SILE: A new typesetting system

Simon Cozens

Abstract

SILE is a new typesetting system, influenced by TeX but written from scratch in Lua. While still in the early stages of development, it holds potential as a typesetting system designed for unsupervised automated typesetting, especially in non-Latin scripts. SILE can be obtained from http://www.sile-typesetter.org/.

1 Introduction

In 2012, I wrote a typesetting system by mistake.

As part of my work for a small publishing company, I wrote a simple Perl script to automate the production of book covers. However, I soon discovered that the typesetting of the back cover blurb was unacceptable without proper justification. I ported Bram Stein's JavaScript version [8] of the original TeX justification algorithm [5] to Perl. Since there was already a Perl implementation [4] of TeX's hyphenation algorithm [6], I added support for hyphenation at the same time.

Now I had something which could reliably typeset paragraphs to PDF ... well, you can probably guess the rest. Adding a page builder was the obvious next step, and soon penalties, skips, glues and the rest followed. The project was rewritten in JavaScript, and then finally in Lua.

Why does the world need another typesetting system? Of course, it doesn't. But sometimes it's a good idea to reinvent the wheel; that's how we get better wheels. If we never reinvented wheels in the software industry, this journal would be called *troffboat*. And a friend who works in Bible typesetting let me know about a number of things that current automated typesetters can't do well — column balancing with multi-page lookbehind *and* grid typesetting; layout of parallel polyglots across page spreads; and so on — which gave me a number of goals.

Because of these goals and my own interest in non-Latin scripts, SILE has developed a focus on multilingual typesetting, particularly with complex and minority scripts, and the unsupervised layout of large, complex documents. SILE will see a 1.0.0 release when it is capable of taking a Unified Scripture XML [7] Bible translation and an appropriate class file, and producing a print-ready Bible of quality equivalent to that of a human typesetter. Even if I never achieve it, I'm having fun trying.

2 SILE's Component Parts

One of the advantages of writing a typesetting system in 2012 rather than in 1982 is that most of the hard work is already done for you. As we have mentioned, core typesetting algorithms are readily available; Unicode, together with its standard annexes and technical reports, describes good solutions to many of the problems of multilingual data representation; OpenType fonts and shaping engines help with the layout of complex scripts; embedded, interpreted languages won out over macro processors; and the world has effectively standardised on PDF as a document format.

A bird's eye view of SILE is shown in fig. 1. Text is consumed, and is reordered according to the Unicode Bidirectional Algorithm [9]. Then each run of text, together with its font, language, direction and other settings, is passed to the HarfBuzz [1] shaping engine. HarfBuzz returns a stream of glyph IDs and metrics, which are then assembled into a list of nodes, either by language-specific processors or by the default Unicode processor. The nodes are fed to the familiar H&J algorithms and collected into vboxes, vboxes into frames, frames into pages, and pages are finally output as PDF.

The choice of Lua as an implementation language hinged on a number of factors; obviously there are some benefits to using a language which is familiar to a pre-existing community of typesetting software engineers, although I have no strong desire to 'convert' anyone! But there are also benefits to using an interpreted language for implementation: first,

