The
SILE
Book

for SILE version v0.13.0

Simon Cozens

Table of Contents

Chapter 1
What is SILE?

SILE is a typesetting system. Its job is to produce beautiful printed documents from raw content. The best way to understand what SILE is and what it does is to compare it to other systems which you may have heard of.

1.1 SILE versus Word

When most people produce printed documents using a computer, they usually use desktop oriented word processing software such as Microsoft Word, iWork Pages, or LibreOffice Writer. SILE is not a word processor; it is a typesetting system. There are several important differences.

The job of a word processor is to produce a document that looks exactly like what you type on the screen. By contrast, the job of a typesetting system is to take raw content and produce a document that looks as good as possible. The input for SILE is a text document that includes instructions about how the content should be laid out on a page. In order to obtain the typeset result, the input file must be *processed* to render the desired output.

Word processors often describe themselves as WYSIWYG: What You See Is What You Get. SILE is cheerfully *not* WYSIWYG. In fact, you don't see what you get until you get it. Rather, SILE documents are prepared initially in a *text editor*—a piece of software which focuses on the text itself and not what it looks like—and then run through SILE in order to produce a PDF document.

For instance, most word processors are built roughly around the concept of a page with a central content area into which you type and style your content. The overall page layout is controlled by the page size and margins and more fine tuning is done by styling the content itself. You typically type continuously and when you hit the right margin, your cursor will automatically jump to the next line. In this way, the user interface shows you where the lines on the printed page will break.

In SILE the overall page layout is defined with a paper size and a series of one or more content frames. These frame descriptions provide the containers where content will later be typeset, including information about how it might flow from one frame to the next. Content is written separately, and SILE works out automatically how it best flows from frame to frame and from page to page. So when you are preparing content for SILE, you don't know where the lines will break until after it has been processed. You may use your text editor to type and type and type as long a line as you like, and when SILE comes to process your instructions, it will consider your input (up to) three times over in order to work out how to best to break the lines to form a paragraph. For example, if after one pass it finds that it has ended two successive lines with a hyphenated word, it will go back and try again and see if it can find better layout.

The same idea applies to page breaks. When you type into a word processor, at some point you will spill over onto a new page. When preparing content for SILE, you keep typing, because the page breaks are determined after considering the layout of the whole document.

In other words, SILE is a *language* for describing what you want to happen, and an interpreter that will make certain formatting decisions about the best way for those instructions to be turned into print.

1.2 SILE versus TeX

"Ah," some people will say, "that sounds very much like TeX!"[1] If you don't know much about TeX or don't care, you can probably skip this section.

And it's true. SILE owes an awful lot of its heritage to TeX. It would be terribly immodest to claim that a little project like SILE was a worthy successor to the ancient and venerable creation of the Professor of the Art of Computer Programming, but... really, SILE is basically a modern rewrite of TeX.

TeX was one of the earliest typesetting systems, and had to make a lot of design decisions somewhat in a vacuum. Some of those design decisions have stood the test of time—TeX is still an extremely well-used typesetting system more than thirty years after its inception, which is a testament to its design and performance—but many others have not. In fact, most of the development of TeX since Knuth's era has involved removing his early decisions and replacing them with technologies which have become the industry standard: we use TrueType fonts, not METAFONTs (`xetex`); PDFs, not DVIs (`pstex`, `pdftex`); Unicode, not 7-bit ASCII (`xetex` again); markup languages and embedded programming languages, not macro languages (`xmltex`, `luatex`). At this point, the parts of TeX that people actually *use* are 1) the box-and-glue model, 2) the hyphenation algorithm, and 3) the line-breaking algorithm.

SILE follows exactly in TeX's footsteps for each of these three areas that have stood the test of time; it contains a slavish port of the TeX line-breaking algorithm which has been tested to produce exactly the same output as TeX given equivalent input. But as SILE is itself written in an interpreted language,[2] it is very easy to extend or alter the behaviour of the SILE typesetter.

For instance, one of the things that TeX can't do particularly well is typesetting on a grid. This is something of a must-have feature for anyone typesetting bibles. Typesetting on a grid means that each line of text will line up between the front and back of each piece of paper producing much less visual bleed-through when printed on thin paper. This is virtually impossible to accomplish in TeX. There are various hacks to try to make it happen, but they're all horrible. In SILE, you can alter the behaviour of the typesetter and write a very short add-on package to enable grid typesetting.

---

1. Except that, being TeX users, they will say "Ah, that sounds very much like TeX!"
2. And if the phrase `TeX capacity exceeded` is familiar to you, you should already be getting excited.

Of course, nobody uses plain TeX—they all use LaTeX equivalents plus a huge repository of packages available from the CTAN. SILE does not benefit from the large ecosystem and community that has grown up around TeX; in that sense, TeX will remain streets ahead of SILE for some time to come. But in terms of *core capabilities*, SILE is already certainly equivalent to, if not somewhat more advanced than, TeX.

1.3 SILE versus InDesign

The other tool that people reach for when designing printed material on a computer is InDesign (or similar Desktop Publishing software, such as Scribus).
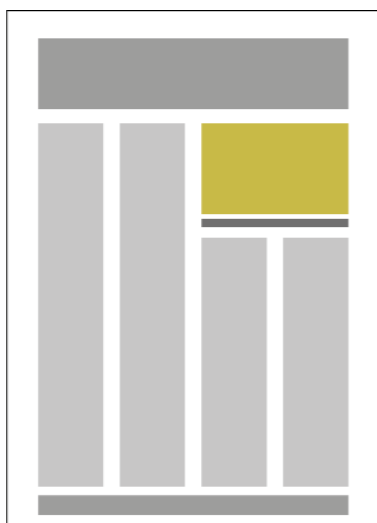
DTP software is similar to Word Processing Software in that they are both graphical and largely WYSIWYG, but the paradigm is different. The focus is usually less on preparing the content than on laying it out on the page--you click and drag to move areas of text and images around the screen.

InDesign is a complex, expensive, commercial publishing tool. SILE is a free, open source typesetting tool which is entirely text-based; you enter commands in a separate editing tool, save those commands into a file, and hand it to SILE for typesetting. And yet the two systems do have a number of common features.

In InDesign, text is flowed into *frames* on the page. On the left, you can see an example of what a fairly typical InDesign layout might look like. SILE also uses the concept of frames to determine where text should appear on the page, and so it's possible to use SILE to generate page layouts which are more flexible and more complex than that afforded by TeX.

Another thing which people use InDesign for is to turn structured data in XML format—catalogues, directories and the like—into print. The way you do this in InDesign is to declare what styling should apply to each XML element, and as the data is read in, InDesign formats the content according to the rules that you have declared.

You can do exactly the same thing in SILE, except you have a lot more control over how the XML elements get styled, because you can run any SILE command you like for a given element, including calling out to Lua code to style a piece of XML. Since SILE is a command-line filter, armed with appropriate styling instructions you can go from an XML file to a PDF in one shot. Which is quite nice.

In the final chapters of this book, we'll look at some extended examples of creating a *class file* for styling a complex XML document into a PDF with SILE.

1.4 Conclusion

SILE[3] takes some textual instructions and turns them into PDF output. It has features inspired by TeX and InDesign, but seeks to be more flexible, extensible and programmable than either of them. It's useful both for typesetting documents (such as this very documentation) written in the SILE language, and as a processing system for styling and outputting structured data.

3.   In case you're wondering, the author pronounces it /saɪəl/, to rhyme with "trial".

Chapter 2
Getting Started

Now that we understand some of what SILE is about and what it seeks to do, let's dive into SILE itself.

2.1 A Basic SILE Document

Before we show you how to use SILE, let's have a look at an example of what SILE documents look like. This is the input that we're going to feed to SILE, which it is going to process and turn into a PDF file. Each of the examples can be saved as a text file and then use them as input to SILE.

These documents are plain text files; when you create your own SILE files, you will need to create them in a plain text editor. Trying to create these files in a word processor such as Word will not work as they will be saved with the word processor's formatting rather than in a plain text format. Examples of good text editors include cross platform editors such as Atom and Sublime Text which are GUI oriented or Vim and Emacs which are more keyboard command oriented. Depending on your operating system there are other good choices such as Notepad++ on Windows or TextMate on OS X. You can also get started more simply with one of the basic editors built into your desktop environment such as Notepad on Windows, TextEdit on OS X, Gedit on Gnome, Kate on KDE, etc. For comparisons of text editors see `http://alternativeto.net/tag/text-editor/` *and select your platform.*

To begin with, here's the most basic SILE file of all:

```
\begin{document}
Hello SILE!
\end{document}
```

We'll pick apart this document in the next chapter, but for now take it on trust that this is what a SILE document looks like.

At the risk of belabouring the obvious, this is going to produce an A4-sized PDF document, with the text Hello SILE at the top left, and the page number (1) centered at the bottom. How are we going to get to that PDF?

2.2 Installing

First of all, we need to get ahold of SILE and get it running on our computer. Ready to use packages are available for macOS and many Linux distributions. Details for those we know about are listed in the sections below. Many other Linux distros without native packages can also take advantage of either Linuxbrew or Nix packaging. For all other systems you will need to follow the steps to download and compile the source yourself. Alternatively Docker containers are also available for use on any system that can run them.

2.2.1 macOS

For macOS the recommended way to install SILE is through the Homebrew package manager. Once you have Homebrew running (see `http://brew.sh`), installing SILE is as easy as running:

```
$ brew install sile
```

The formula also has an option that can compile SILE from the current Git HEAD version. To test the latest unreleased code you can install using:

```
$ brew install sile --HEAD
```

The `brew` package manager is also available as Linuxbrew for many Linux distributions. As an alternative, the `nix` package manager is also available for macOS, see below.

2.2.2 Arch Linux

Arch Linux (and derivatives such as Manjaro, Parabola, and others) have prebuilt packages in the official package repository:

```
$ pacman -S sile
```

The official package builds use Lua 5.4. Alternatively, a package that uses LuaJIT may be built manually from the Arch User Repository using `sile-luajit`. See the note about LuaJIT in the section *Installing From Source* for applicable caveats. A VCS package is also available as `sile-git` to build from the latest Git commit. These may be built and installed like any other AUR[1] package.

2.2.3 Ubuntu

For Ubuntu an official PPA is available with precompiled packages for all currently supported releases since 18.04 (currently Bionic, Focal, Hirsute):

```
sudo add-apt-repository ppa:sile-typesetter/sile
sudo apt-get update
```

---

1.   `https://wiki.archlinux.org/title/Arch_User_Repository`

```
sudo apt-get install sile
```

2.2.4 NetBSD

For NetBSD, package sources are available in `print/sile`. Use the usual command `bmake install` to build and install. A binary package can be installed using `pkgin`

```
pkgin install sile
```

2.2.5 NixOS

Packages are available for NixOS:

```
$ nix-env -i sile
```

The Nix package manager is also available on many other Linux distributions, as well as for macOS.

If you have flakes support[2] enabled, you can run sile with:

```
nix run github:sile-typesetter/sile#sile -- <sile arguments>
```

2.2.6 Void Linux

Void Linux packages are available in the default package manager.

```
$ xbps-install sile
```

2.2.7 Running via Docker

Another way of getting SILE up and running in a pre-compiled state is to use prebuilt Docker containers. If your system has Docker installed already, you can run SILE simply by issuing a run command. The first time it is used Docker will fetch the necessary layers and assemble the image for you. Thereafter only a small amount of CPU time and memory overhead goes into running the container compared to a regular system install.

The catch is that because SILE is running *inside* the container, in order to do anything useful with it you must first pass your sources (including things like fonts) in and give it a way to write files back out. The easiest way to do that is by mounting your entire project inside the container. This makes the actual invocation command quite a mouthful. For most shells, a single alias can be created to hide that complexity and make it pretty simple to run:

```
$ alias sile='docker run -it --volume "$(pwd):/data" --user "$(id -u):$(id
-g)" siletypesetter/sile:latest'
```

---

2.   https://nixos.wiki/wiki/Flakes#Installing_flakes

Docker images are tagged to match releases (e.g. `v0.10.0`). Additionally the latest release will be tagged `latest`, and a `master` tag is also available with the freshest development build. You can substitute `latest` in the alias above to run a specific version.

One notable issue with using SILE from a Docker container is that it will not have access to your system's fonts by default. You can map a folder of fonts (any tree usable by fontconfig) into the container. This could be your system's default font directory, your user one, a project specific folder, or anything of your choosing. You can see where fonts are found on your system using `fc-list`. The path of your choosing from the host system should be passed as a volume mounted on `/fonts` inside the container like this:

```
$ docker run -it --volume "/usr/share/fonts:/fonts" --volume "$(pwd):/data" --
user "$(id -u):$(id -g)" siletypesetter/sile:latest
```

Armed with commands (or aliases) like these to take care of the actual invocation, you should be able to use all other aspects of SILE as documented in the rest of the manual. Just be aware when it comes to things like fonts, images, or other resources about where your files are relative to the container.

2.2.8 Installing From Source

Downloads of SILE can be obtained from the home page at `http://www.sile-typesetter.org/`.

SILE requires a number of other software packages to be installed on the computer before it can work—the Lua programming language, and the Harfbuzz text shaping library. SILE provides its own PDF creation library, which has its own requirements: `freetype`, `fontconfig`, `zlib` and `libpng`.

We suggest you use your distributions's package manager to install as many of the dependencies as possible. On DEB-based Linux machines such as Debian and Ubuntu, you should be able to install all of the needed dependencies by issuing the command:

```
$ apt-get install lua5.2 liblua5.2-dev lua-sec libharfbuzz-dev
libfreetype6-dev libfontconfig1-dev libpng-dev libicu-dev
```

Here's an incantation that will work on most Redhat-based Linux distros:

```
$ yum install harfbuzz-devel make automake libtool gcc
freetype-devel fontconfig-devel lua-devel lua-sec libng-gevel libicu-devel
```

If you are on another system you will have to come up with the correct equivalent packages yourself.

There are a large number of lua dependencies required to run SILE. You may either install them to your system using your system's package manager or `luarocks`, or let the SILE build process fetch and bundle them for you (this is the default unless you specify otherwise). You cannot mix and match these two methods; either the system path has to have all the dependencies, or all of them will be bundled with SILE.

If you choose to install the lua dependencies to your system, you may use any combination of your system's packages and installing them via `luarocks install`. The easiest way is to let Luarocks figure it out based on the included Rockspec:

```
$ luarocks install --deps-only sile-dev-1.rockspec
```

Note that the `luasec` package requires OpenSSL libraries on your system in order to compile. On some systems such as macOS you may need to configure the location of the header files manually it install it:

```
$ luarocks install luasec OPENSSL_DIR=...
```

Once you have these requirements in place, you should then be able to unpack the file that you downloaded from SILE's home page, change to that directory,[3] and configure the build envirenment.

If you supplied all the Lua dependencies yourself, run:

```
$ ./configure --with-system-luarocks
```

Otherwise to go with default of bundling them, just run:

```
$ ./configure
```

SILE can be built to run using LuaJIT under which it runs almost exactly twice as fast! Because SILE lets your documents, classes, and packages use Lua code from your system –and because the ecosystem for Lua software is more developed around the regular Lua versions– we default to using the newest plain Lua installation found. If your system has LuaJIT and you prefer to use it you can ask the configure process to use it like this:

```
$ ./configure --with-luajit
```

Keep in mind that any Lua code you write will need to be JIT compatible. The vast majority of Lua code will be fine, but there are a few limitations. See the documentation for your version of LuaJIT for details.

If that comand was successful, you can build SILE itself:

3. If you downloaded a copy of the SILE source by cloning the git repository rather than downloading one of the release packages, you will also need to run `./bootstrap.sh` to setup the configure script at this point before continuing to the next step.

```
$ make
```

Most users of SILE will want to install the `sile` command and SILE's library files onto their system; this can be done with:

```
$ make install
```

Now the `sile` command is available from any directory.

If you wish you can skip the install step and use the compiled SILE executable diretly from the source directory. As configured above this will only work from a shell with the CWD set to the SILE source. To make it usable from anywhere, you can configure it with the source directory baked in as the installation location.

```
$ ./configure --datarootdir=$(cd ..;pwd)
$ make
```

Now to run SILE from anywhere you just need to supply the full path to the source directory.

```
$ /full/path/to/sile/sile
```

2.2.9 Notes for Windows users

SILE may be built on Windows using CMake and Visual Studio. There is not an installer (yet), but Windows executables are generated using Azure for every commit. You may download these executables by selecting the latest build from `https://simoncozens-github.visualstudio.com/sile/_build` and downloading the "sile" artifact from the Artifacts drop down.

2.3 Running SILE

Let's move to a new directory, and in a text editor, create the file `hello.sil`. Copy in the content above and save the file. Now at your command line run:

```
$ sile hello.sil
```

Once again, this should produce an output file `hello.pdf`. Congratulations—you have just typeset your first document with SILE!

SILE output filenames are generated by filing off the extension from the first input filename and adding .pdf. If you want to write to a different filename altogether, use the -o file.pdf command line option to SILE. You can use -o - to write the PDF to standard output, if you wish to use SILE as part of a pipeline.

2.4 Let's Do Something Cool

In `https://sile-typesetter.org/examples/docbook.xml`, you will find a typical DocBook 5.0 article. Normally turning DocBook to print involves a curious dance of XSLT processors, format object processors and/or strange LaTeX packages. But SILE can read XML files and it also comes with a `docbook` class, which tells SILE how to render (admittedly, a subset of) the DocBook tags onto a page.

Turning `dockbook.xml` into `docbook.pdf` is now as simple as:

```
$ ./sile -I docbook.sil docbook.xml
SILE v0.13.0 (Lua 5.3)
Loading docbook
<classes/docbook.sil><docbook.xml>[1] [2] [3]
```

The `-I` flag adds a *preamble*, which is to say that it loads up a *class* before reading the input file; the `docbook` preamble provides processing expectations for DocBook tags, which means that once this is loaded, DocBook files can be processed directly as SILE documents.

In Chapter 9, we'll look at how the `docbook` class works, and how you can define processing expectations for other XML formats.

2.5 Running SILE remotely as a CI job

It may be useful for some work flows to run SILE remotely on a CI server as part of a job that renders documents automatically from sources. This comes with the caveats mentioned in the section *Running via Docker* above, but if you plan ahead and arrange your projects right it can be quite useful.

There are actually many ways to run SILE remotely as part of a CI work flow. Because packages are available for many platforms, one way would be to just use your platforms native package installation system to pull them into whatever CI runner environment you already use. Another way is to pull in the prebuilt Docker container and run that.

As a case study, here is how a work flow could be setup in GitHub Actions:

```
name: SILE
on: [push, pull_request]
jobs:
  sile:
    runs-on: ubuntu-20.04
```

```
name: SILE
steps:
  - name: Checkout
    uses: actions/checkout@v2
  - name: SILE
    id: sile
    uses: docker://siletypesetter/sile:latest
    with:
      args: my-document.sil
```

Add to your repository as `.github/workflows/sile.yaml`. This work flow assumes your project has a source file `my-document.sil` and will leave behind a `my-document.pdf`. Note that this Actions work flow explicitly uses a container fetched from Docker Hub because this is the fastest way to get rolling, and the comments in the section about Docker regarding tagged versions besides `latest` apply equally here.

Because this repository is itself a GitHub Action you can also use the standard `uses` syntax like this:

```
uses: sile-typesetter/sile@latest
```

But be warned that since GitHub rebuilds containers from scratch on every such invocation, this syntax is not recommended for regular use. Pulling the prebuilt Docker images is recommended instead.

With these ideas in mind, other CI systems should be easy to support as well.

Chapter 3
SILE's Input Language

Let's now go back and reconsider the first SILE file we saw:

```
\begin{document}
Hello SILE!
\end{document}
```

3.1 Defining the paper size

A document starts with a `\begin{document}` command and ends with `\end{document}`. In between, SILE documents are made up of two elements: text to be typeset on the page, such as "Hello SILE!" in our example, and commands.

The default paper size is A4, although each class may override this value. To manually change the paper size, an optional argument may be added to the document declaration:

```
\begin[papersize=letter]{document}
```

SILE knows about the ISO standard A, B and C series paper sizes using names a4, b5, etc. Also available are the traditional sizes letter, note, legal, executive, halfletter, halfexecutive, statement, folio, quarto, ledger, tabloid, the series archA through archE, as well as DL, Comm10, Monarch, flsa, flse, csheet, dsheet, and esheet.

