

The SILE Book

for SILE version 0.9.2

Simon Cozens

Table of Contents

SILE とは？	2
SILE と Word	2
SILE と TeX	2
SILE と InDesign	4
結論	4
さあ始めよう	6
基本的な SILE 文書	6
インストール	6
OS X ユーザのためのノート	7
Linux ユーザのためのノート	7
Windows ユーザのためのノート	8
SILE の実行	8
もうちょっとクールに	8
SILE 文書の作成	10
テキスト	10
コマンド	12
環境	13
XML 書式	13
SILE コマンド	16
フォント	16
文書構造	17
章と節	17
脚注	18
インデントとスペーシング	18
分割	18
言語とハイフネーション	19
ファイルの取り込みと Lua コード	20
SILE パッケージ	22
image	22
rules	23
color	24

rotate	24
features	25
unichar	25
bidirectional	26
raiselower	26
grid	26
verbatim	28
他のパッケージにより利用されるパッケージ	28
footnotes	28
counters	29
pdf	29
frametricks	30
insertions	30
twoside	31
masters	31
infonode	31
inputfilter	32
SILE マクロとコマンド	34
単純なマクロ	34
内容を伴うマクロ	35
ネストしたマクロ	36
SILE の設定	38
スペーシング設定	38
タイプセッタ設定	40
ラインブレーキング設定	41
Lua からの設定	42
SILE の内部	44
ボックスとグルー、ペナルティ	44
Lua インターフェイス	45
フレーム	46
文書クラスの設計	50
コマンドを定義する	52
出力ルーチン	53
エクスポート	56
高度なクラスファイル 1: XML プロセッサとしての SILE	58

表題を扱う	59
Sectioning	60
Other Features	61
Further Tricks	62
Parallel Text	62
Sidenotes	65
SILE As A Library	69
Debugging	70

Chapter 1

SILE とは？

SILE は組版システムです。その目的は美しい文書を生成することにあります。SILE について理解する最も良い方法は、あなたが聞いたことがあるであろう他のシステムと比較することでしょう。

1.1 SILE と Word

多くの人たちはパソコンを使って印刷用に文書を作成するとき、Word (Microsoft Office の一部です) や Writer (OpenOffice や LibreOffice に含まれます) といったソフトウェア、あるいはそれらに似たワープロソフトを利用します。しかしながら、SILE はワープロソフトではありません。それは組版システムです。そこにはいくつかの重要な違いがあります。

ワープロソフトの目的は、あなたがスクリーン上で入力したものと全く同一に見える文書を作成することにあります。一方、SILE はあなたが入力したものを、文書を作成するための指示だとみなし、それをもとに可能な限り良く見える文書を生成します。

少し具体的にみてみましょう。ワープロソフトでは、あなたが入力をしている文章が行の右端にさしかかると、カーソルは自動的に次の行に移動します。ワープロソフトは改行位置をあなたに示してくれます。SILE では、あなたが文章を入力している段階では改行位置を知らせてくれません。その段階ではまだそれは明らかになっていないからです。あなたは好きなだけ長い行を打ち込むことができます。SILE はそれを処理する段階になって、パラグラフを構築するために、文章の最適な改行位置を探します。この処理はひとつの入力に対して (最大で) 3 回行われます。ふたつの連続した行がハイフネートされた語で終わっていないか、などあまりよろしくない状況が考慮され、最適な改行位置が見つかるよう処理が繰り返されます。

ページ分割に対しても同様です。ワープロソフトではいずれあなたは新しいページに移動することになりますが、SILE では入力自体は好きなだけ継続されます。文章がどのようにページに分割されるかは文書全体のレイアウトを検討したのちに決定されるからです。

ワープロソフトはしばしば WYSIWYG—What You See Is What You Get (見たままが得られる)—であると言われます。SILE は全く WYSIWYG ではありません。実際、結果はそれが得られるまで分からないのです。むしろ、SILE 文書はテキストエディター—テキストを入力するためのもので、整形された文書を作成するためのものではない—を用いて準備され、PDF 文書を生成するために SILE によって処理されます。

言い換えると、SILE はあなたが求める結果を記述するための言語であって、SILE はあなたが与えた指示に対し、最良の印刷物を得るための文書整形の処理を行います。

1.2 SILE と TeX

いくつかの人たちは、なんだか TeX のようだ、と思うかもしれません。¹ もしあなたが TeX についてよく知らない、あるいは関心がないのであれば、このセクションは読み飛ばしてもらっても構いません。

1. ひとりの TeX ユーザとして言わせれば“ なんだか TeX のようだ ”だろうか。

実際、TeX のようだというのは正しい意見です。SILE は TeX からかなりのものを引き継いでいます。SILE のような小さなプロジェクトが、TeX という、“The Art of Computer Programming” の著者たる某教授の、偉大な創造物の後継者だと名乗るのはおこがましいかもしれませんが…SILE は TeX の現代的な再生です。

TeX は組版システムのなかでも最初期のもののうちのひとつで、それゆえほとんど何もないところから設計されなければならませんでした。そのうちいくつかは時の試練に耐え — そして TeX はその創造から 30 年以上たった今でも最もよく利用される組版システムのうちのひとつであり、それはその設計とパフォーマンスの証である — 多くはそうではありませんでした。実際、Knuth の時代より続く TeX の発展の歴史の大半は、彼の元々の設計を取り除き、新たな業界標準技術で置き換えることでした。例えば、我々は METAFONT ではなく TrueType フォントを使い (xetex のように)、DVI ではなく PDF を使い (pstex や pdftex)、7 ビットの ASCII ではなく Unicode を使い (これも xetex)、マクロ言語ではなくマークアップ言語や組込みのプログラミング言語を使います (xmltex や luatex)。現在、我々が依然として利用する TeX のオリジナルの部分は、(1) ボックスとグルー・モデル、(2) ハイフネーション・アルゴリズム、(3) 改行 (行分割) 処理アルゴリズムです。

SILE は上記 3 つの点を TeX から受け継いでいます。SILE は TeX の改行処理アルゴリズムのほぼ丸写しな移植を含み、それは同じ入力を与えられたとき、TeX と全く同じ出力が得られるようにテストされています。しかしながら、SILE 自身がスクリプト言語で書かれているため、²SILE の組版エンジンの動作を拡張したり、変更したりすることが容易にできます。

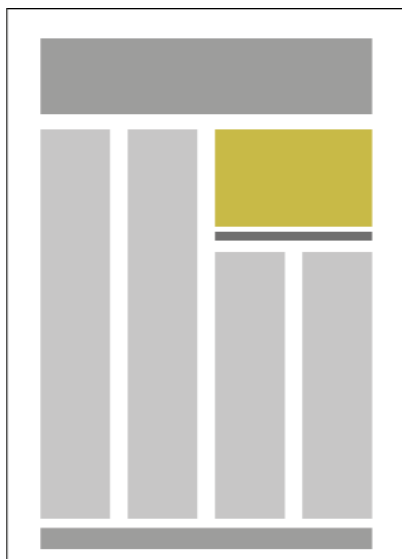
例えば TeX が苦手とすることのひとつとして、グリッド上での組版があります。この機能は聖書を組むような人にとっては重要なものです。これを TeX 上で行う試みはなされてきましたが、どれもひどいものでした。SILE では組版エンジンの動作を変更し、ごく簡単なアドオンパッケージを用意することでグリッド上での組版を可能にします。

もちろん、いまだきだれも plain TeX を使いません — だれもが LaTeX で同様のことを行い、そのうえ CTAN から入手可能な巨大なパッケージ群を活用しています。SILE は未だ TeX が持つような巨大なコミュニティやリソースを持たず、そのようなものを活用することができません。この点において TeX は SILE よりもずっと先を行っています。しかし、可能性という点において、TeX と同等か、あるいはもっと進んでいるとも言えるところがあるかもしれません。

2. もしもあなたが `TeX capacity exceeded` というメッセージに馴染んでいるならば、これはさぞかし興味深いことでしょう。

1.3 SILE と InDesign

人々が出版物をデザインするとき辿りつくツールとして、InDesign（あるいはそれと似た DTP ソフト、例えば Scribus）があります。



InDesign は複雑で高価な商用出版ツールです。それは非常にグラフィカルです — クリックやドラッグといったマウス操作でテキストや画像をスクリーン上で移動させます。SILE は自由なオープンソースの組版ツールで、完全にテキストベースです。SILE ではエディタでコマンドを入力し、それらのコマンドをファイルに保存し、SILE に組版させるために渡します。これらの根本的な違いにかかわらず、この 2 つには共通した特徴があります。

InDesign では文章はページ上のフレームに流しこまれます。左の図は InDesign でよくあるレイアウトがどのようなものかを示しています。

SILE もまたページ上でどこに文章が表示されるべきかを、フレームという概念を用いて決定します。そのため SILE では TeX でできるよりもっと複雑で柔軟なページレイアウトを設計することが可能です。

InDesign で有用な機能として、構造化された XML データ形式を用いたカタログや名簿などの出版があります。InDesign でこれを行うには、まずそれぞれの XML 要素にどのようなスタイルが適用されるか宣言します。データが InDesign に読み込まれると、InDesign は与えられたルールに従ってデータを整形し出力します。

あなたは全く同じことを SILE でできるのです。ただし SILE では XML 要素がどのように整形されるのかをより詳細に制御することができ、これは SILE ではあなたが XML 要素を処理するのに、例えば Lua コードを呼び出したりすることができるからです。SILE はコマンドラインのフィルタープログラムであるため、適切な指示が与えられれば、XML ファイルから PDF へ、いとも簡単に変換することができます。これは素晴らしいことです。

この解説書の最後の章では、複雑な XML 文書をスタイル付して PDF を生成するための クラスファイルのいくつかの例を示します。

1.4 結論

SILE は入力として与えられたテキストの指示をもとに PDF を出力します。SILE は TeX と InDesign にインスパイアされた機能を持ち、かつより柔軟で拡張可能、プログラム可能なものを目指しています。この文書（これは SILE で書かれています）のようなものを作成したり、構造化されたデータを整形して出力するシステムとして有用です。

SILE とは？

Chapter 2

さあ始めよう

さて、SILE とは何か、何をするものなのか、いづれか理解したところで SILE そのものについて話題を移しましょう。

2.1 基本的な SILE 文書

SILE をどうやって使用するのか示す前に、SILE 文書がどのようなものなのかひとつ例を示しましょう。これは SILE に対する入力であり、SILE によって処理され PDF ファイルへと変換されるものです。

これらの文書はプレーンテキストです。あなたがあなた自身の SILE 文書を作成するにはテキストエディタが必要です。Unix 上では例えば、vi や emacs、Mac OS X では Sublime Text、TextMate、あるいは TextEdit など、Windows では Notepad や Notepad+ などです。SILE の入力として用いるにはテキストファイルとして保存する必要があります。Word のようなワープロソフトでは作成できません。それらは文書をプレーンテキストではなく独自のフォーマットで保存するからです。

とりあえず、もっとも簡単な SILE 文書から始めましょう。

```
\begin[papersize=a4]{document}
Hello SILE!
\end{document}
```

今のところは、SILE 文書はこのようなものだというだけにしておいて、詳細は次の章で取り上げましょう。

分かり切ったことを言うようですが、これは左上部に **Hello SILE** と書かれ、ページ番号 (1) がページ下部中央に配置された A4 サイズの PDF 文書を生成します。さて、どうやってその PDF を得るのでしょうか？

2.2 インストール

なにはともあれ、あなたは SILE を手に入れ、あなたのパソコンで走らせなければなりません。SILE はホームページ <http://www.sile-typesetter.org/> から入手できます。

SILE をインストールし、実行するにはいくつか他のソフトウェアが必要です—Lua プログラミング言語のインタプリタと Harfbuzz テキストシェーピング・ライブラリです。

さあ始めよう

SILE にはそれ自身の PDF 生成ライブラリが付属しており、それもまたいくつかのソフトウェアを要求します。**freetype**、**fontconfig**、**libz**、そして**libpng**です。¹

これらの依存ライブラリがインストールされれば、次は Lua ライブラリをそろえる必要があります。

- `luarocks install lpeg`
- `luarocks install luaexpat`

以上のことが済めばようやく本題に移れます。SILE のホームページからダウンロードしたファイルを解凍し、ディレクトリを移動してから以下を実行します。

- `./configure; make`

これが終われば SILE を未インストールの状態で実行できます。

- `./sile examples/simple.sil`

すべてが順調であれば、**examples/simple.pdf** というファイルが生成されるはずです。

SILE を本格的に使うには**sile** コマンドと SILE ライブラリ・ファイルをシステムにインストールします。これを行うには次のようにします。

- `make install`

これで**sile** コマンドがどのディレクトリからも利用可能になりました。

2.2.1 OS X ユーザのためのノート

Homebrew (Mac OS X ではおすすめてです) を利用する Mac OS X 上でこれらをインストールするには、

- `brew install automake libtool harfbuzz fontconfig libpng lua luarocks freetype`