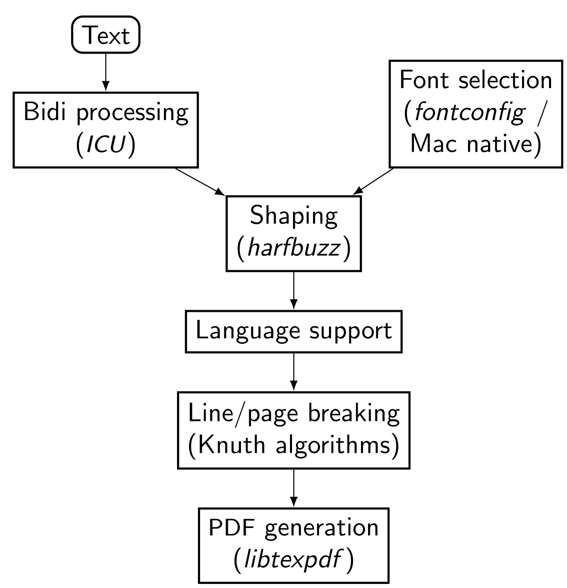


Figure 1: SILE's component parts

```
\SILE{}.registerCommand("tableofcontents:item", function (options, content)
  \SILE{}.settings.temporarily(function ()
    \SILE{}.settings.set("typesetter.parfillskip", \SILE{}.nodefactory.zeroGlue)
    \SILE{}.call("tableofcontents:level" .. options.level .. "item", {}, function ()
      \SILE{}.process(content)
      \SILE{}.call("dotfill")
      \SILE{}.typesetter:typeset(options.pageno)
    end)
  end)
end)
```

Figure 2: Lua code to typeset a TOC entry

Lua is designed as an embedded language, which means that SILE can provide a complex text layout system for embedding within other applications. (For instance, there is a SILE preview plugin for the Glyphs font editor.) It also means that any area of SILE's operation can be overridden or extended, not just those with pre-defined hooks. For instance, the grid typesetting package works by overriding the leading calculation; similarly, when setting Japanese text on a *hanmen* grid, there is no need to apply full best-fit paragraph composition — it's fine to replace the Knuth-Plass algorithm with a simple first-fit line breaker for speed.

SILE's modular design also means that everything is pluggable. I tried a number of different PDF libraries while developing SILE; the first versions used Cairo [2], but Cairo's PDF surface is fairly limited, and does not allow for the generation of PDF annotations, links and outlines — not to mention any of the tagged and structured markup required for accessible PDFs — so I started looking for alternatives: Haru, PoDoFo and others. Since the output system has a well-defined interface, I could easily test a new PDF generation library by slotting a new output implementation in place. Similarly, SILE's regression test system works by plugging in a custom outputter which produces a textual representation instead of a PDF; this can then be compared against the expected results using *diff*.

Incidentally, the PDF library I settled on was both an old one and a new one: I extracted the PDF generation backend from *dvipdfmx* and made it available as a library–*libtexpdf*. This was the only PDF generation system I could find which allowed me to address glyphs by ID, and also to add arbitrary PDF operators to the output.

Apart from C interfaces to HarfBuzz, ICU (the Unicode support library), *fontconfig* and *libtexpdf*, the core of SILE comprises a little under 5,000 lines of Lua code. (10% of which is made up of a somewhat literal port of TeX's line breaking algorithm.) This makes sense — with so much done by third-party libraries, there is relatively little left for SILE to do by itself.

3 Input formats, packages and classes

Just like the output system, SILE's input system is modular. The first input format implemented for SILE was XML — the idea being that SILE is to be used to typeset data produced by other software, such as translation databases, rather than documents constructed by hand; XML is both an easy format to parse and an easy format for other software to output. But while SILE needs to ingest XML, for whatever reason people wanted to hand-generate SILE documents, and so SILE added a parser for a simple, TeX-like input format.

The TeX-like format is only superficially TeX-like. It is, essentially, simply another way of representing an XML tree structure. These two SILE documents are equivalent:

`\begin{foo}`	`<foo>`
`Text`	`  Text`
`\bar[this=that]`	`  <bar this="that"/>`
`\end{foo}`	`</foo>`

The implementation of `<foo>` and `<bar>` is, of course, up to the user. In this sense, SILE is similar to an XML stylesheet processor: alongside a document must come a set of processing expectations which define how the tags will be typeset. SILE's `\define` command provides an extremely restricted macro system for implementing simple tags, but you are deliberately forced to write anything more complex in Lua. (Maxim: Programming tasks should be done in programming languages!) For example, the command to typeset a table-of-contents item is implemented by the code in fig. 2. This expects a command of the form:

```
\tableofcontents:item[level=2,pageno=3]
  {Something}
```

and passes the text and page number separated by leaders to the command which styles a level 2 TOC entry; this command, which is more easily implemented with a `\define` at the SILE level, will in turn set the appropriate font size, style and so on.