If you need a paper size for your document which is not one of the standards, then you can specify it by dimensions:

`papersize=<basic length> x <basic length>`.

SILE knows a number of ways of specifying lengths. A `<basic length>` as mentioned above can be specified as a floating-point number followed by a dimension abbreviation. Acceptable dimensions are printer's points (`pt`), millimeters (`mm`), centimeters (`cm`) or inches. (`in`) (You can even use `ft` and `m` for feet and meters if you are typesetting *particularly* large documents.) For instance, a standard B-format book can be specified by `papersize=198mm x 129mm`.

Once some of the basic document properties have been setup using these fixed size units, other dimensions may be specified relative to them, using *relative units*. In chapter 4 we will meet some of

these relative units, and in chapter 7 we will meet some other ways of specifying lengths to make them stretchable or shrinkable.

3.2 Ordinary text

On the whole, ordinary text isn't particularly interesting—it's just typeset.

TeX users may have an expectation that SILE will do certain things with ordinary text as well. For instance, if you place two straight-backquotes into a TeX document (like this: ``) then TeX will magically turn that into a double opening quote ("). SILE won't do this. If you want a double opening quote, you have to ask for one. Similarly, en- and em-dashes have to be input as actual Unicode characters for en- and em-dashes, rather than the pseudo-ligatures such as -- or --- that TeX later transforms to the Unicode characters.

There are only a few bits of cleverness that happen around ordinary text.

The first is that space is not particularly significant. If you write `Hello    SILE!` with three spaces, you get the same output as if you write `Hello SILE!` with just one. Space at the beginning of a line will be ignored.

Similarly, you can place a line break anywhere you like in the input file, and it won't affect the output because SILE considers each paragraph at a time and computes the appropriate line breaks for the paragraph based on the width of the line available. In other words, if your input file says

---

```
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum.
```

...you might not necessarily get a line break after 'tempor'; you'll get a line break wherever is most appropriate. In the context of this document, you'll get:

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in

voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

In other words, a line break is converted to a space.

Sometimes this conversion is not what you want. If you don't want single line breaks to be converted to a space, use a comment character % *at the end of a line to suppress the additional whitespace.*

When you want to end a paragraph, you need to input two line breaks in a row, like this:

```
Paragraph one.

Paragraph two.
This is not paragraph three.

This is paragraph three.
```

The second clever thing that happens around ordinary text is that a few—four, in fact—characters have a special meaning to SILE. All of these will be familiar to TeX users.

We've seen that a *backslash* is used to start a command, and we'll look into commands in more detail soon. *Left and right curly braces* ({, }) are used for grouping, particularly in command arguments. Finally, a *percent sign* is used as a comment character, meaning that everything from the percent to the end of the line is ignored by SILE. If you want to actually typeset these characters, prepend a backslash to them: \\ produces '\', \{ produces '{', \} produces '}', and \% produces '%'.

The third clever thing is SILE will automatically hyphenate text at the end of a line if it feels this will make the paragraph shape look better. Text is hyphenated according to the current language options in place. By default, text is assumed to be in English unless SILE is told otherwise. In the Latin text above, we turned hyphenation off.

The final clever thing is that, where fonts declare ligatures (where two or more letters are merged into one in order to make them visually more attractive), SILE automatically applies the ligature. So if you type `affluent fishing` then, (depending on your font), your output might look like: 'affluent fishing.' If you specifically want to break up the ligatures, then insert empty groups (using the grouping characters { and }) in the middle of the possible ligatures: `af{}f{}luent f{}ishing`: affluent fishing. See the section on the `features` package for more information on how to control the display of ligatures and other font features.

3.3 Commands

Typically (and we'll unpack that statement later), SILE commands are made up of a backslash followed by a command name, and a document starts with a `\begin{document}` command and ends with `\end{document}`.

A command may also take two other optional components: some *parameters*, and an *argument*. The `\begin` command at the start of the document is an example of this.[1]

```
\begin{document}
```

The parameters to a command are enclosed in square brackets and take the form `key=value`; multiple parameters are separated by commas or semicolons, as in `[key1=value1,key2=value2,…]` Spaces around the keys are not significant; we could equally write that as `[key1 = value1; key2 = value2; …]`. If you need to include a comma or semicolon within the value to a parameter, you can enclose the value in quotes: `[key1 = "value1, still value 1", key2 = value2; …]`.

The optional argument (of which there can only be at most one) is enclosed in curly braces.[2] Because the argument is optional, there is a difference between this: `\command{args}` (which is interpreted as a command with argument `args`) and `\command {args}` (which is interpreted as a command with no arguments, followed by the word `args` in a new group).

Here are a few more examples of SILE commands:

```
\eject                          % A command with no parameters or argument

\font[family=Times,size=10pt]   % Parameters, but no argument

\chapter{Introducing SILE}      % Argument but no parameters
```

---

1.   Strictly speaking `\begin` isn't actually a command but we'll pretend that it is for now and get to the details in a moment.
2.   TeX users may forget this and try adding a command argument "bare", without the braces. This won't work; in SILE, the braces are always mandatory.

```
\font[family=Times,size=10pt]{Hi there!} % Parameters and argument
```

3.4 Environments

Commands like \chapter and \em (emphasises text by making it italic) are normally used to enclose a relatively small piece of text; a few lines at most. Where you want to enclose a larger piece of the document, you can use an *environment*; an environment begins with \begin{name} and encloses all the text up until the corresponding \end{name}. We've already seen an example, the document environment, which must enclose the *entire* document.

Here is a secret: there is absolutely no difference between a command and an environment. In other words, the following two forms are equivalent:

```
\font[family=Times,size=10pt]{Hi there!}

\begin[family=Times,size=10pt]{font}
Hi there!
\end{font}
```

However, in some cases the environment form of the command will be easier to read and will help you to be clearer on where the command begins and ends.

3.5 The XML Flavour

While we're on the subject of alternative forms, SILE can actually process its input in a completely different file format. What we've seen so far has been SILE's "TeX-like flavor", but if the first character of the input file is an angle bracket (<) then SILE will interpret its input as an XML file. (If it isn't well-formed XML, then SILE will get very upset.)

Any XML tags within the input file will then be regarded as SILE commands, and tag attributes are interpreted as command parameters; from then on, the two file formats are exactly equivalent, with one exception: instead of a <document> tag, SILE documents can be enclosed in *any* tag. (Although <sile> is conventional for SILE documents.)

In other words, the XML form of the above document would be:

```
<sile>
Hello SILE!
</sile>
```

Commands without an argument need to be well-formed self-closing XML tags (for instance, <break/>), and commands with parameters should have well-formed attributes. The example above, in XML flavor, would look like this:

```
<font family="Times" size="10pt">Hi there!</font>
```

We don't expect humans to write their documents in SILE's XML flavor—the TeX-like flavor is much better for that—but having an XML flavor allows for computers to deal with SILE a lot more easily. One could create graphical user interfaces to edit SILE documents, or convert other XML formats to SILE.

However, there is an even smarter way of processing XML with SILE. For this, you need to know that you can define your own SILE commands, which can range from very simple formatting to fundamentally changing the way that SILE operates. If you have a file in some particular XML format —let's say it's a DocBook file—and you define SILE commands for each possible DocBook tag, then the DocBook file becomes a valid SILE input file, as-is.

In the final two chapters, we'll provide some examples of defining SILE commands and processing XML documents.

Chapter 4
Some Useful SILE Commands

We're going to organise our tour of SILE by usage; we'll start by giving you the most useful commands that you'll need to get started typesetting documents using SILE, and then we'll gradually move into more and more obscure corners as the documentation progresses.

4.1 Fonts

The most basic command for altering the look of the text is the `\font` command; it takes two forms:

- `\font[parameters…]{argument}`

- `\font[parameters…]`

The first form sets the given argument text in the specified font; the second form changes the font used to typeset text from this point on.

So, for instance:

```
Small text
\font[size=15pt]Big text!
\font[size=30pt]{Bigger text}
Still big text!
```

---

produces

Small text

Big text!

Bigger text

Still big text!

As you can see, one possible attribute is `size`, which can be specified as a SILE `<dimension>`. A `<dimension>` is like a `<basic length>` (described above) but with a few extra possible dimensions. There are dimensions which are relative to the size of the *current* font: an em is the size of the font's

current em square–for a 12pt font, this would be 12 points; an en is half the em square; an *ex* is the height of the character 'x'; a *spc* is the width of the space character.

There are also dimensions which are defined as a percentage of the size of the current page width or height, the current frame width or height, and the line width (`%pw`, `%ph`, `%fw`, `%fh`, and `%lw` respectively). You can specify lengths in terms of the current paragraph skip (`ps`) and baseline skip (`bs`), which will make sense later on. Additional units are available relative to the largest or smallest value of either axis (`%pmax`, `%pmin`, `%fmax`, `%fmin`).

The full list of attributes to the `\font` command are:

• *size*: as above.

• *family*: the name of the font to be selected. SILE should know about all the fonts installed on your system, so that fonts can be specified by their name.

• *filename*: if a filename is supplied, SILE will use the font file provided rather than looking at your system's font library.

• *style*: can be `normal` or `italic`.

• *weight*: a CSS-style numeric weight ranging from 100, through 200, 300, 400, 500, **600, 700, 800** to **900**. Not all fonts will support all weights (many just have two), but SILE will choose the closest.

• *variant*: a font variant (`normal`, `smallcaps`)

• *language*: The two letter (ISO639-1) language code for the text. This will affect both font shaping and hyphenation.

• *direction*: The expected text direction. (`LTR-TTB` for left to right, top to bottom; `RTL-BTT` would set text right to left, bottom to top!)

• *script*: The script family in use. (See chapter 7, "Language", for more on these past three settings.)

It's quite fiddly to be always changing font specifications manually; later we'll see some ways to automate the process. SILE provides the `\em{…}` command as a shortcut for `\font[style=italic]{…}`. There is no shortcut for boldface, because boldface isn't good typographic practice and so we don't want to make it easy for you to make bad books.

4.2 Document Structure

SILE provides a number of different *class*es of document (similar to LaTeX classes). By default, you get the *plain* class, which has very little support for structured documents. There is also the *book* class, which adds support for right and left page masters, running headers, footnotes, and chapter, section and subsection headings.

To use the commands in this section, you will need to request the *book* class by specifying in your `\begin{document}` command '`[class=book]`'; for example, the document you are currently reading begins with the command `\begin[class=book]{document}`.

4.2.1 Chapters and Sections

If you choose the book class, you can divide your document into different sections using the commands \chapter{…}, \section{…} and \subsection{…}. The argument to each command is the name of the chapter or section respectively; chapters will be opened on a new right-hand page, and the chapter name will form the left running header. Additionally, the section name and number will form the right running header.

Chapters, sections and subsections will be automatically numbered starting from 1; to alter the numbering, see the documentation for the counters *package in the next chapter. To produce an unnumbered chapter, provide the parameter* [numbering=false]*.*

This subsection begins with the command \subsection{Chapters and Sections}.

4.2.2 Footnotes

Footnotes can be added to a book with the \footnote{…} command.[1] The argument to the command will be set as a footnote at the bottom of the page; footnotes are automatically numbered from 1 at the start of each chapter.

4.3 Indentation and Spacing

Paragraphs in SILE normally begin with an indentation (by default, 20 points in width). To turn this off, you can use the \noindent command at the start of a paragraph. (This current paragraph doesn't need to call \noindent because \section and \chapter automatically call it for the text following the heading.) A \noindent can be cancelled by following it with an \indent. You can completely turn off indentation for the whole of the document by changing its size to zero; we'll see how to do change the size of the indentation in the settings chapter, but the easiest way to set it to zero for the whole of the document (rather than for just one paragraph) is to issue the command \neverindent.

To increase the vertical space between paragraphs or other elements, the commands \smallskip, \medskip and \bigskip are available to add a 3pt, 6pt and 12pt gap respectively. There will be a \bigskip after this paragraph.

---

1.   Like this: \footnote{Like this.}

There are also some commands to increase the horizontal space in a line; from the smallest to the largest, `\thinspace` (1/6th of an em), `\enspace` (1 en), `\quad` (1 em), and `\qquad` (2em).

You can center a paragraph of text by wrapping it in the `center` environment. (`\begin{center}` …
`\end{center}`). This paragraph is centered on the page.

If you want to add a very long stretchy space, you can use the command `\hfill`. Doing this in conjunction with a line break will cause the line before the break to be flush left, like this.
The command `\cr` is a shortcut for `\hfill\break`.

4.4 Breaks

SILE automatically determines line and page breaks; in later chapters we will introduce some *settings* which can be used to tweak this process. However, SILE's plain class also provides some commands to help the process on its way.

Between paragraphs, the `\break` command requests a *frame break* at the given location. (The commands `\framebreak` and `\eject` are also available as synonyms.) Where there are multiple frames on a page—for instance, in a document with multiple columns—the current frame will be ended and typesetting will recommence at the top of the next frame. `\pagebreak` (also known as `\supereject`) is a more forceful variant, and ensures that a new page is opened even if there are remaining frames on the page. A less forceful variant is `\goodbreak`, which suggests to SILE that this is a good point to break a frame. The opposite is `\nobreak` which requests that, if at all possible, SILE does not break at the given point. A neutral variant is `\allowbreak`, which allows SILE to break at a point that it would otherwise not consider as suitable for breaking.

Within a paragraph, these commands have a different meaning. The `\break` command requests a *line* break at the given location, and, *mutatis mutandis*, so do `\goodbreak`, `\nobreak` and `\allowbreak`. If you want to be absolutely sure that you are inhibiting a *page* break, you can say `\novbreak`.

SILE normally fully-justifies text—that is, it tries to alter the spacing between words so that the text stretches across the full width of the column.[2] An alternative to full justification is ragged

---

2.   This does not mean that text will always exactly fill the width of the column. SILE will choose line breaks and alter the spacing between words up to a certain extent, but when it has done its best, it will typeset the least bad solution; this may involve some of the words jutting slightly out into the margin.

right margin formatting, where the spacing between words is constant but the right hand side of the paragraph may not line up. Ragged right is often used for children's books and for frames with narrow columns such as newspapers. To use ragged right formatting, enclose your text in a \begin {raggedright} environment. This paragraph is set ragged right.

Similarly, there is a raggedleft environment, in which the right-hand margin of the paragraph is fixed, but the left-hand margin is allowed to vary. This paragraph is set ragged left.

4.5 Including Other Files and Code

To make it easier for you to author a large document, you can break your SILE document up into multiple files. For instance, you may wish to put each chapter into a separate file; you may wish to develop a file of user-defined commands (see chapter 6) and keep this separate from the main body of the document. You will then need the ability to include one SILE file from another.

This ability is provided by the \include command. It takes one mandatory parameter, src=<path>, which represents the path to the file. So for instance, you may wish to write a thesis like this:

```
\begin[class=thesis]{document}
\include[src=macros.xml]
\include[src=chap1.sil]
\include[src=chap2.sil]
\include[src=chap3.sil]
…
\include[src=endmatter.sil]
\end{document}
```

\includes may be nested, in that file A can include file B which includes file C.

The contents of an included file should be put in a \sile tag (or <sile> if the file is in XML flavour), like so:

```
\begin{sile}
\chapter{A Scandal In Bohemia}
To Sherlock Holmes she is always {the woman}.
\end{sile}
```

This is because every file is required to contain a valid XML tree, which wouldn't be the case without a common root.

SILE is written in the Lua programming language, and the Lua interpreter is available at run-time. Just as one can run Javascript code from within a HTML document using a `<script>` tag, one can run Lua code from within a SILE document using a `\script` command. (It looks better in XML-flavor.) This command has two forms: `\script[src=<filename>]` which includes a Lua file, and `\script{…}` which runs Lua code inline.

Doing anything interesting inline requires knowledge of the internals of SILE, (thankfully the code is not that hard to read) but to get you started, the Lua function `SILE.typesetter:typeset(…)` will add text to a page, `SILE.call("…")` will call a SILE command, and `SILE.typesetter:leaveHmode()` ends the current paragraph and outputs the text. So, for example:

```
\begin{script}
  for i=1,10 do
    SILE.typesetter:typeset(i .. " x " .. i .. " = " .. i*i .. ". ")
    SILE.typesetter:leaveHmode()
    SILE.call("smallskip")
  end
\end{script}
```

produces the following output:

1 x 1 = 1.
2 x 2 = 4.
3 x 3 = 9.
4 x 4 = 16.
5 x 5 = 25.
6 x 6 = 36.
7 x 7 = 49.
8 x 8 = 64.
9 x 9 = 81.
10 x 10 = 100.

There is one notable caveat when embedding Lua code documents written with the TeX-flavor markup. Since SILE has to first parse the TeX markup to find the start and end of such script commands *without* understanding what's in between, it is strictly necessary that no end tags appear inside the Lua code. This means that for the environment block version (`\begin{script}`) there must

be no instances of \end{script} inside the Lua code block, even inside a string that would otherwise be valid Lua code, (e.g. if it was inside a quoted string or Lua comment). The first instance of such an end tag terminates the block, and there is currently no way to escape it. For the inline comand form (\script) an exact matching number of open and closing braces may appear in the Lua code — the next closing brace at the same level as the opening brace will close the tag, even if it happens to be inside some quoted string in the Lua code. Including any number of nested opening and closing braces is possible as long as they are balanced.

Chapter 5
SILE Packages

SILE comes with a number of packages which provide additional functionality. In fact, the actual "core" of SILE's functionality is very small and very extensible, with most of the interesting features being provided by add-on packages. SILE packages are written in the Lua programming language, and can define new commands, change the way that the SILE system operates, or indeed do anything that it's possible to do in Lua.

As mentioned above, loading a package is done through the `\script` command, which runs Lua code. By convention packages live in the `packages/` subdirectory of either your input file's location, your current working directory or SILE's installation directory. For instance, we'll soon be talking about the `grid` package, which normally can be found as `/usr/local/lib/sile/packages/grid.lua`. To load this, we'd say:

```
\script[src=packages/grid]
```

How SILE locates files

SILE searches for paths in a variety of directories: first, in the directory in which your input file is located, then the current wording directory; next, if the environment variable `SILE_PATH` *is set, it will look in that directory; then it will look in the standard installation directories* `/usr/lib/sile` *and* `/usr/local/lib/sile`. *Unlike TeX, it does not descend into subdirectories when looking for a file, and so if you have arranged your personal macro, class or package files into subdirectories, you will need to provide a full relative path to them.*

5.1 image

As well as processing text, SILE can also include images.

Loading the **image** package gives you the `\img` command, fashioned after the HTML equivalent. It takes the following parameters: `src=⟨file⟩` must be the path to an image file; you may also give `height` and/or `width` parameters to specify the output size of the image on the paper. If the size parameters are not given, then the image will be output at its 'natural' size, honoring its resolution if available.

With the libtexpdf backend (the default), the images can be in JPEG, PNG, EPS or PDF formats.

Here is a 200x243 pixel image output with `\img[src=documentation/gutenberg.png]`. The image has a claimed resolution of 100 pixels per inch, so ends up being 2 inches (144pt) wide on the page:

Here it is with (respectively) `\img[src=documentation/gutenberg.png, width=120pt]`, `\img[src=documentation/gutenberg.png, height=200pt]`, and `\img[src=documentation/gutenberg.png, height=200pt, width=120pt]`:

Notice that images are typeset on the baseline of a line of text, rather like a very big letter.

5.2 folio

The **folio** package (which is automatically loaded by the plain class, and therefore by nearly every SILE class) controls the output of folios—the old-time typesetter word for page numbers.

It provides four commands to users:

• \nofolios: turns page numbers off.

• \nofoliothispage: turns page numbers off for one page, then on again afterward.

• \folios: turns page numbers back on.

• \foliostyle: a command you can override to style the page numbers. By default, they are centered on the page.

If, for instance, you want to set page numbers in a different font you can redefine the command like so:

```
\define[command=foliostyle]{\center{\font[family=Albertus]{\process}}}
```

If you want to put page numbers on the left side of even pages and the right side of odd pages, there are a couple of ways you can do that. The complicated way is to define a command in Lua which inspects the page number and then sets the number ragged left or ragged right appropriately. The easy way is just to put your folio frame where you want it on the master page...

