fangle

Sam Liddicott

sam@liddicott.com

August 2009

Introduction

Fangle is a tool for fangled literate programming. Newfangled is defined as *New and often needlessly novel* by TheFreeDictionary.com.

In this case, fangled means yet another not-so-new[1] method for literate programming.

Literate Programming has a long history starting with the great Donald Knuth himself, whose literate programming tools seem to make use of as many escape sequences for semantic markup as TEX (also by Donald Knuth).

Norman Ramsey wrote the Noweb set of tools (`notangle`, `noweave` and `noroots`) and helpfully reduced the amount of magic character sequences to pretty much just `<<`, `>>` and `@`, and in doing so brought the wonders of literate programming within my reach.

While using the LYX editor for LATEX editing I had various troubles with the noweb tools, some of which were my fault, some of which were noweb's fault and some of which were LYX's fault.

Noweb generally brought literate programming to the masses through removing some of the complexity of the original literate programming, but this would be of no advantage to me if the LYX / LATEX combination brought more complications in their place.

Fangle was thus born (originally called Newfangle) as an awk replacement for notangle, adding some important features, like better integration with LYX and LATEX (and later TEXMACS), multiple output format conversions, and fixing notangle bugs like indentation when using -L for line numbers.

Significantly, fangle is just one program which replaces various programs in Noweb. Noweave is done away with and implemented directly as LATEX macros, and noroots is implemented as a function of the untangler fangle.

Fangle is written in awk for portability reasons, awk being available for most platforms. A Python version[2] was considered for the benefit of LYX but a scheme version for TEXMACS will probably materialise first; as TEXMACS macro capabilities help make edit-time and format-time rendering of fangle chunks simple enough for my weak brain.

As an extension to many literate-programming styles, Fangle permits code chunks to take parameters and thus operate somewhat like C pre-processor macros, or like C++ templates. Name parameters (or even local *variables* in the callers scope) are anticipated, as parameterized chunks — useful though they are — are hard to comprehend in the literate document.

1. but improved.

2. hasn't anyone implemented awk in python yet?

License

Fangle is licensed under the GPL 3 (or later).

This doesn't mean that sources generated by fangle must be licensed under the GPL 3.

This doesn't mean that you can't use or distribute fangle with sources of an incompatible license, but it means you must make the source of fangle available too.

As fangle is currently written in awk, an interpreted language, this should not be too hard.