もしも OS X で Homebrew を利用しているのであれば、環境変数**PKG_CONFIG_PATH**を適切に設定する必要があるかもしれません。(**PKG_CONFIG_PATH=/usr/local/lib/pkgconfig:/usr/local/lib**) あるいはライブラリ依存関係参照のための**pkgconfig** ファイルを提供する**brew link**を使用してください。configure スクリプトはどのライブラリが見つからなかったか警告してくれるでしょう。

SILE をシステムにインストールせずに実行するには、環境変数を次のように設定する必要があります。

`DYLD_LIBRARY_PATH=./libtexpdf/.libs ./sile examples/simple.sil`

2.2.2 Linux ユーザのためのノート

Debian や Ubuntu などの Debian 系 Linux OS では、

- `apt-get install lua5.1 luarocks libharfbuzz-dev libfreetype6-dev libfontconfig1-dev libpng-dev`

Redhat 系 Linux では次のようになるでしょう。

- `yum install harfbuzz-devel make automake gcc freetype-devel fontconfig-devel lua-devel lua-lpeg lua-expat libpng-devel`

1. 代わりに Pango と Cairo を使うようにもできますが、その出力は特に Linux において劣ります。あえてそうする場合は**libcairo-gobject2** と**libpango1.0-0** パッケージをシステムにインストールし、**lgi** Lua モジュールを追加する必要があります。

2.2.3 Windows ユーザのためのノート

Windows でもmingw32 環境で SILE を動作させることができたとのユーザからの報告があります。現在のところ、確実な方法はありませんが、<https://github.com/simoncozens/sile/issues/82> での議論が参考になるでしょう。

2.3 SILE の実行

では新たなディレクトリに移り、テキストエディタを開いて先ほど例示した内容をファイル `hello.sil` に保存しましょう。そしてコマンドを実行します。

- `sile hello`

(SILE は引数のファイル名に拡張子が与えられなければ、自動的に拡張子 `.sil` を追加します)

これによってファイル `hello.pdf` ができるでしょう。あなたはめでたく SILE での最初の文書を作成することができました。

2.4 もうちょっとクールに

`examples/article-template.xml` は典型的な DocBook 5.0 文書です。DocBook を印刷する場合、しばしば、XSLT プロセッサ、FO プロセッサ、そして場合によっては奇妙な LaTeX パッケージに振り回されなければなりません。しかし、SILE は XML ファイルを読み込むことができ、しかも DocBook (実際にはそのサブセット) を処理するための `docbook` クラスが付属しています。

例、`examples/article-template.xml` を `examples/article-template.pdf` に変換するには、単純にこうします。

```
% ./sile -I docbook examples/article-template.xml
This is SILE 0.9.2
Loading docbook
<classes/docbook.sil><examples/article-template.xml>[1] [2] [3]
```

ここで `-I` フラグは入力ファイルを読み込む前に クラス ファイルを読み込むための指示です。 `docbook` クラスファイルが読み込まれたのち、DocBook ファイルは直接読み込まれ、タグは SILE コマンドとして解釈されます。

第 10 章では `docbook` クラスがどのようなものか見てみます。そこでは他の XML フォーマットをいかに処理するか学ぶでしょう。

さあ始めよう

Chapter 3

SILE 文書の作成

さて、ここで最初の例に戻りましょう。

```
\begin[papersize=a4]{document}
Hello SILE!
\end{document}
```

文書は`\begin{document}` コマンドで始まります。それには用紙サイズの指定が必須です。そして文書は`\end{document}` で終わります。その間には2種類の SILE 文書を構成する要素が来ます。ページ上に出されるテキスト、ここでは“Hello SILE!”、とコマンドです。

用紙サイズ

SILE は国際規格 ISO の A・B・C シリーズの用紙サイズを認識します。これに加えて次の伝統的によく用いられる用紙サイズも利用可能です。letter、note、legal、executive、halfletter、halfexecutive、statement、folio、もしも標準的でない用紙サイズを指定したければ、具体的なサイズを直接指定することも可能です。`papersize=<basic length> x <basic length>`。

単位

SILE では長さを指定するいくつかの方法があります。上記`<basic length>` は数と単位（の省略記号）の指定からなります。認識される単位はポイント（pt）、ミリメートル（mm）、センチメートル（cm）、インチ（in）です。例えば、ペーパーバックサイズの B-format は`papersize=198mm x 129mm` のように指定されます。後ほど長さを指定する別の方法についてもみることとなるでしょう。

3.1 テキスト

通常のテキストについてはこれといって述べることはありません。単に入力してください。

TeX ユーザーは SILE がテキストについても何らかの処理を行うものと期待するかもしれません。例えば、あなたが TeX において、ふたつの連続したバッククォート（```）を入力すると、TeX はそれを開始用のダブルクォート（`“`）に置き換えてくれます。SILE はそのようなことは行いません。ダブルクォートを入力してください。同様に en ダッシュと em ダッシュでも、`--` や`---` ではなく、Unicode で該当する文字を入力してください。

テキスト処理においていくつか挙げる点があるとすれば以下のものでしょうか。

まずひとつ目は、スペースの扱いについてです。もしあなたがスペース 3 つを用いて `Hello SILE!` と書いたとしても、それはスペース 1 つ分、`Hello SILE!` と同じ結果になります。また、行の先頭にあるスペースは無視されます。

同様に、改行文字を好きなところに入れることができます。¹SILE はパラグラフ全体を取扱い、与えられた行長で可能な、最適な改行位置を計算します。例として挙げるならば、あなたの入力が仮に

```

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.
```

だったとしても、SILE の出力において 'eiusmod' で改行が起こるとは限りません。改行は常に、適切な位置で行われます。実際の出力は以下のようなものとなるでしょう。

```

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.
```

言うならば、改行文字はスペースに変換されます。(場合によってはこれは望ましいことではありません。もし行の終わりに余分なスペースを加えたいのであれば、行の末尾でコメント文字 `%` を使います) パラグラフを終了する場合は、改行を 2 個続けて入れてください。例えば、

Paragraph one.

Paragraph two.

This is not paragraph three.

This is paragraph three.

注意点として挙げられるふたつ目は、いくつかの (4 つです) 文字は SILE では特別な意味を持つことです。これらは TeX ユーザにとっては馴染み深いものでしょう。

バックスラッシュはコマンドの開始に用いられます。(コマンドの詳細については後ほどすぐに述べましょう) 波括弧 (`{`, `}`) はグループ化に、特にコマンドの引数を扱う際に、用いられます。最後にパーセント記号はコメント行の開始として用いられます。パーセント記号から次の改行文字までは SILE

1. 訳注: わかり書きをする言語ではです。改行文字はスペース 1 個分と同じように扱われます。

によって無視されます。これらの文字を出力したければ、バックスラッシュを前に付けましょう。`\`は`¥`²を、`\{`は`{`を、`\}`は`}`を、そして`\%`は`%`を出力します。

3つ目の点はハイフネーションです。SILEはそれによってパラグラフ全体の見た目が良くなると判断できるときはいつでも、自動的に語をハイフネートして改行します。ハイフネーションはその時の言語の設定が反映されます。特に指定がなければ、SILEはデフォルトで英語を仮定し、ハイフネーション処理を行います。上記のラテン語のテキストの例ではハイフネーションは無効化されています。

最後に挙げる点はリガチャです。(ふたつあるいはそれ以上の文字が、見た目を良くするために、ひとつの文字に結合される) SILEは自動的にリガチャ処理を行います。このため、あなたがもし`affluent fishing`と入力すると(実際には使用するフォントに依存します)、出力結果は`'affluent fishing'`のようになります。リガチャを抑制したい場合は、空のグループ(グループ化文字`{ }`を使って)を挿入します。`af{}f{}luent f{}ishing`では`affluent fishing`のようになります。リガチャやその他の機能の制御に関する詳細についてはOpenType フィーチャの節を参照してください。

3.2 コマンド

典型的な(この点に関しては後ほど再検討しましょう) SILE コマンドは、バックスラッシュで始まり、コマンド名が続く文字列です。そして文書は`\begin{document}` コマンドで始まり、`\end{document}` で終わります。

コマンドはまた、ふたつの必須でない部分を持ちます。それはパラメータと引数です。文書を開始する時の`\begin` コマンドはその良い例です。³

```
\begin[papersize=a4]{document}
```

コマンドのパラメータは角括弧で囲まれ、`key=value` の形をとります。複数のパラメータを指定する場合は、コンマやセミコロンを使って、`[key1=value1,key2=value2,...]` のように続けます。`"key"` の前後のスペースは重要ではありません。`[key1 = value1; key2 = value2; ...]` のように書くこともできます。もしもコンマやセミコロンをパラメータの値に使いたければ、引用符で値全体を囲います。`[key1 = "value1, still value 1", key2 = value2; ...]` のように。

コマンドは引数をとるかもしれませんが、その場合は波括弧で囲います。⁴

SILEはコマンド直後のスペースや改行文字を無視します。

以下にいくつかの SILE コマンドを示しましょう。

<code>\eject</code>	% A command with no parameters or argument
<code>\font[family=Times,size=10pt]</code>	% Parameters, but no argument
<code>\chapter{Introducing SILE}</code>	% Argument but no parameters

2. 訳注: フォントによっては円記号になってしまいます。
3. 厳密に言うと`\begin` はコマンドではありませんが、とりあえず今はそういうことにしましょう。
4. TeX ユーザはつい括弧を忘れてしまうかもしれませんが、それはいけません。SILE では括弧は必須です。

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

3.3 環境

`\chapter` や `\em` (イタリック体による強調) といったコマンドは、せいぜい数行の比較的短いテキストを囲むために用いられます。もっと長い、文書の一部を構成する部分を囲みたい場合は、環境を使います。環境は `\begin{name}` で始まり、対応する `\end{name}` までをその中に含みます。ひとつの例が既に出ていますね。document 環境で、これは文書全体を囲みます。

内緒ですが、コマンドと環境の間には全く違いはありません。いうなれば、以下のふたつは等価なのです。

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

しかしながら、いくつかの場面では、環境を用いたほうが読みやすく、どこからどこまでコマンドが影響するのか認識しやすくなります。

3.4 XML 書式

実際のところ、SILE はこれまで示したものとは完全に異なる入力フォーマットを受け付けます。これまで例示してきたものは「TeX 風書式」でしたが、もし入力ファイルの最初の文字が山括弧 (実際は不等号記号 `<`) であった場合は、SILE は入力ファイルが XML 書式であると捉えます。[もしそれが整形式の (well-formed) XML 文書でなければ、SILE は非常に機嫌を損ねるでしょう]

入力ファイル中のすべての XML タグは、SILE コマンドであると解釈され、属性はパラメータであるとみなされます。このため、ふたつのファイルフォーマットは実際的には等価です。ただひとつの例外を除いては。XML 書式の場合は SILE 文書は任意のタグで始まってもよいのです。(習慣として SILE 文書には `<sile>` を用いるのが好ましいですが)

例えば、XML 形式で前述の例文を示すと、

```
<sile papersize="a4">
Hello SILE!
</sile>
```

引数を取らないコマンドはすべて整形式の self-closing⁵ タグ（例えば<break/>）でなければならず、パラメータ付のコマンドはその属性が整形式でなければなりません。前に挙げた例を XML 書式で書くと、

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

XML 書式は人間が直接書くことを想定しているわけではありませんが—TeX 風書式のほうがそれには向いているでしょう—XML 書式に対応することは、コンピューターで SILE を扱うのをより容易にします。例えば SILE 文書を編集するための GUI インターフェイスを作ったり、他の XML 書式を SILE のそれに変換したり。

しかしながら、SILE においては XML 文書进行处理するためのよりスマートな方法が存在します。そのためには、あなたはあなた自身の SILE コマンド、それは非常に単純な文書整形用のものから SILE の動作を根本から変えるものまでを含む、を定義できることを知る必要があります。あなたが特定の XML 形式のファイル—仮に DocBook としましょう—を持っているとします。あなたはすべての可能な DocBook タグに対する SILE コマンドを定義します。するとあなたの DocBook ファイルは SILE 入力ファイルとしてそのまま使えるようになるのです。

最後の 2 章では、SILE コマンドを定義と XML 文書进行处理する例を示しましょう。

5. 訳注：適切な訳語が分からないが、開始・終了のペアではなく、単体で存在するタグのこと。

Chapter 4

SILE コマンド

さて、それでは SILE の具体的な使用法について見ていきましょう。まずはあなたが SILE で文書を作成し始めるのに最も役立つコマンドから始め、次第により細かな点について進んでいきます。

4.1 フォント

テキストの見た目を変えるもっとも基本的なコマンドは `\font` コマンドです。これは次のような書式をとります。

- `\font[parameters...]{argument}`
- `\font[parameters...]`

最初の書式では引数として与えられたテキストを指定されたフォントで描画します。次の書式ではそれ以降のテキストすべてに影響します。

例として挙げると、

Small text

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

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

Still big text!

は

Small text

Big text!

Bigger text

Still big text!