5.3 rules

The **rules** package draws lines. It provides three commands.

The first command is \hrule, which draws a blob of ink of a given width (length), height (above the current baseline) and depth (below the current baseline). Such rules are horizontal boxes, placed along the baseline of a line of text and treated just like other text to be output. So, they can appear in the middle of a paragraph, like this: ⸺ (that one was generated with \hrule[height=0.5pt, width=20pt].)

The second command provided by this package is \underline, which underlines its contents.

Underlining is horrible typographic practice, and you should <u>never</u> do it.

(That was produced with \underline{never}.)

Finally, \fullrule draws a thin line across the width of the current frame.

5.4 color

The **color** package allows you to temporarily change the color of the (virtual) ink that SILE uses to output text and rules. The package provides a `\color` command which takes one parameter, `color=`⟨color specification⟩, and typesets its argument in that color. The color specification is the same as HTML: it can be a RGB color value in #xxx or #xxxxxx format, where x represents a hexadecimal digit (#000 is black, #fff is white, #f00 is red and so on), or it can be one of the HTML and CSS named colors.

The HTML and CSS named colors can be found at `http://dev.w3.org/csswg/css-color/#named-colors.`

So, for example, this text is typeset with `\color[color=red]{…}`.

Here is a rule typeset with `\color[color=#22dd33]`: ⟶

5.5 background

The **background** package allows you to set the color of the canvas background (by drawing a solid color block the full size of the page on page initialization). The package provides a `\background` command which requires at least one parameter, `color=`⟨color specification⟩, and sets the backgound of the current and all following pages to that color. If setting only the current page background different from the default is desired, an extra parameter `allpages=false` can be passed. The color specification in the same as specified in the **color** package.

So, for example, `\background[allpages=false, color=#e9d8ba]` will set a sepia tone background on the current page.

5.6 rotate

The **rotate** package allows you to rotate things. You can rotate entire frames, by adding the `rotate=`⟨angle⟩ declaration to your frame declaration, and you can rotate any content by issuing the command `\rotate[angle=⟨angle⟩]{⟨content⟩}`, where the angle is measured in degrees.

Content which is rotated is placed in a box and rotated. The height and width of the rotated box is measured, and then put into the normal horizontal list for typesetting. The effect of this is that space is reserved around the rotated content. The best way to understand this is by example: here is some text rotated by _ten,_ _twenty_ and _forty_ degrees.

The previous line was produced by the following code:

```
here is some text rotated by
\rotate[angle=10]{ten}, \rotate[angle=20]{twenty} and \rotate[angle=40]{forty}
degrees.
```

5.7 features

As mentioned in Chapter 3, SILE automatically applies ligatures defined by the fonts that you use. These ligatures are defined by tables of *features* within the font file. As well as ligatures (multiple glyphs displayed as a single glyph), the features tables also declare other glyph substitutions.

The **features** package provides an interface to selecting the features that you want SILE to apply to a font. The features available will be specific to the font file; some fonts come with documentation explaining their supported features. Discussion of OpenType features is beyond the scope of this manual.

These features can be turned on and off by passing 'raw' feature names to the \font command like so:

```
\font[features="+dlig,+hlig"]... % turn on discretionary and historic ligatures
```

However, this is unwieldy and requires memorizing the feature codes. **features** provides two commands, \add-font-feature and \remove-font-feature, which make it easier to access Open-Type features. The interface is patterned on the TeX package fontspec; for full documentation of the OpenType features supported, see the documentation for that package.[1]

Here is how you would turn on discretionary and historic ligatures with the **features** package:

```
\add-font-feature[Ligatures=Rare]\add-font-feature[Ligatures=Discretionary]
...
\remove-font-feature[Ligatures=Rare]\remove-font-feature[Ligatures=Discretionary]
```

5.8 unichar

SILE is Unicode compatible, and expects its input files to be in the UTF-8 encoding. (The actual range of Unicode characters supported will depend on the supported ranges of the fonts that SILE is using to typeset.) Some Unicode characters are hard to locate on a standard keyboard, and so are

---

1.   http://texdoc.net/texmf-dist/doc/latex/fontspec/fontspec.pdf

difficult to enter into SILE documents. The **unichar** package helps with this problem by providing a command to enter Unicode codepoints. After loading **unichar**, the \unichar command becomes available:

```
\unichar{U+263A} % produces ☺
```

If the argument to \unichar begins U+, u+, 0x or 0X, then it is assumed to be a hexadecimal value. Otherwise it is assumed to be a decimal codepoint.

5.9 bidi

Scripts like the Latin alphabet you are currently reading are normally written left to right; however, some scripts, such as Arabic and Hebrew, are written right to left. The **bidi** package, which is loaded by default, provides SILE with the ability to correctly typeset right-to-left text and also documents which mix right-to-left and left-to-right typesetting. Because it is loaded by default, you can use both LTR and RTL text within a paragraph and SILE will ensure that the output characters appear in the correct order.

The **bidi** package provides two commands, \thisframeLTR and \thisframeRTL, which set the default text direction for the current frame. That is, if you tell SILE that a frame is RTL, the text will start in the right margin and proceed leftward. It also provides the commands \bidi-off and \bidi-on, which allow you to trade off bidirectional support for a dubious increase in speed.

5.10 pullquote

The pullquote environment formats longer quotations in an indented blockquote block with decorative quotation marks in the margins.

Here is some text set in a pullquote environment:

> An education is not how much you have committed to memory, or even how much you know. It is being able to differentiate between what you do know and what you do not know.
>
> — *Anatole France*

Optional values are available for:

• author to add an attribution line

• setback to set the bilateral margins around the block

• color to change the color of the quote marks

• scale to change the relative size of the quote marks

If you want to specify what font the pullquote environment should use, you can redefine the `\pullquote:font` command. By default it will be the same as the surrounding document. The font style used for the attribution line can likewise be set redefining `\pullquote:author-font` and the font used for the quote marks can be set redefining `\pullquote:mark-font`.

5.11 raiselower

If you don't want your images, rules or text to be placed along the baseline, you can use the **raiselower** package to move them up and down. (The **footnote** package uses this to superscript the footnote reference numbers.)

It provides two simple commands, `\raise` and `\lower` which both take a `height=(dimension)` parameter. They will respectively raise or lower their argument by the given height. The raised or lowered content will not alter the height or depth of the line.

Here is some text raised by ^{three points}; here is some text lowered by _{four points}.

The previous paragraph was generated by:

```
Here is some text raised by \raise[height=3pt]{three points}; here is
some text lowered by \lower[height=4pt]{four points}.
```

5.12 grid

In normal typesetting, SILE determines the spacing between lines of type according to the following two rules:

• SILE tries to insert space between two successive lines so that their baselines are separated by a fixed distance called the `baselineskip`.

• If this first rule would mean that the bottom and the top of the lines are less than two points apart, then they are forced to be two points apart. (This distance is configurable, and called the `lineskip`)

The second rule is designed to avoid the situation where the first line has a long descender (letters such as g, q, j, p, etc.) which abuts a high ascender on the second line. (k, l, capitals, etc.)

In addition, the `baselineskip` contains a certain amount of 'stretch', so that the lines can expand if this would help with producing a page break at an optimal location, and similarly spacing between paragraphs can stretch or shrink.

The combination of all of these rules means that a line may begin at practically any point on the page.

An alternative way of typesetting is to require that lines begin at fixed points on a regular grid. Some people prefer the 'color' of pages produced by grid typesetting, and the method is often used when typesetting on very thin paper as lining up the lines of type on both sides of a page ensures that ink does not bleed through from the back to the front. Compare the following examples: on the left, the lines are guaranteed to fall in the same places on the recto (front) and the verso (back) of the paper; on the right, no such guarantee is made.

lorem
ipsum
dolor
sit amet

lorem
ipsum
dolor
sit amet

The **grid** package alters the way that the SILE's typesetter operates so that the two rules above do not apply; lines are always aligned on a fixed grid, and spaces between paragraphs etc. are adjusted to conform to the grid. Loading the package adds two new commands to SILE: `\grid[spacing=⟨dimension⟩]` and `\no-grid`. The first turns on grid typesetting for the remainder of the document; the second turns it off again.

At the start of this section, we issued the command `\grid[spacing=15pt]` to set up a regular 15-point grid. Here is some text typeset with the grid set up:

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

And here is the same text after we issue `\no-grid`:

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in

culpa qui officia deserunt mollit anim id est laborum.

5.13 linespacing

SILE's default method of inserting leading between lines should be familiar to users of TeX, but it is not the most friendly system for book designers. The **linespacing** package provides a better choice of leading systems.

After loading the package, you are able to choose the linespacing mode by setting the `linespacing.method` parameter. The following examples have funny sized words in them so that you can see how the different methods interact.

By default, this is set to `tex`. The other options available are:

• `fixed`. This set the lines at a fixed baseline-to-baseline distance, determined by the `linespacing.fixed.baselinedistance` parameter. You can specify this parameter either relative to the type size (e.g. `1.2em`) or as a absolute distance (`15pt`). This paragraph is set with a fixed 1.5em baseline-to-baseline distance.

• `fit-glyph`. This sets the lines solid; that is, the lowest point on line 1 (either a descender like q or, if there are no descenders, the baseline) will touch the highest point of line 2, as in this paragraph. You generally don't want to use this as is.

What you probably want to do is insert a constant (relative or absolute) space between the lines by setting the `linespacing.fit-glyph.extra-space` parameter. This paragraph is set with 5 points of space between the descenders and the ascenders.

• `fit-font`. This inspects each hbox on the line, and asks the fonts it finds for their bounding boxes – the highest ascender and the lower descender. It then sets the lines solid. Essentially each character is treated as if it is the same height, rather like composing a slug of metal type. If there are things other than text on your line, or the text is buried inside other boxes, this may not work so well.

As with `fit-glyph`, you can insert extra space between the lines with the `linespacing.fit-font.extra-space` parameter.

• `css`. This is similar to the method used in browsers; the baseline distance is set with the `linespacing.css.line-height` parameter, and the excess space between this parameter and the actual height of the line is distributed between the top and bottom of the line.

5.14 verbatim

The **verbatim** package is useful when quoting pieces of computer code and other text for which formatting is significant. It changes SILE's settings so that text is set ragged right, with no hyphenation, no indentation and regular spacing. It tells SILE to honor multiple spaces, and sets a monospaced font.

Despite the name, verbatim *does not alter the way that SILE sees special characters. You still need to escape backslashes and braces: to produce a backslash, you need to write* \\.

Here is some text set in the verbatim environment:

```
function SILE.repl()
  if not SILE._repl then SILE.initRepl() end
  SILE._repl:run()
end
```

If you want to specify what font the verbatim environment should use, you can redefine the \verbatim:font command. For example you could change it from XML like this:

```
<define command="verbatim:font">
   <font family="DejaVu Sans Mono" size="9pt"/>
</define>
```

5.15 font-fallback

What happens when SILE is asked to typeset a character which is not in the current font? For instance, we are currently using the "Gentium" font, which covers a wide range of European scripts; however, it doesn't contain any Japanese character. So what if I ask SILE to typeset abc あ?

Many applications will find another font on the system containing the appropriate character and use that font instead. But which font should be chosen? SILE is designed for typesetting situations where the document or class author wants complete control over the typographic appearance of the output, so it's not appropriate for it to make a guess—besides, you asked for Gentium. So where the glyph is not defined, SILE will give you the current font's "glyph not defined" symbol (a glyph called .notdef) instead.

But there are times when this is just too strict. If you're typesetting a document in English and Japanese, you should be able to choose your English font and choose your Japanese font, and if the

glyph isn't available in one, SILE should try the other. The **font-fallback** package gives you a way to specify a list of font specifications, and it will try each one in turn if glyphs cannot be found.

It provides two commands, \font:add-fallback and \font:clear-fallbacks. The parameters to \font:add-fallback are the same as the parameters to \font. So this code:

```
\font:add-fallback[family=Symbola]
\font:add-fallback[family=Noto Sans CJK JP]
```

will add two fonts to try if characters are not found in the current font. Now we can say:

あば x 🐱 Hello world. あ

and SILE will produce:

あば x 🐱 Hello world. あ

\font:clear-fallbacks removes all font fallbacks from the list of fonts to try.

\font:remove-fallback removes the last added fallback from the list of fonts to try.

5.16 boustrophedon

Partly designed to show off SILE's extensibility, and partly designed for real use by classicists, the **boustrophedon** package allows you to typeset ancient Greek texts in the 'ox-turning' layout—the first line is written left to right as normal, but the next is set right to left, then left to right, and so on. To use it, you will need to set the font's language to ancient Greek (grc) and wrap text in a boustrophedon environment:

ΧΑΙΡΕΔΕΜΟΤΟΔΕΣΕΜΑΠΑΤΕΡΕΣΤΕΣΕΘΑΝΟΝΤΟΣΑΝΦΙΧΑΡΕ

(Under normal circumstances, that line would appear as ΧΑΙΡΕΔΕΜΟΤΟΔΕΣΕΜΑΠΑΤΕΡΕΣΤΕΣΕΘΑΝΟΝΤΟΣΑΝΦΙΧΑΡ
.)

5.17 chordmode

This package provides the chordmode environment, which transforms lines like:

```
I've be<G>en a wild rover for many's a <C>year
```

into:
```
       G                          C
I've been a wild rover for many's a year
```

The chords can be styled by redefining the \chordmode:chordfont command, and the offset between the chord name and text set with the chordmode.offset setting.

5.18 converters

The **converters** package allows you to register additional handlers to process included files and images. That sounds a bit abstract, so it's best explained by example. Suppose you have a GIF image that you would like to include in your document. You read the documentation for the **image** package and you discover that sadly GIF images are not supported. What **converters** does is allow you to teach SILE how to get the GIF format into something that *is* supported. We can use the ImageMagick toolkit to turn a GIF into a JPG, and JPGs are supported.

We do this by registering a converter with the \converters:register command:

```
\script[src=packages/converters]
\converters:register[from=.gif,to=.jpg,command=convert $SOURCE $TARGET]
```

And now it just magically works:

```
\img[src=hello.gif, width=50px]
```

This will execute the command convert hello.gif hello.jpg and include the converted hello.jpg file.

This trick also works for text file:

```
\converters:register[from=.md, to=.sil, command=pandoc -o $TARGET $SOURCE]

\include[src=document.md]
```

5.19 cropmarks

When preparing a document for printing, you may be asked by the printer to add crop marks. This means that you need to output the document on a slightly larger page size than your target paper and add printers crop marks to show where the paper should be trimmed down to the correct size. (This is to ensure that pages where the content "bleeds" off the side of the page are correctly cut.)

This package provides the `\crop:setup` command which should be run early in your document file. It takes one argument, `papersize`, which is the true target paper size. It place cropmarks around the true page content.

It also adds a header at the top of the page with the filename, date and output sheet number. You can customize this header by redefining `\crop:header`.

5.20 date

The **date** package provides the `\date` command, which simply outputs the date using the system's date function. You can customize the format by passing the `format` parameter, following the formatting codes in the Lua manual. (`https://www.lua.org/pil/22.1.html`)

5.21 debug

This package provides two commands: `\debug`, which turns on and off SILE's internal debugging flags (similar to using `--debug=...` on the command line); and `\disable-pushback` which is used by SILE's developers to turn off the typesetter's pushback routine, because we don't really trust it very much.

5.22 dropcaps

The **dropcaps** package allows you to format paragraphs with an 'initial capital' (also commonly referred as a 'drop cap'), typically one large capital letter used as a decorative element at the beginning of a paragraph.

It provides the `\dropcap` command. The content passed will be the initial character(s). The primary option is `lines`, an integer specifying the number of lines to span (defaults to 3). The scale of can be adjusted relative to the first line using the `scale` option (defaults to 1.0). The `join` is a boolean for whether to join the dropcap to the first line (defaults to false). If `join` is true, the value of the `standoff` option (defaults to 1spc) is applied to all but the first line. Optionally `color` can be passed to change the typeface color, sometimes useful to offset the apparent weight of a large glyph. To tweak the position of the dropcap, measurements may be passed to the `raise` and `shift` options. Other options passed to `\dropcap` will be passed through to `\font` when drawing the initial letter(s). This may be useful for passing OpenType options or other font preferences.

One caveat is that the size of the initials is calculated using the default linespacing mechanism. If you are using an alternative method from the linespacing package, you might see strange results. Set the `document.baselineskip` *to approximate your effective leading value for best results. If that doesn't work set the size manually. Using* `SILE.setCommandDefaults()` *can be helpful for so you don't have to set the size every time.*

39

5.23 leaders

The **leaders** package allows you to create repeating patterns which fill a given space. It provides the
\dotfill command, which does this:

```
A\dotfill{}B
```

A . . . . B

It also provides the \leaders[width=⟨dimension⟩]{⟨content⟩} command which allow you to
define your own leaders. For example:

```
A\leaders[width=40pt]{/\\}B
```

A /\/\/\/\ B

If the width is omitted, the leaders extend as much as possible (as a \dotfill or \hfill).

Leader patterns are always vertically aligned, respectively to the end edge of the frame they
appear in, for a given font. It implies that the number of repeated patterns and their positions do
not only depend on the available space, but also on the alignment constraint and the active font.

5.24 lorem

Sometimes you just need some dummy text. The command \lorem produces fifty words of "lorem
ipsum"; you can choose a different number of words with the words=⟨number⟩ parameter. Here's
\lorem[words=20]:

lorem ipsum dolor sit amet consetetur sadipscing elitr sed diam nonumy eirmod tempor in-
vidunt ut labore et dolore magna aliquyam

5.25 specimen

SILE has found itself becoming well used by type designers, who often want to create specimen
documents to show off their new fonts. This package provides a few commands to help create test
documents. (The fontproof class, available from the package manager, contains many more tools
for creating specimens.) The \repertoire command prints out every glyph in the font, in a simple
table. The \pangrams command prints out a few pangrams for the Latin script. Finally, \set-to-

width[width=⟨dimension⟩]{⟨content⟩} will process each line of content, changing the font size so that the output is a constant width.

```
\begin[width=4cm]{set-to-width}
CAPERCAILLIE
LAMMERGEYER
CASSOWARY
ACCENTOR DOWITCHER DOTTEREL
\end{set-to-width}
```

CAPERCAILLIE

LAMMERGEYER

CASSOWARY

ACCENTOR DOWITCHER DOTTEREL

5.26 textcase

The **textcase** package provides commands for language-aware case conversion of input text. For example, when language is set to English, then \uppercase{hij} will return HIJ. However, when language is set to Turkish, it will return HİJ.

As well as \uppercase, the package provides the commands \lowercase and \titlecase.

5.27 url

This package enhances the typesetting of URLs in two ways. First, it provides the \href[src=⟨url⟩] {⟨content⟩} command which inserts PDF hyperlinks, like this.

The \href command accepts the same borderwidth, bordercolor, borderstyle and borderoffset styling options as the \pdf:link command from the **pdf** package, for instance like this.

Nowadays, it is a common practice to have URLs in print articles (whether it is a good practice or not is yet *another* topic). Therefore, the package also provides the \url command, which will automatically insert breakpoints into unwieldy URLs like https://github.com/sile-typesetter/sile-typesetter.github.io/tree/master/examples so that they can be broken up over multiple lines.

It allows line breaks after the colon, and before or after appropriate segments of an URL (path elements, query parts, fragments, etc.). By default, the \url command ignores the current language, as one would not want hyphenation to occur in URL segments. If you have no other choice, however, you can pass it a language option to enforce a language to be applied. Note that if French (fr) is selected, the special typographic rules applying to punctuations in this language are disabled.

To typeset an URL and at the same type have it as active hyperlink, one can use the `\href` command without the `src` option, but with the URL passed as argument.

The breaks are controlled by two penalty settings, `url.linebreak.primaryPenalty` for preferred breakpoints and, for less acceptable but still tolerable breakpoints, `url.linebreak.secondaryPenalty` —its value should logically be higher than the previous one.

5.28 Packages usually used by other packages

In addition, there are certain packages that you *probably* don't need to use directly, as their main job is to provide more basic functionality to other packages and classes. Classes such as the `book` class compose functionality from different auxiliary packages.