4a ⟨gpl3-copyright[1](), lang=text⟩ ≡

```
fangle - fully featured notangle replacement in awk

Copyright (C) 2009-2010 Sam Liddicott <sam@liddicott.com>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Table of contents

Introduction . . . 3
License . . . 4

I Using Fangle . . . 9

1 Introduction to Literate Programming . . . 11

2 Running Fangle . . . 13

2.1 Listing roots . . . 13
2.2 Extracting roots . . . 13
2.3 Formatting the document . . . 13

3 Using Fangle with LaTeX . . . 15

4 Using Fangle with LyX . . . 17

4.1 Installing the LyX module . . . 17
4.2 Obtaining a decent mono font . . . 17
4.2.1 txfonts . . . 17
4.2.2 ams pmb . . . 17
4.2.3 Luximono . . . 17
4.3 Formatting your Lyx document . . . 18
4.3.1 Customising the listing appearance . . . 18
4.3.2 Global customisations . . . 18
4.4 Configuring the build script . . . 19
4.4.1 19

5 Using Fangle with TeXmacs . . . 21

6 Fangle with Makefiles . . . 23

6.1 A word about makefiles formats . . . 23
6.2 Extracting Sources . . . 23
6.2.1 Converting from LyX to LaTeX . . . 24
6.2.2 Converting from TeXmacs . . . 24
6.3 Extracting Program Source . . . 25
6.4 Extracting Source Files . . . 25
6.5 Extracting Documentation . . . 27
6.5.1 Formatting TeX . . . 28
6.5.1.1 Running pdflatex . . . 28
6.5.2 Formatting TeXmacs . . . 28
6.5.3 Building the Documentation as a Whole . . . 28
6.6 Other helpers . . . 29
6.7 Boot-strapping the extraction . . . 29
6.8 Incorporating Makefile.inc into existing projects . . . 30
Example . . . 30

II Source Code . . . 33

7 Fangle awk source code . . . 35

7.1 AWK tricks . . . 35
7.2 Catching errors . . . 36

8 TEXMACS args . . . 37

9 LATEX and lstlistings . . . 39

9.1 Additional lstlstings parameters . . . 39
9.2 Parsing chunk arguments . . . 41
9.3 Expanding parameters in the text . . . 42

10 Language Modes & Quoting . . . 45

10.1 Modes to keep code together . . . 45
10.2 Modes affect included chunks . . . 45
10.3 Modes operation . . . 46
10.4 Quoting scenarios . . . 47
10.4.1 Direct quoting . . . 47
10.5 Language Mode Definitions . . . 47
10.5.1 Backslash . . . 48
10.5.2 Strings . . . 49
10.5.3 Parentheses, Braces and Brackets . . . 50
10.5.4 Customizing Standard Modes . . . 50
10.5.5 Comments . . . 50
10.5.6 Regex . . . 51
10.5.7 Perl . . . 52
10.5.8 sh . . . 52
10.5.9 Make . . . 52
10.6 Some tests . . . 54
10.7 A non-recursive mode tracker . . . 54
10.7.1 Constructor . . . 54
10.7.2 Management . . . 55
10.7.3 Tracker . . . 56
10.7.3.1 One happy chunk . . . 59
10.7.3.2 Tests . . . 59
10.8 Escaping and Quoting . . . 59

11 Recognizing Chunks . . . 61

11.1 Chunk start . . . 61
11.1.1 TEXMACS . . . 61
11.1.2 lstlistings . . . 62
11.2 Chunk Body . . . 63
11.2.1 TEXMACS . . . 63
11.2.2 Noweb . . . 64
11.3 Chunk end . . . 64
11.3.1 lstlistings . . . 64
11.3.2 noweb . . . 65
11.4 Chunk contents . . . 65
11.4.1 lstlistings . . . 66

12 Processing Options . . . 69

13 Generating the Output . . . 71

13.1 Assembling the Chunks . . . 72
13.1.1 Chunk Parts . . . 72

14 Storing Chunks . . . 77

15 getopt . . . 79

16 Fangle LaTeX source code . . . 83

16.1 fangle module . . . 83
16.1.1 The Chunk style . . . 83
16.1.2 The chunkref style . . . 84
16.2 Latex Macros . . . 84
16.2.1 The chunk command . . . 85
16.2.1.1 Chunk parameters . . . 86
16.2.2 The noweb styled caption . . . 86
16.2.3 The chunk counter . . . 86
16.2.4 Cross references . . . 89
16.2.5 The end . . . 90

17 Extracting fangle . . . 91

17.1 Extracting from Lyx . . . 91
17.2 Extracting documentation . . . 91
17.3 Extracting from the command line . . . 92

III Tests . . . 93

18 Tests . . . 95

19 Chunk Parameters . . . 97

19.1 LyX . . . 97
19.2 TeXmacs . . . 97

20 Compile-log-lyx . . . 99

Part I

Using Fangle

Chapter 1
Introduction to Literate Programming

Todo: Should really follow on from a part-0 explanation of what literate programming is.

Chapter 2
Running Fangle

Fangle is a replacement for NOWEB, which consists of `notangle`, `noroots` and `noweave`.

Like `notangle` and `noroots`, `fangle` can read multiple named files, or from stdin.

2.1 Listing roots

The -r option causes fangle to behave like noroots.

```
fangle -r filename.tex
```

will print out the fangle roots of a tex file.

Unlike the `noroots` command, the printed roots are not enclosed in angle brackets e.g. `<<name>>`, unless at least one of the roots is defined using the `notangle` notation `<<name>>=`.

Also, unlike noroots, it prints out all roots — not just those that are not used elsewhere. I find that a root not being used doesn't make it particularly top level — and so-called top level roots could also be included in another root as well.

My convention is that top level roots to be extracted begin with `./` and have the form of a filename.

Makefile.inc, discussed in 6, can automatically extract all such sources prefixed with `./`

2.2 Extracting roots

notangle's `-R` and `-L` options are supported.

If you are using LyX or LaTeX, the standard way to extract a file would be:

```
fangle -R./Makefile.inc fangle.tex > ./Makefile.inc
```

If you are using TeXmacs, the standard way to extract a file would similarly be:

```
fangle -R./Makefile.inc fangle.txt > ./Makefile.inc
```

TeXmacs users would obtain the text file with a *verbatim* export from TeXmacs which can be done on the command line with `texmacs -s -c fangle.tm fangle.txt -q`

Unlike the `noroots` command, the `-L` option to generate C pre-preocessor `#file` style line-number directives,does not break indenting of the generated file..

Also, thanks to mode tracking (described in 10) the `-L` option does not interrupt (and break) multi-line C macros either.

This does mean that sometimes the compiler might calculate the source line wrongly when generating error messages in such cases, but there isn't any other way around if multi-line macros include other chunks.

Future releases will include a mapping file so that line/character references from the C compiler can be converted to the correct part of the source document.

2.3 Formatting the document

The noweave replacement built into the editing and formatting environment for TeXmacs, LyX (which uses LaTeX), and even for raw LaTeX.

Use of fangle with TeXmacs, LyX and LaTeX are explained the the next few chapters.

Chapter 3
Using Fangle with LaTeX

Because the noweave replacement is impemented in LaTeX, there is no processing stage required before running the LaTeX command. Of course, LaTeX may need running two or more times, so that the code chunk references can be fully calculated.

The formatting is managed by a set of macros shown in 16, and can be included with:

```
\usepackage{fangle.sty}
```

Norman Ramsay's origial `noweb.sty` package is currently required as it is used for formatting the code chunk captions.

The `listings.sty` package is required, and is used for formatting the code chunks and syntax highlighting.

The `xargs.sty` package is also required, and makes writing LaTeX macro so much more pleasant.

To do: Add examples of use of Macros

Chapter 4
Using Fangle with LyX

LyX uses the same LaTeX macros shown in 16 as part of a LyX module file `fangle.module`, which automatically includes the macros in the document pre-amble provided that the fangle LyX module is used in the document.

4.1 Installing the LyX module

Copy `fangle.module` to your LyX layouts directory, which for unix users will be `~/.lyx/layouts`

In order to make the new literate styles availalble, you will need to reconfigure LyX by clicking Tools->Reconfigure, and then re-start LyX.

4.2 Obtaining a decent mono font

The syntax high-lighting features of LSTLISTINGS makes use of bold; however a mono-space tt font is used to typeset the listings. Obtaining a `bold tt font` can be impossibly difficult and amazingly easy. I spent many hours at it, following complicated instructions from those who had spend many hours over it, and was finally delivered the simple solution on the lyx mailing list.

4.2.1 txfonts

The simple way was to add this to my preamble:

```
\usepackage{txfonts}
\renewcommand{\ttdefault}{txtt}
```

4.2.2 ams pmb

The next simplest way was to use ams poor-mans-bold, by adding this to the pre-amble:

```
\usepackage{amsbsy}
%\renewcommand{\ttdefault}{txtt}
%somehow make \pmb be the command for bold, forgot how, sorry, above line not work
```

It works, but looks wretched on the dvi viewer.

4.2.3 Luximono

The lstlistings documention suggests using Luximono.

Luximono was installed according to the instructions in Ubuntu Forums thread 1159181[1] with tips from miknight[2] stating that `sudo updmap --enable MixedMap ul9.map` is required. It looks fine in PDF and PS view but still looks rotten in dvi view.

4.3 Formatting your Lyx document

It is not necessary to base your literate document on any of the original LyX literate classes; so select a regular class for your document type.

Add the new module *Fangle Literate Listings* and also *Logical Markup* which is very useful.

In the drop-down style listbox you should notice a new style defined, called *Chunk*.

When you wish to insert a literate chunk, you enter it's plain name in the Chunk style, instead of the old NOWEB method that uses `<<name>>=` type tags. In the line (or paragraph) following the chunk name, you insert a listing with: Insert->Program Listing.

Inside the white listing box you can type (or paste using `shift+ctrl+V`) your listing. There is no need to use `ctrl+enter` at the end of lines as with some older LyX literate techniques — just press enter as normal.

4.3.1 Customising the listing appearance

The code is formatted using the LSTLISTINGS package. The chunk style doesn't just define the chunk name, but can also define any other chunk options supported by the lstlistings package `\lstset` command. In fact, what you type in the chunk style is raw latex. If you want to set the chunk language without having to right-click the listing, just add `,lanuage=C` after the chunk name. (Currently the language will affect all subsequent listings, so you may need to specify `,language=` quite a lot).

To do: so fix the bug

Of course you can do this by editing the listings box advanced properties by right-clicking on the listings box, but that takes longer, and you can't see at-a-glance what the advanced settings are while editing the document; also advanced settings apply only to that box — the chunk settings apply through the rest of the document[3].

To do: So make sure they only apply to chunks of that name

4.3.2 Global customisations

As lstlistings is used to set the code chunks, it's `\lstset` command can be used in the pre-amble to set some document wide settings.

If your source has many words with long sequences of capital letters, then `columns=fullflexible` may be a good idea, or the capital letters will get crowded. (I think lstlistings ought to use a slightly smaller font for captial letters so that they still fit).

The font family `\ttfamily` looks more normal for code, but has no bold (an alternate typewriter font is used).

With `\ttfamily`, I must also specify `columns=fullflexible` or the wrong letter spacing is used.

In my LaTeX pre-amble I usually specialise my code format with:

1. http://ubuntuforums.org/showthread.php?t=1159181
2. http://miknight.blogspot.com/2005/11/how-to-install-luxi-mono-font-in.html
3. It ought to apply only to subsequent chunks of the same name. I'll fix that later

19a ⟨document-preamble[1](), lang=tex⟩ ≡

```
\lstset{
numbers=left, stepnumber=1, numbersep=5pt,
breaklines=false,
basicstyle=\footnotesize\ttfamily,
numberstyle=\tiny,
language=C,
columns=fullflexible,
numberfirstline=true
}
```

4.4 Configuring the build script

You can invoke code extraction and building from the LyX menu option Document->Build Program.

First, make sure you don't have a conversion defined for Lyx->Program

From the menu Tools->Preferences, add a conversion from Latex(Plain)->Program as:

```
set -x ; fangle -Rlyx-build $$i |
  env LYX_b=$$b LYX_i=$$i LYX_o=$$o LYX_p=$$p LYX_r=$$r bash
```

(But don't cut-n-paste it from this document or you may be be pasting a multi-line string which will break your lyx preferences file).

I hope that one day, LyX will set these into the environment when calling the build script.

You may also want to consider adding options to this conversion...

