The SILE Book

for SILE version v0.15.9

Simon Cozens
Caleb Maclennan
Olivier Nicole
Didier Willis
& many more contributors...

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 MS Word and friends

When most people produce printed documents using a computer, they usually use desktop oriented word processing software such as Microsoft Word, Apple 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(s) 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 several 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 and company

"Ah," some people will say, "that sounds very much like TeX!"[1]

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 forty 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 behavior of the SILE typesetter.

For instance, one of the things that TeX can't do particularly well is typesetting on a grid. This a must-have feature for anyone typesetting bibles and other documents to be printed on thin paper. 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 a fairly difficult task to accomplish in TeX. There are various solutions trying to address it, but they are complex and have limitations. In SILE, the core behaviors of the typesetter itself can easily be altered, even on the fly in a document. There is a reasonably short add-on package shipped with SILE by default to enable grid typesetting.

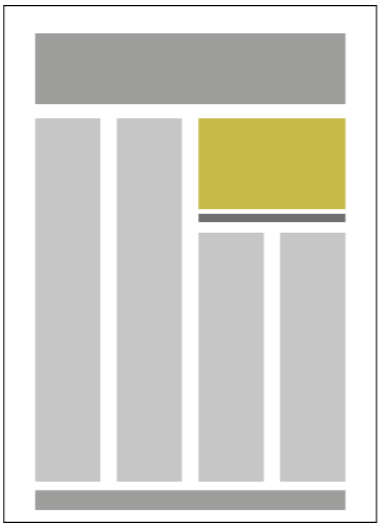
In fact, almost nobody uses plain TeX—they all use LaTeX equivalents.[3] Additionally they leverage

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.
3. Such as `pdflatex`, `xelatex`, `lualatex`, and `context`.

a huge repository of packages available from the The Comprehensive TeX Archive Network (CTAN) archive. SILE does not benefit from the large ecosystem and community that has grown up around TeX.[4] In that sense, TeX will remain ahead of SILE for some time to come. But in terms of *core capabilities*, SILE aims at being at least equivalent to TeX.

1.3 SILE versus InDesign and competitors

The other category of tool that people reach for when designing printed material on a computer desktop publishing software (DTP). Adobe's InDesign is a prominent package in this space, but many others exist. Affinity Publisher is a newcommer but popular alternative. Old timers and newspaper publishers will remember QuarkXPress. Scribus is a capable Open Source entry in this space.



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 advanced and flexible page layouts.

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 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.

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.

4. Nevertheless, SILE does have a small ecosystem of third-party packages—More on the topic later.

1.4 Conclusion

SILE[5] 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 for typesetting structured content whether they are documents written in the SIL input syntax (such as this very documentation), XML, or in some other structured data syntax that needs styling and outputting.

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

Chapter 2
Getting Started

To begin harnessing the power of SILE, now that we have covered some of its key aspects and objectives, let's dive into installing it on your computer, and set up everything you need to start typesetting documents.

2.1 Installing SILE

Ready-to-use packages are available for macOS and many Linux distributions. Details for those we know about are listed in the sections below. If your Linux distribution doesn't have native packages, fear not! You can also use either Linuxbrew or Nix packaging.

For other operating systems, you will need to download and compile the source code yourself, following the steps outlined below. Alternatively, Docker containers are available for use on any compatible system.

2.1.1 macOS

For macOS users, the recommended method for installing SILE is through the Homebrew package manager. Once Homebrew is up and running (see `http://brew.sh`), you can install SILE effortlessly by running:

```
$ brew install sile
```

Additionally, you have the option to compile SILE from the latest (unreleased) source code:

```
$ 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.

2.1.2 Arch Linux

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

```
$ pacman -S sile
```

A VCS package is also available as `sile-git` to build from the latest Git commit. This may be built and installed like any other AUR[1] package.

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

2.1.3 Fedora

A COPR repository[2] is available for Fedora users with packages of SILE and all the necessary dependencies including fonts. Fedora 36 and Fedora 37 are supported. There is work in progress to get the packages added to the official Fedora repository.

```
$ dnf copr enable jonny/SILE
$ dnf install sile
```

2.1.4 OpenSUSE

OpenSUSE has official packages ready to install the usual way:

```
$ zypper install sile
```

2.1.5 Ubuntu

A PPA[3] is available for Ubuntu users with packages of SILE and all the necessary dependencies. We introduced support starting with Bionic (18.04) and maintain packages for all Ubuntu release series since for as long as they are supported by Canonical.

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

2.1.6 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
```

2.1.7 NixOS or under Nix on any platform

In addition to NixOS, the `nix` package manager is available as a standalone package manager on many platforms including most Linux and BSD distributions, macOS, and even for Windows via WSL, and so presents an alternative way to run SILE on most systems.

The `sile` package is available in both the stable and unstable channels, the unstable channel having the latest stable SILE releases and the stable channel being frozen on NixOS releases. You can use all

2. https://copr.fedorainfracloud.org/coprs/jonny/SILE/
3. https://launchpad.net/~sile-typesetter/+archive/ubuntu/sile

the usual Nix tricks, including adding SILE into a `nix shell` environment or executing it directly with `nix run`.

```
$ nix shell nixpkgs/nixpkgs-unstable#sile
$ sile <arguments>
$ nix run nixpkgs/nixpkgs-unstable#sile -- <arguments>
```

The SILE source repository is also a valid Nix Flake[4] which means you can run any specific version or the latest unreleased development code directly:

```
$ nix run github:sile-typesetter/sile/v0.15.0 -- <arguments>
$ nix run github:sile-typesetter/sile -- <arguments>
```

2.1.8 Void Linux

Void Linux packages are available in the default package manager.

```
$ xbps-install sile
```

2.1.9 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 in your sources (including things like fonts) 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" siletypesetter/sile:latest'
$ sile input.sil
```

Docker images are tagged to match releases (e.g. `v0.15.0`). 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 by default it will not have access to your system's fonts. To work around this you can map a folder of fonts (in any organization usable by fontconfig) into the container. This could be your system's default font directory, your user one, a

4. https://wiki.nixos.org/wiki/Flakes#Installing_flakes

folder with project specific resources, 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 mounted as a volume on `/fonts` inside the container like this:

```
$ docker run -it --volume "/usr/share/fonts:/fonts" --volume "$(pwd):/data"
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.

2.1.10 Installing from source

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

SILE is completely programmable using the Lua programming language. As of v0.15.0, the CLI you actually execute is a Rust binary with a Lua VM built in. (For compatibility and demonstration purposes a pure Lua version of the CLI is still available as `sile-lua`.) The Rust binary can be built based on your system's Lua sources or use its own vendored Lua sources. All SILE's Lua code takes a lowest-common-denominator approach to Lua compatibility. Any of Lua 5.1, 5.2, 5.3, 5.4, or LuaJIT (2.0, 2.1, or OpenResty) are fully supported. Compiling it to match your system's Lua version has the advantage of making it easy to access system installed Lua Rocks, but this is not a requirement.

Compiling from sources will require both a Rust toolchain and Lua sources. At runtime no Rust tooling is required, and the system Lua interpreter is not actually used.

It also relies on external libraries to access fonts and write PDF files. HarfBuzz (minimum version 2.7.4) should be available from your operating system's package manager. For HarfBuzz to work you will also need `fontconfig` installed. SILE also requires the `icu` libraries for Unicode handling. SILE provides its own PDF creation library, which has its own requirements: `fontconfig`, `zlib` and `libpng`.

Even if building SILE from source, we suggest you use your distributions's package manager to install as many of the dependencies as possible. Most distros will have all of the system library dependencies and some of them will also have packages for some or all of the Lua dependencies. The `./configure` script will prompt for any dependencies that are missing, but it well only suggest the generic names of tools and libraries you will need. You will need to search the package repositories for the correct package names. Note that many distributions separate "development" or "library" packages from main ones. For example if your distro has "icu" and "libicu-dev" — for the purpose of building SILE you'll need the latter; Once you have built it, it will only need the former to run.

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 file:

```
$ luarocks install --only-deps 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 to 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,[5] and configure the build environment.

If you supplied all the Lua dependencies yourself, run:

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

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

```
$ ./configure
```

Also note that by default, the build process will use a vendored copy of fresh Lua sources. This will probably result in a different version of Lua than the default on your system. In the event you want it to exactly match, you'll need to have the development headers installed matching your system Lua. Once available, add this flag to your configuration:

```
$ ./configure --with-system-lua-sources
```

> *Normally a source build will not actually run until after it is installed. If you want to be able to run it from the source directory without installing it, it is important to configure it for that ahead of time. Setting up the run-time paths such that the source directory are checked allows SILE to run in place after building without installing. This is useful if you want to experiment with running SILE and/or plan on modifying or developing SILE itself. Being able to tweak the sources and re-run SILE immediately to check*

5. 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.

the difference is much faster than having to install after each tweak. You can add `--datarootdir=$(cd ..;pwd)` *which will enable the compiled binary to run directly from the source directory.*

Alternatively another useful option is `--enable-developer-mode`*. This will also accomplish the path handling change (so you don't have to use both) but takes it a few steps farther. It also enables checks on extra dependencies needed for* `testing` *SILE. These can be useful whether just to hack on it for your own use or contribute upstream, but also not all of the tooling is required. For example (among other things) you may not wish to rebuild the Docker image, lint the Lua files, or test the Flake. Individual checks can be skipped:* `--enable-developer-mode NIX=false DOCKER=false LUACHECK=false`*. Using this the developer mode option also enables a number of targets that wouldn't normally be needed by end-users, such as* `make regressions`*.*

By default SILE looks for a LuaJIT installation at configure time. This default is because running it under LuaJIT is nearly twice as fast as under PUC Lua versions. That being said, all SILE's Lua code takes a lowest-common-denominator approach to Lua compatibility. Any of Lua 5.1, 5.2, 5.3, 5.4, or LuaJIT (2.0, 2.1, or OpenResty) are fully supported.

If your system either does not have LuaJIT or you prefer to use a version of PUC lua, you can ask the configure process to pass on the LuaJIT detection:

```
$ ./configure --without-luajit
```

Keep in mind that while SILE and all its dependencies are tested to work on any interpreter, any Lua code you write for your project will need to be compatible with whatever version you choose. Several shims are provided to keeps things compatible, but it is also possible to write Lua expressions that only work in some VMs. The vast majority of Lua code will be fine, but there are a few limitations.

If that command was successful, you can now build SILE itself:

```
$ 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 will be available from any directory.

If you wish you, can skip the install step and use the compiled SILE executable directly 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
```

2.1.11 Notes for Windows users

Nobody is currently maintaining Windows compatibility in SILE and we expect the state to be a bit broken. At present there is no Windows installer. Unless you are experienced building software on Windows, it is probably best to use one of the Linux-based methods under WSL (Windows Subsystem for Linux).

There are persistent rumors from credible users that say they have gotten it working, but the exact steps they used to make it happen are a bit elusive. We would be happy to see better support, but none of the current developers are Windows users or developers. If anyone wants to help in this department, we'd be happy to facilitate contributions.

According to the rumors, SILE may be built on Windows using CMake and Visual Studio. Additionally some Windows executables are supposed to be 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.

2.2 Selecting a text editor

A SILE document is just a *plain text* file. 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 codes rather than as plain text.

Lots of good text editors exist (many of them for free) and any of them will work for SILE documents so which one you use is entirely a matter of preference. You can get started with even the most basic text editors built into your desktop environment such as Notepad on Windows, TextEdit on macOS, Gedit on Gnome, Kate on KDE, etc. However more advanced text editors (sometimes categorized as *code editors*) can offer a lot of features that make the editing process more robust. Editors are typically either graphical (GUI) or terminal (TUI) oriented and range from relatively simple to extremely

6. Still relatively popular, but was discontinued in late 2022.
7. VIM & NeoVIM users can benefit from syntax highlighting and other features via the `vim-sile` plugin at `https://github.com/sile-typesetter/vim-sile`.

complex integrated development environments (IDE). Examples of popular cross-platform GUI oriented editors include VS Code, Sublime Text, and Atom[6]. Examples of popular terminal based editors include VIM[7], Emacs, and GNU Nano. Depending on your operating system there may be platform-specific editors to consider such as Notepad++ on Windows or TextMate on macOS. Many more niche options abound: Lapce, Lite XL, Micro, Geany, BBEdit, UltraEdit, Eclipse, JetBrains IDE(s), Netbrains, Bluefish, CudaText, Leafpad, etc.

For comparisons of editors see `https://alternativeto.net/category/developer-tools/code-editor/` and select your platform.

2.3 Running SILE

Once you have set up an editor, it's time to consider a SILE input file.

2.3.1 A basic document

Let's move to a new directory, and in a text editor, create a file `hello.sil`. Copy in the following content and save the file.

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

It is a the most basic document file of all, in "TeX-like" SIL syntax (more on that later).

Then, at your command line type:

```
$ sile hello.sil
```

This produces an A4-sized PDF document `hello.pdf`, with the text Hello SILE at the top left, and the page number (1) centered at the bottom.

Congratulations—you have just typeset your first document with SILE!

All the available command-line options are documented both in the help output (`sile --help`) and in the man page (`man sile`). This manual will only mention a few in passing as they come up in other other topics.

> *SILE generates output filenames by replacing the extension from the first input filename with the default extension for the outputter. For most outputters this will be* `.pdf` *but, for example, the text backend will append* `.txt` *instead. If you want to write to a different filename altogether, use the* `--output file.pdf` *command line option. You can use* `--output -` *to write the output directly to the system IO stream—useful if you wish to use SILE as part of a pipeline.*

2.3.2 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 complicated dance of XSLT processors, format object processors, and/or strange LaTeX packages. But SILE can read XML files directly, and comes with a **docbook** class, which tells SILE how to render (admittedly, a subset of) the DocBook tags onto a page.

Hence, turning `docbook.xml` into `docbook.pdf` is as simple as:

```
$ sile --class docbook docbook.xml
SILE v0.15.9 (LuaJIT 2.1.ROLLING) [Rust]
Loading docbook
<classes/docbook.sil><docbook.xml>[1] [2] [3]
```

The `-c` flag sets the default class, a necessary step because DocBook XML files do not come wrapped in a tag that specifies a SILE class. The **docbook** class will provide the commands necessary to process the tags typically found in DocBook files.

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

2.3.3 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 properly 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 platform's 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 workflow could be setup in GitHub Actions:

```
name: SILE
on: [ push, pull_request ]
jobs:
  sile:
    runs-on: ubuntu-latest
    name: SILE
    steps:
      - name: Checkout
        uses: actions/checkout@v3
      - name: Render document with SILE
        uses: sile-typesetter/sile@v0
        with:
```

```
          args: my-document.sil
```

Add the block above to your repository as `.github/workflows/sile.yaml`. This workflow assumes your project has a source file named `my-document.sil` and will leave behind a PDF file named `my-document.pdf`. Note that this Actions workflow explicitly uses a container fetched from Docker Hub because this is the fastest way to get rolling. 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
```

However, 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.

