

Chaining contextual substitution. Contextual substitutions, like other substitutions, completely consume their inputs, even if lookups are applied at some positions only. For instance, a contextual substitution acting on `TeX` and simply turning `e` to epsilon will nonetheless consider `X` as processed; the next iteration of the current lookup will begin at the next character. Also, since what came before the current position is considered processed too, contextual substitutions can't take that into account either. In other words, substitutions of that type (and of all the types seen up to now) have no memory, and they can't foresee the future.

A chaining contextual substitution, on the other hand, is precisely that: a contextual substitution with memory and foresight. In other words, it can take into account already-processed glyphs and future glyphs without consuming the latter. Chaining contextual substitutions are so useful that virtually all fonts use them even when simple contextual substitutions would do.

In essence, a chaining contextual substitution works like a simple contextual substitution: it identifies a sequence of glyphs and calls other lookups at some positions in this sequence. But it can also identify preceding glyphs, called the *backtrack sequence*, and/or subsequent glyphs, called the *lookahead sequence*.

An important point to keep in mind is that, whatever the format, the backtrack sequence is always set in reverse order of the direction of writing; for instance, to identify `abc[xyz]def`, where `abc` and `def` are the backtrack and lookahead sequences respectively, and `xyz` the input sequence proper, a lookup in the `glyphs` format (as in the previous section) would have the `glyphs` table organized as:

```
back  = c b a
names = x y z
fore  = d e f
```

Note how the `back`track sequence is displayed.

In Libertine, the `ccmp` tag (*Glyph Composition/ Decomposition*, a somewhat all-purpose tag) activates a lookup, `ks_l_32`, with two subtables. After noting the lookup's `type` (`gsub_contextchain`), let's look at `ks_l_32_c_0`, the first of those two subtables, in `fonttable.lookups`:

```
type = chainsub
format = coverage
rules = {
  1 = {
    lookups  = { 1 = ss_l_0 }
    coverage = {
      current = { 1 = f f_f }
      after   = {
        1 = parenright question T ... } } } }
```

This subtable uses the `coverage` format, which specifies a set of glyphs for each position in the input, backtrack, and lookahead sequences, denoted respectively by subtables called `current`, `before` (missing here) and `after`, whose indices are the position in the sequences (starting from the end in `before`) and the values at those indices are strings representing the glyphs. In this example, the input sequence is made of one glyph, either *f* or *ff*, and the lookahead sequence also contains one glyph, which can be a question mark, a right parenthesis, *T*, and many others elided here. So, *f?, f), fT, ff?* ... will all be identified by this single rule. This works a bit like a regular expression (with `f_f` denoting the ligature): `[ff_f][?)T...]`. If one of the three sequences had contained more than one glyph, it would have had an equal number of entries.

The `lookups` table should be read as in a simple contextual substitution, with indices denoting positions in `current`. Here, `ss_l_0` should be applied at position one; this lookup replaces *f* (both on its own and in the ligature) with a variant whose arm is shorter, so it doesn't touch the next glyph: *f?* becomes *f?*.[42]

One last remark about our example: although the relevant context here is described at entry 1 in `rules`, there can be no other entries in that table, since the `coverage` format allows only one context per subtable. Hence another context would be defined in another subtable. That is not true of the other two formats, `glyphs` and `class`.

Let's turn to an example of the latter. Libertine can turn *etc.* to *&c.* thanks to the `etca` tag (ET CA*etera*, or ETC. *with Ampersand*, or ETC. *Alternate*, or ETC. with a dummy letter only because tags should be made of four characters; I did *not* spend hours trying to find a meaningful name). The core of the feature is a simple ligature ($e + t$ to *&*), but performed if and only if the two letters are at the beginning of a word (to exclude text like *fetch*), and followed by *c* and a period (so *etch* is also excluded).[43]

The `etca` tag points to lookup `ks_l_5`, whose only subtable `ks_l_5_s` is (in `fonttable.lookups`):

```
type         = chainsub
format       = class
before_class = { 1 = space parenleft bracketleft }
current_class = { 1 = e, 2 = t }
after_class  = { 1 = c, 2 = period }
rules = {
  1 = { class = {
          before  = { 1 = 0 }
          current = { 1 = 1, 2 = 2 } } }
  2 = { class = {
          current = { 1 = 1, 2 = 2 }
            after = { 1 = 1, 2 = 2 } }
        lookups = { 1 = ls_l_3 } } }
```