となります。

ここで見たように、属性として可能なものとして、`size` があります。これは、`<dimension>` で指定されます。ここで、`<dimension>` は以前登場した `<basic length>` のようなものですが、これは現在のフォントのサイズに対する相対的な値として指定可能です。例えば、ex ユニット (`ex`)、であったり、em ユニット (`em`)、あるいは en ユニット (`en`) です。

`\font` コマンドで指定可能な属性値は、

- `size` – 先に述べたとおりです。

- `family` – 使用するフォント名が来ます。フォントをその名前で指定するには、SILE はシステムにインストールされたすべてのフォントについて知る必要があります。SILE の XML 書式では、フォントファミリーは CSS 形式のコンマで分離された ‘スタック’ として指定可能です。
- `style` – `normal` または `italic` です。
- `weight` – CSS 形式のウェイトを表す数値が来ます。有効な値は 100 と 200 から 300、400、500、600、700、800、900 までです。フォントによっては全てのウェイトがサポートされているとは限りませんが（ふたつ程度かもしれません）、SILE は最も近いものを選択します。
- `language` – 2 文字からなる (ISO639-1) 言語コードです。これはスペーシングとハイフネーションの両方に影響を与えます。
- `script` – スクリプト（文字体系、用字系）の指定です。後で述べる「言語とハイフネーション」の節を参照してください。

手動で陽にフォント指定を行うのは非常に面倒ですね。後ほどこれを自動化する方法についても見てみましょう。SILE は `\em{...}` コマンドを `\font[style=italic]{...}` のショートカットとして提供します。ボールド体に対するショートカットはありません。なぜならそれはあまり良い習慣とは言えないからです。そのようなものを簡単に行う方法は与えないことにしましょう。

4.2 文書構造

SILE は様々な文書クラス (LaTeX のクラスと似た) を提供します。デフォルトでは、文書の構造化をごくわずかにサポートするのみの、`plain` クラスが用いられます。他には `book` クラスがあり、これは左右のページマスタ、ヘッダと脚注、章、節などのヘッディング（柱）をサポートします。

この節のコマンドを使うには、あなたの文書の `\begin{document}` コマンドで `book` クラスを指定する必要があります。あなたが今読んでいるこの文書は実際に、`\begin[papersize=a4,class=book]{document}` で始まります。

4.2.1 章と節

あなたは文書を `\chapter{...}`、`\section{...}`、そして `\subsection{...}` などのコマンドを使って分割することができます。これらのコマンドは引数として、その章や節の見出しをとります。章は新たな左ページから始まり、章の見出しは左ページのヘッダに表示されます。加えて、節の見出しは右ページのヘッダに表示されます。

章や節は自動的に 1 から番号付けされて開始されます。この動作を変更するには、次の章の `counters` パッケージの解説を参照してください。番号付けを抑制したければ、パラメータ `[numbering=no]` を与えます。

この副節はコマンド `\subsection{章と節}` で開始されています。

4.2.2 脚注

脚注は`\footnote{...}` コマンドでつけることができます。¹脚注コマンドに対する引数はページ下部に表示される脚注の内容です。これは各章ごとに、自動的に 1 から番号付けされます。

4.3 インデントとスペーシング

SILE では、パラグラフは通常インデントされます（デフォルトで 20 ポイント幅です）。これを抑制するには`\noindent` コマンドを、パラグラフの先頭に付与します。（このパラグラフのような、最初のパラグラフでは`\noindent` は必要ありません。なぜなら`\section` と`\chapter` は自動的に、章や節の見出しに続く文章に対してそれを呼ぶからです）`\noindent` は`\indent` コマンドを続けて呼ぶことで打ち消すことができます。

パラグラフ間、あるいはパラグラフと他の要素との間の垂直方向のスペース分量を増やすには、`\smallskip`、`\medskip` および`\bigskip` が使えます。これらはそれぞれ、3pt、6pt、12pt のスペースに相当します。このパラグラフの後に`\bigskip` を入れてみましょう。

水平方向のスペースを行ないに挿入するには、小さなものから大きなものへ順に、`\thinspace`（em の 1/6）、`\enspace`（1em）、`\quad`（1em）、そして`\qquad`（2em）。

`center` 環境中（`\begin{center} ... \end{center}`）では中央寄せとなります。例えばこのパラグラフのように。

4.4 分割

SILE は行とページの分割を自ら決定します。後の章ではこのプロセスを微調整する 設定法 を紹介しましょう。しかしながら、SILE の plain クラスにもそれを助けるためのいくつかの方法が存在します。

パラグラフ間に挿入された`\break` コマンドはフレーム分割を引き起こします。（`\framebreak` と`\eject` という同義のコマンドも存在します）もし、複数のフレームがページ内にあれば、— 例えば、多段組みの文書 — 現在のフレームが終了し、次のフレームの先頭から処理は続けられます。`\pagebreak`（あるいは`\supereject`）はより強制力のあるもので、これはページ上に更なるフレームが残っていても新しいページを開始します。より穏やかな変種としては、`\goodbreak`、これは SILE にそこが良いページ分割点であると教えるもの、があります。それとは反対に、`\nobreak` は分割を抑止する働きがあります。これらの中間的なものとして、`\allowbreak` があり、SILE にページやフレームの分割に適さないかもしれないが、それを許可するよう指示するものとして利用できます。

パラグラフの中では、これらのコマンドは全く別の意味を持ちます。`\break` コマンドは改行を指示し、同様に、`\goodbreak`、`\nobreak`、および`\allowbreak` も行分割に対応します。もしもページ分割を特に禁止したければ、`\novbreak` を使います。

1. このように。 `\footnote{ このように }`。

SILE は通常、両端揃えを行います—すなわち、SILE は一行がちょうど与えられた行長でぴったり収まるように単語間のスペースを調整します。² 両端揃え以外には左揃えがあります。左揃えでは単語間のスペースは均等になるかわり、パラグラフの右端はきれいに揃いません。左揃えはしばしば子供向けの本に用いられたり、新聞のような一行の幅が狭い状況でも用いられます。左揃えを行うには、文章

を `\begin`

`{raggedright}` 環境を囲います。このパラグラフは左揃えで組まれています。

同様に、`raggedleft` 環境もあります。これはパラグラフの右側は揃え、逆に左はがたつきます。このパラグラフは右揃えで組まれています。

4.5 言語とハイフネーション

SILE は現在選択されている言語の設定に基づいてハイフネーションを行います。(言語設定は前に見たように `\font` コマンドで行います) SILE は様々な言語のハイフネーションをサポートしています。また、その言語特有の組版ルールについてもサポートすることを目的としています。

SILE はまた、`xx` という特別な「言語」、を理解します。これはなんのハイフネーションパターンもないものです。この言語に切り替えると、ハイフネーションは行われません。コマンド `\nohyphenation{...}` が `\font[language=xx]{...}` のショートカットとして利用できます。

ハイフネーション以外にも、言語ごとに組版上の規則は異なりますが、SILE はほとんどの言語とスクリプトに対する基本的なサポートを備えます。(もしも SILE が適切に処理出来ない言語やスクリプトがあれば知らせてください。対応します)

いくつかの言語では、同じ文字を使うが異なるように組まれるという状況が生じます。例えば、Sindhi と Urdu はアラビア文字 `heh` を標準的なアラビア語とは異なるやり方で結合します。そのような場合は、あなたは `language` と `script` オプションを `\font` コマンド中で適切に指定しなければなりません。

Standard Arabic

```
\font[family=Scheherazade,language=ar,script=Arab]{\font{...}};
```

Sindi:

```
\font[family=Scheherazade,language=snd,script=Arab]{\font{...}};
```

Urdu:

```
\font[family=Scheherazade,language=urd,script=Arab]{\font{...}}.
```

Standard Arabic: ههه; Sindhi: ههه; Urdu: ههه.

(`script` オプションの完全なリストについては <http://www.simon-cozens.org/content/duffers-guide-fontconfig-shd-harf-buzz> を参照) を行長に厳密に合うようにすることを意味しません。SILE はある程度の調整を行います、最善を尽くした後、最も悪くないと思われる結果を出力します。いくつかの語がわずかに余白に突き出る結果となることもあります。

4.6 ファイルの取り込みと Lua コード

長大な文書を作成するとき、あなたは SILE 文書を複数のファイルに分割して管理したくなるでしょう。例えば、それぞれの章を別のファイルに小分けしたり、ユーザー定義のコマンドを開発し（第 6 章を参照）、それをひとまとめのファイルにして文書の本文とは分けて管理したり。その場合、異なる SILE ファイルを取り込む必要があります。

その機能は `\include` コマンドにより提供されます。これには、必須の `src=<path>` パラメータによりファイルへのパスを示す必要があります。例えば、あなたは学位論文を次のように書きたくなるでしょう。

```
\begin[papersize=a4,class=thesis]{document}
\include[src=macros]
\include[src=chap1]
\include[src=chap2]
\include[src=chap3]
...
\include[src=endmatter]
\end{document}
```

`\include` は入れ子になっても構いません。ファイル A がファイル B を取り込み、それがまたファイル C を取り込んだり。

SILE は Lua プログラム言語で書かれており、Lua インタプリタが実行時に利用可能です。ちょうど HTML 文書中で Javascript コードを `<script>` タグで実行するように、SILE 文書中では Lua コードを `\script` コマンドを用いて実行可能です。（XML 書式ではちょうどよく見えるでしょう）このコマンドはふたつの形態をとります。ひとつは `\script[src=<filename>]` で Lua スクリプトをファイルごとに取り込み、もうひとつは `\script{...}` で、インラインの Lua コードです。

インラインで何か面白いことをやるには SILE の内部に関する知識が必要です（幸運なことにコードはそれほど複雑ではない）が、とりあえず手始めに、Lua 関数 `SILE.typesetter:typeset(...)` を使ってみましょう。これはページにテキストを加えます。`SILE.call("...")` は SILE コマンドを呼び出し、`SILE.typesetter:leaveHmode()` は現在のパラグラフを終了し、テキストを出力します。例として、

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

SILE コマンド

は以下を出力します。

```
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.
```

Chapter 5

SILEパッケージ

SILEには付加的な機能を提供する様々なパッケージが付属しています。事実、SILEの実際の「中核」(“core”)となる機能はかなりコンパクトで拡張性に富み、ほとんどの重要な機能はアドオンパッケージとして提供されています。SILEパッケージはLuaプログラミング言語で書かれており、新たなコマンドを定義したり、SILEの動作を変更したり、実際のところ、Luaでできることは何でもできます。

先に述べたとおり、パッケージのロードは`\script`コマンドで行われ、これはLuaコードを実行します。規約として、パッケージはあなたの作業ディレクトリ、またはSILEのインストールディレクトリの、`packages/`サブディレクトリに入れることになります。例えば、すぐ後に述べる`grid`パッケージは通常は`/usr/local/lib/sile/packages/grid.lua`にあります。これをロードするには、

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

とします。

SILEのパス検索

SILEは様々なディレクトリを検索します。まずはカレントディレクトリ、次に、もし環境変数`SILE_PATH`が設定されていれば、SILEはそのディレクトリを、そして標準的なインストールディレクトリ、`/usr/lib/sile`や`/usr/local/lib/sile`。TeXとは異なり、SILEはサブディレクトリを再帰的に検索しません。このためもしあなたが、あなたのマクロやクラス、パッケージファイルなどをサブディレクトリに置いたら、あなたはその完全な相対パスを指定しなければなりません。

5.1 image

テキスト以外にもSILEは画像を挿入することができます。

`image`パッケージをロードすることで、HTMLのそれと同様の、`\img`コマンドが使えるようになります。`img`は次のふたつのパラメータを取ります。`src=...`は画像ファイルへのパスで、またオプションとして、`height=...`あるいは`width=...`パラメータを表示する画像のサイズとして指定します。もしもサイズが指定されなければ、画像はその「自然な」¹ピクセルサイズで表示されます。

デフォルトの`libtexpdf`バックエンドでは、画像はJPEG、PNG、EPS、PDFフォーマットがサポートされます。`Pango/Cairo`バックエンドではPNGのみです。

1. 訳注：原著は‘natural’です。どういう意味なのかよく分かりません。

それでは 200x243 ピクセルの画像を、`\img[src=documentation/gutenberg.png]` で表示させてみましょう。



それと、(それぞれ) `\img[src=documentation/gutenberg.png,width=120px]`、
`\img[src=documentation/gutenberg.png,height=200px]`、
`\img[src=documentation/gutenberg.png,width=120px,height=200px]` です。



画像は、あたかも非常に大きな文字のように、テキストのベースラインに沿って配置されることに注意してください。

5.2 rules

`rules` パッケージは罫線を描画します。これはふたつのコマンドを提供します。

まずは `\hrule` で、これは与えられた太さ（高さ）と長さ（幅）の線分を描画します。

罫線は他のテキストと全く同じように取り扱われます。このため、パラグラフの途中で、このように `——` 描画することも可能です。（これは `\hrule[width=20pt, height=0.5pt]` で生成されました）

画像と同じく、罫線はテキストのベースラインに沿って配置されます。

`rules` パッケージで提供される、ふたつ目のコマンドは `\underline` で、これは下線を引くものです。

「下線」は文書を作成する上で、あまり良い習慣とは言えません。決して使わないでください。

(これは `\underline{決して}` で生成されました)

5.3 color

`color` パッケージは、テキストや罫線の色を一時的に変えるためのものです。このパッケージはひとつのパラメータを取る、`\color` コマンドを提供します。パラメータは `color=<color specification>` で引数として与えられたテキストや罫線を、その色で描画します。色の指定法は HTML と同じです。16 進数値 `x` を使った RGB 値で `#xxx` または `#xxxxxx` の形式 (例えば、`#000` は黒で `#fff` は白、`#f00` は赤)、または HTML と CSS における名前付きの色指定が可能です。

HTML と CSS の名前付き色のリストは <http://dev.w3.org/csswg/css-color/#named-colors> にあります。

例として挙げると、このテキストは `\color{color=red}{...}` での出力です。

罫線の例も挙げておきましょう。 `\color{color=#22dd33}`: 

5.4 rotate

`rotate` パッケージは回転機能を提供します。これにより、`rotate=<angle>` をフレーム宣言に加えることで、フレーム全体を回転させることができます。また、`\rotate[angle=<angle>]{...}` コマンドで、あらゆるものを回転させることができます。ここで `<angle>` は角度を「度」で表したものです。

回転される描画物はボックスの中に配置され、回転されます。その高さや幅は再度計算され、組版のための通常の水平リストに送られます。このため、回転される内容の周囲には余白が確保されます。このことを理解するために実例をいくつか示しましょう。 here is some text rotated by *ten*, *twenty* and *forty* degrees.