2.4 Installing third-party packages

Third-party SILE packages can be installed using the `luarocks` package manager. Packages may be hosted anywhere, either on the default `https://luarocks.org` repository or (as in the example below) listed in a specific server manifest. For example, to install markdown.sile[8] (a plugin that provides a SILE inputter that reads and processes Markdown documents) one could run:

```
$ luarocks install --server=https://luarocks.org/dev markdown.sile
```

By default, this will try to install the package to your system. This may not be desired (and usually requires root access), but there are two other ways to install plugins. First you make add `--tree ./` to install them in the current directory. In this case, assuming this is the same directory as your document, SILE will automatically find such plugins. Additionally you may install them to your user profile by adding `--local` when installing. In this case you will also need to modify your user environment to check for plugins in that path since Lua does not do so by default. This can be done by running `eval $(luarocks path)` before running SILE (or from your shell's initialization script).

2.4.1 Finding Lua version in use for running SILE

Third party packages must be installed for the same version of Lua that SILE uses. On systems with more than one Lua version installed, *and* where SILE does not use the default one, you may need to specify the version manually. To determine which Lua version is used for the execution of SILE:

```
$ export LUA_VERSION=$(sile -e 'print(SILE.lua_version);os.exit()' 2> /dev/null)
```

8. https://github.com/Omikhleia/markdown.sile

```
$ luarocks install --lua-version $LUA_VERSION ...
```

Chapter 3
SILE's Input

As mentioned earlier, a SILE document is essentially a plain text file. However, you will need some markup to guide SILE in formatting the text. Such markup allows you to emphasize some words, start a new paragraph, introduce a chapter, and so forth.

3.1 Concerning input formats

The default SILE distribution includes support for a proprietary input language known as SIL, which comes in two different "flavors," right out of the box:

- A "TeX-like" SIL syntax, loosely inspired by LaTeX but with notable deviations and different design choices. It looks like this: `\em{content}`, producing *content*.
- An "XML flavor" which is equivalent to the TeX-like syntax but (quite obviously) presented in XML form. The previous example, in XML format, look like this: `<em>content</em>`.

For the purpose of this documentation, we will mostly use the SIL TeX-like input format. The SIL input syntax offers a more convenient and user-friendly alternative to XML, which can often be verbose and tedious to work with by hand. On the other hand, if you are handling data written by some other program, XML might be a much better solution.

But before moving forward, it is essential to note that SILE can actually accept other input markup languages. Third-party packages can also add their own input formats. Thus, SILE is quite versatile and not tied to the default SIL syntax, whether in its TeX-like or XML flavor.

Arbitrary XML schemas may be processed, with appropriate package support.[1] The SILE distribution ships with support for (a subset of) the DocBook specification. There are also existing 3rd-party packages providing support for other XML schemas; such as the TEI (Tex Initiative Encoding) specifications, USX (Unified Scripture XML), and others.

1. It sounds easy when put in those terms, and it is quite true. But of course, most XML-based document formats are fairly large and complex specifications. Thus, implementing support for them may not be as straightforward as it initially appears.

SILE is flexible and can be extended to support other markup languages, beyond XML. For instance, there are 3rd-party collections of modules for the lightweight markup languages Markdown and Djot, and others.

The stipulation is that an "inputter" component parses the content and produces an AST (Abstract Syntax Tree) recognized by SILE. With the right inputter, any markup language could potentially be supported and elevated to the status of a first-class input candidate within SILE.

With that being acknowledged, let's get back to the SIL syntax.

3.2 The SIL flavor

The first SILE file we saw in the Getting Started chapter was in SIL TeX-like input syntax (which we will just refer to as "SIL" from now on). Let's take reconsider it:

Hello SILE!

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.

3.2.1 Defining the paper size

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 like a4 and b5 as well as many other traditional sizes. Here is a full list of papersize preset names: a0, a1, a10, a2, a3, a4, a5, a6, a7, a8, a9, ansia, ansib, ansic, ansid, ansie, archa, archb, archc, archd, arche, arche1, arche2, arche3, b0, b1, b10, b2, b3, b4, b5, b6, b7, b8, b9, c2, c3, c4, c5, c6, c7, c8, comm10, csheet, dl, dsheet, esheet, executive, flsa, flse, folio, halfexecutive, halfletter, ledger, legal, letter, monarch, note, quarto, statement, and tabloid.

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

`papersize=<measurement> x <measurement>`.

SILE knows a number of ways of specifying a measurement. A `<measurement>` as mentioned above can be specified as a floating-point number followed by a unit abbreviation. Acceptable units include printer's points (`pt`), millimeters (`mm`), centimeters (`cm`), or inches (`in`). (You can even use feet (`ft`) or meters (`m`) if you are typesetting *particularly* large documents or twips (`twip`, twentieths of a point) for *particularly* small 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 set up using these fixed size units, other dimensions may be specified relative to them, using *relative units*. For example, once the paper size is set, percentage of page width (`%pw`) and height(`%ph`) become valid units. In Chapter 4 we will meet more of these relative units, and in Chapter 7 we will meet some other ways of specifying lengths to make them stretchable or shrinkable.

3.2.2 Setting orientation as landscape

The orientation of the page is defined as "portrait" by default, but if you want to set it as landscape there is an option for that:

```
\begin[landscape=true]{document}
```

3.2.3 Full bleed printing

When preparing a book for press printing, you may be asked by the professional printer to output the document on a larger sheet than your target paper, and to reserve a trim area around it. This trick is often called "full bleed printing". Your document will be printed on an oversized sheet that will then be mechanically cut down to the target size. You can specify the expected "trim" (or "bleed") dimension, to be distributed evenly on all sides of the document:

`papersize=<paper size>, bleed=<measurement>`.

For instance, a US trade book with an extra 0.125 inch bleed area can be specified by `papersize=6in x 9in, bleed=0.25in`. The output paper size is then 6.25 per 9.25 inches, with the actual 6 per 9 inches inner content centered.

Some packages, such as **background** and **cropmarks**, ensure their content extends over the trim area and thus indeed "bleeds" off the sides of the page, so that artifacts such as blank lines are avoided when the sheets are cut, would they be trimmed slightly differently for some assembling or technical reasons.

Finally, there is also the case when the actual paper sheets available to you are larger than your target paper size, and yet you would want the output document to show properly centered:

`papersize=<paper size>, sheetsize=<actual paper size>`.

For instance, `papersize=6in x 9in, sheetsize=a4` produces an A4-dimensioned document, but with you content formatted as a 6 per 9 inches US trade book. You may, obviously, combine these options and also specify a bleed area.

3.2.4 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'; rather, 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.

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, 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.

3.2.5 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.[2]

```
\begin{document}
```

2. 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.

The parameters to a command are enclosed in square brackets and take the form `key=value`; multiple parameters are separated by commas, 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.[3] Because the argument is optional, there is a difference between this: `\command{args}` (which is interpreted as a command with argument `args`) and this: `\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
\font[family=Times,size=10pt]{Hi there!} % Parameters and argument
```

3.2.6 Environments

Commands like `\chapter` (to start a chapter) and `\em` (to emphasize text) 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. As an example, the following two forms are equivalent:

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

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

3. 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.

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.

3.2.7 SIL grammar specifications

The official grammar for the SIL syntax is the LPEG reference implementation. That being said the reference implementation has some idiosyncrasies and is not the easiest to read. For convenience an ABNF grammar is also provided in the source tree, see `sil.abnf`. This grammar does not completely express the language as it cannot express the way SIL can embed other syntaxes, but it is a decent approximation.

The intent behind many of the syntax choices is to make it easy to have parity with SXML flavors. This means limiting commands to valid XML identifiers (e.g. starting with an ASCII letter, not a digit or special character), requiring a single top level command as the document, and so forth.

3.3 The XML syntax

What we've seen so far has been SILE's "TeX-like" SIL syntax flavor, but it can also directly read and process XML files. (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. Once read and parsed, processing content from either of the two file formats are exactly equivalent.

The XML form of the above document would be:

```
<document>
Hello SILE!
</document>
```

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 SIL 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 organize 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.

For instance:

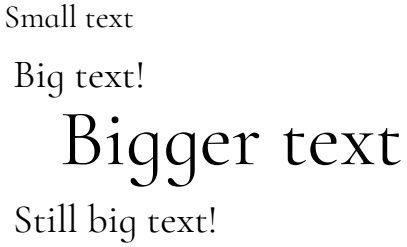
```
Small text

\font[size=15pt]%
Big text!

\font[size=30pt]{Bigger text}

Still big text!
```

produces:



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.
- `features`: Enable or disable OpenType feature flags (`-hlig`, `+ss01`)
- `variant`: A font variant (`normal`, `smallcaps`)
- `variations`: Set OpenType variations axis values used in variable fonts (e.g. `variations="wdth=122"`).[1]
- `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's **plain** class notably provides the `\strong{…}` command as a a shortcut for `\font[weight=700]{…}`, and the `\em{…}` to emphasize text (switching between italic or regular style as needed).

Note for parameters that accept multiple values, values may be separated with commas. Be sure to wrap the value in quotes so the commas don't get parsed as new parameters.

1. Support for variations requires at least HarfBuzz 6. If SILE is built on a system without support, an error will be thrown when trying to render documents using variations.

For example `\font[features="+calt,+ss01"]` will enable OpenType feature flags for both contextual alternatives and alternative style set 1. Similarly values that are themselves key=value pairs the quotation marks will keep them separate from other parameters. For example `\font[variations="wght=150,wdth=122"]` can be used to set both the weight and width axis values.

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, the `class=book` parameter; 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.[2] 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 Paragraph indentation

2. Like this: `\footnote{Like this.}`

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 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`.

4.4 Horizontal spacing

There are also 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).

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`.

4.5 Vertical spacing

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.

Besides this predefined skips, you can also use `\skip[height=(dimension)]` to add a vertical space of a given height.

If you want to add a very long stretchy vertical space, you can use the command `\vfill`.

When playing with vertical spaces, there is however a few additional considerations to take into account. Without entering into the details, they are usually ignored at the beginning of a frame. Would you want to enforce them there, you therefore need to have some initial content. An empty `\hbox` can do the trick. Additionally, there are cases where SILE automatically inserts a `\vfill` command at the end of a frame, so you may need to ensure you terminated a paragraph and introduced your own frame break in order to avoid it. The following example illustrates both techniques.

```
\hbox{}% This is an empty initial line
\skip[height=2cm]
A paragraph around 2 centimeters below the top of the frame.
\vfill
A paragraph pushed at the bottom of the frame.\par
\break
```

4.6 Text alignment

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.[3] An alternative to full justification is ragged 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 `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.

You can center a paragraph of text by wrapping it in the `center` environment. This paragraph is centered on the page.

4.7 Line and page 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.

The following four commands can be used to control line breaks (when used *within* a paragraph), as well as page breaks (when used *between* paragraphs):[4]

- `\break`
- `\goodbreak`
- `\nobreak`
- `\allowbreak`

Within a paragraph, the `\break` command requests a *line* break at the given location.[5] A less forceful

3. 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.
4. The names are similar to those used in (La)TeX, but their semantics differ slightly.
5. Note that `\break` just causes a line break, but might not be what you intended, for instance in a justified paragraph. As previously noted, the `\cr` command might do what you actually expected there.

variant is \goodbreak, which suggests to SILE that this is a good point to break a line. The opposite is \nobreak, which requests that, if at all possible, SILE not break a line 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 line breaking.

Between paragraphs, these commands have a different meaning. The \break command requests a *frame break* at the given location. 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. *Mutatis mutandis*, \goodbreak, \nobreak and \allowbreak affect frame breaking in a similar way.

The following commands are intended to be used between paragraphs and apply to page breaks only:

- \novbreak inhibits a frame break, and is just a convenience over \nobreak (ending a paragraph if need be, to be sure you are indeed inhibiting a *frame* break).
- \framebreak and \eject request a frame break.
- \pagebreak and \supereject request a non-negotiable page break, and are more forceful variants of the previous commands, ensuring that a new page is opened even if there are remaining frames on the page.

With \framebreak and \pagebreak, all vertical stretchable elements[6] are expanded to fill up the remaining space as much as possible. The \eject and \supereject variants insert an infinite vertical stretch, so that all vertical stretchable elements on the page stay at their natural size.

4.8 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, or 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]
```

6. Vertical: Here, in this document in latin script. The more advanced topic of writing directions and foreign scripts is tackled later in this manual.

```
\include[src=chap1.sil]
\include[src=chap2.sil]
\include[src=chap3.sil]
…
\include[src=endmatter.sil]
\end{document}
```

`\include`s may be nested: file A can include file B which includes file C.