As expected, `format` is `class`; but what is a class? First, it has absolutely nothing to do with the glyph classes mentioned above. Here a class is simply a set of glyphs created especially for a lookup. More precisely, classes are defined separately for the input, backtrack and lookahead sequences in (respectively) `current_class`, `before_class` and `after_class`. In each of those tables, a class is denoted by its index, and its content is a string of space-separated glyph names. For instance, class 1 for the backtrack sequence here contains space (recall that we said that space is a glyph in OpenType fonts), left parenthesis and left bracket, while the classes in the other sequences are singletons. In each subtable of `rules`, the context is described analogously to the `coverage` format: indices in the `current`, `before` and `after` subtables denote positions, but here the values point to classes (one class per position); for instance, at position 1 in the input sequence for the first rule, there should be a member of `class` 1 for that sequence, i.e. an *e*. This is, again, a bit like regular expressions, except that sets `[...]` are assigned to variables beforehand and the regexp is defined with those variables.

Classes are a bit special in that they do not intersect, i.e. a glyph belongs to one and only one class (for each sequence, that is). This fact is of little value to us (though quite important to the font designer), except when paired with another one: that there exists a default class, which need not be defined and whose index is 0; this class contains all glyphs, except those present in other classes. Thus, class 0 in `before_class` contains everything but space, left parenthesis and left bracket, class 0 in `current_class` contains everything but *e* and *t*, and class 0 in `after_class` contains everything but *c* and period.

Class 0 is put to good use in rule 1; this rule matches any sequence `Xet` where `X` is anything but `(`, `[` or a space, since class 0 contains all glyphs but them. For instance, if we're inspecting `fetch`, rule 1 will match, preventing rule 2 from being applied at the same position — and that's the only reason why rule 1 exists at all: to prevent rule 2 from being applied in most circumstances; accordingly, rule 1 has no `lookups` table. However, rule 1 will not match with `(et` or `[et` `␣et` (no matter what follows), since the initial glyph in each case doesn't belong to class 0, and rule 2 will perhaps match if the right sequence follows.

The implementation of a chaining contextual substitution is similar to that of a simple contextual substitution, except that we may request surrounding glyphs to identify themselves. No matter how many nodes we scan to compare them to the lookahead sequence, they do not count as processed, and the next iteration of the lookup starts at the glyph to the immediate right of the input sequence; in our example, provided the right context has been found, the lookup will start again at *c*.[44]

At this point, the reader might have a question nagging at the back of his or her mind, namely:

[42] The second subtable in `ks_l_32` substitutes *ı* and *ȷ* for *i* and *j* before accents, which will then be properly positioned.

[43] No word in English ends with *etc*, so the second condition would be enough; but you never know.

[44] Of course here it is impossible for the lookup to match on *c*, and it could have been skipped (i.e. the following *c* and period could very well have been parts of the input — not lookahead — sequence). However, the ampersand substitution could have been part of a much more general lookup implementing a wide range of stylistic variants, one of which might take *c* as its input.

what is the difference between a contextual lookup (chaining or not) with n subtables containing one rule each, as illustrated by `ccmp` above, and the same lookup with one subtable containing n rules, as `etca` here? As far as what the lookup *does* is concerned, the answer is: none, since a subtable matches when one of its rules matches, and a successful subtable or rule prevents the other ones from being applied. But there are some technical differences which may dictate why one implementation is chosen over the other: first, as already mentioned, only one rule per subtable is allowed in the `coverage` format, so a lookup will necessarily use several subtables; second, all rules in a subtable are in the same format (as witnessed by the fact that the `format` entry is on the same level as the `rules` table, and thus holds for all the rules contained in the latter), whereas different subtables in the same lookup may be expressed in different formats; if two contexts are better expressed in two different formats, then different subtables can be used.

12 Lookup types: positioning

At this point, we're done with substitutions (since reverse substitution is in limbo for the present). We turn to positioning lookups, held in the `fonttable.gpos` table.