Lua code is loaded into SILE as packages or classes, similar to LaTeX — classes define the layout and key formatting expectations for tags, while packages provide additional functionality. Classes can be inherited (in the object-oriented programming sense) from other classes; SILE comes with a number of basic document classes but the expectation would be that each substantial document project would define its own class.

Since classes can be loaded even before the document is opened, they can do things such as providing a new input format. The `markdown` class does just this, implementing a parser and providing processing expectations for Markdown documents.

Naturally there are not currently anything like as many packages for SILE as for TeX derivatives. But fig. 3 is (an abridged version of) my favourite. This implements boustrophedon text by overriding the typesetter's function for turning horizontal lists into vertical lists. After the default implementation, the vertical list is inspected, and a custom whatsit (`swap`) is inserted after every vbox. When the whatsit is output, the typesetter's direction is reversed: if the previous line was left-to-right, the next line will be right-to-left, and *vice versa*.

SILE's programmability leads itself to experimentation and implementation of new technologies; support for OpenType color fonts was added as an external package in 85 lines of code, and rudimentary support for OT fonts with SVG outlines has recently been added.

```
local swap = \SILE{}.nodefactory.newVbox({})
swap.outputYourself = function(self,typesetter)
  typesetter.frame.direction =
    typesetter.frame.direction == "LTR-TTB"
      and "RTL-TTB" or "LTR-TTB"
  typesetter.frame:newLine()
end

\SILE{}.typesetter.boxUpNodes = function(self)
  local vboxlist =
    \SILE{}.defaultTypesetter.boxUpNodes(self)
  local nl = {}
  for i=1,#vboxlist do
    nl[#nl+1] = vboxlist[i]
    if nl[#nl]:isVbox() then
      nl[#nl+1] = swap
    end
  end
  return nl
end
```

Figure 3: The `boustrophedon` package, abridged.

4 The language support system

While Harfbuzz and Unicode provides a lot of what SILE needs to support complex scripts, different languages have different typographic conventions. For instance, correctly typesetting Japanese is not a matter of inserting line break opportunities between every pair of characters; Japanese *kinsoku-shori* rules stipulate that some punctuation characters cannot start lines and others cannot end lines. Additionally, characters are generally set on a fixed grid, but spacing is reduced around brackets and commas. These language-specific typesetting conventions are encoded in SILE's language support system, which assembles the stream of glyphs from the shaper into nodes, giving SILE a chance to implement hyphenation points, line breaking opportunities and so on.

Another pertinent example is that of many southeast Asian languages which are written without interword spaces but which line break between graphical syllable clusters, the clusters being determined by morphological analysis. SILE's support for Javanese uses a Parsing Expression Grammar [3] to detect syllable boundaries and insert penalties into the node stream to specify potential break points. Access to ICU means that language-specific casing rules (such as the Turkish i/İ and ı/I combinations) are correctly applied.

SILE does not assume any default directionality, meaning that left-to-right typesetting is not privileged over right-to-left processing. Indeed, supporting Mongolian, which is traditionally written top-to-bottom and left-to-right, is simply a matter of telling the typesetter about the new direction: `\thisframedirection{TTB-LTR}`.

Figure 4 demonstrates SILE's multi-script capabilities; notice how SILE has respected the typographic conventions of each script, and how the RTL texts (Arabic and Hebrew) have been reordered according to the conventions of mixed directionality typesetting. In the source file, each text is marked up with its language so that SILE can select the appropriate set of rules, but the bidi reordering is performed by default and requires no additional markup.