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

```
\begin{sile}

\chapter{A Scandal In Bohemia}

To Sherlock Holmes she is always \em{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 runtime. 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 `\lua` command. (A `\script` command exists, but is being deprecated beginning in SILE v0.15.0.)

This command has three modes:

- A Lua library may be loaded using the Lua package path, as in `\lua[require=module.spec]`.
- A Lua script may run by giving a filesystem path, as in `\lua[src=path/to/file.lua]`.
- Lua code can be provided as inline content, as in `\lua{SILE.typesetter:typeset("foo")}`.

> *Another former use case or `\script[src=…]` was to load SILE packages. This use case has been deprecated in favor of the more robust loader `\use[module=…]`. Be sure to use a module spec with period delimiters not a path with slashes (e.g. `packages.math` not `packages/math`). This will ensure cross-platform compatibility as well as make sure packages don't get loaded multiple times.*

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. For example:

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

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 lua 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{lua}`) there must be no instances of `\end{lua}` 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 command form (`\lua`) 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.

4.9 Including raw inline content

When parsing a SIL file, SILE invokes an "inputter" module, which implements the SIL language grammar and constructs an abstract syntax tree (AST) for processing. This implies that the content of any command or environment is in SIL syntax.

However, there are cases when you may need to pass raw content that should remain unparsed — or, more properly, later parsed by a *different* grammar. While you could escape all special characters

in your content with backslashes to prevent them from being interpreted as SIL constructs, this approach is tedious and cumbersome.

This issue already arises in several scenarios. For instance, the `\lua` command (and the legacy `\script` command) described above fall into this category. In these cases, one expects to use Lua code without the need for escaping it.

Similarly, the content of the `\math` command (for the **math** package) falls outside the scope of the SIL language syntax and requires a different grammar. After all, its content follows the TeX math syntax, with commands with multiple arguments, special use of brackets, and so on. Therefore, we need to instruct the SIL parser that this content should not be interpreted,but rather extracted as a raw string. Later, it will be fed to another dedicated inputter for parsing.[7]

The SIL inputter reserves a few special keywords: `\lua`, `\script`; but also `\ftl`, `\math`, `\sil`, `\use`, `\xml`; and finally `\raw`, which we will discuss here.

It is obvious that we can't reserve too many keywords in advance. However, they must be known *before* parsing a file, which means they can't be dynamic. The reserved keywords can't be overridden or redefined after document parsing has begun. So, how can we achieve extensibility?

SILE provides a mechanism to address this: *raw handlers*. Through the Lua interface, packages and classes can register a function that gets called when a raw command is encountered in the input stream. From within a SIL file, the `\raw[type=…]` command can then be used to invoke that handler, passing the raw content.[8]

Raw handlers are identified by the `type` parameter. By default, SILE comes with a `text` raw handler, which simply typesets its content"verbatim" (as a string) without interpreting it. Packages and classes can register their own additional raw handlers to fulfill specific needs.

7. In the case of math, it is currently a *pseudo*-inputter, but that is an implementation detail.
8. In a certain sense, all things equal, raw handlers are similar to the concept of "CDATA sections" in XML.

Chapter 5
SILE Packages

SILE comes with a number of standard packages which provide additional functionality. In fact, the actual "core" of SILE's functionality is small and extensible, with most of the interesting features being provided by add-on packages. SILE ships with the core libraries plus a small collection of packages covering some common needs; more can be added from 3rd party sources. 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 is possible to do in Lua.

5.1 Loading a package

Loading a package is done through the `\use` command. By convention packages live in a `packages/` For instance, we'll soon be talking about the **grid** package, which normally can be found as `sile/packages/grid/init.lua` in wherever your system installed the SILE resource files. To load this, we'd say:

```
\use[module=packages.grid]
```

By default SILE will look for packages in a variety of directories:

1. *The directory where your input source file is located.*
2. *The current working directory.*
3. *The environment variable* `SILE_PATH`*, if defined.*
4. *The default Lua search path.*
5. *Various directories depending on where and how SILE is installed on your system.*

SILE does not descend into subdirectories when looking for a file. If you have arranged your personal class or package files into subdirectories, you will need to provide a full relative path to them.

5.2 The SILE ecosystem

The SILE installation includes a core collection of modules we hope are generally useful. But there's more out there! As mentioned earlier in this manual, a number of third-party contributed collections of modules can be installed via the LuaRocks package manager.

A non-authoritative list of third-party modules may be consulted at `https://luarocks.org/m/sile`*. To publish your own modules to LuaRocks, see the* `package-template.sile` *repository.*

A SILE compatible LuaRock simply installs the relevant class, package, language, internationalization resources, or similar files in a `sile` directory. This directory could be in your system Lua directory, in your user directory, or any other location you specify.

By default, LuaRocks will install these modules to the Lua search path.

```
$ luarocks install markdown.sile
$ sile ...
```

Depending on your system, this probably requires root permissions. If you either don't have root permissions or don't want to pollute your system's root file system, you can also install as a user. To use packages installed as a user you will need to have LuaRocks add its user tree to your Lua search path before running SILE.

```
$ luarocks --local install markdown.sile
$ eval $(luarocks --local path)
$ sile ...
```

Of course, you can add that eval statement to your shell profile to always include your user directory in your Lua path. You can also add your own entries to the top of the search path list by setting the `SILE_PATH` variable. For example:

```
$ export SILE_PATH="/path/to/my/library/"
$ sile ...
```

Note that modules are not limited to just packages. They can include classes, languages, internationalization resources, or anything else provided by SILE.[1]

5.3 Graphics

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

5.3.1 image

1. Also because external locations are searched before SILE itself, they can even override any core part of SILE itself. As such you should probably make sure you review what a package does before installing it!

● Good maturity

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. The command also supports a `page={number}` option, to specify the selected page in formats supporting several pages (such as PDF).

> *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 two 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.

5.3.2 svg

◍ Usable with limitations

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.

5.3.3 converters

◍ Usable with limitations

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. The **converters** package allows 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 JPEG, and JPEGs are supported directly by SILE.

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

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

And now it just magically works:

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

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

This trick also works for text files:

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

5.4 Text & Characters

This section covers a range of different topics from initial capitals to text transforms, through URL formatting.

5.4.1 dropcaps

● Good maturity

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 the characters can be adjusted relative to the first line using the `scale` option (defaults to `1.0`). The `join` parameter 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, which is 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.

Some fonts have capitals — such as, typically, Q and J — hanging below the baseline. By default, the dropcap fits the specified number of lines and the characters are typeset in a smaller size to fit these descenders.

With the `strict=false` option, the characters are scaled with respect to their height only, and extra hanged lines are added to the dropcap in order to accommodate the descenders. The dropcap is allowed to overflow the baseline by a reasonable amount, before triggering the addition of extra lines, for fonts that have capitals very slightly hanging below the baseline. This tolerance is computed based on the font metrics. If you want to bypass this mechanism and adjust the tolerance, you can use the `dropcaps.bsratio` setting.

Moreover, some fonts, such as EB Garamond Initials, have *all* capitals hanging below the baseline. To take this case into account in non-strict mode, the depth adjustment of the dropcap is empirically corrected based on that of a character which shouldn't have any, by default an I. The character(s) used for this depth adjustment correction can be specified using the `depthadjust` option.

> *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.*

5.4.2 lorem

● Good maturity

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 invidunt ut labore et dolore magna aliquyam

5.4.3 textcase

● Good maturity

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`.

5.4.4 unichar

● Good maturity

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 difficult to enter into SILE documents.

The **unichar** package helps with this problem by providing the `\unichar` command to enter Unicode codepoints.

```
\unichar{U+263A}
```

This produces: ☺

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

5.4.5 url

● Good maturity

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 a URL and also make it an active hyperlink, 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.

The `\urlstyle` command hook may be overridden to change the style of URLs. By default, they are typeset as “code”.

5.4.6 gutenberg

○ Experimental

Johann Gutenberg’s 42-line Bible is considered a masterpiece of early printing in part 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.

5.5 Colors

Color perception is a complicated topic, depending on many factors. SILE currently provides a few packages for handling coloring, in a simple acceptation of the term.

5.5.1 color

◍ Usable with limitations

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 one of the following:

- A RGB color in `#xxx` or `#xxxxxx` format, where `x` represents a hexadecimal digit, as often seen in HTML/CSS (`#000` is black, `#fff` is white, `#f00` is red, and so on);
- A RGB color as a series of three numeric values between 0 and 255 (e.g. `0 0 139` is a dark blue) or as three percentages;
- A CMYK color as a series of four numeric values between 0 and 255 or as four percentages;
- A grayscale color as a numeric value between 0 and 255;
- A (case-insensitive) named color, as one of the 148 keywords defined in the CSS Color Module Level 4. (Named colors resolve to RGB in the actual output.)

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

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

5.5.2 background

◍ Usable with limitations

As its name implies, the **background** package allows you to set the color of the page canvas background or to use a background image extending to the full page width and height.

The package provides a `\background` command which requires one of the following parameters:

- `color=⟨color specification⟩` sets the background of the current and all following pages to that color. The color specification has the same syntax as specified in the **color** package.
- `src=⟨file⟩` sets the background of the current and all following pages to the specified image. The latter will be scaled to the target dimension.

The background extends to the page trim area (“page bleed”) if the latter is defined. This is to ensure that it indeed “bleeds” off the sides of the page, so as to avoid thin white lignes on an otherwise full color page when the paper sheet is cut to dimension but some pages are trimmed slightly more than others. If setting only the current page background different from the default is desired, an extra parameter `allpages=false` can be passed.

So, for example, `\background[allpages=false, color=#e9d8ba]` will set a sepia tone background on the current page. The `disable=true` parameter allows disabling the background on the following pages. It may be useful when `allpages` is active from a previous invocation.

5.6 Fillers & Rules

Line-filling patterns or rules, rectangular blobs of inks... What else to say?

5.6.1 leaders

● Good maturity

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.

5.6.2 rules

● Good maturity

The **rules** package provides several line-drawing commands.

The `\hrule` command 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 `\underline` command underlines its content.

The `\strikethrough` command ~~strikes~~ its content.

Both commands support paragraph content spanning multiple lines.

> *The position and thickness of the underlines and strikethroughs are based on the metrics of the current font, honoring the values defined by the type designer.*

The `\hrulefill` inserts an infinite horizontal rubber, similar to an `\hfill`, but—as its name implies—filled with a rule (that is, a solid line). By default, it stands on the baseline and has a thickness of 0.2pt, below the baseline. It supports optional parameters `raise=⟨dimension⟩` and `thickness=⟨dimension⟩` to adjust the position and thickness of the line, respectively. The former accepts a negative measurement, to lower the line. Alternatively, use the `position` option, which can be set to `underline` or `strikethrough`. In that case, it honors the current font metrics and the line is drawn at the appropriate position and, by default, with the relevant thickness. You can still set a custom thickness with the `thickness` parameter.

For instance, `\hrulefill[position=underline]` gives: _____

Finally, `\fullrule` draws a thin standalone rule across the width of a full text line. Accepted parameters are `raise` and `thickness`, with the same meanings as above.

5.7 Boxes & Effects

You can manipulate boxed elements to achieve a variety of effects.

5.7.1 raiselower

● Good maturity

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.

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}.
```

5.7.2 rebox

● Good maturity

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.

Look I'm not !