前行は以下のコードで生成されました。

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

5.5 features

Chapter 3 で述べたように、SILE は自動的に、フォントで定義されたリガチャを適用します。これらのリガチャは、フォントファイルの フィーチャ² テーブルで定義されています。リガチャ（複数のグリフがひとつのグリフとして表示される）の他にも、フィーチャテーブルでは様々なグリフ置換が定義されています。

features パッケージは、SILE があなたが選択したフォントに対し、有効にするフィーチャを選択する機構を提供します。どのようなフィーチャが存在するかはフォントに依存します。いくつかのフォントでは、それがどのようなフィーチャをサポートしているのかを説明するマニュアルが付属しています。OpenType フィーチャに関する議論はこの文書の範疇を超えているのでここでは行いません。

フィーチャは、`\font` コマンドのオプションで、「そのままの」フィーチャ名を渡すことで有効・無効にできます。

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

しかしながら、この方法は扱いづらく、フィーチャコードを覚えておく必要が出てきます。**features** はふたつのコマンドを提供します。`\add-font-feature` と `\remove-font-feature` で、OpenType フィーチャへのより簡単なアクセスを実現します。インターフェイスは TeX のパッケージの **fontspec** に由来しています。サポートされている OpenType フィーチャの完全な解説については、**fontspec** パッケージの文書を参照してください。³

以下に **features** パッケージで任意の (discretionary) リガチャと歴史的 (historic) リガチャを制御する例を示します。

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

5.6 unichar

SILE は Unicode 対応であり、その入力 は UTF-8 エンコーディングです。（サポートされる Unicode の範囲に関しては、使用するフォントに依存します）Unicode で定義されたいくつかの文字はキーボードから直接入力するのは困難です。このため、**unichar** パッケージはこの問題に対処するための方法、Unicode コードを直接指定するコマンド、を提供します。**unichar** をロードすると、`\unichar` コマンドが利用可能となります。

2. 訳注：原文は単に features。OpenType などの高度な組版拡張機能のこと

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

`\unichar{U+263A} % produces ☺`

`\unichar` の引数が `U+`、`u+`、`0x` あるいは `0X` で始まる場合は、それは 16 進数値だとみなされます。そうでなければ 10 進数であると仮定されます。

5.7 bidi

ラテン文字などの用字系では文章は左から右へ (LTR) 進みます。しかしながら、いくつかの用字系、特にアラビア語やヘブライ語では、右から左に (RTL) 進みます。`bidi` パッケージ、これはデフォルトでロードされます、は右から左へ書き進める場合の文書や、LTR と RTL を混在させるような文書を、正しく組むための機能を提供します。これはデフォルトで読み込まれるため、パラグラフ中で LTR と RTL テキストの両方を書くことができ、SILE は正しい順序でそれらの文字が出力されることを保証します。

`bidi` パッケージはふたつのコマンド、`\thisframeLTR` と `\thisframeRTL`、これらは現在のフレームのデフォルトの書字方向を設定する、を提供します。すなわち、もしあなたが SILE にフレームが RTL であると指示するならば、文章は右端から始まり、左に進みます。このパッケージはまた、`\bidi-off` と `\bidi-on` コマンドを提供します。bidirectional サポートを無効にすることで、もしかしたら処理速度を向上させることができるかもしれません。

5.8 raiselower

もしあなたが、画像や罫線、テキストなどがベースライン上に沿って配置させてくれないならば、`raiselower` パッケージを使ってそれらを上げ下げすることができます。(footnote パッケージは脚注の参照番号を、上付き数字として表示するため、このパッケージを利用しています。)

これはふたつの単純なコマンドを提供します。`\raise` と `\lower` で、どちらも `height=<dimension>` をパラメータとして取ります。それぞれ引数となるものを、与えられた量だけ持ち上げたり、下げたりします。これらは行の高さや深さに影響しません。

Here is some text raised by three points; here is some text lowered by four points

前のパラグラフは以下のコードで生成されます。

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

5.9 grid

SILE は通常、以下に示すふたつのルールに従って行と行の間のスペースを決めます。

- SILE はベースライン間が、`baselineskip` と呼ばれるある固定された距離となるように、連続したふたつの行の間にスペースの挿入を試みる。

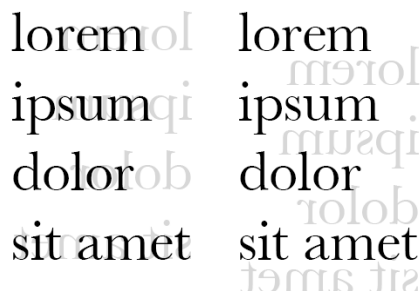
- もしも最初のルールによる結果によって、行間の距離が2ポイント以下になるならば、それを2ポイントとなるように強制する。(この量は`lineskip`の設定で調整可能)

2番目のルールは、前の行が大きなディセンダー（例えば、g、q、j、pなどの文字が含まれる）場合に、高いアセンダーを持つ（k、l、と大文字など）次の行とぶつかるのを防ぐためにあります。

これに加えて、`baselineskip`はある量の「伸張性」(‘stretch’)を持ちます。これはページ分割処理を最適化させるために役に立つ場合に、行間を広げたり、同様にパラグラフ間の距離を伸縮させたりします。

これらのルールの組み合わせにより、行はページの様々な位置で開始することになるのです。

これとは別の組版の流儀では、行は規則正しいグリッド上の決まった位置で始まることを要求します。いくらかの人々はグリッド上の組版によって生じる、ページの「カラー」(‘color’)を好みます。そしてこの手法は、しばしば非常に薄い紙に印字するときに、インクが裏から滲まないように工夫する目的として、行の位置をきっちり揃えるために用いられます。次の例を比べてみてください。左のものは行が紙の両面で同じ位置にくるようにしたもので、右はそうになっていないものです。



loremol lorem
ipsumqi ipsum
dolorob dolor
sit amet sit amet

`grid` パッケージは SILE の組版エンジンが、上記ふたつのルールを適用しないように動作を変更させます。その結果、行は常にグリッド上に整列され、パラグラフ間のスペースなどは常に、グリッド上行が規則正しく並ぶように調整されます。パッケージをロードすることで、ふたつの新たな SILE コマンドが追加されます。`\grid[spacing=<dimension>]` と `\no-grid` です。最初のもはグリッド上の組版を、それが実行された時点以降から、有効にして、次のものはそれを無効にします。

このセクションのはじめで、`\grid[spacing=15pt]` によって、15ポイントのグリッドが設定されています。グリッドが有効になったときの例文を示しましょう。

Lorem ipsum dolor sit amet, consectetur adipiscing 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.

そして以下は、`\no-grid` にした後のものです。

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.

5.10 verbatim

verbatim パッケージはプログラムのコードを引用したりするなど、フォーマットが重要な文章を表示したりする際に役に立ちます。これは SILE の設定を、左揃え、ハイフネーションなし、インデントなしの規則的なスペーシングとなるようにします。これはまた、SILE にスペースの数を勝手に減らしたりしないように伝え、また等幅フォントを使うように設定します。

この名前にも関わらず、**verbatim** は SILE が特殊文字を扱うやり方を変えません。あなたは **verbatim** 中でもバックスラッシュと波括弧をエスケープする必要があります。例えば、`\\` のように。

それでは **verbatim** 環境の例を示してみましょう。

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

verbatim 環境の中で使用するフォントを指定したければ、**verbatim:font** コマンドで再定義することができます。この文書では、

```
<define command="verbatim:font">
  <font family="Consolas" size="10pt"/>
</define>
```

です。

5.11 他のパッケージにより利用されるパッケージ

これらに加えて、おそらくあなたが直接は利用しないであろうパッケージもあります。それらのパッケージは、他のパッケージやクラスにより基本的な機能を提供するという目的のために存在します。例えば **book** クラスは、他の補助的なパッケージからの機能で構成されています。

5.11.1 footnotes

例えば、**book** クラスでは、`\footnote` コマンドで脚注を加えることができることをみました。このコマンドは実際には、**footnotes** パッケージにより提供されています。**book** クラスはこのパッケージをロードし、どこに脚注を置くべきか伝え、そして **footnotes** パッケージは脚注を整形し表示させます。これは他にも、以下に述べる数々のパッケージを利用しながら行われます。

5.11.2 counters

SILE の様々な部分、例えば `footnotes` パッケージや章・節用のコマンド、はカウンタを利用します。現在の脚注番号、章番号、などに使うためです。 `counters` パッケージは、カウンタを設定し、カウンタを増加させ、表示させたりすることに使えます。これは以下のようなコマンドを提供します。

- `\set-counter[id=<counter-name>,value=<value>]` — `<counter-name>` という名前のカウンタを与えられた値で設定します。
- `\increment-counter[id=<counter-name>]` — `\set-counter` と同様に、しかし `value` パラメータがなければ、カウンタを 1 だけ増加させます。
- `\show-counter[id=<counter-name>]` — これはカウンタの値を、宣言された表示形式で整形し、表示します。

カウンタパッケージのすべてのコマンドはオプションとして、`display=<display-type>` パラメータをとり、これはカウンタの表示形式を設定します。

可能な表示形式は、デフォルトで `arabic`、アルファベットのカウンタとして `alpha`、小文字のローマ数字 `roman`、そして大文字のローマ数字 `Roman` です。

例えば、次のような SILE コードは、

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

以下のようになります。

```
2
iii
```

5.11.3 pdf

`pdf` パッケージは（基本的な）PDF リンクや目次の機能を実現します。これは 3 つのコマンドを提供します。 `\pdf:destination`、`\pdf:link`、そして `\pdf:bookmark` です。

コマンド `\pdf:destination` はリンク先（ターゲット）を生成します。これはパラメータとして `name` をとり、リンク先を特定するための一意的な名前を指定します。文書中のある場所にリンクを張るには、`\pdf:link[dest=name]{内容}` を使います。

もし、`pdf` パッケージが `tableofcontents` パッケージのあとにロードされたら（例えば、`book` クラスを用いた文書の中）、PDF 文書はしおり（アウトライン）付のものとなります。

5.11.4 frametricks

最初の章で述べたように、SILE はページのどの部分にテキストを置くか指定するために、フレームを使います。**frametricks** パッケージはパッケージの著者に、フレームを扱うための数々のコマンドを提供します。

とにかく有用なのは**showframe** です。これは出力エンジンに、フレームを線で囲いラベルを付けるように指示します。これはオプションでパラメータ `id=<frame id>` をとります。もしこのオプションが与えられなければ、現在のフレームが使用されます。もし ID が **all** であれば、現在のクラスの中で宣言されたすべてのフレームが表示されます。

コマンド `\breakframevertical` は現在のフレームを、与えられた地点で上下2分割にします。現在のフレームが ID **main** を持つとしましょう。フレームが分割されると、**main** は上のフレーム（コマンド挿入以前）となり、下のフレーム（コマンド挿入以降）は **main_** という ID となります。このパラグラフの先頭では、`\breakframevertical` コマンドを実行しています。そしてここで、コマンド `\showframe` を実行してみましょう。見てわかる通り、現在のフレームは **content_** で、ちょうどこのパラグラフの開始位置で始まります。

同様に、`\breakframehorizontal` コマンドは、フレームを左右に分割します。このコマンドは必須でない引数として、`offset=<dimension>` をとり、これはどの位置でフレームを分割するかを指定します。もしこれが与えられなければ、フレームはその行の現在地で分割されます。

コマンド `\shiftframeedge` は、現在のフレームを左右に再配置します。これは **left=** と（または）**right=** パラメータをとり、パラメータの値は正か負の長さです。このコマンドはフレームの先頭で使用されなければなりません。それはこのコマンドが組版エンジンを再初期化するからです。

それらをすべて組み合わせた `\float` コマンドは、現在のフレームを分割し、フロートオブジェクトを保持する小さなフレーム（この文章のはじめのドロップキャップのように）を生成し、文章を周囲のフレームに流し込み、文章がフロートオブジェクトの脇を過ぎ去れば、フレームを元のように戻します。`\float` コマンドは必須でないふたつのパラメータを取ります。`bottomboundary=<dimension>` と `rightboundary=<dimension>` で、フレームの周りに余白を付与します。このパラグラフのはじめでは、コマンド `\float[bottomboundary=5pt]{\font[size=50pt]{こ}}` が実行されています。

最後に、本文とは分離された、サイドバーのようなフレームを定義する方法を示しましょう。後の章でどのようにしてそれが実現されるか見るでしょう。⁴ **frametricks** は `\typeset-into` コマンドを提供します。これはテキストを指定されたフレームに描画します。

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

5.11.5 insertions

footnotes パッケージは、補助的なもの（すなわち脚注の内容）を内容として受け取り、フレームを適当なサイズに縮小して脚注フレームに挿入します。このような作業は **insertions** パッケージの力で実現されています。これはユーザの目に見える SILE コマンドを提供しませんが、他のパッケージに Lua 機能を
4. 訳注：原文は 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?