5.28.1 footnotes

We've seen that the `book` class allows you to add footnotes to text with the `\footnote` command. This functionality exists in the class because the class loads the **footnotes** package. The `book` class loads up the **insertions** package and tells it which frame should recieve the footnotes that are typeset. After commands provided by the **footnotes** package take care of formatting the footnotes.

5.28.2 color-fonts

The **color-fonts** package adds support for fonts with a `COLR` OpenType table. This package is automatically loaded when such a font is detected.

5.28.3 counters

Various parts of SILE such as the **footnotes** package and the sectioning commands keep a counter of things going on: the current footnote number, the chapter number, and so on. The counters package allows you to set up, increment and typeset named counters. It provides the following commands:

• `\set-counter[id=⟨counter-name⟩, value=⟨value⟩]` — sets the counter with the specified name to the given value.

• `\increment-counter[id=⟨counter-name⟩]` — does the same as `\set-counter` except that when no `value` parameter is given, the counter is incremented by one.

• `\show-counter[id=⟨counter-name⟩]` — this typesets the value of the counter according to the counter's declared display type.

All of the commands in the counters package take an optional `display=⟨display-type⟩` *parameter to set the display type of the counter.*

The available built-in display types are:

• `arabic`, *the default;*

- alpha, *for lower-case alphabetic counting;*

- Alpha, *for upper-case alphabetic counting;*

- roman, *for lower-case Roman numerals; and,*

- Roman *for upper-case Roman numerals.*

The ICU library also provides ways of formatting numbers in global (non-Latin) scripts. You can use any of the display types in this list: http://www.unicode.org/repos/cldr/tags/latest/common/bcp47/number.xml. *For example,* display=beng *will format your numbers in Bengali digits.*

So, for example, the following SILE code:

```
\set-counter[id=mycounter, value=2]
\show-counter[id=mycounter]

\increment-counter[id=mycounter]
\show-counter[id=mycounter, display=roman]
```

produces:

2

iii

5.28.4 pdf

The **pdf** package enables (basic) support for PDF links and table-of-contents entries. It provides the four commands \pdf:destination, \pdf:link, \pdf:bookmark, and \pdf:metadata.

The \pdf:destination parameter creates a link target; it expects a parameter called name to uniquely identify the target. To create a link to that location in the document, use \pdf:link[dest=⟨name⟩]{⟨content⟩}.

The \pdf:link command accepts several options defining its border style: a borderwidth length setting the border width (defaults to 0, meaning no border), a borderstyle string (can be set to "underline" or "dashed", otherwise a solid box), a bordercolor color specification for this border (defaults to blue), and finally a borderoffset length for adjusting the border with some vertical space above the content and below the baseline (defaults to 1pt). Note that PDF renderers may vary on how they honor these border styling features on link annotations.

It also has an `external` option for URL links, which is not intended to be used directly—refer to the **url** package for more flexibility typesetting external links.

To set arbitrary key-value metadata, use something like `\pdf:metadata[key=Author, value=J. Smith]`. The PDF metadata field names are case-sensitive. Common keys include `Title`, `Author`, `Subject`, `Keywords`, `CreationDate`, and `ModDate`.

5.28.5 ifattop

This package provides two commands: `\ifattop` and `\ifnotattop`. The argument of the command is processed only if the typesetter is at the top of a frame or is not at the top of a frame respectively.

5.28.6 frametricks

As we mentioned in the first chapter, SILE uses frames as an indication of where to put text onto the page. The **frametricks** package assists package authors by providing a number of commands to manipulate frames.

The most immediately useful is `\showframe`. This asks the output engine to draw a box and label around a particular frame. It takes an optional parameter `id=⟨frame id⟩`; if this is not supplied, the current frame is used. If the ID is `all`, then all frames declared by the current class are displayed.

It's possible to define frames such as sidebars which are not connected to the main text flow of a page. We'll see how to do that in a later chapter, but this raises the obvious question: if they're not part of the text flow, how do we get stuff into them? **frametricks** provides the `\typeset-into` command, which allows you to write text into a specified frame:

```
\typeset-into[frame=sidebar]{ ... frame content here ... }
```

---

frametricks also provides a number of commands which, to be perfectly honest, we *thought were going to be useful, but haven't quite ended up being as useful as all that.*

The command `\breakframevertical` breaks the current frame in two at the specified location into an upper and lower frame. If the frame initially had the ID `main`, then `main` becomes the upper frame (before the command) and the lower frame (after the command) is called `main_`. We just issued a `\breakframevertical` command at the start of this paragraph, and now we will issue the command `\showframe`. As you can see, the current frame is called `content_` and now begins at the start of the paragraph.

Similarly, the `\breakframehorizontal` command breaks the frame in two into a left and a right frame. The command takes an optional argument `offset=⟨dimension⟩`, specifying where on the line the frame should be split. If it is not supplied, the frame is split at the current position in the line.

The command `\shiftframeedge` allows you to reposition the current frame left or right. It takes `left` and/or `right` parameters, which can be positive or negative dimensions. It should only be used at the top of a frame, as it reinitializes the typesetter object.

44

Combining all of these commands, the `\float` command breaks the current frame, creates a small frame to hold a floating object, flows text into the surrounding frame, and then, once text has descended past the floating object, moves the frame back into place again. It takes two optional parameters, `bottomboundary=⟨dimension⟩` and/or `rightboundary=⟨dimension⟩`, which open up additional space around the frame. At the start of this paragraph, I issued the command

`\float[bottomboundary=5pt]{\font[size=60pt]{C}}`.

To reiterate: we started playing around with frames like this in the early days of SILE and they have not really proved a good solution to the things we wanted to do with them. They're great for arranging where content should live on the page, but messing about with them dynamically seems to create more problems than it solves. There's probably a reason why InDesign and similar applications handle floats, drop caps, tables and so on inside the context of a content frame rather than by messing with the frames themselves. If you feel tempted to play with **frametricks**, there's almost always a better way to achieve what you want without it.

5.28.7 insertions

The **footnotes** package works by taking auxiliary material (the footnote content), shrinking the current frame and inserting it into the footnote frame. This is powered by the **insertions** package; it doesn't provide any user-visible SILE commands, but provides Lua functionality to other packages. TeX wizards may be interested to realise that insertions are implemented by an external add-on package, rather than being part of the SILE core.

5.28.8 twoside

The `book` class described in chapter 4 sets up left and right mirrored page masters; the **twoside** package is responsible for swapping between the two left and right frames, running headers and so on. It has no user-serviceable parts.

5.28.9 masters

The masters functionality is also itself an add-on package. It allows a class to define sets of frames and switch between them either temporarily or permanently. It defines the commands `\define-master-template` (which is pattern on the `\pagetemplate` function we will meet in chapter 8), `\switch-master` and `\switch-master-one-page`. See `tests/masters.sil` for more about this package.

5.28.10 infonode

This package is only for class designers.

While typesetting a document, SILE first breaks a paragraph into lines, then arranges lines into a page, and later outputs the page. In other words, while it is looking at the text of a paragraph, it is not clear what page the text will eventually end up on. This makes it difficult to produce indexes, tables of contents and so on where one needs to know the page number for a particular element.

To get around this problem, the **infonode** package allows you to insert *information nodes* into the text stream; when a page is outputted, these nodes are collected into a list, and a class's output routine can examine this list to determine which nodes fell on a particular page. **infonode** provides the \info command to put an information node into the text stream; it has two required parameters, category=(name) and value=(any object). Categories are used to group similar sets of node together.

As an example, when typesetting a Bible, you may wish to display which range of verses are on each page as a running header. During the command which starts a new verse, you would insert an information node with the verse reference:

```
SILE.call("info", { category = "references", value = ref }, {})
```

During the endPage method which is called at the end of every page, we look at the list of "references" information nodes:

```
local refs = SILE.scratch.info.thispage.references
local runningHead = SILE.shaper.shape(refs[1] .. " - " .. refs[#refs])
SILE.typesetNaturally(rhFrame, runningHead);
```

5.28.11 inputfilter

The **inputfilter** package provides ways for class authors to transform the input of a SILE document after it is parsed but before it is processed. It does this by allowing you to rewrite the abstract syntax tree representing the document.

Loading **inputfilter** into your class with class:loadPackage("inputfilter") provides you with two new Lua functions: transformContent and createCommand. transformContent takes a content tree and applies a transformation function to the text within it. See https://sile-typesetter.org/examples /inputfilter.sil for a simple example, and packages/chordmode.sil for a more complete one.

5.28.12 break-firstfit

SILE's normal page breaking algorithm is based on the Knuth-Plass "best-fit" method, which tests a variety of possible paragraph constructions before deciding on the visually optimal one. That guarantees great results for texts which require full justification, but some languages don't need that degree of complexity. In particular, Japanese is traditionally typeset on a grid system with charac-

ters being essentially monospaced. You don't need to do anything clever to break that into lines: just stop when you get to the end of the line and start a new one (the "first-fit" method). This package implements the first-fit technique. It's currently designed to be used by other packages so it doesn't currently provide any user-facing commands.

5.28.13 chapterverse

The **chapterverse** package is designed as a helper package for book classes which deal with versified content such as scriptures. It provides commands which will generally be called by the higher-level \verse and \chapter (or moral equivalent) commands of the classes which handle this kind of content:

• \save-book-title takes its argument and squirrels it away as the current book name.

• \save-chapter-number and \save-verse-number does the same but for the chapter and verse reference respectively.

• \format-reference is expected to be called from Lua code with a content table of {book = ..., chapter = ..., verse = ...} and typesets the reference in the form cc:vv. If the parameter showbook= true is given then the book name is also output. (You can override this command to output your references in a different format.)

• \first-reference and \last-reference typeset (using \format-reference) the first reference on the page and the last reference on the page respectively. This is helpful for running headers.

5.28.14 parallel

The **parallel** package provides the mechanism for typesetting diglot or other parallel documents. When used by a class such as classes/diglot.lua, it registers a command for each parallel frame, to allow you to select which frame you're typesetting into. It also defines the \sync command, which adds vertical spacing to each frame such that the *next* set of text is vertically aligned. See https:// sile-typesetter.org/examples/parallel.sil and the source of classes/diglot.lua for examples which makes the operation clear.

5.28.15 rebox

This package provides the \rebox command, which allows you to lie to SILE about the size of content. You can change the width, height, or depth of your content with the respective parameters, or make it invisible by setting the phantom parameter to true.

For example:

```
Hello \rebox[width=0pt]{world}overprint.
Look I'm not \rebox[phantom=true]{here}!
```

Hello world print.

Look I'm not   !

5.28.16 tableofcontents

The **tableofcontents** package provides tools for class authors to create tables of contents. When you are writing sectioning commands such as \chapter or \section, your classes should call the \tocentry[level=(number), number=(strings)]{(title)} command to add a table of contents entry. At the end of each page the class will call a hook function (moveTocNodes) that collates the table of contents entries from that pages and logs which page they're on. At the end of the document another hook function (writeToc) will write this data to a file. The next time the document is built, any use of the \tableofcontents (typically near the beginning of a document) will be able to read that index data and output the TOC. Because the toc entry and page data is not available until after rendering the document, the TOC will not render until at least the second pass. If by chance rendering the TOC itself changes the document pagination (e.g. the TOC spans more than one page) it might be necessary to run SILE 3 times to get accurate page numbers shown in the TOC.

The \tableofcontents command accepts a depth option to control the depth of the content added to the table.

If the **pdf** package is loaded before using sectioning commands, then a PDF document outline will be generated. Moreover, entries in the table of contents will be active links to the relevant sections. To disable the latter behavior, pass linking=false to the \tableofcontents command.

Class designers can also style the table of contents by overriding the following commands:

• \tableofcontents:headerfont - the font used for the header.

• \tableofcontents:level1item, \tableofcontents:level2item, etc. - styling for entries.

• \tableofcontents:level1number, \tableofcontents:level2number, etc. - deciding what to do with entry section number, if defined: by default, nothing (so they do not show up in the table of contents).

5.28.17 xmltricks

In chapter 9, we're going to use SILE to typeset existing XML documents. Most of the work of typesetting XML with SILE is creating processing expectations for particular XML tags. **xmltricks** makes the process somewhat easier by providing commands to handle two common cases.

\xmltricks:ignore{tag1 tag2 tag3} instructs SILE not to bother investigating the given tags. For instance, when processing a HTML document, you probably don't need to typeset the head tag or anything inside it.

Similarly, some tags are just wrappers; you want to process their content, but there's nothing specific about the tag itself that needs any styling. List those tags in a \xmltricks:passthru{...} command, and SILE will descend into the content without requiring a specific command for the tag itself.

5.29 Experimental packages

5.29.1 autodoc

This package extracts documentation information from other packages. It's used to construct the SILE manual. Keeping package documentation in the package itself keeps the documentation near the implementation, which (in theory) makes it easy for documentation and implementation to be in sync.

For that purpose, it provides the `\package-documentation[src=⟨package⟩]` command.

Properly documented packages should export a `documentation` string containing their documentation, as a SILE document.

For documenters and package authors, it also provides commands that can be used in their package documentation to present various pieces of information in a consistent way.

Setting names can be fairly long (e.g. *namespace.area.some-stuff*). The `\autodoc:setting` command helps line-breaking them automatically at appropriate points, so that package authors do not have care about them manually.

With the `\autodoc:command` command, one can pass a simple command, or even a full commands (with parameters and arguments), without the need for escaping special characters. This relies on SILE's AST (abstract syntax tree) parsing, so you benefit from typing simplicity, syntax check, and even more –such as styling[2]. Moreover, for text content in parameter values or command arguments, if they are enclosed between angle brackets, they will be presented with an distinguishable style. Just type the command as it would appear in code, and it will be nicely typeset. It comes with a few caveats, though: parameters are not guaranteed to appear in the order you entered them and some purely syntactic sequences are just skipped and not reconstructed. Also, it is not adapted to math-related commands. So it comes with many benefits, but also at a cost.

The `\autodoc:environment` command just takes an environment name, so basically a command, but just displays it without leading backslash.

The `\autodoc:setting`, `\autodoc:command` and `\autodoc:environment` commands all check the validity and existence of their inputs. Would you want to disable this feature (e.g. to refer to a setting or command defined in another package or module that might not be loaded at this point), you can set the optional parameter `check` to false. Note, however, that for commands, it is applied recursively to the parsed AST (so it is a all-or-none trade-off).

The `\autodoc:parameter` commands takes either a parameter name, possibly with a value (which as above, may be bracketed) and typesets it in the same fashion.

5.29.2 balanced-frames

This package attempts to ensure that the main content frames on a page are balanced; that is, that they have the same height. In your frame definitions for the columns, you will need to ensure that they have the parameter `balanced` set to a true value. See the example in `tests/balanced.sil`.

The current algorithm does not work particularly well, and a better solution to the column problem is being developed.

5.29.3 bibtex

BibTeX is a citation management system. It was originally designed for TeX but has since been integrated into a variety of situations.

This experimental package allows SILE to read and process BibTeX `.bib` files and output citations and full text references. (It doesn't currently produce full bibliography listings.)

To load a BibTeX file, issue the command `\loadbibliography[file=⟨whatever.bib⟩]`

To produce an inline citation, call `\cite{⟨key⟩}`, which will typeset something like "Jones 1982". If you want to cite a particular page number, use `\cite[page=22]{⟨key⟩}`.

To produce a full reference, use `\reference{⟨key⟩}`.

Currently, the only supported bibliography style is Chicago referencing, but other styles should be easy to implement if there is interest. Check out `packages/bibtex/styles/chicago.lua` and adapt as necessary.

5.29.4 gutenberg

One of the reasons why Johann Gutenberg's 42 line Bible is considered a masterpiece of early printing is due to the quality of justification of every line. To achieve perfect justification color, Gutenberg used a number of ligatures, abbreviations, substitutions and so on. As an experiment in extending SILE's justification engine, the `gutenberg` package allows SILE to choose between a number of different options for a particular piece of text, depending on what would improve the line fitting.

For instance, issuing the command `\alternative{{and}{&}}` would insert either the text *and* or an ampersand, depending on what best fits the current line.

5.29.5 indexer

An index is essentially the same thing as a table of contents, but sorted. This package provides the `\indexentry` command, which can be called as either `\indexentry[label=⟨text⟩]` or `\indexentry{⟨text⟩}` (so that it can be called from a macro). Index entries are collated at the end of each page, and the command `\printindex` will deposit them in a list. The entry can be styled using the `\index:item` command.

Multiple indexes are available and an index can be selected by passing the `index=⟨name⟩` parameter to `\indexentry` and `\printindex`.

Classes using the indexer will need to call its exported function `buildIndex` as part of the end page routine.

5.29.6 math

This package provides typesetting of formulas directly in a SILE document.

Mathematical typesetting in SILE is still in its infancy. As such, it lacks some features and may contain bugs. Feedback and contributions are always welcome.

To typeset mathematics, you will need an OpenType math font installed on your system[3]. By default, this package uses Libertinus Math, so it will fail if Libertinus Math can't be found. Another font may be specified via the setting `math.font.family`.

The first way to typeset math formulas is to enter them in the MathML format. MathML is a standard for encoding mathematical notation for the Web and for other types of digital documents. It is supported by a wide range of tools and represents the most promising format for unifying the encoding of mathematical notation, as well as improving its accessibility (e.g. to blind users).

To render an equation encoded in MathML, one simply has to put it in a `mathml` command. For example, the formula $a^2 + b^2 = c^2$ was typeset by the following command:

```
\mathml{
    \mrow{
        \msup{\mi{a}\mn{2}}
        \mo{+}
        \msup{\mi{b}\mn{2}}
        \mo{=}
        \msup{\mi{c}\mn{2}}
    }
}
```

In an XML document, we would have used the more classical XML syntax:

```
<mathml>
    <mrow>
        <msup> <mi>a</mi> <mn>2</mn> </msup>
        <mo>+</mo>
        <msup> <mi>b</mi> <mn>2</mn> </msup>
        <mo>=</mo>
        <msup> <mi>c</mi> <mn>2</mn> </msup>
    </mrow>
</mathml>
```

By default, formulas are integrated into the flow of text. To typeset them on their own line, one may use the `mode=display` option:

$$a^2 + b^2 = c^2$$

As this example code illustrates, MathML is not really intended to be written by humans and quickly becomes very verbose. That is why this package also provides a `math` command, which understands a syntax similar to the math syntax of TeX. To typeset the above equation, one only has to type `\math{a^2 + b^2 = c^2}`.

Here is a slightly more involved equation:

```
\begin[mode=display]{math}
    \sum_{n=1}^\infty \frac{1}{n^2} = \frac{\pi^2}{6}
\end{math}
```

This renders as:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

The general philosophy of the TeX-like syntax is to be a simple layer on top of MathML, and not to mimic perfectly the syntax of the LaTeX tool. Its main difference from the SILE syntax is that `\mycommand{arg1}{arg2}{arg3}` is translated into MathML as `<mycommand> arg1 arg2 arg3 </mycommand>` whereas in normal SILE syntax, the XML equivalent would be `<mycommand>arg1</mycommand> arg2 arg3`.

`\sum`, `\infty` and `\pi` are only shorthands for the Unicode characters $\sum$, ∞ and π. If it's more convenient, you can use these Unicode characters directly. The symbol shorthands are the same as in the TeX package `unicode-math`.

`{formula}` is a shorthand for `\mrow{formula}`. Since parentheses — among other glyphs — stretch vertically to the size of their englobing `mrow`, this is useful to typeset parentheses of different sizes on the same line:

```
\Gamma (\frac{\zeta}{2}) + x^2(x+1)
```

---

renders as

$$\Gamma\!\left(\frac{\zeta}{2}\right) + x^2\!\left(x + 1\right)$$

which is ugly. To keep parentheses around $x + 1$ small, you should put braces around the expression:

```
\Gamma (\frac{\zeta}{2}) + x^2{(x+1)}
```

$$\Gamma\!\left(\frac{\zeta}{2}\right) + x^2(x + 1)$$

To print a brace in a formula, you need to escape it with a backslash.