Single positioning. This kind of feature moves a glyph horizontally and/or vertically and modifies its horizontal and/or vertical advance. Typographically, that means modifying the glyph's sidebearings (moving a glyph left/right increases/decreases the left sidebearing, and increasing/decreasing its width does the same for the right sidebearing). For instance, capital spacing in Libertine (activated by the `cpsp` tag) is implemented by lookup `sp_l_0` (with type `gpos_single`), whose only subtable `sp_l_0_s` is detailed in each uppercase glyph's `lookups` table, e.g. *A* (65):

```
sp_l_0_s = { 1 = {
    type = position
    specification = { x = 2, h = 5 } } }
```

As usual, what should be done is detailed in the `specification` subtable. For single positioning, the table may have up to four values: `x` is the horizontal displacement of the glyph; `h` is its width correction; `y` and `v` are the same things in the vertical direction.

Now suppose we're implementing capital spacing. We browse all capitals in a node list, or rather (and more generally), all glyphs that have this particular lookup. For each such glyph we add a kern of `x` units before if it follows a similar glyph, and a kern of `h` units after if it precedes a similar glyph (the *if*-clauses translate the fact that space adjustment should take place only *between* capitals). We could also merge the two kerns for each pair.

A caveat: capital spacing does not replace other forms of kerning, particularly the default kerning (denoted by the `kern` tag). If that kerning is done automatically by LuaTEX (because the `kerning` callback is left untouched or the `node.kerning` function is used), that should occur before capital spacing, otherwise our additional kerns will prevent it. In turn, when looking for the preceding or following glyph, we should ignore intervening kerns (which have the special subtype 0).

An example of vertical positioning can be found in `sp_l_1` (triggered by `onum`, whose only subtable `sp_l_1_s` lowers mathematical operators so they are better placed with “oldstyle” numbers). Thus the `specification` for this subtable in glyphs $=$, $+$, $-$, $\times$ and $\div$ (61, 43, 8722, 215 and 247 respectively) contains a single entry at `y` whose value is `-100`; then it suffices to assign (or rather, add) -100 units to the glyph nodes' `yoffset` field to implement the lookup. I leave it at that here, since `yoffset` is detailed more thoroughly below, along with its horizontal twin `xoffset`.

Pair positioning (kerning). This second type of adjustment is well known: it is the tiny amount of space (possibly negative) added between two letters that look badly set when just left next to each other, for instance between *T* and *o* (compare kerned *To* with “natural” *T o*).

Kerning pairs go in two formats. First, there is kerning information for individual pairs, which is found in the glyph table for the left member of the pair, more precisely in the `kerns` (not `lookups`) subtable. For instance, looking precisely at this subtable for *T* (84):

```
kerns = {
 1 = { char = udieresis
       off  = -44
       lookup = { 1=pp_l_2_g_0, 2=pp_l_2_k_1 } }
 2 = { char = odieresis
       off  = -44
       lookup = { 1=pp_l_2_g_0, 2=pp_l_2_k_1 } }
 3 = { char = adieresis
       off  = -36
       lookup = { 1=pp_l_2_g_0, 2=pp_l_2_k_1 } }
 ... 22 more
 }
```

This means that *ü* and *ö* should be brought closer to a preceding *T* by 44 units, whereas for *ä* it should be 36 units.[45]

[45] The `lookup` table shouldn't be a table at all, but a string identical to the table's first entry, i.e. `pp_l_2_g_0`; this hap-

However, different pairs often share the same amount of kerning. Better yet, classes of glyphs on the left will often have the same kerning when followed by classes of glyphs on the right. For instance, *A*, *À*, *Á* ... should behave similarly, because here accents make little difference (this is not true for a lowercase letter). Accordingly, kern pairs are often defined in kern tables; for instance, here's the entry at index 3 in `fonttable.gpos` (much abridged):

```
3 = {
  type = gpos_pair
  flags = { ignorecombiningmarks = true }
  name = pp_l_2
  features = { ... }
  subtables = {
    1 = { name = pp_l_2_g_0 }
    2 = {
      name = pp_l_2_k_1
      kernclass = {
        1 = {
          firsts = {
            2 = r v w y yacute ydieresis ...
            3 = a i u ...
            ...  }
          seconds = {
            2 = a aogonek
            3 = c e o q ccedilla
            ...
            11 = exclam parenright ...  }
          offsets = { 13 = -15
                      14 = -10
                      18 = -29
                      ...  }
          lookup = pp_l_2_k_1  } } } } }
```

We discover *en passant* that such kerning is associated with the `kern` tag, which activates both the per-glyph kerning (see the first subtable with lookup `pp_l_2_g_0` — also associated with the individual kerning pairs for *T* in the previous code snippet) we've seen above and the kerning by class that we're interested in; it is of course crucial that the per-glyph subtable take precedence over the kerning table: the latter is powerful yet indiscriminate, while the former is limited but accurate. Kerning tables deal with glyphs massively, and individual kerning pairs set the exceptions.