提供します。TeXnician たちは、これが SILE のコアではなく、外部のアドオンパッケージとして実装されていることに興味を持つかもしれません。

5.11.6 twoside

Chapter 4 で述べた **book** クラスは左右の対となるページマスタ⁵を設定します。**twoside** パッケージはヘッダやその他の位置の入れ替えなどを行います。これは一般ユーザからは利用されません。

5.11.7 masters

ページマスタ設定機能もそれ自身、アドオンパッケージのひとつです。それはクラス中で、フレームのセットを定義し、一時的あるいは恒久的にそれらの間を切り替えることを可能にします。これはコマンド `\define-master-template` (これは Chapter 8 で見る `\pagetemplate` をまねたもの)、`\switch-master`、そして `\switch-master-one-page` を定義します。このパッケージについて、より詳しくは、`tests/masters.sil` を参照してください。

訳注：原文（よく意味が分かりません）

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

5.11.8 infonode

このパッケージはクラスを設計する人にのみ有用です。

文書进行处理するとき、SILE はまずパラグラフを行に分割します。それから行をページ内に配置し、最終的にページを出力します。すなわち、SILE がパラグラフ中のテキスト进行处理している間では、そのテキストがどのページに現れるかは明らかではないのです。これはインデックスや目次などを生成する際に困難をもたらします。なぜならそれらは、ある特定のテキストなどの要素が現れるページ番号を必要とするからです。

この問題に対処するため、**infonode** は、あなたが入力テキスト・ストリーム中に情報ノードを挿入することを可能にします。ページが出力されるとき、これらのノードはあるリストに集約され、クラスの出カルーチンは、どのノードが特定のページに出現するか決定するために、このリストを調べることができます。**infonode** は `\info` コマンドを提供します。これはテキスト・ストリームに情報ノードを挿入するためのもので、ふたつのパラメータを取ります。**category=** と **value=** です。カテゴリは似たような種類のノードをグループ化するのに使用されます。

例として、聖書を考えましょう。どの範囲の節がページに含まれるかヘッダに出力たいとします。このとき、新たな節を開始するコマンドで、情報ノードを節への参照とともに挿入します。

5. 訳注：適切な訳がわかりません。

```
SILE.Commands["info"] ( category = "references", value = ref , )
```

各ページの最後に呼ばれる `endPage` メソッドで、“references” 情報ノードのリストを調べます。

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

5.11.9 inputfilter

`inputfilter` パッケージは、クラスの製作者に、SILE によって処理される前の入力データに、フィルタをかける方法を提供します。これは文書を表す抽象構文木 (abstract syntax tree) を書き換えることで可能となります。

`inputfilter` を、あなたのクラスで `class:loadPackage("inputfilter")` によりロードすることで、ふたつの新たな Lua 関数が利用可能となります。`transformContent` と `createCommand` です。`transformContent` は文書の内容を構成する木構造 (content tree) に対し、その内容となるテキストに変換関数を適用します。簡単な例として、`examples/inputfilter.sil` を、より完全なものとして、`packages/chordmode.sil` を見てください。

Chapter 6

SILE マクロとコマンド

我々がコンピュータを利用する理由のひとつは、それが反復作業を得意とすることでしょう。おそらくコンピュータを利用するなかで最も重要な技術は、特にプログラミングにおいて、繰り返しなされる部分を見極め、人間の代わりにコンピュータにそれをやらせることです。すなわち、Don't Repeat Yourself。

SILE においてもこれは同じことです。しばらく SILE を使っていれば、入力にはパターンがあり、何度も繰り返しなされることがあるということに気づくでしょう。

6.1 単純なマクロ

仮に SILE のちょっとした「でこぼこ道」ロゴをデザインしたいとしましょう。（ \TeX の熱烈なファンにはおなじみの）我々のロゴは「 \SILE 」です。あまり素晴らしいものとは言えませんが、この節ではこのロゴを例題として用いましょう。

このロゴを出力するには、我々は \SILE に、「S」を、少し下げて（ちょうど ex の半分）「I」を、続けて「L」を、ちょっと戻って少し小さな「E」をやや持ちあげて、出力するように指示します。

\SILE コードではこれは、

```
S\lower[height=0.5ex]{I}L\glue[width=-.2em]\raise[height=0.6ex]{\font[size=0.8em]{E}}
```

（ \glue コマンドについては気にしないでください。後で説明します）

もう既に 4 回も、この章ではこのロゴを入力しています。何度もこのコードを繰り返し入力したくはないですね。我々がすべきことは、コンピュータに、「これは \SILE のロゴ。 \SILE を入力したら、 $\text{S\lower[height=0.5ex]{I}L\glue[width=-.2em]\raise[height=0.6ex]{\font[size=0.8em]{E}}$ と解釈せよ」と指示することです。

つまり、コマンドを定義するのです。

SILE¹ では、コマンドを定義するふたつの方法があります。最も単純なコマンドは、例として挙げたような \SILE コマンドで、「 \x と書いたら、かわりに x \y z で置き換える」というものです。これらはマクロと呼ばれ、置き換えの作業をマクロ展開と呼びます。

あなたはこの種のマクロを、SILE ファイルの中で定義することができます。ちょうどここでは、

```
\define[command=SILE]{  
S\lower[height=0.5ex]{I}L\glue[width=-.2em]  
\raise[height=0.6ex]{\font[size=0.8em]{E}}  
}
```

1. ロゴを使うのはもう辞めましょう。

としました。

ここでは、ビルトインの SILE コマンド `\define` を使用しています。`\define` はオプションとして `command` を取り、その値は定義するコマンドの名前です。`\define` コマンドの内容は、コマンドが使用される時に実行されるべき SILE インストラクションです。

ここで、SILE コマンドで利用可能な名前について知っておくべきでしょう。

XML 風書式の入力ファイルでは、コマンド名は XML タグ名として許されるものでなければなりません。TeX 風書式ではアルファベットとアラビア数字、ハイフンとコロンが使用可能です。これに加えて、任意の 1 文字コマンド名が利用可能です。(つまり、バックスラッシュを書くのに `\\` が使えます)

コマンドを定義する際に、XML 風書式でそれを行うと、どんなコマンド名でも可能です — 例えそれが XML 風書式の入力ファイルで利用不可であっても！(XML 風書式の SILE ファイルで奇妙な名前のコマンドを定義し、TeX 風書式でそれをつかうことはできます) TeX 風書式では、それはパラメータ値として有効なものでなければなりません。パラメータ値はコンマ、セミコロン、あるいは閉じ括弧まで続いてしまうかもしれません。

6.2 内容を伴うマクロ

それでは次の段階に進みましょう。コマンドを定義していく中で、時々 単純な置き換え以上の、引数を伴うものが必要となることがあります。例として、`color` パッケージを考えましょう。我々はテキストを **このように** 赤色で描画したいとします。通常のやり方では、これは、

```
\color{color=red}{このように}
```

のようになります。

しかしながら、我々は常に「このように」という語をハイライトしたいわけではありません。別のテキストをハイライトしたい場合もあります。そこで我々は、ある与えられたものを、コマンド `\color{color=red}{ ... }` で囲む必要が出てきます。つまり、引数を取るコマンドを定義したいのです。

これを SILEで行うには、`\process` コマンドを使います。`\process` は `\define` 中でのみ有効です。(その他の場所で使おうとするとぐちゃぐちゃな結果となるでしょう) これは、「このコマンドの引数として与えられたものを何でも実行せよ」という意味です。すなわち、赤でなにか描画するコマンドを定義するには、

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

`\process` コマンドはひとつのマクロ中で何度も呼ぶことはできません。

このため、`\chapter` コマンドは単純なマクロとしては実装できません。コマンドを定義する別の方法は、Lua プログラミング言語を使ってそれを書くことです。そしてそれは実際に`\chapter` コマンドの定義で行われています。後の章でどのようにそれを行うか見るでしょう。.

`\define` コマンドは、本当に単純なことをやるためのものです。

6.3 ネストしたマクロ

マクロの中でマクロを呼び出すことができます。マクロは単に現在処理中のステップにおける置き換えでしかありません。SILE がマクロコマンドを読み込むとき、SILE はあたかもあなたがマクロの定義内容を入力していたかのように振舞います。もちろん、そのようなマクロ定義は他のコマンドを含むかもしれません。

このため、ごく単純なマクロの処理で、実に様々な自動化を行うことができます。

例えば、この文書ではたくさんのノートがあります。そこではパラグラフは、イタリック体²で、左マージン付きでふたつの太い線の間に囲まれて表示されます。これは`\note` コマンドによって実現されています。このコマンドはビルトイン・コマンドではなく、この文書で読み込まれている、`documentation/macros.sil` ファイル中で定義されています。その定義は、XML 風書式で、

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

2. 訳注：和訳ではイタリックではありません。

ここで初めて登場するコマンドは`\set`です。これについてはすぐに議論します。

Chapter 7

SILEの設定

コマンドの他にも、SILEはその動作に影響を与える様々な仕組みを提供します。SILEのパラメータを操作することで、その出力結果に対し、些細なものから劇的なものまで、実に様々な変化を与えることができます。外部パッケージはそれ自身の設定を宣言することができます（SILEに同梱されているパッケージはそうではありませんが）が、ここではSILE本体に組み込みの設定のみ解説することにしましょう。

SILEの設定は名前空間を持ちます。これは、(1)その設定がシステムのどのエリアに影響するのか、その名前から明らかなようにする、(2)各パッケージが、他のパッケージやSILE内部に干渉する心配なしに設定を行えるようにする、ために必要なことです。それぞれの設定パラメータは`area.name`の形をとります—例えば`typesetter.orphanpenalty`は、タイプセッタがオーファン¹に対するペナルティを課す際の設定です。

SILE文書中から設定を変更するインターフェイスは、`\set` コマンドです。これはふたつのオプションを取ります。どの設定が変更されるのかを表すパラメータと、それがどのように変更されるのかを示す値です。例えば、

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

もし、`\set` コマンドが引数とともに与えられるならば、それは引数として与えられたブロックのみの、局所的な変更であるとみなされます。言うならば、

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

はオーファン・ペナルティを250に設定し、`\lorem` コマンドで与えられる50語からなるダミーのテキストを組み、オーファン・ペナルティを元の値に戻します。

それでは各ビルトインの設定がどのようなものなのか見てみましょう。まずはより明瞭なものから始め、次第に理解が難しいものへ移ります。

7.1 スペーシング設定

`\note` 環境の`document.lskip` パラメータを例にとりましょう。これはグルーパラメータで、各行の左側に与えられるスペースです。この値を正の値に設定することで、左マージンを実効的に増加させること

1. 訳注：パラグラフの最終行が、ごく少数の語句のみから構成されてしまい、それが最終行に「取り残された」ようになること。または、パラグラフの先頭行のみが、前のページに分離して取り残されることを指す。

ができます。同様に、`document.rskip` は各行の右側にスペースを与えます。

グルー

グルー パラメータは、通常の長さとは若干異なります。グルーは基本的には「スペース」で、長さとして表されますが、ふたつの付加的な構成要素を持ちます。ストレッチとスキップ²で、`<dimension> plus <dimension> minus <dimension>` の形で指定されます。最初の長さ (dimension) は基本となる長さで、ストレッチはそれに加えることができる最大の長さ、シュリンクは逆に差し引くことができる長さです。例えば、`12pt plus 6pt minus 3pt` は理想的には 12 ポイントだが、最大で 18 ポイント、最小で 9 ポイントまで伸び縮みが許容されます。

ここで、`center` 環境がどのように実装されているか考えましょう。まずは我々は非常に柔軟に伸びるグルーを左右のマージンに追加します。

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

これは以下のような結果を出力します。

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.