In the `math` syntax, every individual letter is an identifier (MathML tag `mi`), every number is a... number (tag `mn`) and all other characters are operators (tag `mo`). If it does not suit you, you can explicitly use the `\mi`, `\mn` or `\mo` tags. For instance, `sin(x)` will be rendered as $sin(x)$, because SILE considers the letters s, i and n to be individual identifiers, and identifiers made of one character are italicized by default. To avoid that, you can specify that sin is an identifier by writing `\mi{sin}(x)` and get: $\sin(x)$. If you prefer it in "double struck" style, this is permitted by the `mathvariant` attribute: `\mi[mathvariant=double-struck]{sin}(x)` renders as $\sin(x)$.

To save you some typing, the math syntax lets you define macros with the following syntax:

```
\def{macro-name}{macro-body}
```

---

where in the macro's body #1, #2, etc. will be replaced by the macro's arguments. For instance:

```
\begin[mode=display]{math}
    \def{diff}{\mfrac{\mo{d}#1}{\mo{d}#2}}
    \def{bi}{\mi[mathvariant=bold-italic]{#1}}
```

53

```
    \diff{\bi{p}}{t} = ∑_i \bi{F}_i
\end{math}
```

results in:

$$\frac{\mathrm{d}\boldsymbol{p}}{\mathrm{d}t} = \sum_i \boldsymbol{F}_i$$

Finally, tabular math can be typeset using the `table` command (or equivalently the `mtable` MathML tag). For instance, to typeset a matrix:

```
\begin[mode=display]{math}
    (
    \table{
        1 & 2 & 7 \\
        0 & 5 & 3 \\
        8 & 2 & 1 \\
    }
    )
\end{math}
```

will yield:

$$\begin{pmatrix} 1 & 2 & 7 \\ 0 & 5 & 3 \\ 8 & 2 & 1 \end{pmatrix}$$

Tables may also be used to control the alignment of formulas:

```
\begin[mode=display]{math}
    \{
    \table[columnalign=right center left]{
        u_0 &=& 1 \\
        u_1 &=& 1 \\
        u_n &=& u_{n−1} + u_n−2, \forall n ⩾ 2 \\
    }
\end{math}
```

$$\begin{cases} u_0 &= 1 \\ u_1 &= 1 \\ u_n &= u_{n-1} + u_{n-2}, \forall n \geqslant 2 \end{cases}$$

Tables currently do not support all attributes required by the MathML standard, but they do allow to control spacing using the `rowspacing` and `columnspacing` options.

Finally, here is a little secret. This notation:

```
\table{
    1 & 2 & 7 \\
    0 & 5 & 3 \\
    8 & 2 & 1 \\
}
```

is strictly equivalent to this one:

```
\table{
        {1} {2} {7}
    }{
        {0} {5} {3}
    }{
        {8} {2} {1}
    }
}
```

In other words, the notation using `&` and `\\` is only a syntactic sugar for a two-dimensional array constructed with braces.

When macros are not enough, creating new mathematical elements is quite simple: one only needs to create a new class deriving from `mbox` (defined in `packages/math/base-elements.lua`) and define the `shape` and `output` methods. `shape` must define the `width`, `height` and `depth` attributes of the element, while `output` must draw the actual output. An `mbox` may have one or more children (for instance, a fraction has two children — its numerator and denominator). The `shape` and `output` methods of the children are called automatically.

This package still lacks support for some mathematical constructs, but hopefully we'll get there. Among unsupported constructs are: decorating symbols with so-called accents, such as arrows or hats, "over" or "under" braces, and line breaking inside a formula.

5.29.7 pdfstructure

For PDF documents to be considered accessible, they must contain a description of the PDF's docu-

ment structure. This package allows structure trees to be created and saved to the PDF file. Currently this provides a low-level interface to creating nodes in the tree; classes which require PDF accessibility should use the `\pdf:structure` command in their sectioning implementation to declare the document structure.

See `tests/pdf.sil` for an example of using the **pdfstructure** package to create a PDF/UA compatible document.

5.29.8 svg

This package provides two commands.

The first is `\svg[src=⟨file⟩]`. This loads and parses an SVG file and attempts to render it in the current document. Optional `width` or `height` options will scale the SVG canvas to the given size calculated at a given `density` option (which defaults to 72 ppi). For example, the command `\svg[src= packages/svg/smiley.svg, height=12pt]` produces the following:

The second is a more experimental `\svg-glyph`. When the current font is set to an SVG font, SILE does not currently render the SVG glyphs automatically. This command is intended to be used as a means of eventually implementing SVG fonts; it retrieves the SVG glyph provided and renders it.

In both cases the rendering is done with our own SVG drawing library; it is currently very minimal, only handling lines, curves, strokes and fills. For a fuller implementation, consider using a **converters** registration to render your SVG file to PDF and include it on the fly.

5.30 The Package Manager

While your SILE installation will include a comprehensive collection of SILE packages, there's more out there! SILE comes with a simple package manager which can help you to download and install new packages. The package manager is accessed from the SILE REPL, like so:

```
 $ sile
SILE v0.13.0 (Lua 5.3)
> installPackage("endnotes")
Loading catalogue from https://raw.githubusercontent.com/simoncozens/sile-packages/master/
packages.lua
```

There will then follow a load of information which should end up with the `endnotes` package and all its dependencies being installed. SILE will also ensure that downloaded packages are added to the package search path. There's currently no way to list or search the available packages. We'll get there.

Chapter 6
SILE Macros and Commands

One of the reasons that we use computers is that they are very good at doing repetitive jobs for us, so that we don't have to. Perhaps the most important skill in operating computers, and particularly in programming computers, is noticing areas where an action is being repeated, and allowing the computer to do the work instead of the human. In other words, Don't Repeat Yourself.

The same is true in operating SILE. After you have been using the system for a while, you will discover that there are patterns of input that you need to keep entering again and again.

6.1 A simple macro

For instance, let's suppose that we want to design a nice little "bumpy road" logo for SILE. (Afficionados of TeX and friends will be familar with the concept of bumpy road logos.) Our logo will look like this: S$_\text{I}$L$^\text{E}$. It's not a great logo, but we'll use it as S$_\text{I}$L$^\text{E}$'s logo for the purposes of this section.

To typeset this logo, we need to ask S$_\text{I}$L$^\text{E}$ to: typeset an 'S'; typeset an 'I' lowered by a certain amount (half an ex, as it happens); typeset an 'L'; walk backwards along the line a tiny bit; typeset a smaller-sized 'E' raised by a certain amount, using the `features` package to choose a small capital 'E'.

In S$_\text{I}$L$^\text{E}$ code, that looks like:

```
S%
\lower[height=0.5ex]{I}%
L%
\kern[width=-.2em]\raise[height=0.6ex]{\font[features=+smcp]{e}}%
```

(Don't worry about the `\kern` command for the moment; we'll come back to that later. The `%`'s prevent newlines from becoming spaces.)

We've used our logo four times already in this chapter, and we don't want to have to input that whole monstrosity each time we do so. What we would like to do is tell the computer "this is S$_\text{I}$L$^\text{E}$'s logo; each time I enter `\SILE`, I want you to interpret that as `S\lower[height=0.5ex]{I}L\kern[width=-.2em]\raise[height=0.6ex]{\font[features=+smcp]{e}}`".

In other words, we want to define our own commands.

SILE[1] allows you to define your own commands in two ways. The simplest commands of all are

those like \SILE above: "when I write \x, I want SILE to pretend that I had written X \Y Z instead." These are called *macros,* and the process of pretending is called *macro expansion.*

You can define these kinds of macros within a SILE file itself. In this very file, we entered:

```
\define[command=SILE]{%
S%
\lower[height=0.5ex]{I}%
L%
\kern[width=-.2em]\raise[height=0.6ex]{\font[features=+smcp]{e}}%
}
```

We are using the built-in SILE command \define. \define takes an option called command; its value is the name of the command we are defining. The content of the \define command is a series of SILE instructions to be executed when the command is used.

At this point it's worth knowing the precise rules for allowable names of SILE commands.

Commands in XML-flavor input files must be allowable XML tag names or else your input files will not be well formed. Command names in TeX-flavor input files may consist of any number of alphanumeric characters, hyphens or colons. Additionally, any single character is a valid TeX-flavor command name. (Hence \\ for typesetting a backslash.)

When it comes to defining commands, commands defined by an XML-flavor file can actually have any name that you like—even if they are not accessible from XML-flavor! (You may define oddly-named commands in a XML-flavor SILE file and then use them in a TeX-flavor SILE file.) Commands defined in TeX-flavor obviously have to have names which are valid parameter values, or else they will not parse correctly either; parameter values happen to consist of any text up until the nearest comma, semicolon or closing square bracket.

6.2 Macro with content

Now let's move on to the next level; sometimes you will want to create commands which are not simply replacements, but which have arguments of their own. As an example, let's say we use the color package to turn a bit of text red like this. The usual way to do that is to say

```
\color[color=red]{like this}
```

However, we're not always going to want to be highlighting the words 'like this'. We might want to be able to highlight other text instead. We need the ability to wrap the command \color [color=red]{ ... } around our chosen content. In other words, we want to be able to define our own commands which take arguments.

The way we express this in SILE is with the \process command. \process is only valid within the context of a \define command (you'll mess everything up if you try using it somewhere else), and it basically means 'do whatever you were planning to do with the arguments to this command.' So if we want to a command which makes things red, we can say:

```
\define[command=red]{\color[color=red]{\process}}
…
Making things red is a \red{silly} way to emphasise text.
```

You can't call \process *more than once within the same macro.*

In the definition of the \chapter *command, we want to 1) display the chapter name in a big bold font, and 2) use the chapter name as the left running header. If you try writing the* \chapter *command as a macro, you will get stuck—once you've* \process*ed the chapter name to display it in bold, you won't be able to process it again to set the running header.*

So the \chapter *command cannot be written as a simple macro. The other way to implement your own commands is to write them in the Lua programming language, which is what happens for* \chapter*. This is deliberate: the* \define *command really is meant to be used just for simple things, because we believe that programming tasks should be done in a programming language. So don't be afraid to write your own commands in Lua–it's not too difficult, and if you're creating any serious document format yourself (rather than processing a document using a class that someone else has written or adding minor formatting tweaks through customization hooks that classes give you) you should expect to write it in Lua, as you're almost certainly going to need to do so. We will see how to do this in later chapters.*

6.3 Nesting macros

That said, one thing you can do is to call a macro within a macro. This should be obvious, because a macro is just a replacement for the current processing step—when SILE reads a macro command, it behaves as if you had entered the definition of the macro instead, and of course such a definition can contain other commands.

So it is possible even within the simple scope of macro processing to achieve quite a lot of automation.

For instance, within this book, there have been a number of notes—italicized paragraphs between two heavy lines with a left margin. These have been typeset with the \note command; this is not a built-in command but a macro specified within the documentation/macros.sil file included by this document. Here is one way to define \note, in XML flavour:

```
<define command="line">
  <par/><smallskip/><noindent/>
  <hrule width="450pt" height="0.3pt"/><par/>
  <novbreak/><smallskip/><novbreak/>
</define>
<define command="narrower">
  <set parameter="document.lskip" value="24pt"/>
  <process/>
  <set parameter="document.lskip" value="0pt"/>
</end>
<define command="notefont"><font style="italic" size="10pt"><process/></font></notefont>
<define command="note">
  <narrower>
    <line/>
    <notefont><process/></notefont>
    <line/>
  </narrower>
</define>
```

The only command we have not yet met here is \set, which we will now investigate.

Chapter 7
SILE Settings

As well as commands, SILE offers a variety of knobs and levers which affect how it does its job. Changing these parameters can have anything from a subtle to a dramatic effect on the eventual document. External packages may declare their own settings, which are documented accordingly. Here we will run through the settings which are built into the SILE system itself.

Settings in SILE are *namespaced* so that 1) the name of the setting gives you some kind of clue as to what area of the system it will affect, and 2) packages can define their own settings without worrying that they will be interfering with other packages or the SILE internals. Namespacing of settings takes the form area.name—so for instance, typesetter.orphanpenalty is the setting which changes how the typesetter penalizes orphan (end-of-paragraph) lines.

The interface to changing settings from within a SILE document is the \set commmand. It takes several options, the most basic one being *parameter*, which expresses which setting is being changed. The *value* option expresses the value to which the setting is being changed. As an example:

```
\set[parameter=typesetter.orphanpenalty, value=250]
```

Two additional options are accepted. The *makedefault* option can added so that whatever value you set sticks as the new default. The *reset* can be used without a *value* option to reset whatever the current value is back to the default Note these two options are mutually exclusive.

```
\set[parameter=typesetter.orphanpenalty, value=250, makedefault=true]
```

or:

```
\set[parameter=typesetter.orphanpenalty, reset=true]
```

If the \set command is provided with any content, then the change of setting is localised to the content of the argument. In other words, this code:

```
\set[parameter=typesetter.orphanpenalty, value=250]{\lorem}
```

will change the orphan penalty to 250, typeset 50 words of dummy text, and then return the orphan penalty to its previous value.

If you are working in Lua, you have two choices to work with. As with any registered command you can call it using `SILE.call()`. For example:

```
SILE.call("set",  parameter = "typesetter.orphanpenalty", value = 250 )
```

There is nothing wrong with this and it allows you to optionally pass content that is wrapped in those settings. However there is also a slightly lower level function that is more ideomatic of Lua code than SILE that uses positional arguments instead of named options:

```
SILE.settings:set("typesetter.orphanpenalty", 250)
```

The 3rd and 4th optional arguments are for *makedefault* and *reset* respectively.

Now, let's begin looking at what each of the built-in settings does, starting from the most obvious and moving towards the most subtle.

7.1 Spacing Settings

In our \note example, we saw the setting `document.lskip`. This is a *glue* parameter which is added to the left side of every line. Setting this to a positive length effectively increases the left margin of the text. Similarly, `document.rskip` adds some space to the right side of every line.

Note that these skip settings are not the same as page margins. The `document.lskip` and `document.rskip` values are applied inside of the current frame and are relative to the edge of the frame, not to the edge of the page. They are best used for temporary adjustments to the margins relative to the normal margins, such as to indent a pull-quote. They can also be negative, pulling the effective margin outside of the current frame.

Glue

A glue parameter is slightly different from an ordinary dimensioned length. Glue basically means 'space,' but as well as signifying a length, it also has two additional optional components: stretch and shrink, specified as `<dimension>` *plus* `<dimension>` *minus* `<dimension>`*. The first dimension is the basic length; the stretch is the maximum length that can be added to it, and the shrink is some length that can be taken away from it. For instance,* `12pt plus 6pt`

> `minus 3pt` *specifies a space that would ideally by 12 points, but can expand or contract from a minimum of 9 points to a maximum of 18 points.*

Let's think about how the `centering` environment is implemented. First, we will add incredibly stretchable glue to the left and right margins, like so:

```
\set[parameter=document.lskip,value=0pt plus 100000pt]
\set[parameter=document.rskip,value=0pt plus 100000pt]
```

This produces the following:

Here is some text which is almost centered. However, there are three problems: first, the normal paragraph indentation is applied, meaning the first line of text is indented. Second, the space between words is stretchable, meaning that the lines are stretched out so they almost seem justified. Finally, by default SILE adds very large glue at the end of each paragraph so that when the text is justified, the spacing of the last line is not stretched out of proportion. This makes the centering of the last line look a bit odd. We will deal with these three issues in the following paragraphs.

The indentation at the start of each paragraph is controlled by the setting `document.parindent`; this is a glue parameter, and by default it's set to 20pt with no stretch and shrink. Actually, the amount added to the start of the paragraph is `current.parindent`. After each paragraph, `current.parindent` is reset to the value of `document.parindent`. The `\noindent` command works by setting `current.parindent` to zero.

How would you make a paragraph like this with a 'hanging' indentation? We've set the `document.lskip` to 20 points, and the `current.parindent` to *minus* 20 points. (In other words, we called: `\set[parameter=document.lskip,value=20pt]` and `\set[parameter=current.parindent, value=-20pt]`.)

The space between *paragraphs* is set with the glue parameter `document.parskip`. It's normally set to five points with one point of stretchability.

7.1.1 Line spacing settings

As we mentioned in the section on grid typesetting, the rules for spacing between *lines* within a paragraph is determined by two rules. Let's reiterate those rules now in terms of settings:

• SILE tries to insert space between two successive lines to make their baselines exactly `document.baselineskip` apart.

• If this first rule would mean that the bottom and the top of the lines are less than `document.lineskip` apart, then they are forced to be `document.lineskip` apart.

This linebreaking method is fiddly, and book designers may prefer to work with the tools provided by the `linespacing` *package.*

7.1.2 Word spacing settings

There are multiple ways of defining the space between words. By default, the space between words is determined by the width of the space character in the current font. To help with justifying the text, the spaces are shrinkable and stretchable. Specifically, if the width of a space in the current font settings is `<space>`, then the width of the space between words is `<shaper.spaceenlargementfactor> *` `<space> plus <shaper.spacestretchfactor> * <space> minus <shaper.spaceshrinkfactor> * <space>`. The default values of these settings make the space width `1.2 <space> plus 0.5 <space> minus 0.333` `<space>`.

If you want to set the word space width explicitly, you can set the `document.spaceskip` setting. You will also need to turn *off* the setting `shaper.variablespaces`, which allows the width of a space to vary based on context (otherwise known as "space kerning"). If you want to go back to the default (measuring the space character of the font), then you need to turn on `shaper.variablespaces` (set it to a true value) and also *unset* the setting `document.spaceskip`. To unset it, just call `\set` with no `value` parameter: `\set[parameter=document.spaceskip]`.

7.1.3 Letter spacing settings

You can also put spaces in between *letters* with the `document.letterspaceglue` setting.

This paragraph is set with `document.letterspaceglue` set to `0pt plus 4pt`, which allows the typesetter to insert tiny bits of spacing between the letters to improve the fitting of the paragraph, even though it would prefer to keep the letterspacing at zero points if possible. (Letter spacing is not considered a preferable way to solve justification problems.)

This paragraph is set with `document.letterspaceglue` set to `0.3pt`, which *forces* the typesetter to insert tiny bits of spacing between the letters. Frederic Goudy is credited with saying that anyone who would letterspace lowercase would steal sheep.[1]

7.2 Typesetter settings

The settings which affect SILE's spacing controls have the most obvious effect on a document; the typesetter itself has some knobs that can be twiddled:

`typesetter.widowpenalty` and `typesetter.orphanpenalty`[2] affect how strongly SILE is averse to leaving stray lines at the start and end of pages. A *widow* happens when a page is broken leaving one line at the bottom of a page; an *orphan* line is the last line in a paragraph broken off at the top of the page. By default, the *penalty* attached to breaking the page at one of these places is 150 penalty points. This value can be any number up to `10000`, which means "never break at this point."

SILE automatically inserts a piece of massively stretchable glue at the end of each paragraph; without this, the justification algorithm would apply justification to the entire paragraph, including the last line, and produce a fully justified paragraph. (Normally we want the last line of a justified paragraph to be left-aligned.) The size of this glue is defined in the setting `typesetter.parfillskip`. Its default value is `0pt plus 10000pt` but for this current paragraph, we have unset it.

Now we can finally complete our implementation of centering:

```
\set[parameter=document.lskip,value=0pt plus 100000pt]
\set[parameter=document.rskip,value=0pt plus 100000pt]
\set[parameter=document.spaceskip,value=0.5en]
\set[parameter=current.parindent,value=0pt]
\set[parameter=document.parindent,value=0pt]
\set[parameter=typesetter.parfillskip,value=0pt]
```

And this is (more or less) how the `center` environment is defined in the plain class: we make the margins able to expand but the spaces not able to expand; we turn off indenting at the start of the paragraph, and we turn off the filling glue at the end of the paragraph.

7.3 Linebreaking settings

SILE's linebreaking algorithm is lifted entirely from TeX, and so maintains the same level of customizability as TeX. Only the API interfaces and units have been adapted as appropriate. Here is a quick run-down of the settings applicable to the line-breaking algorithm. You are expected to know what you are doing with these.

• `linebreak.tolerance`: How bad a breakpoint is before it is rejected by the algorithm. (Default: 500)

• `linebreak.parShape`: Whether to utilize a callback to `SILE.linebreak:parShape()` to get a customized shape for each line in a paragraph. (Default: false)

• `linebreak.pretolerance`: If there are no breakpoints better than this, the paragraph is considered for hyphenation. (Default: 100)

• `linebreak.hangIndent`: How far to indent initial line(s) of a paragraph. (Default: 0)

• `linebreak.hangAfter`: An integer count of how many lines should have `linebreak.hangIndent` applied. (Default: nil)

• `linebreak.adjdemerits`: Additional demerits which are accumulated in the course of paragraph building when two consecutive lines are visually incompatible. In these cases, one line is built with much space for justification, and the other one with little space. (Default: 10000)

• `linebreak.looseness`: How many lines the current paragraph should be made longer than normal. (Default: 0)

• `linebreak.prevGraf`: The number of lines in the paragraph last added to the vertical list.