```
parselog=/usr/share/lyx/scripts/listerrors
```

...but if you do you will lose your stderr[4].

Now, a shell script chunk called `lyx-build` will be extracted and run whenever you choose the Document->Build Program menu item.

This document was originally managed using LyX and lyx-build script for this document is shown here for historical reference.

```
lyx -e latex fangle.lyx && \
  fangle fangle.lyx > ./autoboot
```

This looks simple enough, but as mentioned, fangle has to be had from somewhere before it can be extracted.

4.4.1 ...

When the lyx-build chunk is executed, the current directory will be a temporary directory, and `LYX_SOURCE` will refer to the tex file in this temporary directory. This is unfortunate as our makefile wants to run from the project directory where the Lyx file is kept.

We can extract the project directory from `$$r`, and derive the probable Lyx filename from the noweb file that Lyx generated.

19b ⟨lyx-build-helper[1](), lang=sh⟩ ≡ 91b▷

```
PROJECT_DIR="$LYX_r"
LYX_SRC="$PROJECT_DIR/${LYX_i%.tex}.lyx"
```

4. There is some bash plumbing to get a copy of stderr but this footnote is too small

19b ⟨lyx-build-helper[1](), lang=sh⟩ ≡ 91b▷

```
TEX_DIR="$LYX_p"
TEX_SRC="$TEX_DIR/$LYX_i"
```

And then we can define a lyx-build fragment similar to the autoboot fragment

20a ⟨lyx-build[1](), lang=sh⟩ ≡ 91a▷

```
#! /bin/sh
⟨lyx-build-helper  19b⟩
cd $PROJECT_DIR || exit 1

#/usr/bin/fangle -filter ./notanglefix-filter \
#  -R./Makefile.inc "../../noweb-lyx/noweb-lyx3.lyx" \
#  | sed '/NOWEB_SOURCE=/s/=.*/=samba4-dfs.lyx/' \
#  > ./Makefile.inc
#
#make -f ./Makefile.inc fangle_sources
```

Chapter 5
Using Fangle with TEXMACS

To do: Write this chapter

Chapter 6
Fangle with Makefiles

Here we describe a `Makefile.inc` that you can include in your own Makefiles, or glue as a recursive make to other projects.

`Makefile.inc` will cope with extracting all the other source files from this or any specified literate document and keeping them up to date.

It may also be included by a `Makefile` or `Makefile.am` defined in a literate document to automatically deal with the extraction of source files and documents during normal builds.

Thus, if `Makefile.inc` is included into a main project makefile it add rules for the source files, capable of extracting the source files from the literate document.

6.1 A word about makefiles formats

Whitespace formatting is very important in a Makefile. The first character of each action line must be a TAB.

```
target: pre-requisite
↦       action
↦       action
```

This requires that the literate programming environment have the ability to represent a TAB character in a way that fangle will generate an actual TAB character.

We also adopt a convention that code chunks whose names beginning with `./` should always be automatically extracted from the document. Code chunks whose names do not begin with `./` are for internal reference. Such chunks may be extracted directly, but will not be automatically extracted by this Makefile.

6.2 Extracting Sources

Our makefile has two parts; variables must be defined before the targets that use them.

As we progress through this chapter, explaining concepts, we will be adding lines to ⟨Makefile.inc-vars 23b⟩ and ⟨Makefile.inc-targets 24c⟩ which are included in ⟨./Makefile.inc 23a⟩ below.

23a ⟨./Makefile.inc[1](), lang=make⟩ ≡

```
1 ⟨Makefile.inc-vars 23b⟩
2 ⟨Makefile.inc-default-targets 28a⟩
3 ⟨Makefile.inc-targets 24c⟩
```

We first define a placeholder for the tool `fangle` in case it cannot be found in the path.

23b ⟨Makefile.inc-vars[1](), lang=⟩ ≡ 24a▷

```
1 FANGLE=fangle
```

We also define a placeholder for `LITERATE_SOURCE` to hold the name of this document. This will normally be passed on the command line.

24a ⟨Makefile.inc-vars[2]() ⇑23b, lang=⟩ +≡ ◁23b 24b▽