The `kernclass` table has the information we need, namely a kerning table,[46] composed of four entries: `firsts` is an array of glyph classes on the left; `seconds` is an array of glyph classes on the right; `offset` is the amount of kern needed for each pair of classes; and `lookup` is just redundant; we can ignore it and stick to the `name` field. Let f_i, s_j, o_k stand for entries `firsts[i]`, `seconds[j]` and `offsets[k]` respectively; then kerning tables can be read as follows:

$$\begin{array}{lllll} & s_1 & s_2 & \dots & s_n \\ f_1 & o_1 & o_2 & \dots & o_n \\ f_2 & o_{n+1} & o_{n+2} & \dots & o_{n+n} \\ \dots & \dots & \dots & \dots & \dots \\ f_m & o_{mn-n+1} & o_{mn-n+2} & \dots & o_{mn} \end{array}$$

In words, the amount of kerning between classes i and j is $o_{(i-1)n+j}$, where n is the length of the table `seconds` (i.e. its highest index; do not use the Lua length operator `#` here, it will return random values, since the entry at index 1 is always missing for reasons explained just below). Thus, between *r* (in class f_2) and *e* (in class s_3), there should be a kern whose width is specified in $o_{(2-1)\times 11+3} = o_{14} = -10$ units — unless, of course, such information is overridden by per-glyph kerning, as seen for *T* above.

When o_i would be 0, i.e. no kerning is specified between two classes, the entry is omitted in the `offsets` table. Another important point is that f_1 and s_1 are special: like class 0 in contextual substitutions above, they hold all the glyphs that don't appear in other classes, and thus aren't explicitly defined. Kerning for those classes is seldom, if ever, specified, for reasons that should be obvious.

LuaTEX is able to handle such kerning by itself; to do so, we fill the `kerns` table that each character may have; e.g. for *r* it would contain the following information:

```
kerns = {
  [name_to_unicode.a]       = -9830,
  [name_to_unicode.aogonek] = -9830,
  [name_to_unicode.c]       = -6554,
  ... }
```

assuming the font has been loaded at 10pt and there are 1000 units per em, so that the first value is $\frac{-15*10*65536}{1000} = 9830.4$ scaled points (rounded, since there is nothing smaller than a scaled point). However, LuaTEX won't insert a kern if anything occurs between two glyph nodes (barring discretionary nodes, which it will certainly do better than us), and we have just seen such intervening material in the previous subsection with capital spacing.[47] Also,

pens quite regularly with similar `lookup` entries. The correct information can be reliably retrieved with:

```
local lk = (type(lookup) == "table" and lookup[1])
           or lookup
```

[46] Actually, `kernclass` is made of subtables, each one being a kerning table; but only the first is valid, the others are defined elsewhere and should be ignored when repeated as the non-first subtables.

[47] Capital spacing is a single positioning lookup. But given that the tag has additional semantics, namely that the positioning should occur if and only if the uppercase letter is preceded and/or followed by another capital letter, we can reinterpret it as kerning with the pairs *A/A*, *A/B* ... *Z/Y*, *Z/Z*. Then, for each glyph m on the left and glyph n on the right, we can set entry n in `kerns`, as $h_m + x_n + k_{mn}$, where

the kerning lookup here is instructed to ignore marks (see the `flags` entry), and LuaTEX can't do that.

Mark positioning. We turn now to the placement of diacritics, i.e. glyphs that exist only relative to another. Only the latter should be visible to the justification engine afterward; the diacritic itself may only contribute to the vertical dimension of the resulting combination.

Since Unicode defines many precomposed characters, independent diacritics can often be avoided. Yet situations still abound where mark positioning is crucial — if only because the font has no glyph for a given precomposed character. As an example, the International Phonetic Alphabet uses a ring below a symbol to denote a voiceless sound (when no independent symbol exists), e.g. *m̥*.[48] If we want to do phonetics with Libertine, we'll have to use mark positioning, because the symbol doesn't exist in that font (or in Unicode, for that matter).