• `linebreak.emergencyStretch`: Assumed extra stretchability in lines of a paragraph. (Default: 0)

• `linebreak.linePenalty`: Penalty value associated with each line break. (Default: 10)

• `linebreak.hyphenPenalty`: Penalty associated with break at a hyphen. (Default: 50)

• `linebreak.doubleHyphenDemerits`: Penalty for consecutive lines ending with a hyphen. (Default: 10000)

7.4 Shaper settings

As well as the settings for varying word space (see above), there is one additional option which affects the shaping of text.[3] The default shaping engine, Harfbuzz, can actually call out to other shaping engines instead of doing the shaping itself. SILE provides an interface (through the `harfbuzz. subshapers` setting) to select the shaping engine in use. To get a list of the subshapers enabled in your build of Harfbuzz, run `sile --debug=versions` on any file:

```
% $ ./sile --debug=versions hello.sil
...
Harfbuzz version: 2.4.0
Shapers enabled: graphite2, ot, coretext, coretext_aat, fallback
...
```

If I wanted to test out the OS X's CoreText shaper instead of using Graphite and Harfbuzz's own OpenType shaper, I could set:

```
\set[parameter=harfbuzz.subshapers,value=coretext]
```

This is one of those situations where for 99% of people it isn't useful at all but the other 1% of people will really appreciate it: specifically, if you are designing fonts with complex text layout and you want to check how they will appear on different rendering systems. If that's not you, don't worry about this setting; if it is, you're welcome.

7.5 Settings from Lua

Most of the time you will not be fiddling with these settings at the SILE layer, because complex layout commands are expected to be implemented in Lua. The following SILE functions access the settings system from inside Lua:

• `SILE.settings:set(<parameter>, value)`: sets a setting.

You should note that, while in the SILE layer, the `\set` *command does its best to turn the textual description of a type into the appropriate Lua type for the value.* `SILE.settings:set` *does not do that; it expects the value to be of the appropriate type: lengths need to be a* `SILE.length` *object, glue must be* `SILE.nodefactory.glue` *and so on.*

• `SILE.settings:get(<parameter>)`: retrieves the current setting of the parameter.

• `SILE.settings:temporarily(function)`: Saves all settings, runs the function and then restores all settings afterwards.

• `SILE.settings:declare(<specification>)`: Declares a new setting. See the base settings in `settings.lua` for examples of how to call this. A class or package should namespace its settings with `<package>.<setting>`.

Chapter 8
Multilingual Typesetting

One thing we're pretty proud of is SILE's language support. Typesetting conventions differ both from script to script and from language to language. SILE aims to support quality typesetting across all script and language families. As an open source project we can collaborate on support for locales that commercial systems do not consider worthwhile. We want to make it easy for minority languages and scripts to implement their own typographic conventions.

8.1 Selecting languages

For SILE to know how to typeset text you will need to tell it what language your text is in! There are two ways to do this: as part of the `\font[language=...]` command as detailed in chapter 4, or by use of the `\language[main=...]` command. Both of these expect an ISO639-1 language code such as `en` for English, `ar` for Arabic, and so on.

Selecting a language by either method loads up the *language support* file(s) for that language. These in turn enable various localization and typesetting conventions. Language support may include and combination of:

• hyphenation patterns,

• line breaking and justification schemes,

• frame advance and writing direction,

• spacing,

• choice of glyphs within a font,

• localization of programatially inserted strings,

• and more.

For example, Sindhi and Urdu users will expect the Arabic letter *heh* (ه) to combine with other letters in different ways to standard Arabic shaping. In those cases, you should ensure that you select the appropriate language before processing the text:

```
Standard Arabic:
\font[family=LateefGR,language=ar]{…};
then in Sindi:
\font[family=LateefGR,language=snd]{…};
then in Urdu:
\font[family=LateefGR,language=urd]{…}.
```

Standard Arabic: هيه; then in Sindi: همه; then in Urdu: بہ.

8.2 Direction

SILE is written to be *direction-agnostic*, which means that it has no fixed idea about which way text should flow on a page: Latin scripts are generally written left-to-right with individual lines starting from the top of the page and advancing towards the bottom. Japanese can be written in the same way, but traditionally is typeset down the page with lines of text moving from the right of the page to the left.

To describe this, SILE uses the concept of a *writing direction*, which denotes the way each individual line appears on the page —left to right for Latin scripts, right to left for Arabic, Hebrew and so on, top to bottom for traditional Japanese— and a *page advance direction*, which denotes the way the lines "stack up". Each of these directions can take one of four values: LTR, RTL, TTB and BTT. A *direction specification* is made up of either a writing direction (LTR or RTL), in which case the page advance direction is understood to be TTB, or a writing direction and a page advance direction joined by a hyphen.

Each frame has its own writing direction. By default, this is LTR-TTB. Normally you would set the writing direction once, in the master frames of your document class. One easy way to do this in the plain document class is to pass the direction=... parameter to the \begin{document} command. For example, Mongolian is written top to bottom with text lines moving from the left to the right of the page, so to create a Mongolian document, use:

```
\begin[direction=TTB-LTR]{document}
\font[language=mo,family=Noto Sans Mongolian]
...
\end{document}
```

To change the writing direction for a single frame, use \thisframedirection[direction=...].

SILE uses the Unicode bidirectional algorithm to handle texts written in mixed directionalities. See https://sile-typesetter.org/examples/i18n.sil for an example which brings together multiple scripts and directionalities.

8.3 Hyphenation

SILE hyphenates words based on its current language. (Language is set using the `\font` command above.) SILE comes with support for hyphenating a wide variety of languages, and also aims to encode specific typesetting knowledge about languages.

The default hyphen character is "-", which can be tweaked by the `\font` parameter `hyphenchar`. It accepts a Unicode character or Unicode codepoint in `[Uu]+<code>` or Hexadecimal `0[Xx]<code>` format – for instance `\font[family=Rachana,language=ml,hyphenchar=U+200C]`.

SILE comes with a special "language" called `und`, which has no hyphenation patterns available. If you switch to this language, text will not be hyphenated. The command `\nohyphenation{…}` is provided as a shortcut for `\font[language=und]{…}`.

The hyphenator uses the same algorithm as TeX and can use TeX hyphenation pattern files if they are converted to Lua format. To implement hyphenation for a new language, first check to see if TeX hyphenation directionaries are available; if not, work through the resources at `http://tug.org/docs/liang/`.

8.4 Localization

A small handfull of strings may be programatically added to documents depending on language, context, and options. For example by default in English the `book` class will prepend "Chapter " before chapter numbers output by the `\chapter` command. These localized strings are managed internally using the Fluent localization system. See Project Fluent (`https://projectfluent.org`) for details on the data format and uses. Some default localizations are provided for a handful of languages, but it is quite likely SILE will not (yet) have your language. Even if it does, it may not use the localization of your choice.

All default localizations can be easily overridden and new locales can easily be added in your document or project. Additionally the Fluent localization system is exposed and can be used for your localization purposes.

To set a new value for a message (or messages), simply use the `\ftl` command. The contents passed to the command will be parsed as new messages and loaded in the locale for the current document language. Optionally messages may be loaded into a different locale with `\ftl[locale=<locale>]`. You can also load messages from an external ftl file with `\ftl[src=<filename>]`.

To output a localized message, pass the message id to the `\fluent` command. The current document languages determines the locale used, or a locale option may be passed. Fluent parameters may also be passed as options.

For example a `hello` message is available in SILE, and in an English context such as this manual `\fluent[name=World]{hello}` will output "Hello World!". To get the localization in Turkish, try `\fluent[name=World,locale=tr]{hello}` to get "Merhaba World!". Now lets change the message with

`\ftl[locale=tr]{hello = Selam { $name }!}` and try again. This time `\fluent[name=Dünyalılar,locale=tr]{hello}` will output "Selam Dünyalılar!".

A particularly common string to override might be the table of contents heading:

```
\ftl{toc-heading = Table of Contents}
\tableofcontents
```

8.5 Support for specific languages

The following section shows some of the support features that SILE provides for specific languages apart from hyphenation and language-specific glyph selection:

8.5.1 Amharic

SILE inserts word break opportunities after Ethiopic word spaces and full stops. Amharic can be typeset in two styles: with space surrounding punctuation or space after punctuation. You can set the setting `languages.am.justification` to either `left` or `centered` to control which style is used. The default is `left`

```
\font[family=Noto Sans Ethiopic,language=am]
ሰላም:ዪዑዪ
\set[parameter=languages.am.justification,value=centered]
ሰላም:ዪዑዪ
```

ሰላም: ዪዑዪ

ሰላም : ዪዑዪ

8.5.2 French

In French typesetting, there is normally a non-breakable space between text and "high" punctuation (a thin fixed space before question marks, exclamation marks, semicolons and an inter-word space before colons), and also spaces within "guillemets" (quotation marks). SILE will automatically apply the correct space. The size of these spaces is determined by `languages.fr.thinspace`, `languages.fr.colonspace` and `languages.fr.guillspace`.

8.5.3 Japanese / Chinese

SILE aims to conform with the W3G document "Requirements for Japanese Text Layout" (`https://www.w3.org/TR/jlreq/`) which describes the typographic conventions for Japanese (and also Chinese) text. Breaking rules (kinzoku shori) and intercharacter spacing is fully supported on selecting the

Japanese language. The easiest way to set up the other elements of Japanese typesetting such as the *hanmen* grid and optional vertical typesetting support is by using the `jplain` class.

Japanese documents are traditionally typeset on a grid layout called a `hanmen`, with each character essentially monospaced inside the grid. (It's like writing on graph paper.) The **hanmenkyoshi** package provides tools to Japanese class designers for creating hanmen frames with correctly spaced grids. It also provides the `\show-hanmen` command for debugging the grid.

The name *hanmenkyoshi* is a terrible pun.

The **tate** package provides support for Japanese vertical typesetting. It allows for the definition of vertical-oriented frames, as well as for two specific typesetting techniques required in vertical documents: `\latin-in-tate` typesets its content as Latin text rotated 90 degrees, and `\tate-chu-yoko` places (Latin) text horizontally within a single grid-square of the vertical *hanmen*.

Japanese texts often contain pronunciation hints (called *furigana*) for difficult kanji or foreign words. These hints are traditionally placed either above (in horizontal typesetting) or beside (in vertical typesetting) the word that they explain. The typesetting term for these glosses is *ruby*.

The **ruby** package provides the `\ruby[reading=⟨ruby text⟩]{⟨base text⟩}` command which sets a piece of ruby above or beside the base text. For example:

```
\ruby[reading=れいわ]{令和}
```

Produces:

れいわ
令和

8.5.4 Syllabic languages

SILE implements syllable-based line breaking for Burmese and Javanese text.

8.5.5 Uyghur

Uyghur is the only Arabic script based language which uses hyphenation, and SILE supports hyphenation. Because Arabic fonts aren't normally designed with hyphenation in mind, you may need to tweak some settings to ensure that Uyghur is typeset correctly. As well as choosing the `hyphenchar` (see the hyphenation section above), the setting `languages.ug.hyphenoffset` inserts a space between the text and the hyphen.

Chapter 9
The Nitty Gritty

We are finally at the bottom of our delve into SILE's commands and settings. Here are the basic building blocks out of which all of the other operations in SILE are created.

At this point, it is expected that you are a class designer, and will be able to follow the details of how SILE implements these commands and features; we will also explain how to interact with these components at the Lua level.

9.1 Boxes, Glue and Penalties

SILE's job is, looking at it in very abstract terms, all about arranging little boxes on a page. Some of those boxes have letters in them, and those letters are such-and-such a number of points wide and such-and-such a number of points high; some of the boxes are empty but are there just to take up space; when a horizontal row of boxes has been decided (i.e. when a line break is determined) then the whole row of boxes is put into another box and the vertical list of boxes are then arranged to form a page.

Conceptually, then, SILE knows about a few different basic components: horizontal boxes (such as a letter); horizontal glue (the stretchable, shrinkable space between words); vertical boxes (a line of text); vertical glue (the space between lines and paragraphs); and penalties (information about where and when not to break lines and pages).[1]

The most immediately useful of these are horizontal and vertical glue. It is possible to explicitly add horizontal and vertical glue into SILE's processing stream using the \glue and \skip commands. These take a width and a height parameter respectively, both of which are glue dimensions. So, for instance, the \smallskip command is the equivalent of \skip[height=3pt plus 1pt minus 1pt]; \thinspace is defined as being \glue[width=0.16667em].

Similarly, there is a \penalty command for inserting penalty nodes; \break is defined as \penalty[penalty=-10000] and \nobreak is \penalty[penalty=10000].

You can also create horizontal and vertical boxes from within SILE. One obvious reason for doing so would be to explicitly avoid material being broken up by a page or line break; another reason for doing so would be that once you box some material up, you then know how wide or tall it is. The \hbox and \vbox commands put their contents into a box; when called from Lua, they also *return* the new box.

9.1.1 From Lua

SILE's Lua interface contains a `nodefactory` for creating boxes and glue. Before we get into that, however, you need to know that glue measurements in SILE should always be specified in terms of `SILE.length` objects; these are "three-dimensional" lengths, in that they consist of a base length plus stretch and shrink. To construct a `SILE.length`:

```
local l = SILE.length({ length = x, stretch = y, shrink = z})
```

Now we can construct horizontal and vertical glue:

```
local glue  = SILE.nodefactory.glue({ width =  l})
local vglue = SILE.nodefactory.vglue({ height = l})
```

SILE's typesetting is organised by the `SILE.typesetter` object; it maintains two queues of material that it is still working on: the node queue the output queue. Material in these queues is content that has been parsed but not yet rendered to the canvas and can still be manipulated. The node queue (`SILE.typesetter.state.nodes`) contains new horizontal boxes and glue that have not yet been broken up into lines. The output queue (`SILE.typesetter.state.outputQueue`) consists of vertical material (lines) which have not yet been broken up into pages. Line breaking and page breaking happen when the typesetter moves between horizontal and vertical mode.

As new content is parsed is is added to the node queue in as small chunks as possible—chunks that must remain together no matter where they end up on a line. For example this might individual symbols, sylables, or objects such as images. As soon as new content which requires a vertical break is encountered, the node queue is processed to derivce any missing shaping information about each node, then the sequence of node is broken up into lines. Once all the "horizontal mode" nodes are broken into lines and those lines are added to the output queue, the other new vertical content can be processed. At any point you can force the current queue of horizantal content (the node queue) to be shaped into lines and added to the vertical output queue by calling the function `SILE.typesetter:leaveHmode()`. This is a handy move for writing custom functions, but it is a fairly low level control (i.e. it is unlikely to be useful while writing a document). A related but higher level command, \par, is more frequently used when writing a document and embeded in the content. The

\par command first calls `SILE.typesetter:leaveHmode()`, then inserts a vertical skip according to the `document.parskip` setting, then goes on to reset a number of settings that are typically paragraph related such as hanging indents.

When writing a custom command, if you want to manually add a vertical space to the output first ensure that material in the current paragraph has been all properly boxed-up and moved onto the output queue by calling `SILE.typesetter:leaveHmode()`; then add your desired glue to the output queue. This is exactly what the `\skip` and similar commands do.

Adding boxes and glue to the typesetter's queues is such a common operation that the typesetter has some utility methods to construct the nodes and add them for you:

```
SILE.typesetter:leaveHmode()
SILE.typesetter:pushVglue({ height = l })
```

Adding boxes yourself is a little more complicated, because boxes need to know how to display themselves on the page. To facilitate this, they normally store a `value` and an `outputYourself` member function. For instance, the `image` package actually does something very simple; it adds a horizontal box to the node queue which knows the width and height of the image, the source, and instructions to the output engine to display the image:

```
SILE.typesetter:pushHbox({
  width= …,
  height= …,
  depth= 0,
  value= options.src,
  outputYourself= function (this, typesetter, line)
    SILE.outputter.drawImage(this.value,
      typesetter.frame.state.cursorX, typesetter.frame.state.cursorY-this.height,
      this.width,this.height
    );
    typesetter.frame:advanceWritingDirection(this.width)
end});
```

Adding horizontal and vertical penalties to the typesetter's queues is similarly done with the `SILE.typesetter:pushPenalty({penalty = x})` and `SILE.typesetter:pushVpenalty({penalty = y})` methods.

9.1.2 Kerns

\kern's are a type of \glue. The difference between the two is that, while a \glue can be broken

at the end of a line, a \kern can't be. Hearkening back to our $S_I{}^E$ example from the *Macros and Commands* chapter, consider that example, repeated enough times to cause a linebreak, but with \glue's everywhere \kern's are used instead:

$S_I{}^E S_I{}^E$ $S_I{}^L$
^{E}$S_I{}^E S_I{}^L$
$^E S_I{}^E S_I{}^E S_I{}^E S_I{}^E S_I{}^E$

The 27th of the 60 $S_I{}^E$'s, in red, is broken between its 'L' and '^{E}'. Meanwhile, if we typeset the same line, except using \kern's as we had originally instead...

$S_I{}^E S_I{}^E$ $S_I{}^L{}^E$$S_I{}^E S_I{}^E S_I{}$

The line just continues on right off the page. Why this is a useful feature is more obvious if there are spaces between them:

Glues:

$S_I{}^E\ S_I{}^E$ $S_I{}^L$
^{E} $S_I{}^E\ S_I{}^E$
$S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E$

Kerns:

$S_I{}^E\ S_I{}^E$ $S_I{}^E$
$S_I{}^E\ S_I{}^E$
$S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E\ S_I{}^E$

9.2 Frames

As we have previously mentioned, SILE arranges text into frames on the page. The overall layout

of a page, including the apparent margins between content and the page edge and other content regions, is controlled by defining the position of the frame or frames into which the content will be flowed.

Normally those frames are defined by your document class, but you can actually create your own frames on a per-page basis using the \pagetemplate and \frame commands. There are very few situations in which you will actually want to do this, but if you can understand this, it will help you to understand how to define your own document classes.

For instance, in a couple of pages time we're going to implement a two-column layout. SILE uses a *constraint solver* system to declare its frames, which means that you can tell it how the frames relate to each other and it will compute where the frames should be physically placed on the page.

Here is how we will go about it. We need to start with a page break, because SILE will not appreciate you changing the page layout after it's started to determine how to put text onto that page.[2] How do we get to the start of a new page? Remember that the \eject (another word for \break in vertical mode) only adds a penalty to the end of the output queue; page breaking is triggered when we leave horizontal mode, and the way to do that is \par. So we start with \eject\par and then we will begin a \pagetemplate. Within \pagetemplate we need to tell SILE which frame to begin typesetting onto.

```
\eject\par
\begin[first-content-frame=leftCol]{pagetemplate}
```

Now we will declare our columns. But we're actually going to start by declaring the gutter first, because that's something that we know and can define; we're going to stipulate that the gutter width will be 3% of the page width:

```
\frame[id=gutter,width=3%pw]
```

Declarations of frame dimensions are like ordinary SILE <dimension>*s, except with three additional features:*

• You can refer to properties of other frames using the `top()`, `bottom()`, `left()`, `right()`, `height()` *and* `width()` *functions. These functions take a frame ID. SILE magically pre-defines the frame* page *to allow you to access the dimensions of the whole page.*

• You can use arithmetic functions: plus, minus, divide, multiply, and parenthesis have their ordinary arithmetic meaning. To declare that frame b *should be half the height of frame* a *plus 5 millimeters, you can say* `height=5mm + (height(b) / 2)`. *However, as we will see later, it is usually better to structure your declarations to let SILE make those kind of computations for you.*

• Since book design is often specified in terms of proportion of a page, you can use the shortcut `width=5%pw` *instead of* `width=0.05 * width(page)` *and* `height=50%ph` *instead of* `height=0.5 * height(page)`.