```
2 LITERATE_SOURCE=
```

Fangle cannot process LyX or TEXMACS documents directly, so the first stage is to convert these to more suitable text based formats[1].

6.2.1 Converting from LyX to LaTeX

The first stage will always be to convert the LyX file to a LaTeX file. Fangle must run on a TeX file because the LyX command `server-goto-file-line`[2] requries that the line number provided be a line of the TeX file and always maps this the line in the LyX docment. We use `server-goto-file-line` when moving the cursor to error lines during compile failures.

The command `lyx -e literate fangle.lyx` will produce `fangle.tex`, a TeX file; so we define a make target to be the same as the LyX file but with the `.tex` extension.

The `EXTRA_DIST` is for automake support so that the TeX files will automaticaly be distributed with the source, to help those who don't have LyX installed.

24b ⟨Makefile.inc-vars[3]() ⇑23b, lang=⟩ +≡ △24a 24d▽

```
3 LYX_SOURCE=$(LITERATE_SOURCE) # but only the .lyx files
4 TEX_SOURCE=$(LYX_SOURCE:.lyx=.tex)
5 EXTRA_DIST+=$(TEX_SOURCE)
```

We then specify that the TeX source is to be generated from the LyX source.

24c ⟨Makefile.inc-targets[1](), lang=⟩ ≡ 25a▷

```
1 .SUFFIXES: .tex .lyx
2 .lyx.tex:
3 ↦      lyx -e latex $<
4 clean_tex:
5 ↦      rm -f -- $(TEX_SOURCE)
6 clean: clean_tex
```

6.2.2 Converting from TEXMACS

Fangle cannot process TEXMACS files directly[3], but must first convert them to text files.

The command `texmacs -c fangle.tm fangle.txt -q` will produce `fangle.txt`, a text file; so we define a make target to be the same as the TEXMACS file but with the `.txt` extension.

The `EXTRA_DIST` is for automake support so that the TeX files will automaticaly be distributed with the source, to help those who don't have LyX installed.

24d ⟨Makefile.inc-vars[4]() ⇑23b, lang=⟩ +≡ △24b 25b▷

```
6 TEXMACS_SOURCE=$(LITERATE_SOURCE) # but only the .tm files
7 TXT_SOURCE=$(LITERATE_SOURCE:.tm=.txt)
8 EXTRA_DIST+=$(TXT_SOURCE)
```

To do: Add loop around each $< so multiple targets can be specified

1. LyX and TEXMACS formats are text-based, but not suitable for fangle
2. The Lyx command `server-goto-file-line` is used to position the Lyx cursor at the compiler errors.
3. but this is planned when TEXMACS uses xml as it's native format

25a ⟨Makefile.inc-targets[2]() ⇑24c, lang=⟩ +≡ ◁24c 25d▽

```
.SUFFIXES: .txt .tm
.tm.txt:
↦       texmacs -s -c $< $@ -q
.PHONEY: clean_txt
clean_txt:
↦       rm -f -- $(TXT_SOURCE)
clean: clean_txt
```

6.3 Extracting Program Source

The program source is extracted using fangle, which is designed to operate on text or a LaTeX documents[4].

25b ⟨Makefile.inc-vars[5]() ⇑23b, lang=⟩ +≡ ◁24d 25c▽

```
FANGLE_SOURCE=$(TXT_SOURCE)
```

The literate document can result in any number of source files, but not all of these will be changed each time the document is updated. We certainly don't want to update the timestamps of these files and cause the whole source tree to be recompiled just because the literate explanation was revised. We use `CPIF` from the *Noweb* tools to avoid updating the file if the content has not changed, but should probably write our own.

However, if a source file is not updated, then the fangle file will always have a newer time-stamp and the makefile would always re-attempt to extact a newer source file which would be a waste of time.

Because of this, we use a stamp file which is always updated each time the sources are fully extracted from the LaTeX document. If the stamp file is newer than the document, then we can avoid an attempt to re-extract any of the sources. Because this stamp file is only updated when extraction is complete, it is safe for the user to interrupt the build-process mid-extraction.

We use `echo` rather than `touch` to update the stamp file beause the `touch` command does not work very well over an `sshfs` mount that I was using.

25c ⟨Makefile.inc-vars[6]() ⇑23b, lang=⟩ +≡ △25b 26a▷

```
FANGLE_SOURCE_STAMP=$(FANGLE_SOURCE).stamp
```

25d ⟨Makefile.inc-targets[3]() ⇑24c, lang=⟩ +≡ △25a 26b▷