At this point, it might be useful to review how TEX positions diacritics. The `\accent` primitive, used as:

`\accent` ⟨*number*⟩ ⟨*char*⟩

places the character at position ⟨*number*⟩ in the font on the character at position ⟨*char*⟩ as follows: first, if ⟨*char*⟩'s height differs from the font's x-height, i.e. `\fontdimen5` (a.k.a. `parameters.x_height`), then the accent is put into a box shifted vertically by:

$$shift = height_{chr} - xheight$$

Then the accent is surrounded by kerns so that it is centered on the accentee, modulo the declared slant per point (`\fontdimen1` a.k.a. `parameters.slant`). If we denote the width of the first kern by k_1 and the second by k_2, they can be computed as:

$$k_1 = \frac{width_{chr} - width_{acc}}{2} + \frac{slant}{1pt} \times shift$$
$$k_2 = -(k_1 + width_{acc})$$

We can see that the second kern is used only to cancel whatever horizontal displacement was caused by the first kern and the accent. The entire operation can be justified by saying that TEX expects an accent to be designed to fit an ascenderless lowercase letter.

h and x are the corresponding entries in `specification` for the single positioning lookup and k is the (real) kern between the two glyphs. Then we can let LuaTEX do the kerning by itself, although, of course, this is just a special case.

[48] A voiceless consonant is produced with the vocal folds open, so that the air flows through soundlessly; in a voiced consonant, the vocal folds are closed and flap under air pressure, producing a vibration. In each of the pairs *t*/*d*, *p*/*b* and *f*/*v*, both sounds are identical except that the first one is voiceless while the second is voiced. To produce a voiceless *m*, either say *m* without humming, or say *p* while expelling air through the nose, or whisper *mama*, although technically whispering isn't voicelessness.

OpenType fonts don't work this way; instead, glyphs have anchors, and a diacritic is placed relative to a base glyph by aligning those anchors. Also, following Unicode, a diacritic is expected to follow the character it modifies: `e^` denotes *ê*. What road shall we follow: TEX's diacritic–base or Unicode's base–diacritic? And does it make any difference?

To see the problem in action, suppose we're scanning a node list to perform mark positioning. If marks were denoted in TEX's way, then they can easily be spotted thanks to the kerns used to position them, which have `subtype` 2. We can undo whatever TEX tried to do and reposition the marks. If we follow Unicode, however, TEX will do nothing and would-be diacritics will simply stand suspended in mid-air with nothing to distinguish them.

But OpenType fonts are a little bit more clever than that: We can rely on glyph classes. Whenever a "mark" glyph is encountered, it should be attached to the preceding glyph (if the corresponding feature has been activated). Thus, when scanning a node list, we should check the class of each glyph node (actually, we've been implicitly doing that all along, since we know some lookups ignore some glyph classes). Not all fonts have classes and anchors, however, only those that implement mark positioning. For the rest (e.g. Latin Modern), one should rely on TEX's `\accent`, because we won't be able to do better than that. But otherwise the OpenType method should be followed because if we put diacritics before their bases *à la* TEX, as in `A`⟨*mark*⟩`BC` where OpenType expects `AB`⟨*mark*⟩`C`, then they might mess with a lookup designed to act on `A` and `B` and not instructed to ignore marks because there shouldn't be one to begin with.

Let's get back to voiceless *m* (109). If we explore the letter itself, we'll see it has an `anchors` subtable (also, not shown here is the glyph's `class`: it is `base`):

```
anchors = {
 basechar = {
  Anchor-2 = { x = 522, y = 645,  lig_index = 0 }
  Anchor-5 = { x = 157, y = 9,    lig_index = 0 }
  Anchor-6 = { x = 382, y = -105, lig_index = 0 }
 } }
```

The anchors here all belong to the same category, `basechar`, meaning that they are the anchors to be used when a mark is to be attached to the letter. The ring-below glyph (755), on the other hand, has the following anchors (and its `class` is `mark`):

```
anchors = {
  mark = { Anchor-6 = { x = 92, y = -116,
                        lig_index = 0 } } }
```

The only anchor here is in the mark category, meaning it should be used when the glyph is moved relative to another. In both cases, the anchor category might seem redundant given the glyph's class, but we'll see below that it is not: there might be anchors with different categories.