Next we declare the left and right column frames. The book class gives us some frames already, one of which, content, defines a typeblock with a decent size and positioning on the page. We will use the boundaries of this frame to declare our columns: the left margin of the left column is the left margin of the typeblock; the right margin of the right column is the right margin of the typeblock. But we also want a few other parameters to ensure that:

• the gutter is placed between our two columns

• the two columns have the same width (We don't know what that width is, but SILE will work it out for us.)

• after the left column is full, typesetting should move to the right column.

```
\frame[id=leftCol, left=left(content), right=left(gutter),
       top=top(content), bottom=bottom(content),
       next=rightCol]
\frame[id=rightCol, left=right(gutter), right=right(content),
       top=top(content), bottom=bottom(content),
       width=width(leftCol)]
```

And now finally we can end our pagetemplate.

```
\end{pagetemplate}
```

Let's do it.

leftCol

So there we have it: a two-column page layout.

In the next chapter we'll use the knowledge of how to declare frames to help us to create our own document class files. In the meantime, here is some dummy text to demonstrate the fact that text does indeed flow between the two columns naturally:

lorem ipsum dolor sit amet consetetur sadipscing elitr sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat sed diam voluptua at vero eos et accusam et justo duo dolores et ea rebum stet clita kasd gubergren no sea takimata sanctus est lorem ipsum dolor sit amet lorem ipsum dolor sit amet consetetur sadipscing elitr sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat sed diam voluptua at vero eos et accusam et justo duo dolores et ea rebum stet clita kasd gubergren no sea takimata sanctus est lorem ipsum dolor sit amet lorem ipsum dolor sit amet consetetur sadipscing elitr sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat sed diam voluptua at vero eos et accusam et justo duo dolores et ea rebum stet clita kasd gubergren no sea takimata sanctus est lorem ipsum dolor sit amet

duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat vel illum dolore eu feugiat nulla facilisis at vero eros et accusan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi lorem ipsum dolor sit amet consectetuer adipiscing elit sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat

ut wisi enim ad minim veniam quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat duis autem

rightCol

vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi

nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum lorem ipsum dolor sit amet consectetuer adipiscing elit sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat ut wisi enim ad minim veniam quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat

duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat vel illum dolore eu feugiat nulla facilisis

at vero eos et accusam et justo duo dolores et ea rebum stet clita kasd gubergren no sea takimata sanctus est lorem ipsum dolor sit amet lorem ipsum dolor sit amet consetetur sadipscing elitr sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat sed diam voluptua at vero eos et accusam et justo duo dolores et ea rebum stet clita kasd gubergren no sea takimata sanctus est lorem ipsum dolor sit amet lorem ipsum dolor sit amet consetetur sadipscing elitr at accusam aliquyam diam diam dolore dolores duo eirmod eos erat et nonumy sed tempor et et invidunt justo labore stet clita ea et gubergren kasd magna no rebum sanctus sea sed takimata ut vero voluptua est lorem ipsum dolor sit amet lorem ipsum dolor sit amet consetetur sadipscing elitr sed diam nonumy eirmod tempor invidunt ut labore

Chapter 10
Designing Basic Class Files

Now we know how to define a frame layout for a single page, let's try to define one for an entire document.

Document classes are Lua files, and live somewhere in the `classes/` subdirectory of either where your input file is located, your current working directory or your SILE path (typically `/usr/local/share/sile`). We're going to create a simple class file which merely changes the size of the margins and the typeblock. We'll call it `bringhurst.lua`, because it replicates the layout of the Hartley & Marks edition of Robert Bringhurst's *The Elements of Typographical Style.*

We are designing a book-like class, and so we will inherit from SILE's standard `book` class found in `classes/book.lua`. Let's briefly have a look at `book.lua` to see how it works. First, a table is populated with a description of the default frameset.

```
book.defaultFrameset = {
  content = {
    left = "8.3%pw",
    right = "86%pw",
    top = "11.6%ph",
    bottom = "top(footnotes)"
  },
  folio = {
    left = "left(content)",
    right = "right(content)",
    top = "bottom(footnotes)+3%ph",
    bottom = "bottom(footnotes)+5%ph"
  },
  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"
  }
}
```

So there are four frames directly declared. The first is the content frame, which by SILE convention is called content. Directly abutting the content frame at the bottom is the footnotes frame. The top of the typeblock and the bottom of the footnote frame have fixed positions, but the boundary between typeblock and footnote is variable. Initially the height of the footnotes is zero (and so the typeblock takes up the full height of the page) but as footnotes are inserted into the footnote frame its height will be adjusted; its bottom is fixed and therefore its top will be adjusted, and the bottom of the main typeblock frame will also be correspondingly adjusted. The folio frame (which holds the page number) lives below the footnotes, and the running headers live above the content frame.

Normally, as in the plain class and anything inheriting from it, this would be enough to populate the pages' frameset. Instead the book class includes its own extension to the class with a callback init function. We see that it loads the masters package and generates a master frameset using the above defined default frameset.

```
function book:_init (options)
  self:loadPackage("masters")
  self:defineMaster({
      id = "right",
      firstContentFrame = self.firstContentFrame,
      frames = self.defaultFrameset
    })
  ...
  plain._init(self, options)
  return self
end
```

Next, we use the twoside package to mirror our right-page master into a left-page master:

```
  self:loadPackage("twoside", { oddPageMaster = "right", evenPageMaster = "left" })
  self:mirrorMaster("right", "left")
```

The book class also loads the table of contents package which sets up commands for sectioning, and declares various things that need to be done at the start and end of each page. Since we will be

inheriting from the book class, we will have all these definitions already available to us. All we need to do is set up our new class, and then define what is different about it. Here is how we set up the inheritance:

```
local book = require("classes.book")
local bringhurst = pl.class(book)
bringhurst._name = "bringhurst"
...
return bringhurst
```

Now we need to define our frame masters.

The LaTeX memoir class' *A Few Notes On Book Design* tells us that Bringhurst's book has a spine margin one thirteenth of the page width; a top margin eight-fifths of the spine margin; a front margin and bottom margin both sixteen-fifths of the spine margin. We can define this in SILE terms like so:

```
bringhurst.defaultFrameset = {
  content = {
    left = "width(page) / 13",
    top = "width(page) / 8",
    right = "width(page) * .75",
    bottom = "top(footnotes)"
  },
  folio = book.defaultFrameset.folio,
  runningHead = {
    left = "left(content)",
    right = "right(content)",
    top = "top(content) / 2",
    bottom = "top(content) * .75"
  },
  footnotes = book.defaultFrameset.footnotes
}
```

Note that we've deliberately copied the frame definitions for the folio and footnote frames from the book class, but if we had tried to reuse the runningHead frame definition it would have been too high because the typeblock is higher on the page than the standard book class, and the running heads are defined relative to them. So, we needed to change the definition the running header frame to bring them down a bit lower.

If all we want to do in our new class is to create a different page shape, this is all we need. The init function inherited from the book class will take care of setting these frames up with mirrored masters.

If we had wanted to load additionall packages into our class as, say, the `bible` class does, we would need to define our own init function and call our parent class's init function as well. For example to load the `infonode` package into our class, we could add this init function:

```
function bringhurst:_init(options)
  book._init(self, options)
  self:loadPackage("infonode")
  return self
end
```

Note that the official SILE classes have some extra tooling to handle legacy class models trying to inherit from them. You don't need those deprecation shims if your own classes when following these examples.

10.1 Defining Commands

However, it's usually the case that a class will want to do more than just change the page shape; a class will typically want to do some other things as well: define additional commands, alter the output routine, store and investigate bits of state information, and so on. We'll look briefly at some of the principles involved in those things here, and in the next chapters will run through some worked examples.

To define your own command at the Lua level, you use the `SILE.registerCommand` function. It takes three parameters: a command name, a function to implement the command, and some help text. The signature of a function representing a SILE command is fixed: you need to take two parameters, `options` and `content` (of course you can name your parameters whatever you like, but these are the most common names). Both of these parameters are Lua tables. The `options` parameter contains the command's parameters or XML attributes as a key-value table, and the `content` is an abstract syntax tree reflecting the input being currently processed.

So in the case of `\mycommand[size=12pt]{Hello \break world}`, the first parameter will contain the table `{size = "12pt"}` and the second parameter will contain the table:

```
{
  "Hello ",
  {
```

```
    attr = {},
    id = "command",
    pos = 8,
    tag = "break"
  },
  " world"
}
```

Most commands will find themselves doing something with the `options` and/or calling `SILE.process(content)` to recursively process the argument. Here's a very simple example; an XML `<link>` tag may take an XLink `xl:href` attribute[1] We want to render `<link xl:href="http://...">` `Hello</link>` as Hello (`http://...`). So, first we need to render the content, and then we need to do something with the attribute:

```
bringhurst.registerCommands = function (self)
  book.registerCommands(self)
  SILE.registerCommand("link", function(options, content)
    SILE.process(content)
    if (options["xl:href"]) then
     SILE.typesetter:typeset(" (")
     SILE.call("code", {}, {options["xl:href"]})
     SILE.typesetter:typeset(")")
    end
  end)
end
```

We use the usual `SILE.typesetter:typeset` and `SILE.call` functions to output text and call other commands.

If you do need to do something with a dimension, you can use `SILE.toPoints` to parse a basic length and `SILE.parseComplexFrameDimension` to parse a frame dimension, and turn them into points.

10.2 Output Routines

As well as defining frames and packages, different classes also alter the way that SILE performs its output—what it should do at the start or end of a page, for instance, which controls things like swapping between different master frames, displaying page numbers, and so on.

The key methods for defining the *output routine* are:

• newPar and endPar are called at the start and end of each paragraph.

• newPage and endPage are called at the start and each of each page.

• init and finish are called at the start and end of the document.

Once again this is done in an object-oriented way, with derived classes overriding their superclass' methods where necessary.

Some packages may provide functions that need to be run as part of the class output routines. One way packages can accomplish this is registering hook functions that get run an known locations in the provided classes. Class authors may also provide their own hook locations for packages, or run any set of registered hooks in their own outputs. The other way is to export functions that get added to the class itself and can be run manually from the class.

For examples check out the **tableofcontents** package for the hooks it sets, but also the \tocentry command it registeres that gets called manually in the book class.

Let's demonstrate roughly how the tableofcontents package works. We'll be using the infonodes package to collect the information about which pages contain table of content items.

First, we set up our infonodes by creating a command that can be called by sectioning commands. In other words, \chapter, \section, etc. should call \tocentry to store the page reference for this section.

```
SILE.registerCommand("tocentry", function (options, content)
  SILE.call("info", {
    category = "toc",
    value = {
      label = content, level = (options.level or 1)
    }
  })
end)
```

Infonodes work on a per-page basis, so if we want to save them throughout the whole document, at the end of each page we need to move them from the per-page table to our own table. We also need to make sure we store their page numbers!

SILE provides the `SILE.scratch` *variable for you to store global information in; you should use a portion of this table namespaced to your class or package.*

So, here is a routine we can call at the end of each page to move the TOC nodes.

```
SILE.scratch.tableofcontents = { }
-- Gather the tocentries into a big document-wide TOC
local moveNodes = function(self)
  local n = SILE.scratch.info.thispage.toc
  for i=1,#n do
    n[i].pageno = class:formatCounter(SILE.scratch.counters.folio)
    table.insert(SILE.scratch.tableofcontents, n[i])
  end
end
```

We're going to take the LaTeX approach of storing these items out as a separate file, then loading them back in again when typesetting the TOC. So at the end of the document, we serialize the `SILE.scratch.tableofcontents` table to disk. Here is a function to be called by the `finish` output routine.

```
local writeToc = function (self)
  local t = pl.pretty.write(SILE.scratch.tableofcontents)
  saveFile(t, SILE.masterFilename .. '.toc')
end
```

And then the `\tableofcontents` command reads that file if it is present, and typesets the TOC nodes appropriately:

```
SILE.registerCommand("tableofcontents", function (options, content)
  local toc = loadFile(SILE.masterFilename .. '.toc')
```

```
  if not toc then
    SILE.call("tableofcontents:notocmessage")
    return
  end
  SILE.call("tableofcontents:header")
  for i = 1,#toc do
    local item = toc[i]
    SILE.call("tableofcontents:item", {level = item.level, pageno = item.pageno},
item.label)
  end
end)
```

And the job is done. Well, nearly. The `tableofcontents` package now contains a couple of functions –`moveNodes` and `writeToc`– that need to be called at various points in the output routine of a class which uses this package. How do we do that?

10.3 Exports

Packages which are primarily used for providing functionality to other classes and packages need a way of supplying these composible bits of functionality to the code which is going to use them. This is called the *export mechanism.*

As well as defining commands, each package may also return a Lua table consisting of two entries, `init` and `exports`.

The `init` function allows you to perform some initialization actions, optionally based on arguments supplied by the loading class. When the package is loaded with `class:loadPackage(package, args)`, the initializer is called with two arguments, `class` and `args`. For instance, the `twoside` package receives information about the IDs of the main right and left master frames so that it can set up the code to switch masters on page change. In our case, we will want to ensure that the `infonode` package is loaded into our caller:

```
return {
  init = function (class)
    class:loadPackage("infonode")
  end,
```

The other entry to be returned from the package is `exports`, which contains names and functions to be mixed into the class namespace. In other words, after:

```
  exports = { writeToc = writeToc, moveTocNodes = moveNodes }
}
```

any class which loads `tableofcontents` can call `self:writeToc()` and `self:moveTocNodes()` (note that we renamed this function when exporting it). It is the class's responsibility for calling these methods at the appropriate point into the output routine.

Chapter 11
Advanced Class Files 1: SILE As An XML Processor

Now we are ready to look at a worked example of writing a class to turn an arbitrary XML format into a PDF file. We'll be looking at the DocBook processor that ships with SILE. DocBook is an XML format for structured technical documentation. DocBook itself doesn't encode any presentation information about how its various tags should be rendered on a page, and so we shall have to make all the presentation decisions for ourself.

The first thing you should know is that it makes your life significantly easier if you consider writing the class file in *two* files; the first being a SILE file in TeX format, and the second as Lua code. This allows you to dispose of all the easy jobs in a convenient format, and then deal with the hard jobs in Lua. When you use the `-I classname.sil` command line option to SILE, SILE first looks for `classname.sil` and uses that as a wrapper around your file to be processed. If that then begins

```
\begin[class=classname]{document}
```

then SILE will also load up `classes/classname.lua` as normal.

Now we can start defining SILE commands to render XML elements. Most of these are fairly straightforward so we will not dwell on them too much. For instance, DocBook has a tags like `<code>`, `<filename>`, `<guimenu>` which should all be rendered in a monospaced typewriter font. To make it easier to customize the class, we abstract out the font change into a single command:

```
\define[command=docbook-ttfont]{\font[family=Inconsolata,size=2ex]{\process}}
```

Now we can define our `<code>` (etc.) tags:

```
\define[command=code]{\docbook-ttfont{\process}}
\define[command=filename]{\docbook-ttfont{\process}}
\define[command=guimenu]{\docbook-ttfont{\process}}
\define[command=guilabel]{\docbook-ttfont{\process}}
\define[command=guibutton]{\docbook-ttfont{\process}}
\define[command=computeroutput]{\docbook-ttfont{\process}}
```

If an end user wants things to look different, they can redefine the `docbook-ttfont` command and get a different font.

11.1 Handling Titles

So much for simple tags. Things get interesting when there is a mismatch between the simple format of SILE macros and the complexity of DocBook markup.

We have already seen an example of the `<link>` tag where we also need to process XML attributes, so we will not repeat that here. Let's look at another area of complexity: the apparently-simple `<title>` tag. The reason this is complex is that it occurs in different contexts, sometimes more than once within a context; it should often be rendered differently in different contexts. So for instance `<article><title>`... will look different to `<section><title>`.... Inside an `<example>` tag, the title will be preferenced by an example number; inside a `<bibliomixed>` bibliography entry the title should not be set off as a new block but should be part of the running text, and so on.

What we will do to deal with this situation is provide a very simple definition of `<title>`, but when processing the containing elements of `<title>` (such as `<article>`, `<example>`), we will process the title ourselves.

For instance, let's look at `<example>`, which has the added complexity of needed to keep track of an article number.

```
SILE.registerCommand("example", function(options,content)
  SILE.call("increment-counter", {id="Example"})
  SILE.call("bigskip")
  SILE.call("docbook-line")
  SILE.call("docbook-titling", {}, function()
    SILE.typesetter:typeset("Example".." " ..
class:formatCounter(SILE.scratch.counters.Example]))
```

`\docbook-line` is a command that we've defined in the `docbook.sil` macros file to draw a line across the width of the page to set off examples and so on. `\docbook-titling` is a command similarly defined in `docbook.sil` which sets the default font for titling and headers; once again, if people want to customize the look of the output we make it easier for them by giving them simple, compartmentalized commands to override.

So far so good, but how do we extract the `<title>` tag from the `content` abstract syntax tree? SILE does not provide XPath or CSS-style selectors to locate content form within the DOM tree;[1] instead there is a simple one-level function called `SILE.findInTree` which looks for a particular tag or command name within the immediate children of the current tree:

```
local t = SILE.findInTree(content, "title")
```

```
  if t then
    SILE.typesetter:typeset(": ")
    SILE.process(t)
```

We've output $\text{Example } 123$ so far, and now we need to say : *Title*. But we also need to ensure that the `<title>` tag doesn't get processed again when we process the content of the example:

```
    docbook.wipe(t)
```

`docbook.wipe` is a little helper function which nullifies all the content of a Lua table:

```
function docbook.wipe(tbl)
  while((#tbl) > 0) do tbl[#tbl] = nil end
end
```

Let's finish off the `<example>` example by skipping a bit between the title and the content, processing the content and drawing a final line across the page:

```
    end
  end)
  SILE.call("smallskip")
  SILE.process(content)
  SILE.call("docbook-line")
  SILE.call("bigskip")
```

Now it happens that the `<example>`, `<table>` and `<figure>` tags are pretty equivalent: they produce numbered titles and then go on to process their content. So in reality we actually define an abstract `countedThing` method and define these commands in terms of that.

11.2 Sectioning

DocBook sectioning is a little different to the SILE book class. `<section>` tags can be nested; to start a subsection, you place another `<section>` tag inside the current `<section>`. So in order to know what level we are currently on, we need a stack; we also need to keep track of what section number we are on at *each* level. For instance:

```
<section><title>A</title> : 1. A
  <section><title>B</title>: 1.1 B
  </section>
  <section><title>C</title>: 1.2 C
    <section><title>D</title>: 1.2.1 D
    </section>
  </section>
  <section><title>E</title>: 1.3 E
```

95

```
    </section>
  <section><title>F</title>:  2. F
```

So, we will keep two variables: the current level, and the counters for all of the levels so far. Each time we enter a `section`, we increase the current level counter:

```
SILE.registerCommand("section", function (options, content)
  SILE.scratch.docbook.seclevel = SILE.scratch.docbook.seclevel + 1
```

We also increment the count at the current level, while at the same time wiping out any counts we have for levels above the current level (if we didn't do that, then E in our example above would be marked 1.3.1):

```
  SILE.scratch.docbook.seccount[SILE.scratch.docbook.seclevel] =
    (SILE.scratch.docbook.seccount[SILE.scratch.docbook.seclevel] or 0) + 1
  while #(SILE.scratch.docbook.seccount) > SILE.scratch.docbook.seclevel do
    SILE.scratch.docbook.seccount[#(SILE.scratch.docbook.seccount)] = nil
  end
```

Now we find the title, and prefix it by the concatenation of all the `seccounts`:

```
  local title = SILE.findInTree(content, "title")
  local number = table.concat(SILE.scratch.docbook.seccount, '.')
  if title then
    SILE.call("docbook-section-"..SILE.scratch.docbook.seclevel.."-title",
{},function()
      SILE.typesetter:typeset(number.." ")
      SILE.process(title)
    end)
    docbook.wipe(title)
  end
```

Finally we can process the content of the tag, and decrease the level count as we leave the `</section>` tag:

```
  SILE.process(content)
  SILE.scratch.docbook.seclevel = SILE.scratch.docbook.seclevel - 1
end)
```

11.3 Other Features

SILE's DocBook implementation is a work in progress, and there is more that can be done. For instance, there is a basic implementation of lists, which equally need to be able to handle nesting; we

implement another stack to take care of the type of list and list counter at each level of nesting.