訳

この文章はほぼ中央寄せとなっています。しかしながら、これには3つの問題があります。まずは、通常のパラグラフのように、最初の行にはインデントが付加されています。次に単語間のスペースが伸張可能となっているため、各行はほぼ行端揃えされたように見えてしまいます。最後に、デフォルトでは、SILE はパラグラフの終わりに非常に大きなグルーを挿入します。これはテキストが行端揃えの場合に、最後の行が行長に合わせて伸張されてしまうのを避けるためです。このため最後の行は、中央寄せとしてはやや不自然に見えてしまいます。この3つの点について、続けてその対処法について見てみましょう。

各パラグラフの最初のインデント量は、`document.parindent` で制御されます。これはグルーパラメータで、デフォルトでは 20 ポイントで、伸び縮みしません。実際には、パラグラフのはじめに挿入される空

2. 訳注：シュリンクの間違いではないかと思われます。

白の量は`current.parindent`です。各パラグラフの出力後に、`current.parindent`は、`document.parindent`の値でリセットされます。`\noindent` コマンドは`current.parindent`をゼロに設定することで実現されます。

さて、このように「ぶら下げ」インデントにするのはどうすればよいでしょう。ここでは、`document.lskip`を20ポイントに、`current.parindent`をマイナス20ポイントに設定しています。（つまり、`\set[parameter=document.lskip,value=20pt]`と`\set[parameter=current.parindent,value=-20pt]`を実行します）

パラグラフ間のスペースはグルーパラメータ`document.parskip`で設定されます。これは通常は、1ポイントのストレッチを持つ、5ポイントのグルーとして設定されます。

グリッド上の組版の節で述べたように、パラグラフ内の行間のスペースはふたつのルールにより決定されます。ここでは、それを設定パラメータを用いて再び説明してみましょう。

- SILEはふたつの連続した行の間の距離が、ちょうど`document.baselineskip`となるように試みます。
- もし最初のルールを適用した結果として、パラグラフの最後の行と次のパラグラフの最初の行の間の距離が、`document.lineskip`以下となるようであれば、それが`document.lineskip`だけ確保されるように調整します。

最後のスペーシング設定は、`document.spaceskip`です。通常、単語間のスペースは、現在のフォントの空白文字の幅より決定されますが、行端揃えを実現するために、このスペースは伸縮可能となります。具体的には、もし仮に現在のフォントでのスペース幅が、`<space>`だとすると、`document.spaceskip`のデフォルト値は、`1.2 <space> plus 0.5 <space> minus 0.333 <space>`と設定されます。³

これを陽に設定したければ、`document.spaceskip`により設定を行います。もしもデフォルト値に戻したければ（つまりフォントの空白文字の幅を計り直す）、設定をクリアする必要があります。これを行うには、`\set` コマンドを`value` パラメータなしで呼び出します。`\set[parameter=document.spaceskip]` のようにです。

7.2 タイプセット設定

SILEのスペーシングを制御する設定は、生成される文書に対して明白な影響を与えます。タイプセットにも操作するスイッチがあります。

`typesetter.widowpenalty`と`typesetter.orphanpenalty`⁴は、SILEがどの程度、ページの始まりや終わりにはぐれた行があるのを抑止するかを決定します。ウィドウは改ページの際に、1行だけ前のページの終わりに取り残されることを、オーフオンはパラグラフの最後の行だけが次のページに引き離されてしまうことを指します。デフォルトでは、これらに対し、150ポイントの`penalty`が与えられています。この値は10000までの値を取り、最大値は「ウィドウとオーファンの禁止」を意味します。

SILEは非常によく伸張するグルーを、各パラグラフの末尾に自動的に挿入します。これなくしては、行端揃えのアルゴリズムが最終行を含むパラグラフ全体に渡って適用され、パラグラフのすべての行が両端揃えとなってしまいます。（通常、我々は最後の行は左揃えにします）このグルーの量は`typesetter.parfillskip`により設定されています。デフォルト値は`0pt plus 10000pt`ですが、このパラグラフではゼロにしています。

3. この幾分適当な値の設定はバグとも言えるでしょう。

4. TeX ユーザはリネームされていることに注意

それでは、ようやく中央寄せの実装を完結させましょう。

```
\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 started of the paragraph, and we turn off the filling glue at the end of the paragraph.

訳

そしてこれがほぼ、plain クラスで定義されている **center** 環境そのものです。マージンは伸張可能ですが、スペースは伸張せず、パラグラフの最初のインデントやパラグラフの最後のグルーは無効化されています。

最後に、タイプセッタは入力におけるパラグラフ分割方法を知る必要があります。これは通常、ふたつの連続した改行文字です。しかしながら、XML 入力を扱う際には、この仮定は適切ではありません。このような場合は、**typesetter.parseppattern** を Lua パターンを使って指定することができます。これはデフォルトでは `\n\n+` です。これに加えて、入力における複数のスペースがひとつのスペースに変換されるやり方を定義するのが **shaper.spacepattern** の設定です。これは `%s+` をデフォルト値として持ちます。(任意の数のスペースがひとつのスペースとみなされる) もしあなたが、スペースの数をそのまま保ちたいならば、スペースパターンを `%s` に設定すると良いでしょう。

7.3 ラインブレーキング設定

SILE のラインブレーキング (行分割) アルゴリズムは TeX からの借り物であり、そして同等の水準のカスタマイズ性を備えます。とにかくラインブレーキング・アルゴリズムに適用可能な設定を列挙してみましょう。あなたはこれらについてよく知っているものとします。

- **linebreak.tolerance** : アルゴリズムによって分割点が却下されるまでの限界値です。(デフォルト : 500)
- **linebreak.pretolerance** : これよりも良い分割点がければ、ハイフネーションが検討されます (デフォルト : 100)

- `linebreak.adjdemerits` : ふたつの引き続く行が視覚的に不整合であった場合に、パラグラフ構築中に累積される付加的なデメリット値の量です。このケースでは、ジャスティフィケーションのためによるスペーシングの不均一のために起こります。⁵ (デフォルト : 10000)
- `linebreak.looseness` : 現在のパラグラフの行数が、通常よりどのくらい多くなるべきか設定します。(デフォルト : 0)
- `linebreak.prevGraf` : 現在のパラグラフにおいて、垂直リストに追加された行数。
- `linebreak.emergencyStretch` : パラグラフの各行に追加されるかもしれないストレッチの量。
- `linebreak.linePenalty` : 各ラインブレイクに付随したペナルティの値。(デフォルト : 10)
- `linebreak.hyphenPenalty` : ハイフネーションに伴うペナルティの値。(デフォルト : 50)
- `linebreak.doubleHyphenDemerits` : 連続した行がともにハイフンで分割される場合のペナルティ。(デフォルト : 10000)

7.4 Lua からの設定

これらの設定を SILE レイヤで操作することはまずないでしょう。複雑なレイアウトコマンドは Lua スクリプトで実装されることが想定されています。以下の SILE 関数が Lua からアクセス可能です。

- `SILE.settings.set(<parameter>, value)` : パラメータの設定を行います。

SILE レイヤでは、`\set` コマンドは文字列による表現を、可能な限り適切な Lua 型に変換してくれますが、`SILE.settings.set` はそれを行いません。Lua ではパラメータは適切な型で与えられることが想定されています。例えば、長さは `SILE.Length` オブジェクトでグルーは `Glue` でなければなりません。

- `SILE.settings.get(<parameter>)` : 現在のパラメータの設定を取得します。
- `SILE.settings.temporarily(function)` : すべての設定を一旦保存し、関数を実行した後、それを回復します。
- `SILE.settings.declare(<specification>)` : 新たな設定パラメータを宣言します。使い方については、例として `settings.lua` における基本的な設定を参照してください。クラスパッケージは名前空間を用い、`<package>.<setting>` とすべきです。

5. 訳注 : よく分かりません。原文 : 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.

Chapter 8

SILEの内部

我々の SILE に関する探索は、そろそろ最深部にたどり着きました。ここからは、SILE のすべての基礎となる、基本的構成要素についてみていきましょう。

ここでは、あなたはクラスの設計者であるとしましょう。あなたは SILE がこれから説明するコマンドや機能をどう実装しているかについての細部を、きちんと理解できるようになるものとしましょう。我々はまた、Lua レベルでこのコンポーネントにどうアクセスするのか見るでしょう。

8.1 ボックスとグルー、ペナルティ

SILE の仕事は、大雑把に言うと、小さなボックスをページに配置するだけのことです。これらのボックスのうちいくつかは、その内部に文字を持ち、そしてそれらの文字はある幅や高さを持ち、またボックスはスペースを占有するだけで空っぽであったりします。ボックスの、ひとつの水平な集まりが出来上がる（つまりラインブレイクが起こる）と、それは別のボックスの中に押し込まれ、それらが集まってできるボックスの垂直なリストは、ページを構成するよう配置されます。

概念的に、SILE はいくつかの基本的なコンポーネントを知っています。水平ボックス（文字など）、水平グルー（伸縮可能な単語間のスペース）、垂直ボックス（テキストから構成される行）、垂直グルー（行間やパラグラフ間のスペース）、そしてペナルティ（いかなるときに行やページを分割するかの情報）です。¹

とにかく使い勝手がいいのは、水平および垂直グルーです。SILE の入力ストリームに直接これらのグルーを挿入するには、`\glue` と `\skip` コマンドを使います。水平・垂直グルーはそれぞれ、`width` と `height` パラメータを取ります。このパラメータはグルー・ディメンジョン（寸法）です。例えば、`\smallskip` コマンドは、`\skip[height=3pt plus 1pt minus 1pt]` で、`\thinspace` は `\glue[width=0.16667em]` と定義されています。

同様に、`\penalty` コマンドが、ペナルティノードを挿入するのに使えます。`\break` は `\penalty[penalty=-10000]` と、`\nobreak` は `\penalty[penalty=10000]` と定義されています。

水平あるいは垂直ボックスを生成することもできます。これを行うひとつの明白な動機は、改ページや改行によって分割されて欲しくないものがあることでしょう。別の理由としては、一度ボックスとして構築されると、それがどのくらいの大きさを占めるのかが明らかになることです。`\hbox` と `\vbox` コマンドはその内容をボックスに配置し、Lua から呼び出されるときは新たなボックスを返し（return し）ます。

1. これらに加えて、あと3つのボックスの種類があります。N-ノード、アンシェイブド・ノード、そしてディスプレイショナリー（任意の）ノードです。

8.1.1 Lua インターフェイス

SILE の Lua インターフェイスには、ボックスとグルーを扱うための `nodefactory` があります。この話題に移る前に、次のことを理解しておいてください。グルー量は常に `SILE.length` オブジェクトで指定されます。これは、基本長とストレッチ、シュリンクの3要素からなるという意味での「3次元」量です。 `SILE.length` を構築するには、

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

のようにします。

水平グルーや垂直グルーを作るには、

```
local glue = SILE.nodefactory.newGlue ({ width = 1})
local vglue = SILE.nodefactory.newVglue({ height = 1})
```

SILE の組版エンジンは `SILE.typesetter` オブジェクトにより構成されます。これはふたつのキューを管理します — ノードキュー (`SILE.typesetter.state.nodes`) は、すぐに行に分割されることになる、新たな水平ボックスと水平グルーの集まりです。出力キュー (`SILE.typesetter.state.outputQueue`) はページに分割されることになる垂直要素 (行) から構成されます。ラインブレーキングやページブレーキングはタイプセッターが水平モードと垂直モードの間を遷移する際に起こります。あなたはこれを `SILE.typesetter:leaveHmode()` で強制することができます。SILE レベルでは、パラグラフの終了を強制するコマンドは `\par` です。

あなたが垂直なスペースを加えたいとしましょう。この場合、まずは `SILE.typesetter:leaveHmode()` によって、現在のパラグラフの処理対象が適切にボックス化され、出力キューに送られることを確実にします。その後あなたは、望みのグルーを出力キューに加えることができます。

ボックスとグルーをキューに送るということは、よくあるオペレーションで、そのための特別なメソッドが存在します。

```
SILE.typesetter:leaveHmode()
SILE.typesetter:pushVglue({ height = 1 })
```

ボックスをキューに追加することはやや複雑です。なぜならボックスは何らかのやり方でページに描画される必要があるからです。これを容易にするために、ボックスは通常、`value` と `outputYourself` メンバ関数を備えます。例えば、`image` パッケージはこれを非常に単純なやり方で実現します。画像の幅と高さ、画像のソース、画像を表示するための出力エンジンへの指示、を記した水平ボックスをノードキューに送ります。