Now, if we want to position the ring (the mark) relative to *m* (the base), we have to align the anchor of category basechar in the latter with the corresponding anchor of category mark in the former; in this case, the anchor is Anchor-6. So, assuming we're doing it the Unicode way, i.e. base followed by mark, we have to align the two glyphs at the origin, which, in a left-to-right writing system, means that we must move the mark left by the base's width, then shift it horizontally by $x_{base} - x_{mark}$ and vertically by $y_{base} - y_{mark}$. Finally, all this movement should be invisible, so that after the operation we end up at the base's right border. We could use kerns and an hbox to do that, but LuaTEX offers a much simpler solution, manipulating the mark's xoffset and yoffset fields (which are present in all glyph nodes). We still need a kern after the mark to cancel its width, though.

But we've moved a bit too fast. Now that we're more knowledgeable about lookups, we know that we should always check their subtables, because they give us the order of operations; so let's have a look at mb_l_3 in gpos (triggered by the mark tag):

```
name = mb_l_3
type = gpos_mark2base
features = { ... }
flags = { }
subtables = {
  1 = { name = mb_l_3_a_0, anchor_classes = 1 }
  2 = { name = mb_l_3_a_1, anchor_classes = 1 }
  ⟨... five more ...⟩
  }
```

(We furtively note the lookup's type.) The subtables have names, as usual; however, we haven't seen those names in the glyph's information for this lookup, as was the case for other types. That's because the link between the lookup in gpos and its details in affected glyphs is indirect in this case; subtables are tied to anchors (as indicated, somewhat redundantly, by the anchor_classes field), which anchors we then find in the glyphs, as we have already seen. Anchors are enumerated in fonttable.anchor_classes; here are the first two:

```
1 = { type = mark
      name = Anchor-0
      lookup = { 1 = mb_l_3_a_0, ... } }
2 = { type = mark
      name = Anchor-1
      lookup = { 1 = mb_l_3_a_1, ... } }
```

(Again, lookup is buggy; the correct value is the first entry.) Now, to retrace our steps: the mb_l_3 lookup has subtables, each associated with one or more anchor(s), which anchors we then find in some glyph(s). Thus, to implement mark positioning on a given mark glyph: for each subtable of the lookup, and for each anchor in that subtable, we check if the mark has this anchor in its anchors.mark subtable and if the nearest preceding base glyph has this anchor in its anchors.basechar subtable. If so, we align the two anchors (and the lookup, as usual, is considered processed: subsequent anchors and subtables are ignored). Now, the reader can check that the seventh subtable in mb_l_3 points to Anchor-6, which is exactly the one we need to, at long last, put that ring below *m*.

Now suppose we're in a node list, with a node mark to be positioned on the glyph base immediately on its left. We've retrieved the necessary anchor for each glyph, which we denote with ma for the mark's anchor and ba for the base's anchor. Then here's how the positioning is to be performed:

```
mark.xoffset = ba.x - ma.x - base.width
mark.yoffset = ba.y - ma.y
local kern = node.new("kern", 2)
kern.kern = -mark.width
head = node.insert_after(head, mark, kern)
```

The kern has subtype 2, as a reminder that it is used for accent placement. Note that xoffset is invisible to TEX, so there is no need to take it into account in the kern's width.[49] Also, marks often have no width (i.e. they are drawn entirely to the left of their bounding boxes), in which case the kern may be avoided entirely.[50]

On the other hand, yoffset does have an effect, but only on height, not depth; this means that if we're dealing with a mark placed under the baseline, and yoffset is non-zero, then the depth of the horizontal box containing the character might not be properly computed. Devising an alternate solution, using kerns and boxes as TEX natively does and computing depth properly, is left as an exercise to the reader.[51]

49 In right-to-left typesetting, we would have to move the mark right by its width, not the base's width, since glyphs are always drawn with the cartesian origin at the bottom left corner; however, we would still use a negative value for xoffset, because this field follows the writing direction.

50 Actually, no matter what fonttable says, we could set the widths of all mark glyphs to 0, given that they are supposed to be non-spacing glyphs. Then we could do without the kern altogether, thus tinkering less with the nodelist.