5.7.3 rotate

◍ Usable with limitations

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 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.
```

5.7.4 scalebox

● Good maturity

The **scalebox** package allows to scale any content by some horizontal and vertical ratios, by issuing the command `\scalebox[xratio=⟨number⟩, yratio=⟨number⟩]{⟨content⟩}`, where the ratios are optional non-null numbers (defaulting to 1). The content is placed in a box and scaled.

Here is an example.

The previous line was produced by the following code:

```
Here is an \scalebox[xratio=0.75, yratio=1.25]{example}.
```

5.8 Mathematical formulas

◍ Usable with limitations

The **math** 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. 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`. If required, you can set the font style and weight via `math.font.style` and `math.font.weight`. The font size can be set via `math.font.size`. The `math.font.script.feature` setting can be used to specify OpenType features for the math font, which are applied to the smaller script styles. It defaults to `ssty` (script style alternates), notably to ensure that some symbols such as the prime, double prime, etc. are displayed correctly. The default setting applies to Libertinus Math and well-designed math fonts, but some fonts may require different features. (The STIX Two Math font has a stylitic set `ss04` from primes only, but also supports, according to its documentation, `ssty`, which provides other optical adjustments.)

MathML.

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, simply 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 could use 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, use the

`mode=display` option:

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

TeX-like syntax.
As the previous examples illustrate, 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`.

The TeX-like syntax also supports several familiar constructs, pre-defined with appropriate spacing, movable limits and other properties, such as `\sin`, `\cos`, `\lim`, etc. These are just macro-definitions (see further below); for instance, `\lim` is a shorthand for `\mo[atom=op, movablelimits=true]{lim}`.

$$\sin 2\theta = 2 \sin\theta \cos\theta$$

$$\lim_{n \to \infty} F(n) = 0$$

`{formula}` is a shorthand for `\mrow{formula}`. Delimiters—among other glyphs—stretch vertically to the size of their englobing `mrow`, which is useful for their size to adapt to the content. SILE automatically wraps paired delimiters in such a construct, so these adapt to their inner content only.

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

directly renders as

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

which is neat. But for cases when stretchy delimiters are not paired in an obvious way, these can end up too large. To keep them small, you should put braces around the expression:

```
\Vert v \Vert = \sqrt{x^2 + y^2} \text{ vs. } {\Vert v \Vert} = \sqrt{x^2 + y^2}
```

$$\left\|v\right\| = \sqrt{x^2 + y^2} \text{ vs. } {\|v\|} = \sqrt{x^2 + y^2}$$

Alternatively, you can use the `\left` and `\right` commands to automatically adjust the size of the delimiters to the inner content. Since SILE does it automatically for paired delimiters, it only really useful if you took a TeX formula using these commands and want to keep it as is, or if you want to use delimiters that are not paired in an obvious way. In this construct, the period is also supported for a null delimiter, as with TeX.

```
\left\rangle \frac{\zeta}{2} \right\langle \quad \left\}\frac{\zeta}{2} \right.
```

$$\left\rangle \frac{\zeta}{2} \right\langle \quad \left\}\frac{\zeta}{2} \right.$$

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

Token kinds.

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 this 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 $\mathbb{sin}(x)$.

Atom types and spacing.

The current implementation does not follow the MathML rules for spacing, but rather the rules defined in the TeXbook, based on atom types. Each token automatically gets assigned an atom type from the list below:

- `ord`: `mi` and `mn` tokens, as well as unclassified operators

- `op`: large operators like '$\sum$' or '$\prod$'
- `bin`: binary operators like '+' or '%'
- `rel`: relation operators like '=' or '<'
- `open`: opening operators like '(' or '['
- `close`: closing operators like ')' or ']'
- `punct`: punctuation operators like ','

The spacing between any two successive tokens is set automatically based on their atom types, and hence may not reflect the actual spacing used in the input. To make an operator behave like it has a certain atom type, you can use the `atom` attribute. For example, `a \mo[atom=bin]{div} b` renders as

$$a \operatorname{div} b.$$

Spaces in math mode are defined in "math units" (mu), which are 1/18 of an em of the current *math* font (and are independent of the current text font size). Standard spaces inserted automatically between tokens come in three varieties: thin (3 mu), medium (4 mu) and thick (5 mu). If needed, you can insert them manually with the `\thinspace` (or `\,`), `\medspace` (or `\>`), and `\thickspace` (or `\;`) commands. Negative space counterparts are available as `\negthinspace` (or `\!`), `\negmedspace`, and `\negthickspace`. The `\enspace`, `\quad`, and `\qquad` commands from normal text mode are also available, but the spaces they insert scale relative to the text font size. Finally, you can add a space of any size using the `\mspace[width=<dimension>]` command.

Macros.
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}}

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

results in:

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

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.

Matrices, aligned equations, and other tables.

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.

Numbered equations.
Equations can be numbered in display mode.

When `numbered=true`, equations are numbered using a default "equation" counter:

$$e^{i\pi} = -1 \tag{1}$$

A different counter can be set by using the option `counter=⟨id⟩`, and this setting will also enable numbering.

It is also possible to impose direct numbering using the `number=⟨value⟩` option.

The default numbering format is (n), but this style may be overridden by defining a custom `\math:numberingstyle` command. The `counter` or the direct value `number` is passed as a parameter to this hook, as well as any other options.

Missing features.
This package still lacks support for some mathematical constructs, but hopefully we'll get there. Among unsupported features, we can mention line breaking inside a formula.

5.9 Specialized environments

SILE's standard set of packages provides a few high-level environment. Some are quite expected from a typesetting system, and other also possibly serve as an illustration for class and package designers, regarding how to use varying techniques.

5.9.1 lists

● Good maturity

The **lists** package provides enumerated and itemized (also known as *bulleted lists*) which can be nested together.

Itemized lists

The `itemize` environment initiates a itemized list. Each item, unsurprisingly, is wrapped in an `\item` command.

The environment, as a structure or data model, can only contain `item` elements or other lists. Any other element causes an error to be reported, and any text content is ignored with a warning.

- Lorem
  - Ipsum
    - Dolor

On each level, the indentation is defined by the `lists.itemize.leftmargin` setting (defaults to `1.5em`) and the bullet is centered in that margin. Note that if your document has a paragraph indent enabled at this point, it is also added to the first list level.

The package has a default bullet style for each level, but you can explicitly select a bullet symbol of your choice to be used by specifying the options `bullet=⟨character⟩` on the `itemize` environment. You can also force a specific bullet character to be used on a specific item with `\item[bullet=⟨character⟩]`.

Enumerated lists

The `enumerate` environment initiates an enumeration. Each item shall, again, be wrapped in an `\item` command. This environment too is regarded as a structure, so the same rules as above apply.

The enumeration starts at one, unless you specify the `start=⟨integer⟩` option (a numeric value, regardless of the display format).

1. Lorem
   i. Ipsum
      a. Dolor

On each level, the indentation is defined by the `lists.enumerate.leftmargin` setting (defaults to `2em`). Note, again, that if your document has a paragraph indent enabled at this point, it is also added to the first list level.

The `lists.enumerate.labelindent` setting specifies the distance between the label and the previous indentation level (defaults to `0.5em`). Tune these settings at your convenience depending on your styles. If there is a more general solution to this subtle issue, we accept patches.[2]

The package has a default number style for each level, but you can explicitly select the display type (format) of the values (as `arabic`, `roman`, or `alpha`), and the text prepended or appended to them, by specifying the options `display=⟨display⟩`, `before=⟨string⟩`, and `after=⟨string⟩` to the `enumerate` environment.

Nesting

Both environments can be nested. The way they do is best illustrated by an example.

1. Lorem
    i. Ipsum
        - Dolor
            a. Sit amet
                - Consectetur

Vertical spaces

The package outputs lists starting after a line break, but it does not enforce a paragraph break before or after the list. If you want the usual value of `document.parskip` to apply before and/or after your list leave a blank line in your source document separating paragraphs as usual. Between list items, however, the paragraph skip is switched to the value of the `lists.parskip` setting.

Other considerations

Do not expect these fragile lists to work in any way in centered or ragged-right environments, or with fancy line-breaking features such as hanged or shaped paragraphs. Please be a good typographer. Also, these lists have not yet been tried in right-to-left or vertical writing direction.

5.9.2 pullquote

◍ Usable with limitations

The **pullquote** package 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

2. TeX typesets the enumeration label ragged left. Most word processing software do not.

- `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`.

5.9.3 verbatim

◍ Usable with limitations

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 `\\`. See the use of the `\autodoc:environmentraw` with a verbatim type handler for more literal verbatim behavior.*

Here is some text set in the `verbatim` environment:

```
local function init (class, _)
  class:loadPackage("rebox")
  class:loadPackage("raiselower")
end
```

If you want to specify what font the verbatim environment should use, you can redefine the `\verbatim:font` command. Unless otherwise set, the default verbatim font will be *Hack*. For example you could change it from XML like this:

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

This handles spaces, newlines, tabs and other similar whitespace literally in a way that SILE would otherwise have handled specially. If additionally you want to ignore nested SILE content (e.g. SIL commands in SIL) then you need to use a raw enviroment instead:

```
\begin[type=verbatim]{raw}
Sile commands like \em{emphasis} will not be intercepted.
```

```
\end{raw}
```

Displays as:

```
Sile commands like \em{emphasis} will not be intercepted.
```

5.9.4 specimen

● Good maturity

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

5.9.5 boustrophedon

● Good maturity

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:

ΧΑΙΡΕΔΕΜΟΤΟΔΕΣΕΜΑΠΑΤΕΡΕΣΤΕΣΕΘΑΝΟΝΤΟΣΑ

ΙΔΙΑΦΣΟΝΕΜΟΡΘΦΟΛΟΑΔΙΑΠΝΟΘΑΓΑΣΕΡΑΧΙΦΝ

ΜΟΣΕΠΟΙΕ

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

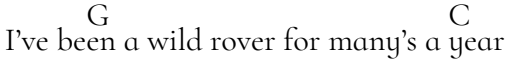
5.9.6 chordmode

● Good maturity

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:



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

5.10 Advanced font features

The following packages leverage SILE's font default handling and the `\font` command with new capabilities.

5.10.1 features

● Good maturity

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 standard `\font` command 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.

The **features** package provides two commands, `\add-font-feature` and `\remove-font-feature`, which make it easier to access OpenType features. The interface is patterned on the TeX package `fontspec`; for full documentation of the OpenType features supported, see the documentation for that

package.[3]

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]
```

5.10.2 font-fallback

◍ Usable with limitations

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 characters. 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 worlあd.
```

and SILE will produce:

あば 😈x Hello worlあd.

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

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

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

5.11 Advanced line-spacing

We will later document the default line-spacing algorithm used by SILE and the available settings that may be tuned. Still, some packages are proposed for *altering* that algorithm and may be useful in some contexts.

5.11.1 grid

◍ Usable with limitations

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

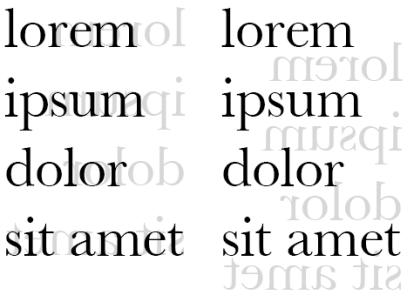
- SILE tries to insert space between two successive lines so that their baselines are separated by a fixed distance called the `baselineskip`.
- If this first rule would mean that the bottom and the top of the lines are less than two points apart, then they are forced to be two points apart. (This distance 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.



The **grid** package alters the operation of SILE's typesetter 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.

5.11.2 linespacing

○ Experimental

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 (`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 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.

5.12 Document parts

You *probably* don't need to load the auxiliary packages in this section directly. Their main job is to provide more basic functionality to other packages and classes. Classes compose functionality from different auxiliary packages. Nevertheless, these packages also provide several user-facing commands of interest.

5.12.1 folio

● Good maturity

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.

5.12.2 footnotes

◍ Usable with limitations

The **footnotes** package allows you to add footnotes to text with the `\footnote` command. Other commands provided by the package, not described here, take care of formatting the footnotes.

Usually, a document class is responsible for automatically loading this package. Minimally, upon initialization, it needs a frame identifier for the the footnotes, and one or more frame(s) which will be reduced as the footnotes take place. By default, it uses, respectively, the `footnotes` and `content` frames, which are assumed to be present in the default standard layout.

For the record, it internally relies on the **insertions** package and tells it which frame should receive the footnotes that are typeset.

5.12.3 tableofcontents

◍ Usable with limitations

The **tableofcontents** package provides tools for class authors to create tables of contents (TOCs). When you are writing sectioning commands such as `\chapter` or `\section`, your classes should call the `\tocentry[level=⟨number⟩, number=⟨strings⟩]{⟨title⟩}` command to register 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 records 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 will be necessary to run SILE a third time 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).

5.13 Bibliographies & Indexes

This section is devoted to packages collating references, in a broad sense.

5.13.1 bibtex

◍ Usable with limitations

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 Bib(La)TeX `.bib` files and output citations and full text references.

Loading a bibliography

To load a BibTeX file, issue the command `\loadbibliography[file=⟨whatever.bib⟩]`. You can load multiple files, and the entries will be merged into a single bibliography database.

Producing citations and references (legacy commands)

The “legacy” implementation is based on a custom rendering system. The plan is to eventually deprecate it in favor of the CSL implementation.

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 bibliographic reference, use `\reference{⟨key⟩}`.

The `bibtex.style` setting controls the style of the bibliography. It currently defaults to `chicago`, the only style supported out of the box. It can however be set to `csl` to enforce the use of the CSL implementation on the above commands.

This implementation doesn’t currently produce full bibliography listings. (Actually, you can use

the `\printbibliography` introduced below, but then it always uses the CSL implementation for rendering the bibliography, differing from the output of the `\reference` command.)

Producing citations and references (CSL implementation)

While an experimental work-in-progress, the CSL (Citation Style Language) implementation is more powerful and flexible than the legacy commands.

You should first invoke `\bibliographystyle[lang=⟨lang⟩, style=⟨style⟩]`, where `style` is the name of the CSL style file (without the `.csl` extension), and `lang` is the language code of the CSL locale to use (e.g., `en-US`).

The command accepts a few additional options:

- `localizedPunctuation` (default `false`): whether to use localized punctuation – this is non-standard but may be useful when using a style that was not designed for the target language;
- `italicExtension` (default `true`): whether to convert `_text_` to italic text ("à la Markdown");
- `mathExtension` (default `true`): whether to recognize `$formula$` as math formulae in (a subset of the) TeX-like syntax.

The locale and styles files are searched in the `csl/locales` and `csl/styles` directories, respectively, in your project directory, or in the Lua package path. For convenience and testing, SILE bundles the `chicago-author-date` and `chicago-author-date-fr` styles, and the `en-US` and `fr-FR` locales. If you don't specify a style or locale, the author-date style and the `en-US` locale will be used.

To produce an inline citation, call `\csl:cite{⟨key⟩}`, which will typeset something like "(Jones 1982)". If you want to cite a particular page number, use `\csl:cite[page=22]{⟨key⟩}`. Other "locator" options are available (article, chapter, column, line, note, paragraph, section, volume, etc.) – see the CSL documentation for details. Some frequent abbreviations are also supported (art, chap, col, fig...)

To produce a bibliography of cited references, use `\printbibliography`. After printing the bibliography, the list of cited entries will be cleared. This allows you to start fresh for subsequent uses (e.g., in a different chapter). If you want to include all entries in the bibliography, not just those that have been cited, set the option `cited` to false.

To produce a bibliographic reference, use `\csl:reference{⟨key⟩}`. Note that this command is not intended for actual use, but for testing purposes. It may be removed in the future.

Notes on the supported BibTeX syntax

The BibTeX file format is a plain text format for bibliographies.

The `@type{…}` syntax is used to specify an entry, where `type` is the type of the entry, and is case-insensitive. Any content outside entries is ignored.

The `@preamble` and `@comment` special entries are ignored. The former is specific to TeX-based systems, and the latter is a comment (everything between the balanced braces is ignored).

The `@string{key=value}` special entry is used to define a string or "abbreviation," for use in other subsequent entries.

The `@xdata` entry is used to define an entry that can be used as a reference in other entries. Such

entries are not printed in the bibliography. Normally, they cannot be cited directly. In this implementation, a warning is raised if they are; but as they have no known type, their formatting is not well-defined, and might not be meaningful.

Regular bibliography entries have the following syntax:

```
@type{key,
  field1 = value1,
  field2 = value2,
  …
}
```

The entry key is a unique identifier for the entry, and is case-sensitive. Entries consist of fields, which are key-value pairs. The field names are case-insensitive. Spaces and line breaks are not important, except for readability. On the contrary, commas are compulsory between any two fields of an entry.

String values shall be enclosed in either double quotes or curly braces. The latter allows using quotes inside the string, while the former does not without escaping them with a backslash.

When string values are not enclosed in quotes or braces, they must not contain any whitespace characters. The value is then considered to be a reference to an abbreviation previously defined in a `@string` entry. If no such abbreviation is found, the value is considered to be a string literal. (This allows a decent fallback for fields where curly braces or double quotes could historically be omitted, such as numerical values, and one-word strings.)

String values are assumed to be in the UTF-8 encoding, and shall not contain (La)TeX commands. Special character sequences from TeX (such as ` assumed to be an opening quote) are not supported. There are exceptions to this rule. Notably, the ~ character can be used to represent a non-breaking space (when not backslash-escaped), and the `\&` sequence is accepted (though this implementation does not mandate escaping ampersands). With the CSL renderer, see also the non-standard extensions above.

Values can also be composed by concatenating strings, using the # character.

Besides using string references, entries have two other *parent-child* inheritance mechanisms allowing to reuse fields from other entries, without repeating them: the `crossref` and `xdata` fields.

The `crossref` field is used to reference another entry by its key. The `xdata` field accepts a comma-separated list of keys of entries that are to be inherited.

Some BibTeX implementations automatically include entries referenced with the `crossref` field in the bibliography, when a certain threshold is met. This implementation does not do that.

Depending on the types of the parent and child entries, the child entry may inherit some or all fields from the parent entry, and some inherited fields may be reassigned in the child entry. For instance, the `title` in a `@collection` entry is inherited as the `booktitle` field in a `@incollection` child entry. Some BibTeX implementations allow configuring the data inheritance behavior, but this

implementation does not. It is also currently quite limited on the fields that are reassigned, and only provides a subset of the mappings defined in the BibLaTeX manual, appendix B.

Here is an example of a BibTeX file showing some of the abovementioned features:

```
@string{JIT = "Journal of Interesting Things"}
...
This text is ignored
...
@xdata{jit-vol1-iss2,
  journal = JIT # { (JIT)},
  year    = {2020},
  month   = {jan},
  volume  = {1},
  number  = {2},
}
@article{my-article,
  author  = {Doe, John and Smith, Jane}
  title   = {Theories & Practices},
  xdata   = {jit-1-2},
  pages   = {100--200},
}
```

Some fields have a special syntax. The `author`, `editor` and `translator` fields accept a list of names, separated by the keyword `and`. The legacy `month` field accepts a three-letter abbreviation for the month in English, or a number from 1 to 12. The more powerful `date` field accepts a date-time following the ISO 8601-2 Extended Date/Time Format specification level 1 (such as `YYYY-MM-DD`, or a date range `YYYY-MM-DD/YYYY-MM-DD`, and more).

5.13.2 indexer

○ Experimental

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.

5.14 Miscellaneous utilities

This section introduces packages that could not fit in another category.

5.14.1 date

● Good maturity

The **date** package provides the `\date` command, which simply outputs a date using the system's date function. It defaults to the current date and time, but can be used to format any other input time as well using the `time` parameter. 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`).

5.14.2 debug

● Good maturity

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.

5.14.3 ifattop

○ Experimental

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.

5.14.4 retrograde

● Good maturity

From time to time, the default behavior of a function or value of a setting in SILE might change with a new release. If these changes are expected to cause document reflows they will be noted in release notes as breaking changes. That generally means old documents will have to be updated to keep rending the same way. On a best-effort basis (not a guarantee) this package tries to restore earlier default behaviors and settings.

For settings this is relatively simple. You just set the old default value explicitly in your document or project. But first, knowing what those are requires a careful reading of the release notes. Then you have to chase down the incantations to set the old values. This package tries to restore as many previous setting values as possible to make old documents render like they would have in previous releases without changing the documents themselves (beyond loading this package).

For functions things are a little more complex, but for as many cases as possible we'll try to allow swapping old versions of code.

None of this is a guarantee that your old document will be stable in new versions of SILE. All of this is a danger zone.

From inside a document, use `\use[module=packages.retrograde, target=v0.15.9]` to load features from SILE v0.15.9.

This can also be triggered from the command line with no changes to a document:

```
$ sile -u 'packages.retrograde[target=v0.15.9]'
```

5.15 Frames and page layouts

As we mentioned in the first chapter, SILE uses frames as an indication of where to put text onto the page.

5.15.1 cropmarks

○ Experimental

When preparing a document for printing, you may be asked by the printer add crop marks. This means that you need to output the document on a slightly larger page size than your target paper and add crop marks to show where the paper sheet should be trimmed down to the correct size.

Actual paper size, true page content area and bleed/trim area can all be set via class options.

This package provides the `\cropmarks:setup` command which should be run early in your document file. It places crop marks around the true page content. The crop marks are guaranteed to stay outside the bleed/trim area, when defined. 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 `\cropmarks:header`.

5.15.2 frametricks

○ Experimental

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.

content_

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 parameter `offset=⟨dimension⟩`, specifying where on the line the frame should be split. If `offset` 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.

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.

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.

5.15.3 twoside

◍ Usable with limitations

A book-like class usually sets up left and right mirrored page masters. The **twoside** package is then responsible for swapping between the two left and right frames, running headers, and so on. As it is normally loaded and initialized by a document class, its main function in mirroring master frames does not provide any user-serviceable parts. It does supply a few user-facing commands for convenience.

The `\open-double-page` ejects whatever page is currently being processed, then checks if it landed on an even page. If so, it ejects another page to assure content starts on an odd page.

The `\open-spread` is similar but a bit more tailored to use in book layouts. By default, headers and folios will be suppressed automatically on any empty pages ejected, making them blank. It can also accept three parameters. The `odd` parameter (default `true`) can be used to disable the opening page being odd, hence opening an even page spread. The `double` parameter (default `true`) can be used to always output at least one empty even page before the starting an odd page. The `blank` parameter (default `true`) can be used to not suppress headers and folios on otherwise empty pages.

Lastly the `\open-spread-eject` command can be overridden to customize the output of blank pages. By default it just runs `\supereject`, but you could potentially add decorative content or other features in the otherwise empty space.

5.15.4 masters

● Good maturity

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 patterned 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.

5.15.5 break-firstfit

◍ Usable with limitations

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 characters 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. This package implements this "first-fit" method. It's designed to be used by other packages so it doesn't currently provide any user-facing commands.

5.15.6 balanced-frames

○ Experimental

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 `true`. 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.

5.16 Low-level internal packages

In addition, there are packages that you *very probably* don't need to use directly when typesetting documents.

5.16.1 bidi

● Good maturity

Scripts like the Latin alphabet you are currently reading are normally written left to right (LTR); however, some scripts, such as Arabic and Hebrew, are written right to left (RTL). 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. 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.

5.16.2 color-fonts

● Good maturity

The **color-fonts** package adds support for fonts with multi-colored glyphs (that is, OpenType fonts with `COLR` and `CPAL` tables). This package is automatically loaded when such a font is detected.

5.16.3 counters

● Good maturity

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. The command takes an optional `display=⟨display-type⟩` parameter to set the display type of the counter (see below).
- `\increment-counter[id=⟨counter-name⟩]`: Increments the counter by one. The command creates the counter if it does not exist and also accepts setting the display type.
- `\show-counter[id=⟨counter-name⟩]`: Typesets the value of the counter according to the counter's declared display type.

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
- `ROMAN`, for upper-case Roman numerals
- `greek`, for Greek letters in alphabetical order (not Greek 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, display=roman]
\show-counter[id=mycounter]
```

produces:

2

iii

The package also provides multi-level (hierarchical) counters, of the kind used in sectioning commands:

- `\set-multilevel-counter[id=⟨counter-name⟩, level=⟨level⟩, value=⟨value⟩]`: Sets the multi-level counter with the specified name to the given value at the given level. The command also takes an optional `display=⟨display-type⟩`, also acting at the given level.
- `\increment-multilevel-counter[id=⟨counter-name⟩]`: Increments the counter by one at its current (deepest) level. The command creates the counter if it does not exist. If given the `level=⟨level⟩` parameter, the command increments that level, clearing any lower level (and filling previous levels with zeros, if they weren't properly set). It also accepts setting the display type at the target level.
- `\show-multilevel-counter[id=⟨counter-name⟩]`: Typesets the value of the multi-level counter according to the counter's declared display types at each level. By default, all levels are output; option `level=⟨level⟩` may be used to display the counter up to a given level. Option `noleadingzeros=true` skips any leading zero (which may happen if a counter is at some level, without previous levels having been set).

5.16.4 insertions

◍ Usable with limitations

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 realize that insertions are implemented by an external add-on package, rather than being part of the SILE core.

5.16.5 infonode

● Good maturity

> *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);
```

5.16.6 inputfilter

● Good maturity

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 `https://sile-typesetter.org/examples/chordmode.sil` for a more complete one.

5.16.7 chapterverse

○ Experimental

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.

5.16.8 parallel

○ Experimental

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 make the operation clear.

5.16.9 autodoc

● Good maturity

The **autodoc** 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{(package)}` command.

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

For documenters and package authors, **autodoc** 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 to do so manually.

With the `\autodoc:command` command, one can pass a simple command, or even an extended command 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.[4] Moreover, for text content in parameter values or command arguments, if they are enclosed between angle brackets, they will be presented in a 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 simply 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 takes an environment name or a command, but displays it without a leading backslash.

The `\autodoc:setting`, `\autodoc:command`, and `\autodoc:environment` commands all check the validity and existence of their inputs. If you want to disable this feature (e.g., to refer to a setting or command defined in another package or module that might not yet be loaded), 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.

The `autodoc:codeblock` environment allows typesetting a block of code in a consistent way. This is not a true verbatim environment, and you still have to escape SILE's special characters within it (unless calling commands is what you really intend doing there, obviously). For convenience, the package also provides a `raw` handler going by the same name, where you do not have to escape the special characters (backslashes, braces, percents).

The `\autodoc:example` marks its content as an example, possibly typeset in a different choice of font.

The `\autodoc:note` outputs its content as a note, in a dedicated framed and indented block. The `\autodoc:package` and `\autodoc:class` commands are used to format a package and class name.

5.16.10 pdf

◍ Usable with limitations

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⟩}`.

4. If the **color** package is loaded and the `autodoc.highlighting` setting is set to `true`, you get syntax highlighting.

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`.

5.16.11 pdfstructure

◍ Usable with limitations

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

5.17 Highly experimental packages

The following packages are not documented here: **complex-spaces**, **pagebuilder-bestfit**, **pandoc**, **simpletable**, **xmltricks**.

> *These packages are not ready for use in production and are subject to change without notice in future versions.*

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. (Aficionados of TEX and friends will be familiar with the concept of bumpy road logos.) Our logo will look like this: SILE. It's not a great logo, but we'll use it as SILE's logo for the purposes of this section.

To typeset this logo, we need to ask SILE 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 SILE 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 SILE'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

1. Let's give up on the logo at this point.

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 must 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 emphasize 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, let's build a macro that italicizes its content and wraps it in a narrower text block. Here is one way to define such a `<note>` macro, in XML flavor:

```
<define command="narrower">
<set parameter="document.lskip" value="24pt"><process/><par/></set>
</define>
```

```
<define command="notefont">
<font style="italic" size="10pt"><process/></font>
</define>

<define command="note">
<narrower><notefont><process/></notefont></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 the name of the setting gives you some kind of clue as to what area of the system it will affect, and so that 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` command. 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 that 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 localized 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 idiomatic of Lua code than SILE that uses positional arguments instead of named options:

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

The third and fourth 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

The `document.lskip` and `document.rskip` settings are *glue* parameters which are added, respectively, to the left and right side of every line. Setting `document.lskip` 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.