5 Frames

In our overview of SILE's component parts, we mentioned in passing that vboxes are assembled into frames and frames are assembled into pages. Frames

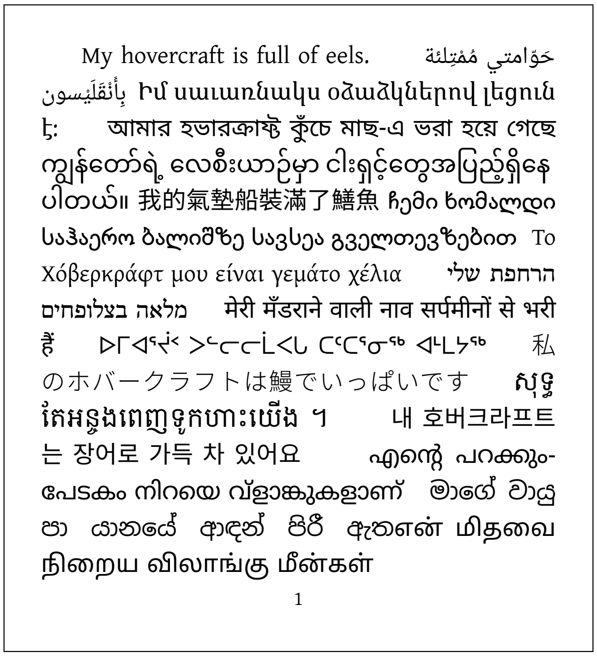


Figure 4: SILE's multi-script capabilities

are areas on the page where text is to be set. The frames for a page are generally defined by the document's class, but can be modified dynamically. For instance, footnotes are generally implemented by placing a zero-height frame at the bottom of the main content area. As footnotes are placed on a page, the footnote frame expands and the content frame shrinks. This allows for interesting layout possibilities: in a two column layout, the footnotes can be placed at the end of the second frame, or across the bottom of two frames, or in a separate area of the page altogether.

Frame layouts are generally specified relatively rather than absolutely; for instance, fig. 5 shows the frame declarations of the standard book class. The dimensions `%pw` and `%ph` refer to percentage of page width and page height respectively.

6 Various neat hacks

SILE packages implement a number of interesting ideas; in no particular order: best-fit page breaking; access to OpenType font features; parallel polyglot layout (see fig. 6); justification alternates (rewriting the text to improve justification); font fallback on missing glyphs; grid typesetting; Japanese vertical typesetting and ruby (*furigana*) support; automatic generation of font specimens; a BibTeX-like bibliography manager implemented at runtime within the typesetter; support for producing Structured and Tagged PDFs.

7 Challenges remaining

While SILE has been used successfully to produce a number of print books and articles, it is still not a finished product. The development team is very small; I'm the primary developer, with Caleb Maclennan and Khaled Hosny as notable contributors. Development happens on github and we are very open to issues and pull requests. However, there are a number of challenges remaining.

The lack of a Windows distribution is currently hampering adoption; the pushback routine (which disgorges the vertical list back onto the horizontal list after every frame break, in case the next frame has a different width) is a perennial source of mind-boggling bugs; and currently we lack a good solution for moving between multi-column and single-column layout and back again, which is a blocker for serious Bible typesetting.

We are also still working out how best to use SILE. For instance, I initially anticipated that the frames feature could be used to solve all kinds of layout challenges — drop caps, wrapping text around floated figures and so on. After some investigation we have found that frames are better suited to page-level layout, and other solutions such as nested vboxes and custom packages work better for altering layout within a frame. As more packages are developed, idiomatic use of the system will become more clearly defined.

I am often asked about typesetting of mathematics. My usual glib answer is that there is not very much mathematics in the Bible! However, I would like to see an implementation of math typesetting for SILE. The current plan is to find a way to call out to MathJax to perform the layout computations, and have SILE lay out the resulting nodes. However, SILE's development is essentially driven by user requests; I don't need math for the kind of books I'm typesetting, but if this is something you need, I would be glad to help you implement it!