51 Said reader may also be glad to learn that LuaTEX has primitives (inherited from PDFTEX) that set the height and depth of paragraph lines independently of their contents:

Mark to mark positioning. Positioning a mark on a base glyph is not the only possibility in diacritic placement; we may want to position a mark, let's call it `mark1`, relative to another one, `mark2`; this occurs for instance if you want a tone marker and an accent on the same vowel in Vietnamese: the first should be placed relative to the second, not relative to the third.[52] The process is similar to what we've just seen, except that: the lookup's type in `fonttable.gpos` is `gpos_mark2mark`, and in `anchor_classes` the type is `mkmk`; `mark2`'s anchor is to be found in the `basemark` subtable of the `anchors` table (parallel to the `basechar` subtable for a base glyph); and most importantly, since `mark2` itself is very likely to be moved on the nearest base glyph, this positioning, reflected in the `xoffset` and/or `yoffset` of `mark2`, should be taken into account when moving `mark1` (actually, we should have done that with mark-to-base placement too, because it may well happen that a base glyph is moved, if only because the end user does so by hand). Libertine contains no lookup of this type.

Mark to ligature positioning. Finally, a mark may be positioned relative to a component in a ligature. Recall the "`c¸h`" example discussed above: the ligaturing lookup (which ignores marks) will turn it into "*ch¸*", and we should position the cedilla so as to obtain "*çh*". This is not simple mark-to-base positioning, because the mark could (theoretically, at least) be set on either of the two original components. That is why an anchor in a ligature has several instances of itself, each associated with one of the original components (not all components need to have an anchor, though); so, if a mark should be placed relative to the first component, the first instance of the anchor shall be used, while for the second component the second instance is the right one, and so on.

Of course, this implies the ligaturing lookup has stored the necessary information, namely the component with which the mark was originally associated. To do so, we can set an attribute in the mark node; in our case, we would set this attribute to 1, indicating that the mark is associated with the first component.

In Libertine, the `mklg` tag activates the `ml_l_4` lookup, whose `type` is `gpos_mark2ligature`.[53] This lookup has one subtable associated with `Anchor-7` (whose type in `anchor_classes` is `mark`, but should be `mklg` — no relation to the tag). If we look at the cedilla (184), we'll see that it has this anchor in the `mark` subtable — so we can at least conclude that, no matter what X is in "mark-to-X positioning", the mark's anchor always has the same category. But the `anchors` table for the ligature (57, 403) is different:

```
anchors = {
  baselig = {
    Anchor-7 = { 1 = { x = 212
                       y = 2
                       lig_index = 0 }
                 2 = { x = 470
                       y = 2
                       lig_index = 1 } } } }
```

Anchors of type `baselig` are made of subtables, each specifying one instance of a given anchor. The index of each subtable correctly identifies the associated component in the ligature, because subtables aren't continuously numbered. If `Anchor-7` was defined for the second component only, the subtable would still be at index 2, not 1. So the `lig_index` field can be done without, all the more as it has the inhuman habit of starting counting at 0.

As for the implementation, it is identical to what we've already seen: anchors should be aligned. The only difference is in how to find the right anchor.

Cursive attachment. In a script (cursive) font, letters should be properly attached together. The Latin alphabet poses no problem: all letters are on the baseline anyway, so it is up to the font designer to ensure that exit points and entry points match, at least vertically (horizontal adjustment can be left to kerning); in other words, entry and exit points should be at the same height. However, that is not true of some Arabic scripts, such as Nasta'līq,[54]

`\pdfeachlineheight` and `\pdfeachlinedepth`. The wrongly computed depth can then be ignored, except when building an independent hbox.

[52] Tone is the use of pitch as a lexical device (i.e. to distinguish between words), just like phonemes do in non-tonal (and, of course, tonal) languages; that is different from intonation, used in all languages, which do not distinguish words: if you say *cat* and then repeat it with a raising intonation, as in a question, it's still the same word.

A little less off-topic: there exist precomposed characters for Vietnamese in Unicode, and they have glyphs in Libertine, so the example is slightly spurious.

[53] Mark to ligature positioning is usually activated with the `mark` tag, just like mark to base positioning. Here I've used a different tag just so users can test it independently.