> *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 `center` 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. (In fact, 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]`.

Note that non-breaking spaces (U+00A0), following the guidelines of Unicode Annex 14 (UAX 14), are treated by default as stretchable or shrinkable akin to regular inter-word spaces, contributing to text justification and alignment for consistent layout.

If you want to disable this behavior, the `languages.fixedNbsp` setting may be set to `true` to enforce fixed-width non-breaking spaces.

Some typography conventions use an em-dash at the start of a paragraph line to denote a speaker change in a dialogue. This is the case in particular in French and Turkish typography. By default, all spaces following an em-dash at the beginning of a paragraph in your input are replaced by a single *fixed* inter-word space, so that subsequent dialogue lines all start identically, while other inter-word spaces may still be variable for justification purposes. To cancel this behavior, the `typesetter.fixedSpacingAfterInitialEmdash` setting may be set to `false`.

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.

1. He was probably talking about blackletter, but it's still true.

7.2.1 Paragraphing

`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.

7.2.2 Automated italic correction

When an italicized word is followed or preceded by non-italicized text, the spacing may need to be adjusted, or "corrected", so that the characters do not overlap. You might thus want to insert some additional space between italicized words and non-italicized ones.

Here is some gibberish exemplifying the issue.

(*fluff*) ¡*fancyful proof*! [*puff*] *fluff*[n]

Inserting the necessary spaces manually is quite tedious, not to say impractical. However, the concept of italic correction does not exist in OpenType fonts, and they do not provide the metrics that would allow a typesetting system to implement it easily.[3] A heuristic approach is to use the difference between a glyph's bounding box and its advance width (when switching from italics to roman) or

2. TeX users, please notice the renaming.
3. This would apply to Graphite fonts too. More generally, there is no known font format supporting kerning across font style changes. (Well, some TeX fonts have such a possibility, but it does not really help here.)

its bearing width (the other way round). Assuming italics is slanted forward (in left-to-right writing direction) and that italicized glyphs usually reach their maximum extent to the right towards their top (and towards their bottom, on their left side), then it is possible to approximate a fairly decent correction.

Be aware, nevertheless, that this solution cannot be made perfect, even assuming a reasonable choice of fonts. Pathological cases may still occur, even in latin scripts, for which there is no solution but using manual kerning.

To enable automated italic correction, you can set the `typesetter.italicCorrection` setting to `true`. Let's turn it on and check how our previous gibberish now behaves.

(fluff) ¡fancyful proof! [puff] fluff^{n}

Note that this setting only works on full paragraphs, or with horizontal boxes constructed with `\hbox{(content)}`.[4] In other terms, turning it on or off around just a few words in a sentence will not have the intended effect.

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)

4. That is, when characters are actually *shaped*, so that SILE knows their properties and metrics.

- `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.[5] 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 macOS 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

5. Shaping is the process of selecting and positioning the glyphs from a font–turning the text that we type into the boxes that SILE puts together on a line.

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.types.length` *object, glue must be* `SILE.types.node.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=⟨code⟩]` command as detailed in Chapter 4, or by use of the `\language[main=⟨code⟩]` 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* files for that language. These in turn enable various localization and typesetting conventions. Language support may include:

- hyphenation patterns
- line breaking and justification schemes
- frame advance and writing direction
- spacing
- choice of glyphs within a font
- localization of programmatically inserted strings

For example, Sindhi and Urdu users will expect the Arabic letter *heh* (ہ) 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]{ههه};
then in Sindi:
\font[family=LateefGR,language=snd]{ههه};
then in Urdu:
\font[family=LateefGR,language=urd]{ههه}.
```

Standard Arabic: ههه; then in Sindi: ههه; then in Urdu: ہہہ.

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`, or `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=(dir)]`.

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, hyphenchar=U+200C, language=ml]`.

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 dictionaries are available; if not, work through the resources at `http://tug.org/docs/liang/`.

Note on Unicode soft hyphens — By default, soft hyphens (U+00AD) are interpreted as discretionary breaks, allowing line-breaking at that point (using the current font's hyphen character).

However, issues may arise when soft hyphens are used in ligatures, causing breaks between constituent characters and disrupting the ligature's integrity. Rather than relying on soft hyphens, for instances requiring hyphenation in unknown words, consider adding an exception to the hyphenation rules instead, with `\hyphenator:add-exceptions{(text)}` (where the text is a lowercase representation of the word, with dashes where hyphenation is allowed).

Moreover, typists sometimes manually insert soft hyphens to rectify line-breaking issues in other typesetting systems. In SILE, leveraging language-specific hyphenation rules tends to be more reliable. Setting `typesetter.softHyphen` to `false` ignores soft hyphens entirely in the text, alleviating potential issues arising from their manual insertion.

Soft hyphens can be inadvertently inserted by text editors or software, remaining invisible in the source text and causing unexpected output. Setting `typesetter.softHyphenWarning` to `true` triggers warnings upon encountering soft hyphens, aiding users in identifying and rectifying such instances, regardless of the previous setting.

8.4 Localization

A small handful of strings may be programmatically 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.[1] 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>]`.

1. See Project Fluent (`https://projectfluent.org`) for details on the data format and uses.

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{tableofcontents-title = 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፡ልዑል

\set[parameter=languages.am.justification,value=centered]
ሰላም፡ልዑል
```

ሰላም፡ ልዑል

ሰላም ፡ ልዑል

8.5.2 Croatian

According to Croatian typography conventions, when a break occurs at an explicit hyphen, the hyphen gets repeated at the beginning of the new line. SILE automatically handles this.

8.5.3 Czech

According to Czech typography conventions, when a break occurs at an explicit hyphen, the hyphen gets repeated at the beginning of the new line. SILE automatically handles this.

8.5.4 Esperanto

Esperanto typesetting is quite straight forward; however one feature of the language is unique: the requirement that *all* adjectives, including numerals, have the suffix "ª". This includes numbers standing on their own. For example, "the 15th of March" is, in Esperanto, "la 15ª de marto". As there is lack of agreement[2] on how to typeset this, you have options: `languages.eo.ordinal.raisedsuffix` when made true will use ª (as in "Ĉapitro 1ª") while `languages.eo.ordinal.hyphenbefore` will prepend a hyphen (as in "Ĉapitro 15-a").

8.5.5 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, and semicolons, and an interword 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`.

8.5.6 Polish

According to Polish typography conventions, when a break occurs at an explicit hyphen, the hyphen gets repeated at the beginning of the new line. SILE automatically handles this.

8.5.7 Portuguese

According to Portuguese typography conventions, when a break occurs at an explicit hyphen, the hyphen gets repeated at the beginning of the new line. SILE automatically handles this.

8.5.8 Slovak

According to Slovak typography conventions, when a break occurs at an explicit hyphen, the hyphen gets repeated at the beginning of the new line. SILE automatically handles this.

8.5.9 Spanish

2. Wikipedia prefers "15-a" while most professional books and posters prefer "15ª". Some authors even write "15a", as the underlying word is "dekkvina".

According to Spanish typography conventions, when a break occurs at an explicit hyphen, the hyphen gets repeated at the beginning of the new line. SILE automatically handles this.

8.5.10 Turkish

According to Turkish typography conventions, when a break occurs at an existing apostrophe, the break point is allowed but no hyphenation character is shown. SILE behaves this way default. Some publisher style guides suggest an alternative behavior replacing the apostrophe with the hyphenation character. This alternative behavior can be achieved by setting `languages.tr.replaceApostropheAtHyphenation` to `true`.

8.5.11 Japanese / Chinese

SILE aims to conform with the W3G document "Requirements for Japanese Text Layout"[3] 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** or **jbook** classes. For other languages with similar layout requirements, more generic **tplain** and **tbook** classes are available that setup the layout elements without also setting the default language and font to Japanese specific values. These are also good candidates to use as base classes and extend for more language-specific classes.

Japanese documents are traditionally typeset on a grid layout called a *hanmen*, with each character essentially monospaced inside the grid (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}
```

3. https://www.w3.org/TR/jlreq/

Produces:
れいわ
令和

8.5.12 Syllabic languages

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

8.5.13 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 or package 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 Measurements and lengths

Before dabbling into more advanced topics, let's introduce "measurements" and "lengths" in SILE, the two available Lua constructs for representing dimensions.

Measurements are specified in terms of `SILE.types.measurement` objects. It is a basic construct with an amount and a unit. Let us illustrate two common ways for creating such an object in Lua (from a string, with same syntax as in command parameters; or from a Lua table).

```
local m1 = SILE.types.measurement("10pt")
local m2 = SILE.types.measurement({ amount = 10, unit = "pt" })
```

SILE also provides a more advanced construct specified in terms of `SILE.types.length` objects; these are "three-dimensional" dimensions, in that they consist in a base measurement plus stretch and shrink measurements. They are therefore composed of three `SILE.types.measurement`.

```
local l1 = SILE.types.length("10pt plus 2pt minus 1pt")
local l2 = SILE.types.length({ length = "10pt", stretch = "2pt", shrink = "1pt" })
```

Both of these are used for various purposes. In many cases, they are nearly interchangeable. Casting from one to the other is straightforward: casting a length to a measurement returns just the base measurement and discards the stretch and shrink properties; casting a measurement to a length sets its stretch and shrink properties to zero.

```
local l3 = SILE.types.length(SILE.types.measurement("10pt")) -- 10pt, without stretch and
shrink
local m3 = SILE.types.measurement(SILE.types.length("10pt plus 2pt minus 1pt")) -- 10pt
```

Proper casting is important, for your code to remain portable across the various versions of the

Lua language.

9.2 Boxes, glue, and penalties

SILE's job, looking at it in very abstract terms, is 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 (such as the stretchable or shrinkable space between words)
- Vertical boxes (typically, a line of text)
- Vertical glue (such as the space between lines and paragraphs)
- Penalties (information about where and when not to break lines and pages)

Additionally, horizontal boxes are further specialized.[1]

- Discretionaries (special construct used when a word is hyphenated)
- N-nodes and unshaped nodes (text content shaped according to a certain font, or not yet shaped and measured)
- Migrating boxes (such as foonote content)

The most immediately useful of these are horizontal and vertical glue. Horizontal and vertical glue can be explicitly added 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. 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 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.

At a Lua coding level, SILE's Lua interface contains a `types.node` for creating boxes and glue. Here

1. The math support in SILE also defines additional types of boxes, not discussed here.

is one way (among others) for you to construct horizontal and vertical glue:

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

9.3 Kerns

`\kern`'s are a type of `\glue`, only different in that while a `\glue` can be broken at the end of a line, a `\kern` can't. Hearkening back to our SILE 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:

SILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESIL
ESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESIL
ESILESILESILESILESILESILESILESILESILESILESILE

Note end of lines where SILE is broken between its 'L' and 'E'. Instead, if we typeset the same line using `\kern`'s as we had originally:

SILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILESILE

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:

SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE

Kerns:

SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE SILE

9.4 The typesetter

SILE's typesetting is organized by the `SILE.typesetter` object. It maintains two queues of material that it is still working on: the node queue and 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 it is added to the node queue in as small chunks as possible. These chunks must remain together no matter where they end up on a line. This might include individual symbols, syllables, or objects such as images. As soon as new content which requires a vertical break is encountered, the node queue is processed to derive 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 horizontal content (the node queue) to be shaped into lines and added to the vertical output queue by calling the function `SILE.typesetter:leaveHmode()`.

When writing a custom command, if you want to manually add a vertical space to the output, first ensure that the 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.

It might be a good point to better explain here the actual difference between just leaving horizontal mode, and the related, but higher level, `\par` command. The latter is more frequently used when writing a document. It first calls `SILE.typesetter:leaveHmode()`, but then also inserts a vertical skip according to the `document.parskip` setting, and goes on to reset a number of settings that are typically paragraph-related such as hanging indents. When designing you own commands, there are therefore some cases when you may just need to call `SILE.typesetter:leaveHmode()` and handle everything else in your own code; and situations when invoking `SILE.call("par")` might be more adequate, resulting in an effective paragraph to be terminated.

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 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.

9.5 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 page's 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:

2. You can use the **frametricks** package to get around this limitation—split the current frame and start fiddling around with the positions of the new frames that **frametricks** created for you.

```
\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 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 parentheses symbols 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, and 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 accumsan 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 vel eum iriure dolor in hendrerit in vulputate velit

rightCol

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 Packages & Classes

This chapter describes how to implement your own add-on packages and classes in the Lua programming language, for you to extend the way that the SILE system operates, define new commands and page layouts, or indeed do anything that is possible to do in Lua.

The default formatting in SILE documents is usually determined by the class used by that document. This default look can be changed, and more functionalities can be added by means of a package. Sometimes it's hard to make a decision when it comes to choose whether to write a package or a class, and the difference may seem subtle. The basic rule is that if your file contains commands that control the look of the *logical structure* of a given type of document, then it's a class. Otherwise, if your file adds features that are independent of the document type, then it's rather a package.[1]

SILE relies on the Penlight Object-Oriented Programming (OOP) framework. Many components are therefore implemented as Penlight classes (here, in the usual OOP sense). Their use below is straightforward and is expected to be covered by examples, but you might also want to read more about it before you start.[2]

10.1 Designing a package

Packages live somewhere in the `packages/` subdirectory of either where your first input file is located, your current working directory, or your SILE path.