8 Conclusion

I have deliberately avoided making comparisons between SILE and TeX derivatives in this article, attempting to introduce SILE on its own merits. In a sense there is no comparison. TeX is an incredibly mature and stable code base with a large and vibrant community; SILE is new, fast-moving and buggy, with few active developers. But I believe that, with time and development, SILE has the potential to provide better output than TeX for complex automated layout of non-Latin documents. It's also really fun to mess about with!

Simon Cozens

```
content = {
  left = "8.3%pw", right = "86%pw",
  top = "11.6%ph", bottom = "top(footnotes)"
},
runningHead = {
  left = "left(content)", right = "right(content)",
  top = "top(content)-8%ph", bottom = "top(content)-3%ph"
},
footnotes = {
  left = "left(content)", right = "right(content)",
  height = "0", bottom = "83.3%ph"
},
folio = {
  left = "left(content)", right = "right(content)",
  top = "bottom(footnotes)+3%ph",
  bottom = "bottom(footnotes)+5%ph"
}
```

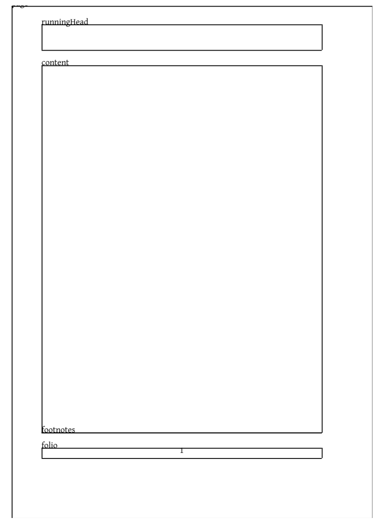
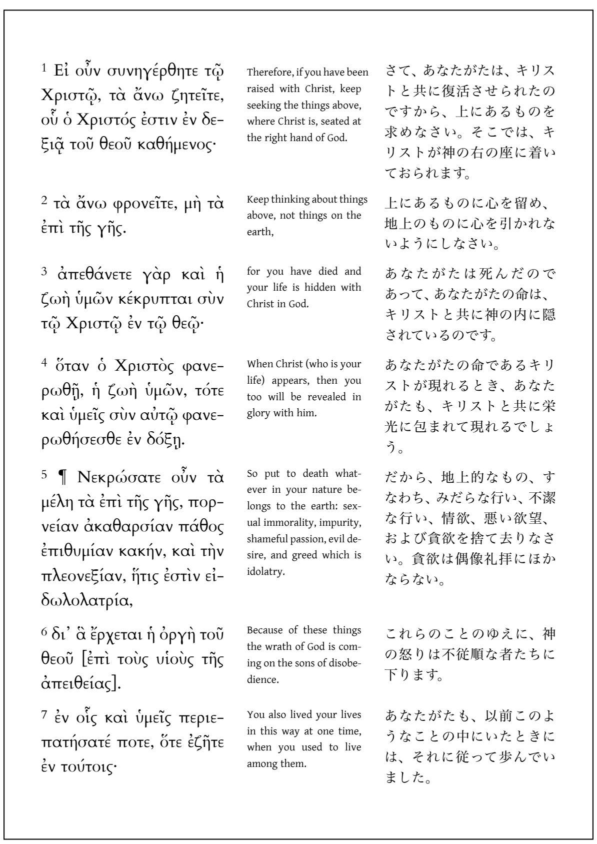