```
$(FANGLE_SOURCE_STAMP): $(FANGLE_SOURCE) \
↦                       $(FANGLE_SOURCES) ; \
↦       echo -n > $(FANGLE_SOURCE_STAMP)
clean_stamp:
↦       rm -f $(FANGLE_SOURCE_STAMP)
clean: clean_stamp
```

6.4 Extracting Source Files

We compute `FANGLE_SOURCES` to hold the names of all the source files defined in the document. We compute this only once, by means of `:=` in assignent. The sed deletes the any `<<` and `>>` which may surround the roots names (for compatibility with Noweb's noroots command).

4. LaTeX documents are just slightly special text documents

As we use chunk names beginning with `./` to denote top level fragments that should be extracted, we filter out all fragments that do not begin with `./`

Note 1. `FANGLE_PREFIX` is set to `./` by default, but whatever it may be overridden to, the prefix is replaced by a literal `./` before extraction so that files will be extracted in the current directory whatever the prefix. This helps namespace or sub-project prefixes like `documents:` for chunks like `documents:docbook/intro.xml`

To do: This doesn't work though, because it loses the full name and doesn't know what to extact!

26a ⟨Makefile.inc-vars[7]() ⇑23b, lang=⟩ +≡ ◁25c 26e▽

```
FANGLE_PREFIX:=\./
FANGLE_SOURCES:=$(shell \
  $(FANGLE) -r $(FANGLE_SOURCE) |\
  sed -e 's/^[<][<]//;s/[>][>]$$//;/^$(FANGLE_PREFIX)/!d' \
      -e 's/^$(FANGLE_PREFIX)/\.\//' )
```

The target below, `echo_fangle_sources` is a helpful debugging target and shows the names of the files that would be extracted.

26b ⟨Makefile.inc-targets[4]() ⇑24c, lang=⟩ +≡ ◁25d 26c▽

```
.PHONY: echo_fangle_sources
echo_fangle_sources: ; @echo $(FANGLE_SOURCES)
```

We define a convenient target called `fangle_sources` so that `make -f fangle_sources` will re-extract the source if the literate document has been updated.

26c ⟨Makefile.inc-targets[5]() ⇑24c, lang=⟩ +≡ △26b 26d▽

```
.PHONY: fangle_sources
fangle_sources: $(FANGLE_SOURCE_STAMP)
```

And also a convenient target to remove extracted sources.

26d ⟨Makefile.inc-targets[6]() ⇑24c, lang=⟩ +≡ △26c 27e▷

```
.PHONY: clean_fangle_sources
clean_fangle_sources: ; \
        rm -f -- $(FANGLE_SOURCE_STAMP) $(FANGLE_SOURCES)
```

We now look at the extraction of the source files.

This makefile macro `if_extension` takes 4 arguments: the filename `$(1)`, some extensions to match `$(2)` and a shell command to return if the filename does match the exensions `$(3)`, and a shell command to return if it does not match the extensions `$(4)`.

26e ⟨Makefile.inc-vars[8]() ⇑23b, lang=⟩ +≡ △26a 26f▽

```
if_extension=$(if $(findstring $(suffix $(1)),$(2)),$(3),$(4))
```

For some source files like C files, we want to output the line number and filename of the original LaTeX document from which the source came[5].

To make this easier we define the file extensions for which we want to do this.

26f ⟨Makefile.inc-vars[9]() ⇑23b, lang=⟩ +≡ △26e 27a▷

```
C_EXTENSIONS=.c .h
```

5. I plan to replace this option with a separate mapping file so as not to pollute the generated source, and also to allow a code pretty-printing reformatter like `indent` be able to re-format the file and adjust for changes through comparing the character streams.

We can then use the `if_extensions` macro to define a macro which expands out to the `-L` option if fangle is being invoked in a C source file, so that C compile errors will refer to the line number in the TEX document.

27a ⟨Makefile.inc-vars[10]() ⇑23b, lang=⟩ +≡ ◁26f 27b▽

```
TABS=8
nf_line=-L -T$(TABS)
fangle=$(FANGLE) $(call if_extension,$(2),$(C_EXTENSIONS),$(nf_line)) -R"$(2)" $(1)
```

We can use a similar trick to define an indent macro which takes just the filename as an argument and can return a pipeline stage calling the indent command. Indent can be turned off with `make fangle_sources indent=`

27b ⟨Makefile.inc-vars[11]() ⇑23b, lang=⟩ +≡ △27a 27c▽

```
indent_options=-npro -kr -i8 -ts8 -sob -l80 -ss -ncs
indent=$(call if_extension,$(1),$(C_EXTENSIONS), | indent $(indent_options))
```

We now define the pattern for extracting a file. The files are written using noweb's `cpif` so that the file timestamp will not be touched if the contents haven't changed. This avoids the need to rebuild the entire project because of a typographical change in the documentation, or if none or a few C source files have changed.

27c ⟨Makefile.inc-vars[12]() ⇑23b, lang=⟩ +≡ △27b 27d▽