1. Obviously there's nothing new here for seasoned (La)TeX users, but there's no harm either stating it for a more general audience.
2. See `https://lunarmodules.github.io/Penlight/libraries/pl.class.html`
3. Programmers will recognize the delegation over inheritance paradigm here. If you intend to develop a complete family of packages sharing several common methods, then of course you might be interested in first implementing all of these in a parent package, that your other packages will inherit.

10.1.1 Implementing a bare package

A minimum working package inherits from the **base** package. While it is possible to inherit from another existing package, let's ignore this advanced use case in this primer.[3]

We need to declare the name of our new package, override the package's initialization method (that is, its class constructor) and possibly other methods as well, set a documentation string, and finally return our new package.

While its presence is not mandatory, the documentation string usually comes in the form of an embedded SIL document, explaining the purpose of the package and possibly illustrating some of its features. It is extracted by the **autodoc** package for presenting the package in a manual such as this one. We recommend writing it, when you feel ready to share your package with other users.

Also note that the package's initialization methods accepts an `options` table. It allows passing parameters when loading and instantiating that package. This is already a somewhat advance use case too, and we are not going to cover it here.

This being said, let's proceed as mentioned, and simply create a file `packages/mypkg/init.lua` with the following content.

```
local base = require("packages.base")

local package = pl.class(base)
package._name = "mypkg"

function package:_init (options)
  -- Things you might want to do before the parent initialization.
  base._init(self)
  -- Things you might want to do after the parent initialization.
end

-- Additional methods will later come here.

package.documentation = [[
\begin{document}
...
\end{document}
]]

return package
```

You have just written you very first package, and you can already use it in a document (for instance, loading it with `\use[module=packages.mypkg]`)... Although this package doesn't do anything interesting yet.

10.1.2 Defining commands

To define your own command at the Lua level, you overload the `registerCommands` package method.

```
function package:registerCommands ()
  -- Our own commands come here
end
```

Within it, use the `self: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`.[4] Both of these parameters are Lua tables. The `options` parameter contains the command's parameters as a key-value table, and the `content` parameter 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 ",
  {
    options = {},
    id = "command",
    pos = …,
    col = …,
    lno = …,
    command = "break"
  },
  " world"
}
```

Most commands will find themselves doing something with the `options` and/or calling `SILE.process(content)` to recursively process and render the argument.

Here's a very simple example: a `\link` command may take an `href` attribute. We want to render `\link[href=http://...]{Hello}` as Hello (`http://...`). First we need to render the content, and then we need to do something with the attribute. We use the `SILE.typesetter:typeset` and `SILE.call` functions to output text and call other commands.

```
self:registerCommand("link", function(options, content)
```

4. Of course ou can name your parameters whatever you like, but these are the most common names.

```
    SILE.process(content)
    if (options.href) then
      SILE.typesetter:typeset(" (")
      SILE.call("code", {}, { options.href })
      SILE.typesetter:typeset(")")
    end
  end)
```

Now, let's (re-)design a `blockquote` environment implementing indented (and possibly nested) quotations. You do remember, right, that an environment in SILE is not much different from a command? So a command be it, without any option this time, but playing with vertical skip, measurements, glue, (temporary) left and right margin settings. (If these concepts elude you, consider re-reading the previous chapters where they are introduced.)

```
self:registerCommand("blockquote", function (_, content)
    SILE.call("smallskip")
    SILE.settings:temporarily(function ()
      local indent = SILE.types.measurement("2em"):absolute()
      local lskip = SILE.settings:get("document.lskip") or SILE.types.node.glue()
      local rskip = SILE.settings:get("document.rskip") or SILE.types.node.glue()
      SILE.settings:set("document.lskip",
        SILE.types.node.glue(lskip.width + indent))
      SILE.settings:set("document.rskip",
        SILE.types.node.glue(rskip.width + indent))
      SILE.process(content)
      SILE.typesetter:leaveHmode() -- gather paragraphs now.
    end)
    SILE.call("smallskip")
  end, "A blockquote environment")
```

10.1.3 Defining settings

To define your own settings at the Lua level, you overload the `declareSettings` package method; and within it, use the `SILE.settings:declare` function. It takes a setting specification as argument.

In our custom quotation environment above, note that we hard-coded the indentation. Say you'd prefer allowing users to specify their preferred value here. You would have more than one way to achieve it. A command option is one of them, but you'd be right in thinking that a SILE setting might be more user-friendly and appropriate in this very case, so one could for instance do `\set[parameter=mypkg.blockindent, value=2em]` to configure it globally (or within a given scope). Let's do this. Change the line setting the indentation in your custom command...

```
local indent = SILE.settings:get("mypkg.blockindent"):absolute()
```

... and declare the corresponding setting:

```
function package:declareSettings ()
  SILE.settings:declare({
    parameter = "mypkg.blockindent",
    type = "measurement",
    default = SILE.types.measurement("2em"),
    help = "Blockquote indentation"
  })
end
```

10.1.4 Defining raw handlers

"Raw handlers" allow packages to register new handlers (or callbacks) for use with the `raw` environment, which content is read as-is by SILE, without being interpreted. This is intended for advanced use cases where you may want to provide a way for users to embed arbitrary content (likely in another syntax), and you will provide the complete parsing and handling for it.[5]

You can define your own raw handlers at the Lua level. Overload the `registerRawHandlers` package method; and within it, use the `self:registerRawHandler` function. It takes two parameters: a handler type name, and a function to implement the handler. The signature of the handler function is the same as for a SILE command.

Here is a handler that just typesets the content as-is, for you to just get the idea.

```
function package:registerRawHandlers ()
  self:registerRawHandler("mypkg:noop", function(options, content)
    -- contains everything within the raw environment as unparsed text.
    local text = content[1]
    SILE.typesetter:typeset(text)
  end)
end
```

10.1.5 Loading other packages

Above, when introducing the `_init` method, we left a few placeholder comments. Let's say you want

5. This may be used to implement a "clever" verbatim environment. It is also used, for instance, by the **markdown.sile** 3rd-party collection to embed Markdown or Djot content directly in a (SIL or XML) document.

to ensure the **color** package is also loaded, so that the custom `\link` command you implemented can safely invoke it in a `SILE.call`.

```
function package:_init ()
  base._init(self)
  -- Load some dependencies
  self:loadPackage("color")
end
```

The `self:loadPackage` methods takes as argument a package name, and optionally packages options (as a table).

10.1.6 Registering class hooks

Some packages may provide additional functions that need to be automatically called at various points in the output routine of the document class. But let's return to that topic later, when describing how to set up you own custom class. For now, we can conclude our primer on packages, as you should already have all the tools to design great packages.

10.2 Designing a document class

Document classes live somewhere in the `classes/` subdirectory of either where your input file is located, your current working directory, or your SILE path.

10.2.1 Implementing a bare class

A minimum working class inherits from the **base** class. Most of the time, however, you will prefer inheriting at least from the **plain** class, which already provides a lot of things users will expect, including most of the basic commands presented early in this manual. Let's assume this is the case, and simply create a file `classes/myclass.lua` with the following content.

```
local plain = require("classes.plain")

local class = pl.class(plain)
class._name = "myclass"

function class:_init (options)
  -- your stuff here (if you want it before the parent init)
  plain._init(self, options) -- Note: passing options
  -- your stuff here (if you want it after the parent init)
end

-- Additional methods will later come here.
```

```
return class
```

Note that it is very similar to what we previously did when designing a package.

A notable difference is that `options` always need to be propagated to the parent in the initialization method. Not only can your document class implement its own additional options, you indeed also want standard options to be honored, such as the paper size, etc. In other methods that we will later override, we will also invoke the corresponding parent method, for it also to do its own things.

That's it. You have implemented a working bare bones class. The next step is to start adding or overriding class functions to do what you want.

10.2.2 Defining commands, settings, etc.

A document class can define commands, declare settings, register raw handlers and load additional packages at initialization.

For all of these, the logic is exactly the same as for packages, so we are not repeating it here.

10.2.3 Defining class options

Your document class can also define specific options. To define your own class option, you overload the `declareOptions` class method; and within it, use the `self:declareOption` function. It takes two arguments, an option name and a function. The latter acts as a setter or getter, so a minimal code will usually look as follows.

```
function class:declareOptions ()
  base.declareOptions(self) -- Note: support parent class options

  self:declareOption("myoption", function (_, value)
    if value then
      self.myoption = value
      -- Possibly perform other processing when the value is set.
    end
    return self.myoption
  end)
end
```

Would you also want this option to have a default value, then overload the `setOptions` method. In that case, do not forget invoking the superclass method, so that its own options are also properly initialized.

```
function class:setOptions (options)
  options.myoption = options.myoption or "default"
  base.setOptions(self, options) -- Note: set parent options
```

```
end
```

10.2.4 Changing the default page layout

We earlier learned how to define a frame layout for a single page, let's try to define one for an entire document. 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.[6] 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"
  }
```

6. 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 in your own classes when following these examples.

```
}
```

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 which loads the **masters** package and generates a master frameset using the default frameset defined above.

```
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 classes' *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, and 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 **book** class will take care of setting these frames up with mirrored masters.

If we had wanted to load additional 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 function:

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

10.2.5 Modifying class output routines

As well as defining frames and packages, classes may also alter the way that SILE performs its output —for instance, what it should do at the start or end of a page, 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 end of each page.
- `finish` is called at the end of the document.

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

10.2.6 Interacting with class hooks

Some packages may provide functions that need to be run as part of the class output routines. They can accomplish this is by registering hook functions that get run at known locations in the provided classes. In the default implementation, three hooks are provided:[7]

- The `newpage` hook is run at the start of each page.
- The `endpage` hook is run at the end of each page.
- The `finish` hook is called at the end of the document.

For an example, we will check out the **tableofcontents** package for the hooks it sets, but also the `\tocentry` command it registers that gets called manually in the **book** class. Let's demonstrate roughly how the that package works. We'll be using the **infonode** 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.

```
self:registerCommand("tocentry", function (options, content)
  -- (Simplified from the actual implementation.)
  SILE.call("info", {
    category = "toc",
    value = {
```

7. We will not cover it here, but class authors may also provide their own hook locations for packages, or run any set of registered hooks in their own outputs.

```
      label = SU.ast.stripContentPos(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. In order to be useful, 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.*

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
function package:moveTocNodes ()
  local node = SILE.scratch.info.thispage.toc
  if node then
    for i = 1, #node do
      node[i].pageno = self.packages.counters:formatCounter(SILE.scratch.counters.folio)
      table.insert(SILE.scratch.tableofcontents, node[i])
    end
  end
end
```

We're going to take the LaTeX approach of storing these items 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:

```
function package.writeToc (_)
  -- (Simplified from the actual implementation.)
  local tocdata = pl.pretty.write(SILE.scratch.tableofcontents)
  local tocfile, err = io.open(pl.path.splitext(SILE.input.filenames[1]) .. '.toc', "w")
  if not tocfile then return SU.error(err) end
  tocfile:write("return " .. tocdata)
  tocfile:close()
end
```

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

```
self:registerCommand("tableofcontents", function (options, _)
  -- (Simplified from the actual implementation.)
  local toc = self:readToc()
  if toc == false 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)
```

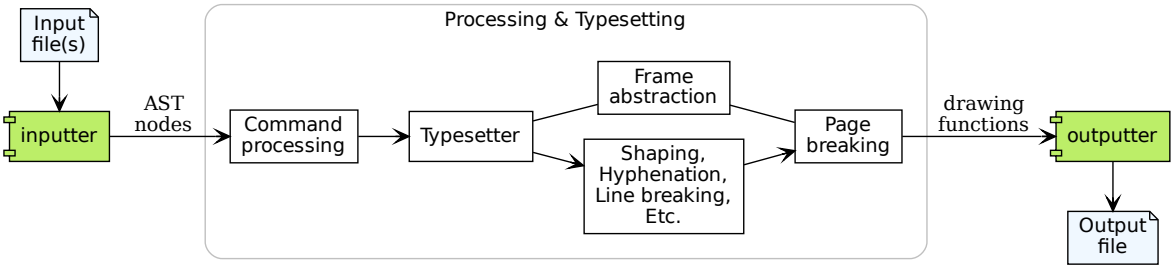
And the job is done. Well, nearly. Our **tableofcontents** package now contains a couple of methods —`moveTocNodes` and `writeToc`—that need to be called at various points in the output routine of a class which uses this package. How do we do that? We simply have to register these methods for them to be called at the intended points.