How would you handle a tag like `<xref>` which renders a cross-reference to another part of the document? For instance, `<xref linkend="ch02"/>` should generate something like Chapter 2, "The Second Chapter". This is another problem which can be handled using the `infonode` package to collect and store cross-reference information about chapter numbers and titles.

Chapter 12
Further Tricks

We'll conclude our tour of SILE by looking at some tricky situations which require further programming.

12.1 Parallel Text

The example `https://sile-typesetter.org/examples/parallel.sil` contains a rendering of chapter 1 of Matthew's Gospel in English and Greek. It uses the `diglot` class to align the two texts side-by-side. `diglot` provides the `\left` and `\right` commands to start entering text on the left column or the right column respectively, and the `\sync` command to ensure that the two columns are in sync with each other. It's an instructive example of what can be done in a SILE class, so we will take it apart and see how it works.

The key thing to note is that the SILE typesetter is an object (in the object-oriented programming sense). Normally, it's a singleton object—i.e. one typesetter is used for typesetting everything in a document. But there's no reason why we can't have more than one. In fact, for typesetting parallel texts, the simplest way to do things is to have two typesetters, one for each column, and have them communicate with each other at various points in the operation.

Let's begin `diglot.lua` as usual by setting up the class and declaring our frames:

```lua
local plain = require("classes.plain");
local diglot = pl.class(plain)
diglot._name = "diglot"

function diglot:_init (options)
  plain._init(self, options)
  self:loadPackage("counters")
    SILE.scratch.counters.folio = { value = 1, display = "arabic" };
    diglot:declareFrame("a",    {left = "8.3%pw", right = "48%pw",
              top = "11.6%ph", bottom = "80%ph" });
    diglot:declareFrame("b",    {left = "52%pw",  right = "100% - left(a)",
              top = "top(a)", bottom = "bottom(a)" });
    diglot:declareFrame("folio",{left = "left(a)", right = "right(b)",
              top = "bottom(a)+3%ph",bottom = "bottom(a)+8%ph" });
  return self
end
```

Now we create two new typesetters, one for each column, and we tell each one how to find the other:

```
diglot.leftTypesetter = SILE.defaultTypesetter {}
diglot.rightTypesetter = SILE.defaultTypesetter {}
diglot.rightTypesetter.other = diglot.leftTypesetter
diglot.leftTypesetter.other = diglot.rightTypesetter
```

Each column needs its own font, so we provide commands to store this information. The \leftfont and \rightfont macros simply store their options to be passed to the \font command every time \left and \right are called. (Because the fonts are controlled by global settings rather than being typesetter-specific.)

```
SILE.registerCommand("leftfont", function(options, content)
  SILE.scratch.diglot.leftfont = options
end, "Set the font for the left side")

SILE.registerCommand("rightfont", function(options, content)
  SILE.scratch.diglot.rightfont = options
end, "Set the font for the right side")
```

Next come the commands for sending text to the appropriate typesetter. The current typesetter object used by the system is stored in the variable SILE.typesetter; many commands and packages call methods on this variable, so we need to ensure that this is set to the typesetter object that we want to use. We also want to turn off paragraph detection, as we will be handling the paragraphing manually using the \sync command:

```
SILE.registerCommand("left", function(options, content)
  SILE.settings:set("typesetter.parseppattern", -1)
  SILE.typesetter = diglot.leftTypesetter;
  SILE.call("font", SILE.scratch.diglot.leftfont, {})
end, "Begin entering text on the left side")

SILE.registerCommand("right", function(options, content)
  SILE.settings:set("typesetter.parseppattern", -1)
  SILE.typesetter = diglot.rightTypesetter;
  SILE.call("font", SILE.scratch.diglot.rightfont, {})
end, "Begin entering text on the right side")
```

The meat of the diglot package comes in the sync command, which ensures that the two typesetters are aligned. Every time we call sync, we want to ensure that they are both at the same position on the page. In other words, if the left typesetter has gone further down the page than the right one, we need to insert some blank space onto the right typesetter's output queue to get them back in sync, and vice versa.

SILE's page builder has a method called SILE.pagebuilder:collateVboxes which bundles a

bunch of vertical boxes into one; we can use this to bundle up each typesetter's output queue and measure the height of the combined vbox. (Of course it's possible to sum the heights of each box on the output queue by hand, but this achieves the same goal a bit more cleanly.)

```
SILE.registerCommand("sync", function()
  local lVbox =
SILE.pagebuilder:collateVboxes(diglot.leftTypesetter.state.outputQueue)
  local rVbox =
SILE.pagebuilder:collateVboxes(diglot.rightTypesetter.state.outputQueue)
  if (rVbox.height > lVbox.height) then
    diglot.leftTypesetter:pushVglue({ height = rVbox.height - lVbox.height })
  elseif (rVbox.height < lVbox.height) then
    diglot.rightTypesetter:pushVglue({ height = lVbox.height - rVbox.height })
  end
```

Next we end each paragraph (we do this after adding the glue so that `parskips` do not get in the way), and go back to handling paragraphing as normal:

```
  diglot.rightTypesetter:leaveHmode();
  diglot.leftTypesetter:leaveHmode();
  SILE.settings:set("typesetter.parseppattern", "\n\n+")
end)
```

Now everything is ready apart from the output routine. In the output routine we need to ensure, at the start of each document and the start of each page, that each typesetter is linked to the appropriate frame:

```
function diglot:_init (options)
  self.leftTypesetter:init(SILE.getFrame("a"))
  self.rightTypesetter:init(SILE.getFrame("b"))
  return plain.init(self)
end
```

(`SILE.getFrame` retrieves a frame that we have declared.)

The default `newPage` routine will do this for one typesetter every time we open a new page, but it doesn't know that we have another typesetter object to set up as well; so we need to to make sure that, no matter which typesetter causes an new-page event, the other typesetter also gets correctly initialised:

```
diglot.newPage = function(self)
  plain.newPage(self)
  if SILE.typesetter == diglot.leftTypesetter then
    SILE.typesetter.other:initFrame(SILE.getFrame("b"))
    return SILE.getFrame("a")
  else
```

```
    SILE.typesetter.other:initFrame(SILE.getFrame("a"))
    return SILE.getFrame("b")
  end
end
```

And finally, when one typesetter causes an end-of-page event, we need to ensure that the other typesetter is given the opportunity to output its queue to the page as well:

```
diglot.endPage = function(self)
  SILE.typesetter.other:leaveHmode(1)
  plain.endPage(self)
end
```

Similarly for the end of the document, but in this case we will use the emergency `chuck` method; whereas `leaveHmode` means "call the page builder and see there's enough material to build a page", `chuck` means "you must get rid of everything on your queue now." We add some infinitely tall glue to the other typesetter's queue to help the process along:

```
diglot.finish = function(self)
  table.insert(SILE.typesetter.other.state.outputQueue, SILE.nodefactory.vfillglue())
  SILE.typesetter.other:chuck()
  plain.finish(self)
end
```

And there you have it; a class which produces balanced parallel texts using two typesetters at once.

12.2 Sidenotes

One SILE project needed two different kinds of sidenotes, margin notes and gutter notes.

Chapter 9

The Transfiguration

9:1 Matt. 16:28; Mark 13:26; Luke 9:27

[1]And Jesus was saying[xxx] to them," **Truly** I say to you, ⌈there are some of those who are standing here who will **not** taste death until they see the kingdom of God after it has come➠ with power⌋."

✟

Sidenotes can be seen as a simplified form of parallel text. With a true parallel, neither the left or the right typesetter is "in charge"—either can fill up the page and then inform the other

typesetter that they need to catch up. In the case of sidenotes, there's a well-defined main flow of text, with annotations having to work around the pagination of the typeblock.

There are a variety of ways that we could implement these sidenotes; as it happened, I chose a different strategy for the margin notes and the gutter notes. Cross-references in the gutter could appear fairly frequently, and so needed to "stack up" down the page—they need to be *at least* on a level with the verse that they relate to, but could end up further down the page if there are a few cross-references close to each other. Markings in the margin, on the other hand, were guaranteed not to overlap.

We'll look at the margin marking first. We'll implement this as a special zero-width hbox (what TeX would call a \special) which, although it lives in the output stream of the main typeblock, actually outputs itself by marking the margin at the current vertical position in the typeblock. In the example above, there will be a special hbox just before the word "there" in the first line.

First we need to find the appropriate margin frame and, find its left boundary:

```
discovery.typesetProphecy = function(symbol)
  local margin = discovery:oddPage() and
                 SILE.getFrame("rMargin") or SILE.getFrame("lMargin")
  local target = margin:left()
```

Next, we call another command to produce the symbol itself; this allows the book designer to change the symbols at the SILE level rather than having to mess about with the Lua file. We use the \hbox command to wrap the output of the command into a hbox. \hbox returns its output, but also puts the box into the typesetter's output node queue; we don't want it to appear in the main typeblock, so we remove the node again, leaving our private copy in the hbox variable.

```
  local hbox = SILE.call("hbox",{}, function()
    SILE.call("prophecy-"..symbol.."-mark")
  end)
  table.remove(SILE.typesetter.state.nodes)
```

What we *do* want in the output queue is our special hbox node which will put the marking into the margin. This special hbox has no impact on the current line—it has no width, height, or depth —and it contains a copy of the symbol that we stored in the hbox variable.

```
  SILE.typesetter:pushHbox({
    width= 0,
    height = 0,
    depth= 0,
    value= hbox,
```

Finally we need to write the routine which outputs this hbox. Box output routines receive three parameters: the box itself, the current typesetter (which knows the frame it is typesetting into,

and the frame knows whereabouts it has got to), and a variable representing the stretchability or shrinkability of the line. (We don't need that for this example.)

What our output routine should do is: save a copy of our horizontal position, so that we can restore it later as we carry on outputting other boxes; jump across to the left edge of the margin, which we computed previously; tell the symbol that we're carrying with us to output *itself*; and then jump back to where we were:

```
    outputYourself= function (self, typesetter, line)
      local saveX = typesetter.frame.state.cursorX;
      typesetter.frame.state.cursorX = target
      self.value:outputYourself(typesetter,line)
      typesetter.frame.state.cursorX = saveX
    end
  })
```

This was a quick-and-dirty version of sidenotes (in twenty lines of code!) which works reasonably well for individual symbols which are guaranteed not to overlap. For the gutter notes, which are closer to true sidenotes, we need to do something a bit more intelligent. We'll take a similar approach to when we made the parallel texts, by employing a separate typesetter object.

As before we'll create the object, and ensure that at the start of the document and at the start of each page it is populated correctly with the appropriate frame:

```
discovery.innerTypesetter = SILE.defaultTypesetter {}

discovery.init = function()
  local gutter = discovery:oddPage() and
                 SILE.getFrame("rGutter") or SILE.getFrame("lGutter")
  discovery.innerTypesetter:init(gutter)
  ...
  return SILE.classes.base:init()
end
discovery.newPage = function ()
  ...
  discovery.innerTypesetter:leaveHmode(1)
  local gutter = discovery:oddPage() and
                 SILE.getFrame("rGutter") or SILE.getFrame("lGutter")
  discovery.innerTypesetter:init(gutter)
  ...
  return SILE.classes.base.newPage(discovery);
end
```

Now for the function which actually handles a cross-reference. As with the parallels example, we start by totting up the height of the material processed on the current page by both the main typesetter and the cross-reference typesetter.

```
discovery.typesetCrossReference = function(xref)
  discovery.innerTypesetter:leaveHmode(1)
  local innerVbox =
SILE.pagebuilder:collateVboxes(discovery.innerTypesetter.state.outputQueue)
  local mainVbox = SILE.pagebuilder:collateVboxes(SILE.typesetter.state.outputQueue)
```

This deals with the material which has already been put into the output queue: in other words, completed paragraphs. The problem here is that we do not want to end a paragraph between two verses; if we are mid-paragraph while typesetting a cross-reference, we need to work out what the height of the material *would have been* if we were to put it onto the output queue at this point. So, we take the `SILE.typesetter` object on a little excursion.

First we take a copy of the current node queue, and then we call the typesetter's `pushState` method. This initializes the typesetter anew, while saving its existing state for later. Since we have a new typesetter, its node queue is empty, and so we feed it the nodes that represent our paragraph so far. Then we tell the typesetter to leave horizontal mode, which will cause it to go away and calculate line breaks, leading, paragraph height and so on. We box up its output queue, and then return to where we were before. Now we have a box which represents what would happen if we set the current paragraph up to the point that our cross-reference is inserted; the height of this box is the distance we need to add to `mainVbox` to get the vertical position of the cross-reference mark.

```
  local unprocessedNodes = pl.tablex.deepcopy(SILE.typesetter.state.nodes)
  SILE.typesetter:pushState()
  SILE.typesetter.state.nodes = unprocessedNodes
  SILE.typesetter:leaveHmode(1)
  local subsidiary =
SILE.pagebuilder:collateVboxes(SILE.typesetter.state.outputQueue)
  SILE.typesetter:popState()
  mainVbox.height = mainVbox.height + subsidiary.height
```

The 1 *argument to* `leaveHmode` *means "you may not create a new page here."*

In most cases, the cross-reference typesetter hasn't got as far down the page as the body text typesetter, so we tell the cross-reference typesetter to shift itself down the page by the difference.

Unlike the parallel example, where either typesetter could tell the other to open up additional vertical space, in this case it's OK if the cross-reference appears a bit lower than the verse it refers to.

```
  if (innerVbox.height < mainVbox.height) then
    discovery.innerTypesetter:pushVglue({ height = mainVbox.height - innerVbox.height
})
  end
```

At this point the two typesetters are now either aligned, or the cross-reference typesetter has gone further down the page than the verse it refers to. Now we can output the cross-reference itself.

```
  SILE.settings:temporarily(function()
    SILE.settings:set("document.baselineskip", SILE.nodefactory.vglue("7pt"))
    SILE.call("font", {size = "6pt", family="Helvetica", weight="800"}, {})
discovery.innerTypesetter:typeset(SILE.scratch.chapter..":"..SILE.scratch.verse.." ")
    SILE.call("font", {size = "6pt", family="Helvetica", weight="200"}, {})
    discovery.innerTypesetter:typeset(xref)
    discovery.innerTypesetter:leaveHmode()
    discovery.innerTypesetter:pushVglue({ height = SILE.length({length = 4}) })
  end)
end
```

We haven't used `SILE.call` *here because it performs all its operations on the default typesetter. If we wanted to make things cleaner, we could swap typesetters by assigning* `discovery.innerTypesetter` *to* `SILE.typesetter` *and then calling ordinary commands, rather than doing the settings and glue insertion "by hand."*

In the future it may make sense for there to be a standard `sidenotes` package in SILE, but it has been instructive to see a couple of 'non-standard' examples to understand how the internals of SILE can be leveraged to create such a package. Your homework is to create one!

12.3 SILE As A Library

So far we've been assuming that you would want to run SILE as a processor for an existing document. But what if you have a program which produces or munges data, and you would like to produce PDFs from within your application? In that case, it may be easier and provide more flexibility to use SILE as a library.

At `https://sile-typesetter.org/examples/byhand.lua`, you will find an example of a Lua script which produces a PDF from SILE. It's actually fairly simple to use SILE from within Lua; the difficult part is setting things up. Here's how to do it.

```
require("core.sile")
SILE.outputFilename = "byhand.pdf"
local plain = require("plain", "classes")
SILE.documentState.documentClass = plain;
local firstFrame = plain:init()
SILE.typesetter:init(firstFrame)
```

Loading the SILE core library also loads up all the other parts of SILE. We need to set the output file name and load the class that we want to use to typeset the document with. We then need to tell SILE what class we are actually using, call `init` on the class to get the first frame for typesetting, and then initialize the typesetter with this frame. This is all that SILE does to get itself ready to typeset.

After this, all the usual API calls will work: `SILE.call`, `SILE.typesetter:typeset` and so on.

```
SILE.typesetter:typeset(data)
```

The only thing to be careful is the need to call the `finish` method on your document class at the end of processing to finish off the final page:

```
plain:finish()
```

12.4 Debugging

When you are experimenting with SILE and its API, you may find it necessary to get further information about what SILE is up to. SILE has a variety of debugging switches that can be turned on by the command line or by Lua code.

Running SILE with the `--debug facility` switch will turn on debugging for a particular area or areas of SILE's operation:

• `ast` provides information about how SILE parsed the document into an abstract syntax tree.

• `break` provides (copious) information about the line breaking algorithm.

• `fonts` shows what font families and attributes were attempted and what font files were used to supply them.

• `frames` draws red boxes around the frames on the page.

• `hboxes` draws red boxes around all the hboxes on the page.

- `pagebuilder` helps to debug problems when determining page breaks.

- `macros` notes when new functions are defined from TeXlike macro syntax.

- `makedeps` lists resources that were determined to be dependencies (use with `-m`).

- `profile` turns on Lua profiling, which gives a report on where the Lua interpreter is spending its time while processing your document. It also makes SILE go really, really slow.

- `pushback` notes how already-shaped content that didn't fit in frames is processed as it migrates to following ones.

- `tokenizer` shows how input content gets broken up into segments before shaping.

- `typesetter` provides general debugging for the typesetter: turning characters into boxes, boxes into lines, lines into paragraphs, and paragraphs into pages.

- `vboxes` provides even more information about page break decisions, showing you what vboxes were in SILE's queue when considering the breaking.

- `versions` gives a report on the versions of libraries and fonts in use; please include this information when reporting bugs in SILE!

- Any package or other area of SILE's operation may define their own debugging tags; the `insertions` package does this, as do the Japanese and Uyghur language support systems (`--debug uyghur`). Often the debug flag is the name of the package or the function.

Multiple facilities can be turned on by adding the flag multiple times or by separating them with commas: `--debug typesetter,break` will turn on debugging information for the typesetter and line breaker.

From Lua, you can add entries to the `SILE.debugFlags` table to turn on debugging for a given facility. This can be useful for temporarily debugging a particular operation:

```
SILE.debugFlags.typesetter = true
SILE.typesetter:leaveHmode()
SILE.debugFlags.typesetter = false
```

From a package's point of view, you can write debugging information by calling the `SU.debug` function (`SU` stands for SILE Utilities, and contains a variety of auxiliary functions used throughout SILE):

```
SU.debug("mypackage", "Doing a thing")
```

When an error occurs, for example when writing custom scripts, its traceback (*stack trace*) can be printed via the `--trace`, or `-t` switch:

```
$ sile -t broken.sil
SILE v0.13.0 (Lua 5.3)
<broken.sil>

Error detected:
packages/inline-footnotes.lua:9: attempt to call a nil value (global
'thisPageInsertionBoxForClass')
stack traceback:
  packages/inline-footnotes.lua:9: in upvalue 'func'
  core/utilities.lua:398: in field '?'
  core/inputs-common.lua:66: in function 'core/sile.process'
  core/inputs-texlike.lua:149: in field 'process'
  core/sile.lua:196: in function 'core/sile.readFile'
  ./sile:56: in function <./sile:56>
  [C]: in function 'xpcall'
  ./sile:56: in main chunk
  [C]: in ?
```

Sometimes it's useful for you to try out Lua code within the SILE environment; SILE contains a REPL (read-evaluate-print loop) for you to enter Lua code and print the results back to you. If you call SILE with no input file name, it enters the REPL:

```
SILE v0.13.0 (Lua 5.3)
> l = SILE.length.parse("22mm")
> l.length
62.3622054
```

At any point during the evaluation of Lua commands, you can call `SILE.repl()` to enter the REPL and poke around; hitting Ctrl-D will end the REPL and return to processing the document.

Two alternative backends are also useful for debugging. Both use the same shaping engine as the default `libtexpdf` backend, but instead of actually generating a PDF they only output some textual information about what's going on. The `debug` backend (activated by calling `sile -b debug <input>`) will generate a log file with a `.debug` extension detailing each string and it's exact output loction. A simpler `text` backend (`sile -b text <input>`) will output a `.txt` file with just the text strings with rough approximations of line breaks. Either may be sent to STDOUT instead of files using `-o /dev/stdout`.

12.5 Conclusion

We've seen not just the basic functionality of SILE but also given you some examples of how to extend it in new directions; how to use the SILE API to solve difficult problems in typesetting. Go

forth and create your own SILE packages!