Figure 5: Frame layout in the standard book class



1 Εἰ οὖν συνηγέρθητε τῷ Χριστῷ, τὰ ἄνω ζητεῖτε, οὗ ὁ Χριστός ἐστιν ἐν δεξιᾷ τοῦ θεοῦ καθήμενος·	Therefore, if you have been raised with Christ, keep seeking the things above, where Christ is, seated at the right hand of God.	さて、あなたがたは、キリストと共に復活させられたのですから、上にあるものを求めなさい。そこでは、キリストが神の右の座に着いておられます。
2 τὰ ἄνω φρονεῖτε, μὴ τὰ ἐπὶ τῆς γῆς.	Keep thinking about things above, not things on the earth,	上にあるものに心を留め、地上のものに心を引かれないようにしなさい。
3 ἀπεθάνετε γὰρ καὶ ἡ ζωὴ ὑμῶν κέκρυπται σὺν τῷ Χριστῷ ἐν τῷ θεῷ·	for you have died and your life is hidden with Christ in God.	あなたがたは死んだのであって、あなたがたの命は、キリストと共に神の内に隠されているのです。
4 ὅταν ὁ Χριστὸς φανερωθῇ, ἡ ζωὴ ὑμῶν, τότε καὶ ὑμεῖς σὺν αὐτῷ φανερωθήσεσθε ἐν δόξῃ.	When Christ (who is your life) appears, then you too will be revealed in glory with him.	あなたがたの命であるキリストが現れるとき、あなたがたも、キリストと共に栄光に包まれて現れるでしょう。
5 ¶ Νεκρώσατε οὖν τὰ μέλη τὰ ἐπὶ τῆς γῆς, πορνείαν ἀκαθαρσίαν πάθος ἐπιθυμίαν κακήν, καὶ τὴν πλεονεξίαν, ἥτις ἐστὶν εἰδωλολατρία,	So put to death whatever in your nature belongs to the earth: sexual immorality, impurity, shameful passion, evil desire, and greed which is idolatry.	だから、地上的なもの、すなわち、みだらな行い、不潔な行い、情欲、悪い欲望、および貪欲を捨て去りなさい。貪欲は偶像礼拝にほかならない。
6 δι᾽ ἃ ἔρχεται ἡ ὀργὴ τοῦ θεοῦ [ἐπὶ τοὺς υἱοὺς τῆς ἀπειθείας].	Because of these things the wrath of God is coming on the sons of disobedience.	これらのことのゆえに、神の怒りは不従順な者たちに下ります。
7 ἐν οἷς καὶ ὑμεῖς περιεπατήσατέ ποτε, ὅτε ἐζῆτε ἐν τούτοις·	You also lived your lives in this way at one time, when you used to live among them.	あなたがたも、以前このようなことの中にいたときには、それに従って歩んでいました。

Figure 6: A parallel triglot: different scripts, different column widths, different font sizes

References

[1] Behdad Esfahbod. HarfBuzz. `http://harfbuzz.org/`, June 2016.

[2] Bryce Harrington. PDF surfaces. `https://www.cairographics.org/manual/cairo-PDF-Surfaces.html`, April 2016.

[3] Roberto Ierusalimschy. Lua parsing expression grammars. `http://www.inf.puc-rio.br/~roberto/lpeg/`, September 2015.

[4] Alex Kapranoff. `Text::Hyphen`. `http://search.cpan.org/perldoc?Text::Hyphen`, October 2015.

[5] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. *Software — Practice and Experience*, 11(11):1119–1184, 1981.

[6] Franklin Mark Liang. *Word Hy-phen-a-tion by Com-put-er*. PhD thesis, Department of Computer Science, Stanford University, 1983. `http://tug.org/docs/liang/`.

[7] United Bible Societies. Unified Scripture XML. `https://ubsicap.github.io/usx/`, 2016.

[8] Bram Stein. TEX line breaking algorithm in JavaScript. `https://github.com/bramstein/typeset`, April 2016.

[9] The Unicode Consortium. UAX #9: Unicode bidirectional algorithm. `http://unicode.org/reports/tr9/`, May 2016.

◇ Simon Cozens
Worldview Center for Intercultural Studies
St Leonards, Tasmania
Australia
simon (at) simon-cozens dot org
http://www.simon-cozens.org