```
function package:_init ()
  -- (Simplified from the actual implementation.)
  base._init(self)
  if not SILE.scratch.tableofcontents then
    SILE.scratch.tableofcontents = {}
  end
  self:loadPackage("infonode")
  ...
  self.class:registerHook("endpage", self.moveTocNodes)
  self.class:registerHook("finish", self.writeToc)
end
```

This concludes our primer on document class design. A few details were’nt addressed, possibly, but you should now have all the tools at your disposal to create your own classes, or start digging into the standard classes and packages with the necessary understanding of their inner working.

Chapter 11
Designing Inputters & Outputters

Let's dabble further into SILE's internals. As mentioned earlier in this manual, SILE relies on "input handlers" to parse content and construct an abstract syntax tree (AST) which can then be interpreted and rendered. The actual rendering relies on an "output backend" to generate a result in the expected target format.



The standard distribution includes "inputters" (as we call them in brief) for the SIL language and its XML flavor,[1] but SILE is not tied to supporting *only* these formats. Adding another input format is just a matter of implementing the corresponding inputter. This is exactly what third party modules adding "native" support for Markdown, Djot, and other markup languages achieve. This chapter will give you a high-level overview of the process.

As for "outputter" backends, most users are likely interested in the one responsible for PDF output. The standard distribution includes a few other backends: text-only output, debug output (mostly used internally for regression testing), and a few experimental ones.

11.1 Designing an input handler

Inputters usually live somewhere in the `inputters/` subdirectory of either where your first input file is located, your current working directory, or your SILE path.

11.1.1 Initial boilerplate

A minimum working inputter inherits from the **base** inputter. We need to declare the name of our new inputter, its priority order, and (at least) two methods.

When a file or string is processed and its format is not explicitly provided, SILE looks for the first inputter claiming to know this format. Potential inputters are queried sequentially according to their

1. Actually, SILE preloads *three* inputters: SIL, XML, and also one for Lua scripts.

priority order, an integer value. For instance,

- The XML inputter has a priority of 2.
- The SIL inputter has a priority of 50.

In this tutorial example, we are going to use a priority of 2. Please note that depending on your input format and the way it can be analyzed in order to determine whether a given content is in that format, this value might not be appropriate. At some point, you will have to consider where in the sequence your inputter needs to be evaluated.

We will return to the topic later below. For now, let's start with a file `inputters/myformat.lua` with the following content.

```
local base = require("inputters.base")

local inputter = pl.class(base)
inputter._name = "myformat"
inputter.order = 2

function inputter.appropriate (round, filename, _)
  -- We will later change it.
  return false
end

function inputter:parse (doc)
  local tree = {}
  -- Later we will work on parsing the input document into an AST tree
  return tree
end

return inputter
```

You have written you very first inputter, or more precisely minimal *boilerplate* code for one. One possible way to use it would be to load it from command line, before processing some file in the supported format:

```
sile -u inputters.myformat somefile.xy
```

However, this will not work yet. We must code up a few real functions now.

11.1.2 Content appropriation

What we first need is to tell SILE how to choose our inputter when it is given a file in our input format. The `appropriate()` method of our inputter is responsible for providing the corresponding logic. It is a static method (so it does not have a `self` argument), and it takes up to three arguments:

- the round, an integer between 1 and 3.
- the file name if we are processing a file (so `nil` in case we are processing some string directly, for instance via a raw command handler).
- the textual content (of the file or string being processed).

It is expected to return a boolean value, `true` if this handler is appropriate and `false` otherwise.

Earlier, we said that inputters were checked in their priority order. This was not fully complete. Let's add another piece to our puzzle: Inputters are actually checked orderly indeed, but three times. This allows for quick compatiblitity checks to supersede resource-intensive ones.

- Round 1 expects the file name to be checked: for instance, we could base our decision on recognized file extensions.
- Round 2 expects some portion of the content string to be checked: for instance, we could base our decision on sniffing for some sequence of characters expected to occur early in the document (or any other content inspection strategy).
- Round 3 expects the entire content to be successfully parsed.

For instance, say you are designing an inputter for HTML. The *appropriation* logic might look as follows.

```
function inputter.appropriate (round, filename, doc)
  if round == 1 then
    return filename:match(".html$")
  elseif round == 2 then
    local sniff = doc:sub(1, 100)
    local promising = sniff:match("<!DOCTYPE html>")
      or sniff:match("<html>") or sniff:match("<html ")
    return promising or false
  end
  return false
end
```

Here, to keep the example simple, we decided not to implement round 3, which would require an actual HTML parser capable of intercepting syntax errors. This is clearly outside the aim of this tutorial.[2] You should nevertheless have a basic understanding of how inputters are supposed to perform format detection.

2. The third round is also the most "expensive" in terms of computing, so clever optimizations like caching the results of fully parsing the content may be called for here, but we are not going to consider the topic now.

11.1.3 Content parsing

Once SILE finds an inputter appropriate for the content, it invokes its `parse()` method. The parser is expected to return a SILE document tree, so this is where your task really takes off. You have to parse the document, build a SILE abstract syntax tree, and wrap it into a document. The general structure will likely look as follows, but the details heavily depend on the input language you are going to support.

```
function inputter:parse (doc)
  local ast = myOwnFormatToAST(doc) -- parse doc and build a SILE AST
  local tree = {{
    ast,
    command = "document",
    options = { ... },
  }}
  return tree
end
```

For the sake of a better illustration, we are going to pretend that our input format uses square brackets to mark italics. Lets say our plain text input format is just all about italics or not, and let us go for a naive and very low-level solution.

```
function inputter:parse (doc)
  local ast = {}
  for token in SU.gtoke(doc, "%[[^]]*%]") do
    if token.string then
      ast[#ast+1] = token.string
    else
      -- bracketed content
      local inside = token.separator:sub(2, #token.separator - 1)
      ast[#ast+1] = {
        [1] = inside,
        command = "em",
        id = "command",
        -- our naive logic does not keep track of positions in the input stream
        lno = 0, col = 0, pos = 0
      }
    end
  end
  local tree = {{
    ast,
    command = "document",
  }}
  return tree
end
```

Of course, real input formats will need more than that, perhaps parsing a complex grammar with LPEG or other tools. SILE also provides some helpers to facilitate AST-related operations. Again, we just kept it as simple as possible here, so as to describe the concepts and the general workflow and get you started.

11.1.4 Inputter options

In the preceding sections, we explained how to implement a simple input handler with just a few methods being overridden. The other default methods from the base inputter class still apply. In particular, options passed to the `\include` commands are passed onto our inputter instance and are available in `self.options`.

11.2 Designing an output handler

Outputters usually live somewhere in the `outputters/` subdirectory of either where your first input file is located, your current working directory, or your SILE path.

All output handlers inherit from a **base** outputter. It is an abstract class, providing just one concrete method, and defining a bunch of methods that any actual outputter has to override for the specifics of its target format.

We first need to declare the name of our new outputter, as well as the default file extension for the output file, which will be appended to the base name of the main input file if the user does not provide an explicit output file name on their command line.

```
local outputter = pl.class(base)
outputter._name = "myformat"
outputter.extension = "ext"
```

And then, we have to provide an implementation for all the low-level output methods for a variety of things (cursor position, page switches, text and image handling, etc.)

We are not going to enter into the details here. First, there are quite a lot of methods to take care of. Moreover, the API is not fully stable here, as needs for other output formats beyond those provided in the core distribution may call for different strategies. Still, you might want to study the **libtexpdf** outputter, by far the most complete in terms of features, which is the standard way to generate a PDF, as it names implies, using a PDF library extracted from the TeX ecosystem and adapted to SILE's need.

Chapter 12
Advanced Class Files 1: SILE As An XML Processor

Now we are ready to look at a working 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.

Since DocDook itself doesn't specify anything about presentation such as paper size, you may need to supply values either on the command line or using a preamble. When you use the `-c docbook` command line option to SILE, SILE will use the **docbook** class in spite of any document declaration. In addition, options such as paper size could be set; for example, `-O papersize=legal`.

The class initialization for DocBoox isn't too fancy; it just loads up a couple packages that will get used later.

> *Much of the example code in this chapter is in SIL format using macros. The actual docbook class currently uses Lua functions to specify these commands. The functionality is the same, but the Lua syntax is more flexible and recommended for most use cases. The SILE* `\define` *macros shown here can still be used in a preamble file if desired.*

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 tags like `<code>`, `<filename>`, and `<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 tags for `<code>` and other similar 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.

12.1 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—and it should often be rendered differently in different contexts. For instance, `<article><title>...` will look different to `<section><title>...`. Inside an `<example>` tag, the title will be prefixed 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>` and `<example>`), we will process the title ourselves.

Let's look at `<example>`, which has the added complexity of needing to keep track of an article number.

```
self: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 someone wants 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 `SU.ast.findInTree()` which looks for a particular tag or command name within the immediate children of the current tree:

```
local t = SU.ast.findInTree(content, "title")
    if t then
      SILE.typesetter:typeset(": ")
```

1. Patches, as they say, are welcome.

```
      SILE.process(t)
```

We've output 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 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")
end)
```

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.

12.2 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, with the expected section numbers and titles in XML comments:

```
<section><title>A</title> : \autodoc:example{1. A}
    <section><title>B</title>: \autodoc:example{1.1 B}
    </section>
    <section><title>C</title>: \autodoc:example{1.2 C}
      <section><title>D</title>: \autodoc:example{1.2.1 D}
      </section>
    </section>
```

```
    <section><title>E</title>: \autodoc:example{1.3 E}
  </section>
  <section><title>F</title>: \autodoc:example{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:

```
self: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 = SU.ast.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)
```

Chapter 13
Further Tricks

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

13.1 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. The latter 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—that is, 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:

```
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%pw - 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" });
end
```

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

```
function diglot:_init (options)
```

```
  self.leftTypesetter = SILE.typesetters.base()
  self.rightTypesetter = SILE.typesetters.base()
  self.rightTypesetter.other = self.leftTypesetter
  self.leftTypesetter.other = self.rightTypesetter
  return plain._init(self)
end
```

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. (This is because the fonts are controlled by global settings rather than being typesetter-specific.)

```
function diglot:registerCommands()
  plain.registerCommands(self)

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

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

  -- Other commands will come here...
end
```

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:

```
self: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")

  self: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 matter 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 method 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.) Next we end each paragraph—after adding the glue so that paragraph skips do not get in the way—and go back to handling paragraphing as normal.

```
self: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

    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. 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 also need to make sure that, no matter which typesetter causes an new-page event, the other typesetter also gets correctly initialized:

```
function diglot:newPage (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
```

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:

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

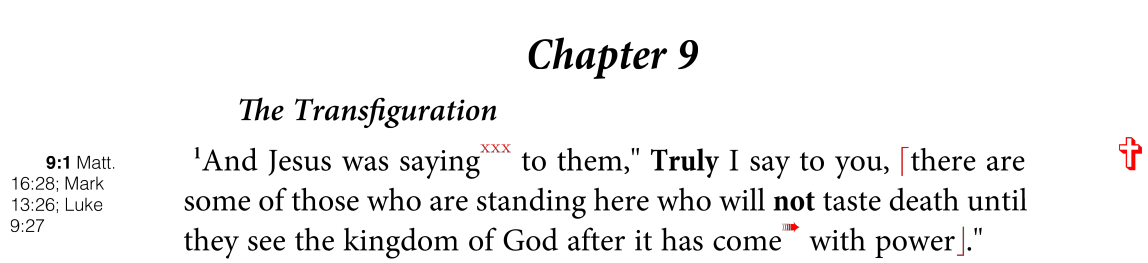
At the end of the document, we will use the emergency `chuck` method. Where `leaveHmode` means "call the page builder and see there's enough material to build a page," `chuck` means "you must get rid of everything on your queue *now*." We add some infinitely tall glue to the other typesetter's queue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

```
function diglot:finish ()
  table.insert(SILE.typesetter.other.state.outputQueue, SILE.types.node.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.

13.2 Sidenotes

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



Sidenotes can be seen as a simplified form of parallel text. With a true parallel layout, 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, we 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:

```
function discovery:typesetProphecy (symbol)
  local margin = self: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 then wrap the output of the command into a hbox. Here, note that we do not use the `\hbox` command: it would put the box into the typesetter's output node queue, but we don't want it to appear in the main typeblock. So we just ask the typesetter to build the box and return it.

```
local hbox = SILE.typesetter:makeHbox(function ()
    SILE.call("prophecy-"..symbol.."-mark")
  end)
```

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 where it must go), 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
  })
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:

```
local base = require("classes.base")

local discovery = pl.class(base)
discovery._name = "discovery"

function discovery:_init ()
  base._init(self)
  local gutter = self:oddPage() and
                 SILE.getFrame("rGutter") or SILE.getFrame("lGutter")
  self.innerTypesetter = self.typesetters.base(gutter)
  ...
  return self
end

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

Now for the function which actually handles a cross-reference. As with the parallels example, we

start by totaling up the height of the material processed on the current page by both the main typesetter and the cross-reference typesetter:

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

This deals with the completed paragraphs which have already been put into the output queue. 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 gotten 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 okay if the cross-reference appears a bit lower than the verse it refers to.

```
if (innerVbox.height < mainVbox.height) then
    self.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.types.node.vglue("7pt"))
    SILE.call("font", {size = "6pt", family="Helvetica", weight="800"}, {})
    self.innerTypesetter:typeset(SILE.scratch.chapter..":"..SILE.scratch.verse.." ")
    SILE.call("font", {size = "6pt", family="Helvetica", weight="200"}, {})
    self.innerTypesetter:typeset(xref)
    self.innerTypesetter:leaveHmode()
    self.innerTypesetter:pushVglue({ height = SILE.types.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!

13.3 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 manipulates 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 of 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()
```

13.4 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 as macros from declarative markup.
- `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. For example, `--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 v0.15.9 (LuaJIT 2.1.ROLLING) [Rust]
<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 names, it enters the REPL:

```
SILE v0.15.9 (LuaJIT 2.1.ROLLING) [Rust]
> l = SILE.types.length("22mm")
> l.length
22mm
> l.absolute()
62.3622054pt
```

At any point during the evaluation of Lua commands, you can call `SILE.repl:enter()` 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`.

13.5 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!