```

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:moveX(this.width)
end});

```

水平・垂直ペナルティをキューに送るには、単に `SILE.typesetter:pushPenalty({penalty = x})` と `SILE.typesetter:pushVpenalty({penalty = y})` メソッドを使用します。

8.2 フレーム

既に述べたように、SILE は文章をページ上のフレームに配置します。通常これらのフレームは、文書クラスで定義されています。しかしながら、あなたはフレームをページごとに作ることができます。これを行うには、`\pagetemplate` と `\frame` コマンドを使います。このようなことを行いたい状況というのは、ごく限られているでしょう。しかしながら、これについて理解することは、文書クラスを自らの手で作成するのに役立つでしょう。

例として、二段組みレイアウトを実現したいとしましょう。SILE はフレームを宣言するのに、constraint solver² システムを利用しています。あなたがフレームが互いにどのように関係しているかを SILE に教えると、SILE はページ上にそれがどのように配置されるべきか計算して教えてくれます。

具体的な手順を見てみましょう。まずは改ページから始まります。なぜなら SILE は、ページにテキストを配置し始めた後にページレイアウトを変えることを許可しないからです。³ どのようにして新しいページを始めることができたでしょう？ `\eject`（垂直モード中の `\break` の別名）が出力キューにペナルティを加えるのみであることを思い出してください。そして、改ページは水平モードを去るときに引き起こされます。それを行うには `\par` です。つまり、`\eject\par` をした後に、`\pagetemplate` を開始できます。 `\pagetemplate` の中では、我々は SILE にどのフレームを用いるのか指示する必要があります。

```

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

```

2. 制約 (constraint) を解決する (solve)

3. もちろんあなたは `frametricks` パッケージを使ってこの制限を回避することができます — 現在のフレームを分割し、`frametricks` がこしらえた新たなフレームを操るのです。

それでは段を宣言しましょう。その前にまず段間を宣言します。なぜならそれは既知のものとして定義可能だからです。ここではそれを、ページ幅の 3% としましょう。

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

フレームのサイズは通常の<dimension>で記述されます。加えて以下の3つも可能です。

- 他のフレームのプロパティを`top()`、`bottom()`、`left()`、`right()`、`height()`、`width()` 関数で参照することができます。この関数はフレーム ID を引数にとります。SILE ではフレーム `page` が定義済みとなっており、これによりページ全体のサイズにアクセスできます。
 - 算術関数が利用可能です。plus、minus、divide、multiply、そして括弧は数式でのそれと同じ意味を持ちます。フレーム `b` がフレーム `a` の半分の高さに 5 ミリ加えたものであると宣言するには、`height=5mm + (height(b) / 2)` とします。しかしながら、後に見るように、SILE にこのような計算を任せるようにしたほうが良いでしょう。
 - 版面のデザインはしばしばページに対する割合で指定されるため、ショートカットとして、`width=5%` を `width=0.05 * width(page)` の代わりに、`height=50%` を `height=0.5 * height(page)` の代わりに使うことができます。SILE はあなたが垂直・水平どちらの割合指定をおこなうつもりなのか、判断できます。
-

次に左右の段のためのフレームを準備しましょう。`book` クラスは既にいくつかのフレームを定義しています。ひとつは`content`で、本文用にちょうど良いサイズと位置に配置されています。我々は、このフレームの境界を段の宣言に使います。左段の左マージンはこのフレームの左マージンで、右段の右マージンはこれの右マージンです。また、我々の目的には追加のパラメータが必要となります。なぜなら以下の要件、

- 段間の余白はふたつの段の間に配置される。
 - ふたつの段は同じ幅を持つ。(この幅は明らかではないが、SILE がうまくやってくれる)
 - 左段がいっぱいになったら、右段に移動して文章を続ける。
- を満たす必要があるからです。

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

そして、ようやく `pagetemplate` を完了させます。

```
\end{pagetemplate}
```

実際に試してみましょう。

LIIJÔĂÑĚ

このページは2段組みになっています。

次の章では、フレームの宣言に関する知識を、独自の文書クラスを作成するために役立ててみましょう。とりあえず、段組みがきちんとできていることを示すために、ダミーの文章を流しておきましょう。

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 consectetur adipiscing elit sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat

ut wisi enim ad minim veniam quis nostrud exercitation 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 eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi

ÔĹĴKÔĂÑĚ

nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum lorem ipsum dolor sit amet consectetur 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 9

文書クラスの設計

The material in this section has changed significantly since the previous release of SILE.

ひとつのページ上でフレームレイアウトを定義する方法を学んだところで、文書全体でそれを定義する方法へと進みましょう。

文書クラスは Lua ファイルで、それはカレントディレクトリ、あるいは SILE 検索パス（典型的には `/usr/local/share/sile`）の `classes/` サブディレクトリにあります。 それでは、マージンと本文領域¹のサイズを変えるだけの、単純なクラスを作成してみましょう。我々はこれを `bringhurst.lua` と名付けます。なぜならばこれは、Hartley & Marks 版、Robert Bringhurst による『The Elements of Typographical Style』のレイアウトを模倣するからです。

我々が設計するのは、本用のクラスです。このため SILE の `book` クラス、`classes/book.lua`、を継承すると良いでしょう。

ざっと `book.lua` の内容を覗いてみましょう。最初のクラス定義の後、`masters` パッケージを読み込んでいます。そしてマスタを以下のフレームとして定義します。（可読性のために適宜、改行を入れています）

```
book:defineMaster({
  id = "right",
  firstContentFrame = "content",
  frames = {
    content = {
      left = "8.3%", right = "86"      top = "11.6%", bottom = "top(footnotes)"
    },
    folio = {
      left = "left(content)", right = "right(content)",
      top = "bottom(footnotes)+3%", bottom = "bottom(footnotes)+5%"
    },
    runningHead = {
      left = "left(content)", right = "right(content)",
      top = "top(content) - 8%", bottom = "top(content)-3%"
    },
    footnotes = {
      left = "left(content)", right = "right(content)",
      height = "0", bottom="83.3%"
    }
  }
})
```

1. 訳注：版面。原文は typeblock です。

```
    }  
  }  
})
```

4つのフレームが宣言されていることが分かります。最初のは本文用のフレームで、SILE では約束事として **content** と呼びます。**content** フレームの下部に隣接するのは、**footnotes** フレームです。本文領域の上辺と注釈用フレームの下辺は固定された位置に来ます。しかしながら、本文領域と注釈の間の境界は可変です。初期値として、注釈用フレームの高さはゼロです。（そして本文領域がページの全領域を占めます）注釈が挿入されると、それらの高さは調整されます。

フォリオフレーム（ページ番号の部分）² は注釈の下に来ます。そしてヘッダは **content** フレームの上に来ます。

次に我々は、**twoside** パッケージで右ページと左ページが対になるようにします。

```
book:loadPackage("twoside", oddPageMaster = "right", evenPageMaster = "left" );  
book:mirrorMaster("left", "right")
```

book クラスはまた、セクショニングのコマンドを提供し、ページの始めと終わりになされるべき様々なことを宣言します。我々は book クラスから継承しているので、これらはすでに利用可能です。すべきことは我々の新たなクラスをセットアップし、異なる部分を定義するのみです。book クラスを継承するやり方は、

```
local book = SILE.require("classes/book")  
local bringhurst = book id = "bringhurst"  
...  
return bringhurst
```

それではフレームマスタの定義に移りましょう。

LaTeX の memoir クラスにおける『A Few Notes On Book Design』によると、Bringhurst の本では、ノドアキ³ はページ幅の 13 分の 1 で、天、地、小口部分ではそれぞれ、ノドアキの 8 分の 5、16 分の 5、16 分の 5 です。SILE ではこれを次のように設定します。

```
bringhurst:defineMaster({  
  id = "right", firstContentFrame = "content",  
  frames = {  
    content = {
```

2. 訳注：ノンブル

3. 原文は spine margin で、本を見開きにした時の綴じ部付近のマージン。


```

    left = "width(page)/13",
    top = "left(content) * 8 / 5",
    right = "100"        bottom = "top(footnotes)"
  },
  folio = { ... as before ... },
  footnotes = { ... as before ... },
  runningHead = { ... as before ... },
},
})
bringhurst:mirrorMaster("right", "left")

```

これで完成です！

実際にこれを使ってみると、ヘッダの位置が高過ぎることに気付くでしょう。なぜなら、本文領域が通常のクラスよりも高い位置に配置してあるにも関わらず、ヘッダの位置がそこから相対的に決定されているからです。

それではヘッダの位置を調整し、少し下げてみましょう。

```

runningHead = {
  left = "left(content)", right = "right(content)",
  top = "top(content) - 4%", bottom = "top(content)-2%"
},

```

ともかく、ページの全体的なレイアウトを変えてみたいだけであれば、これで全てです。

9.1 コマンドを定義する

文書クラスを作成するときは、通常、ページの見た目を変える以上のことを行いたくなるでしょう。新たなコマンドを定義したり、出力ルーチンの動作を変更したり、なんらかの状態に関する情報を保存したり調べたり、などです。次の章では実例を挙げて見ていきます。

Lua レベルであなた自身のコマンドを定義するには、**SILE.registerCommand** 関数を使います。これは3つのパラメータを取ります。コマンド名、コマンドの実装内容である関数、そしてコマンドの説明文です。関数のシグネチャ（呼出し情報）は固定されています。それはふたつのパラメータをとり、**options** と **content**（もちろんパラメータ名は自由に選ぶことができます）です。これらのパラメータはどちらも Lua テーブルです。**options** パラメータはコマンドのパラメータか、キーと値のテーブルとして与えられる XML 属性値です。**content** は現在処理中の入力からなる、抽象構文木です。

つまり、`\mycommand[size=12pt]{Hello \break world}` の場合、最初のパラメータはテーブル `{size = "12pt"}` で、2 番目のパラメータは以下のようなテーブルとなります。

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

ほとんどのコマンドは結局、なんらかの形で`options`を処理し、あるいは場合によっては、`SILE.process(content)`により再帰的に、与えられたコマンド引数を処理することになります。ごく簡単な例を挙げてみましょう。XML `<link>` タグはXLink属性 `x1:href`⁴を取ります。我々は、`<link x1:href="http://...">Hello</link>`を、Hello (<http://...>)と表示したいとしましょう。つまり、まず内容となるテキストを描画し、属性値に対して何らかの処理を行います。

```
SILE.registerCommand("link", function(options, content)
  SILE.process(content)
  if (options["x1:href"]) then
    SILE.typesetter:typeset(" (")
    SILE.call("code", , options["x1:href"])
    SILE.typesetter:typeset(")")
  end
end)
```

ここでは、`SILE.typesetter:typeset`と`SILE.call`関数を使って、テキストの出力と他のコマンドの呼出しを行っています。

ディメンジョン（寸法）を扱う必要がある場合には、`SILE.toPoints`で長さを表す量を、`SILE.parseComplexFrameDimension`でフレーム寸法をパースし、値をポイントに変換します。

9.2 出力ルーチン

フレームやパッケージを定義すると共に、クラスではSILEの出力を行うやり方を変更することができ、ページを始めや終わりで何かを行ったり。例えば、マスタフレームを入れ替えたり、ページ番号を表示したり。

出力ルーチンを定義する基本的なメソッドは、

4. もちろん、文書の著者は別のXML名前空間を選ぶ可能性があります。ここでは物事を単純化しましょう。

- `newPar` と `endPar` はパラグラフの始めと終わりで呼び出される。
- `newPage` と `endPage` はページの始めと終わりで呼び出される。
- `init` と `finish` は文書の始めと終わりで呼び出される。
これはオブジェクト指向的なやり方で行われます。派生クラスはそのスーパークラスのメソッドを、必要であれば書き換えます。

出力ルーチンに影響するパッケージをロードするとき、それらパッケージからクラスが構成されるやり方は、完全には自動化されません。⁵ 言い換えると、パッケージのロードそれ自体は、自動的に出力ルーチンを置き換えるわけではありません。あなたは明示的に、これらのパッケージにより提供される機能を、出力ルーチンに組み込んでいく必要があります。

例えば、`footnote` や `insertions` パッケージは、`outputInsertions` メソッドを提供します。これは各ページの末尾に呼び出される必要があります。もしも `plain` クラスから継承を行い、かつ脚注機能を使いたい場合、`endPage` メソッドを以下のように定義する必要があるでしょう。