[54] Nasta'līq is used mostly for Indo-European languages (especially, Urdu), thus totally unrelated (or rather, to this day not convincingly — for most linguists — related) to Arabic or the Semitic languages more generally or even the Afro-Asiatic family, unless one is willing to accept the Nostratic superfamily or even more remote and controversial linguistic classifications. But then, languages as unrelated to Latin as Basque, Mohawk, Vietnamese, Wolof, and many, many more, are written in the Latin alphabet.

where entry points are generally higher than exit points, and where only the last glyph of a word is set on the baseline:

citehtap si yhpargillac qīl‘atsaN ta tpmetta sihT

The position of each glyph thus depends on the position of the next in the word. To implement that, anchors are used again, but this time each anchor is twofold: there is an entry point and an exit point. For each pair of consecutive glyphs, the entry point of the second glyph should be aligned with the exit point of the first glyph, and a kern and `yoffset` are all we need to do so. Although the operation is similar to mark positioning, it differs in one important respect: all glyphs are spacing here, so there is no kern to compensate for the second glyph's width. Libertine has no cursive positioning, but a typical `anchors` table in a glyph would look like:

```
anchors = {
  centry = {
    Cursive-8 = { x = ⟨x1⟩
                  y = ⟨y1⟩
                  lig_index = 0 } }
  cexit = {
    Cursive-8 = { x = ⟨x2⟩
                  y = ⟨y2⟩
                  lig_index = 0 } }
 ⟨other anchors⟩
}
```

Thus, anchors for cursive attachment are identical to other anchors, except that they belong to the `centry` and `cexit` subtables. As for the lookup itself in `gpos`, it has the `gpos_cursive` type, whereas the anchors' type in `anchor_classes` is `curs`.

As mentioned above, the lookup is mostly used to attach glyphs in words with the last glyph on the baseline; that means that positioning should begin at the end of the word and progress contrary to the direction of writing. This stipulation is not part of the lookup type itself; instead, a flag (in the `flags` subtable of the lookup in `gpos`) is used: `r2l`, meaning "right-to-left".[55] This in turn implies that we have a definition of a "word"; space will do, as discussed a few pages ago.

(Chaining) contextual positioning. We'll omit discussion of these two lookup types, not because there is nothing interesting here, but simply because they are identical to (chaining) contextual substitutions, except that they dispatch to positioning, not substitution, lookups. I hope the reader has the buoyant feeling associated with the last-minute cancellation of a dreaded two-hour class (each Friday at 5 PM). But let's just say that contextual positioning can be used e.g. in *Wörter* (German for *words*) to lower the umlaut so that kerning can be increased with the preceding *W* (this of course requires that the *o* and the accent are separate glyphs, probably due to a multiple substitution), or to add kerning between the period and *T* in *S.A.T.*, the kerning being actually (visually) with the preceding *A*.

13 Conclusion

I hope the reader has found the foregoing journey into the world of OpenType fonts and LuaTEX interesting, informative, and enticing. A larger world lies beyond, especially regarding non-Latin writing systems, not to mention maths, and I'll be satisfied if the reader now feels confident enough to step into that world. As with all of LuaTEX, it may seem intimidating at first but is ultimately extremely rewarding.

References

[1] Yannis Haralambous. *Fonts & Encodings.* O'Reilly, 2007. First edition in French as *Fontes & Codages*, O'Reilly, 2004.

[2] Taco Hoekwater. Math in LuaTEX 0.40. *MAPS*, 38, 2009. An updated version was translated in French as "LuaTEX 0.65 et les mathématiques" in *Cahiers Gutenberg*, 54–55, 2010.

[3] Paul Isambert. LuaTEX: What it takes to make a paragraph. *TUGboat* 32:1, 2011. `tug.org/TUGboat/tb32-1/tb100isambert.pdf`.

[4] LuaTEX team. *LuaTEX Reference.* `www.luatex.org/svn/trunk/manual/luatexref-t.pdf`.

[5] Microsoft OpenType specification. `www.microsoft.com/typography/otspec`.

[6] Linux Libertine. `www.linuxlibertine.org`.

[7] Ulrik Vieth. OpenType math illuminated. *TUGboat* 30:1, 2009. `tug.org/TUGboat/tb30-1/tb94vieth.pdf`.

[8] George Williams. FontForge documentation. `fontforge.sourceforge.net`.

◇ Paul Isambert
`zappathustra (at) free dot fr`

[55] Although cursive attachment is mostly used in a right-to-left writing system, the flag's name (inherited from OpenType files, not made up by LuaTEX or FontForge) assumes a left-to-right system, since it means "contrary to the writing direction".