```
myClass.endPage = function(self)
  myClass:outputInsertions()
  plain.endPage(self)
end
```

それでは、`tableofcontents` パッケージがどのように機能するか、ざっと見てみましょう。我々は、`infonodes` パッケージを使って、どのページが目次に載るべきアイテムを持つか調べます。

まずは、セクショニング・コマンドから呼び出されるコマンドを作り、情報ノードを設定します。具体的には、`\chapter` や `\section` などのコマンドは、その章・節の開始位置を含むページへの参照を記録するため、`\tocentry` を呼び出すことにします。

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

情報ノードはページ単位⁶で機能するため、文書全体に対して有効な情報を保持するためには、それらを各ページの終わりで、何らかの他のテーブルへ移してあげなければなりません。ここでページ番号を記録するのを忘れないようにする必要があります。

5. イベントの順序が重要だからです。

6. 訳注：per-page basis

SILE は `SILE.scratch` 変数を、グローバルな情報を保持する目的のために提供します。あなたのクラスやパッケージのために、名前空間によって区分化して、このテーブルの一部を使います。

目次ノードを移設するために、ページ末尾で呼び出すルーチンは以下のようなものです。

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

これらのアイテムを保持するために、LaTeXのように外部ファイルを使いましょう。目次を出力する際には、このファイルを読み込みます。このためには、文書の終わりで `SILE.scratch.tableofcontents` をファイルに書き出します。以下のような `finish` 出力ルーチンを使いましょう。

```
local writeToc = function (self)
  local t = std.string.pickle(SILE.scratch.tableofcontents)
  saveFile(t, SILE.masterFileName .. '.toc')
end
```

そして、`\tableofcontents` コマンドは、そのファイルが存在すればそれを読み込み、目次ノードを適切な形で整形し、出力します。

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

```
end)
```

これでほぼ完了です。`tableofcontents` パッケージはふたつの関数—`moveNodes` と `writeToc`—を持ち、これらは、このパッケージを使用するクラスの出カルーチンの様々な場所で呼び出される必要があります。それではどのようにそれを行うのでしょうか？

9.3 エクスポート

他のクラスやパッケージに機能を提供するために存在するパッケージは、その機能を、それを必要とするクラスやパッケージにそれを供給する方法を必要とします。これはエクスポート機構と呼ばれます。

コマンドを定義するのと同様に、それぞれのパッケージはふたつのエントリ、`init` と `exports`、からなる Lua テーブルを返すことができます。

`init` は初期化の動作を定義し、それはクラスのロード時に与えられるかもしれないオプションを取ることができます。パッケージが `class:loadPackage(package, args)` でロードされると、イニシャライザはふたつの引数と共に呼び出されます。`class` と `args` です。例えば、`twoside` パッケージは、左右のマスタフレームの ID に関する情報を受け取ります。これはページが変わる際に、マスタを切り替えるコードを実装するのに必要です。今の我々の状況では、呼び出し側で `infonode` パッケージがロードされるのを確実にする必要があります。

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

パッケージが返すもうひとつのエントリは `exports` で、呼び出し側の名前空間に混ぜ込まれる関数とその名前からなります。つまり、

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

となっている場合には、`tableofcontents` パッケージをロードするクラスは `self:writeToc()` と `self:moveTocNodes()` を呼び出すことができます。(エクスポートしたときにリネームされていることに注意) これらのメソッドを出カルーチンの適切な場面で呼び出すのは、そのクラスの責任です。

Chapter 10

高度なクラスファイル1: XMLプロセッサとしてのSILE

今や我々は、任意のXML形式のファイルを、PDFへと変換する実例を解説する段階に来ました。SILEに同梱されているDocBookプロセッサを例にとって見ていきましょう。DocBookは技術文書を作成するためのXML文書形式です。DocBook自身は、それが用いるタグがどのように表示されるべきかについては定めていません。よって我々は、自身で文書の体裁を定めなければなりません。

まずあなたに理解しておいて欲しいことは、2種類のファイルを扱うことが、物事をずっと容易にするということです。ひとつはTeX風書式のSILEファイルで、もうひとつはLuaコードです。これにより、簡単な仕事はそれに都合の良いフォーマットで処理し、困難な仕事はLuaで扱うことを可能とします。SILEのコマンドライン・オプションで`-I classname`を使うと、SILEはまず`classname.sil`というファイルを探し、そしてそれを処理すべきファイルへのラッパーとして用います。そうして、

```
\begin[papersize=a4,class=classname]{document}
```

で文書を始めれば、SILEはまた、いつものように、`classes/classname.lua`をロードします。

それでは、XML要素を描画するSILEコマンドの定義を始めましょう。この作業の大部分は、単純明快なもので、あまりくどくど述べるのはやめましょう。例えば、DocBookは`<code>`、`<filename>`、`<guimenu>`などのタグを定義し、これらは等幅フォントで描画されるべきものです。クラスのカスタマイズを容易にするために、まずはフォントの切り替えをひとつのコマンドで置き換えましょう。

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

そして、`<code>`などのタグを定義します。

```
\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}}
```

もしもユーザが異なるフォントでの表示を望むなら、`docbook-ttfont`を再定義するだけで済みます。

10.1 表題を扱う

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

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

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

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

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

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

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

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

1. Patches, as they say, are welcome.

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 all the content of a Lua table:

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

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

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

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

10.2 Sectioning

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

```
<section><title>A</title> : 1. A
  <section><title>B</title>: 1.1 B
  </section>
  <section><title>C</title>: 1.2 C
    <section><title>D</title>: 1.2.1 D
    </section>
  </section>
  <section><title>E</title>: 1.3 E
</section>
<section><title>F</title>: 2. F
```

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

```
SILE.registerCommand("section", function (options, content)
  SILE.scratch.docbook.secllevel = SILE.scratch.docbook.secllevel + 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.secllevel] =
  (SILE.scratch.docbook.seccount[SILE.scratch.docbook.secllevel] or 0) + 1
while #(SILE.scratch.docbook.seccount) > SILE.scratch.docbook.secllevel do
  SILE.scratch.docbook.seccount[ #(SILE.scratch.docbook.seccount) ] = nil
end
```

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

```
local title = SILE.findInTree(content, "title")
local number = table.concat(SILE.scratch.docbook.seccount, '.')
if title then
  SILE.call("docbook-section-"..SILE.scratch.docbook.secllevel.."-"
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.secllevel = SILE.scratch.docbook.secllevel - 1
end)
```

10.3 Other Features

SILE's DocBook implementation is a work in progress, and there is more that can be done. For instance, there is a basic implementation of lists, which equally need to be able to handle nesting; we implement another stack to take care of the type of list and list counter at each level of nesting.

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

Chapter 11

Further Tricks

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

11.1 Parallel Text

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

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

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

```
local plain = SILE.require("classes/plain");
local diglot = std.tree.clone(plain);
SILE.require("packages/counters");
SILE.scratch.counters.folio = { value = 1, display = "arabic" };
SILE.scratch.diglot = {}
diglot:declareFrame("a",    {left = "8.3%", right = "48%",
                           top = "11.6%", bottom = "80%" });
diglot:declareFrame("b",    {left = "52%", right = "100% - left(a)",
                           top = "top(a)", bottom = "bottom(a)" });
diglot:declareFrame("folio",{left = "left(a)", right = "right(b)",
                           top = "bottom(a)+3%",bottom = "bottom(a)+8%" });
```

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

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

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

```

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

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

```

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

```

SILE.registerCommand("left", function(options, content)
  SILE.settings.set("typesetter.parseppattern", -1)
  SILE.typesetter = diglot.leftTypesetter;
  SILE.Commands["font"](SILE.scratch.diglot.leftfont, )
end, "Begin entering text on the left side")

SILE.registerCommand("right", function(options, content)
  SILE.settings.set("typesetter.parseppattern", -1)
  SILE.typesetter = diglot.rightTypesetter;
  SILE.Commands["font"](SILE.scratch.diglot.rightfont, )
end, "Begin entering text on the right side")

```

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

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

```

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

```

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

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

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

```
diglot.init = function(self)
  diglot.leftTypesetter:init(SILE.getFrame("a"))
  diglot.rightTypesetter:init(SILE.getFrame("b"))
  return SILE.baseClass.init(self)
end
```

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

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

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

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

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

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

```
diglot.finish = function(self)
```

```

table.insert(SILE.typesetter.other.state.outputQueue, SILE.nodefactory.vfillGlue)
SILE.typesetter.other:chuck()
plain.finish(self)
end

```

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

11.2 Sidenotes

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

Chapter 9

The Transfiguration

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

^{xxx}And Jesus was saying to them, "Truly I say to you, there are some of those who are standing here who will **not** taste death until they see the kingdom of God after it has come with power⁺."



Sidenotes can be seen as a simplified form of parallel text. With a true parallel, neither the left or the right typesetter is “in charge”—either can fill up the page and then inform the other typesetter that they need to catch up. In the case of sidenotes, there’s a well-defined main flow of text, with annotations having to work around the pagination of the typeblock.

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

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

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

```

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

```

Next, we call another command to produce the symbol itself; this allows the book designer to change the symbols at the SILE level rather than having to mess about with the Lua file. We use the `\hbox` command

to wrap the output of the command into a hbox. `\hbox` returns its output, but also puts the box into the typesetter's output node queue; we don't want it to appear in the main typeblock, so we remove the node again, leaving our private copy in the `hbox` variable.

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

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

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

Finally we need to write the routine which outputs this hbox. Box output routines receive three parameters: the box itself, the current typesetter (which knows the frame it is typesetting into, and the frame knows whereabouts it has got to), and a variable representing the stretchability or shrinkability of the line. (We don't need that for this example.)

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

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

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

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

```

discovery.innerTypesetter = SILE.defaultTypesetter {}
discovery.init = function()
  local gutter = discovery:oddPage() and
    SILE.getFrame("rGutter") or SILE.getFrame("lGutter")
  discovery.innerTypesetter:init(gutter)
  ...
  return SILE.baseClass:init()
end
discovery.newPage = function ()
  ...
  discovery.innerTypesetter:leaveHmode(1)
  local gutter = discovery:oddPage() and
    SILE.getFrame("rGutter") or SILE.getFrame("lGutter")
  discovery.innerTypesetter:init(gutter)
  ...
  return SILE.baseClass.newPage(discovery);
end

```

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

```

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

```

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

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

```

  local unprocessedNodes = std.tree.clone(SILE.typesetter.state.nodes)

```



```

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

```

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

In most cases, the cross-reference typesetter hasn't got as far down the page as the body text typesetter, so we tell the cross-reference typesetter to shift itself down the page by the difference. Unlike the parallel example, where either typesetter could tell the other to open up additional vertical space, in this case it's OK if the cross-reference appears a bit lower than the verse it refers to.

```

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

```

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

```

SILE.settings.temporarily(function()
  SILE.settings.set("document.baselineskip", SILE.nodefactory.newVglue("7pt"))
  SILE.Commands["font"](size = "6pt", family="Helvetica", weight="800", )
  discovery.innerTypesetter:typeset(SILE.scratch.chapter.."":"..SILE.scratch.verse.."
")
  SILE.Commands["font"](size = "6pt", family="Helvetica", weight="200", )
  discovery.innerTypesetter:typeset(xref)
  discovery.innerTypesetter:leaveHmode()
  discovery.innerTypesetter:pushVglue( height = SILE.length.new(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!

11.3 SILE As A Library

So far we’ve been assuming that you would want to run SILE as a processor for an existing document. But what if you have a program which produces or munges data, and you would like to produce PDFs from within your application? In that case, it may be easier and provide more flexibility to use SILE as a library.

In the **examples/** directory of the SILE distribution, 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("classes/plain")
plain.options.papersize("a4")
SILE.documentState.documentClass = plain;
local ff = plain:init()
SILE.typesetter:init(ff)
```

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. As with an ordinary SILE document, the **papersize** option is mandatory, so we call the **options.papersize** method of our document class to set the paper size. We then need to tell SILE what class we are actually using, call **init** on the class to get the first frame for typesetting, and then initialize the typesetter with this frame. This is all that SILE does to get itself ready to typeset.

After this, all the usual API calls will work: **SILE.call**, **SILE.typesetter:typeset** and so on.

SILE.typesetter:typeset(data)

The only thing to be careful is the need to call the **finish** method on your document class at the end of processing to finish off the final page:

plain:finish()

11.4 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 of SILE's operation:

- **typesetter** provides general debugging for the typesetter: turning characters into boxes, boxes into lines, lines into paragraphs, and paragraphs into pages.
- **pagebuilder** helps to debug problems when determining page breaks.
- **break** provides (copious) information about the line breaking algorithm.
- Any package may define their own debugging facility; currently only **insertions** does this.

Multiple facilities can be turned on by separating them with commas: `--debug typesetter,break` will turn on debugging information for the typesetter and line breaker.

From Lua, you can add entries to the `SILE.debugFlags` table to turn on debugging for a given facility. This can be useful for temporarily debugging a particular operation:

```
SILE.debugFlags.typesetter = 1
SILE.typesetter:leaveHmode()
SILE.debugFlags.typesetter = nil
```

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")
```

Sometimes it's useful for you to try out Lua code within the SILE environment; SILE contains a REPL (read-evaluate-print loop) for you to enter Lua code and print the results back to you. If you call SILE with no input file name, it enters the REPL:

```
This is SILE 0.9.0
> l = SILE.Length.parse("22mm")
> l.Length
62.3622054
```

At any point during the evaluation of Lua commands, you can call `SILE.repl()` to enter the REPL and poke around; hitting Ctrl-D will end the REPL and return to processing the document.

11.5 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!