```
fangle_extract=@mkdir -p $(dir $(1)) && \
  $(call fangle,$(2),$(1)) > "$(1).tmp" && \
  cat "$(1).tmp" $(indent) | cpif "$(1)" \
  && rm -f -- "$(1).tmp" || \
  (echo error fangling $(1) from $(2) ; exit 1)
```

We define a target which will extract or update all sources. To do this we first defined a makefile template that can do this for any source file in the LATEX document.

27d ⟨Makefile.inc-vars[13]() ⇑23b, lang=⟩ +≡ △27c 28b▷

```
define FANGLE_template
  $(1): $(2)
↦       $$(call fangle_extract,$(1),$(2))
  FANGLE_TARGETS+=$(1)
endef
```

We then enumerate the discovered `FANGLE_SOURCES` to generate a makefile rule for each one using the makefile template we defined above.

27e ⟨Makefile.inc-targets[7]() ⇑24c, lang=⟩ +≡ ◁26d 27f▽

```
$(foreach source,$(FANGLE_SOURCES),\
  $(eval $(call FANGLE_template,$(source),$(FANGLE_SOURCE))) \
)
```

These will all be built with `FANGLE_SOURCE_STAMP`.

We also remove the generated sources on a make distclean.

27f ⟨Makefile.inc-targets[8]() ⇑24c, lang=⟩ +≡ △27e 28c▷

```
_distclean: clean_fangle_sources
```

6.5 Extracting Documentation

We then identify the intermediate stages of the documentation and their build and clean targets.

28a ⟨Makefile.inc-default-targets[1](), lang=⟩ ≡

```
.PHONEY : clean_pdf
```

6.5.1 Formatting TEX

6.5.1.1 Running pdflatex

We produce a pdf file from the tex file.

28b ⟨Makefile.inc-vars[14]() ⇑23b, lang=⟩ +≡ ◁27d 28d▽

```
FANGLE_PDF+=$(TEX_SOURCE:.tex=.pdf)
```

We run pdflatex twice to be sure that the contents and aux files are up to date. We certainly are *required* to run pdflatex at least twice if these files do not exist.

28c ⟨Makefile.inc-targets[9]() ⇑24c, lang=⟩ +≡ ◁27f 28e▽

```
.SUFFIXES: .tex .pdf
.tex.pdf:
↦       pdflatex $< && pdflatex $<

clean_pdf_tex:
↦       rm -f -- $(FANGLE_PDF) $(TEX_SOURCE:.tex=.toc) \
↦         $(TEX_SOURCE:.tex=.log) $(TEX_SOURCE:.tex=.aux)
clean_pdf: clean_pdf_tex
```

6.5.2 Formatting TEXMACS

TEXMACS can produce a PDF file directly.

28d ⟨Makefile.inc-vars[15]() ⇑23b, lang=⟩ +≡ △28b 28f▽

```
FANGLE_PDF+=$(LITERATE_SOURCE:.tm=.pdf)
```

To do: Outputting the PDF may not be enough to update the links and page references. I think we need to update twice, generate a pdf, update twice mode and generate a new PDF. Basically the PDF export of TEXMACS is pretty rotten and doesn't work properly from the CLI

28e ⟨Makefile.inc-targets[10]() ⇑24c, lang=⟩ +≡ △28c 29a▷

```
.SUFFIXES: .tm .pdf
.tm.pdf:
↦       texmacs -s -c $< $@ -q

clean_pdf_texmacs:
↦       rm -f -- $(FANGLE_PDF)
clean_pdf: clean_pdf_texmacs
```

6.5.3 Building the Documentation as a Whole

Currently we only build pdf as a final format, but `FANGLE_DOCS` may later hold other output formats.

28f ⟨Makefile.inc-vars[16]() ⇑23b, lang=⟩ +≡ △28d

```
FANGLE_DOCS=$(FANGLE_PDF)
```

We also define `fangle_docs` as a convenient phony target.

29a ⟨Makefile.inc-targets[11]() ⇑24c, lang=⟩ +≡ ◁28e 29b▽

```
.PHONY: fangle_docs
fangle_docs: $(FANGLE_DOCS)
docs: fangle_docs
```

And define a convenient `clean_fangle_docs` which we add to the regular clean target

29b ⟨Makefile.inc-targets[12]() ⇑24c, lang=⟩ +≡ △29a

```
.PHONEY: clean_fangle_docs
clean_fangle_docs: clean_tex clean_pdf
clean: clean_fangle_docs

distclean_fangle_docs: clean_tex clean_fangle_docs
distclean: clean distclean_fangle_docs
```

6.6 Other helpers

If `Makefile.inc` is included into `Makefile`, then extracted files can be updated with this command:

```
make fangle_sources
```

otherwise, with:

```
make -f Makefile.inc fangle_sources
```

6.7 Boot-strapping the extraction

As well as having the makefile extract or update the source files as part of it's operation, it also seems convenient to have the makefile re-extracted itself from *this* document.

It would also be convenient to have the code that extracts the makefile from this document to also be part of this document, however we have to start somewhere and this unfortunately requires us to type at least a few words by hand to start things off.

Therefore we will have a minimal root fragment, which, when extracted, can cope with extracting the rest of the source. This shell script fragment can do that. It's name is `*` — out of regard for NOWEB, but when extracted might better be called `autoupdate`.

To do: De-lyxify

29c ⟨*[1](), lang=sh⟩ ≡

```
#! /bin/sh

MAKE_SRC="${1:-${NW_LYX:-../../noweb-lyx/noweb-lyx3.lyx}}"
MAKE_SRC=‘dirname "$MAKE_SRC"‘/‘basename "$MAKE_SRC" .lyx‘
NOWEB_SRC="${2:-${NOWEB_SRC:-$MAKE_SRC.lyx}}"
lyx -e latex $MAKE_SRC

fangle -R./Makefile.inc ${MAKE_SRC}.tex \
  | sed "/FANGLE_SOURCE=/s/^/#/;T;aNOWEB_SOURCE=$FANGLE_SRC" \
  | cpif ./Makefile.inc

make -f ./Makefile.inc fangle_sources
```

The general Makefile can be invoked with `./autoboot` and can also be included into any automake file to automatically re-generate the source files.

The *autoboot* can be extracted with this command:

```
lyx -e latex fangle.lyx && \
  fangle fangle.lyx > ./autoboot
```

This looks simple enough, but as mentioned, fangle has to be had from somewhere before it can be extracted.

On a unix system this will extract `fangle.module` and the `fangle` awk script, and run some basic tests.

To do: cross-ref to test chapter when it is a chapter all on its own

6.8 Incorporating Makefile.inc into existing projects

If you are writing a literate module of an existing non-literate program you may find it easier to use a slight recursive make instead of directly including `Makefile.inc` in the projects makefile.

This way there is less chance of definitions in `Makefile.inc` interfering with definitions in the main makefile, or with definitions in other `Makefile.inc` from other literate modules of the same project.

To do this we add some *glue* to the project makefile that invokes Makefile.inc in the right way. The glue works by adding a `.PHONY` target to call the recursive make, and adding this target as an additional pre-requisite to the existing targets.

Example Sub-module of existing system

In this example, we are building `module.so` as a literate module of a larger project.

We will show the sort glue that can be inserted into the projects Makefile — or more likely — a regular Makefile included in or invoked by the projects Makefile.

30a ⟨makefile-glue[1](), lang=⟩ ≡ 30b▽

```
1 module_srcdir=modules/module
2 MODULE_SOURCE=module.tm
3 MODULE_STAMP=$(MODULE_SOURCE).stamp
```

The existing build system may already have a build target for `module.o`, but we just add another pre-requisite to that. In this case we use `module.tm.stamp` as a pre-requisite, the stamp file's modified time indicating when all sources were extracted[6].

30b ⟨makefile-glue[2]() ⇑30a, lang=make⟩ +≡ △30a 30c▽

```
4 $(module_srcdir)/module.o: $(module_srcdir)/$(MODULE_STAMP)
```

The target for this new pre-requisite will be generated by a recursive make using `Makefile.inc` which will make sure that the source is up to date, before it is built by the main projects makefile.

30c ⟨makefile-glue[3]() ⇑30a, lang=⟩ +≡ △30b 31a▷

```
5 $(module_srcdir)/$(MODULE_STAMP): $(module_srcdir)/$(MODULE_SOURCE)
6 ↦       $(MAKE) -C $(module_srcdir) -f Makefile.inc fangle_sources LITERATE_SOURCE=$(MODULE_SOURCE)
```

We can do similar glue for the docs, clean and distclean targets. In this example the main prject was using a double colon for these targets, so we must use the same in our glue.

6. If the projects build system does not know how to build the module from the extracted sources, then just add build actions here as normal.

31a ⟨makefile-glue[4]() ⇑30a, lang=⟩ +≡ ◁30c

```
docs:: docs_module
.PHONY: docs_module
docs_module:
↦	$(MAKE) -C $(module_srcdir) -f Makefile.inc docs LITERATE_SOURCE=$(MODULE_SOURCE)

clean:: clean_module
.PHONEY: clean_module
clean_module:
↦	$(MAKE) -C $(module_srcdir) -f Makefile.inc clean LITERATE_SOURCE=$(MODULE_SOURCE)

distclean:: distclean_module
.PHONY: distclean_module
distclean_module:
↦	$(MAKE) -C $(module_srcdir) -f Makefile.inc distclean LITERATE_SOURCE=$(MODULE_SOURCE)
```

We could do similarly for install targets to install the generated docs.

Part II

Source Code

Chapter 7
Fangle awk source code

We use the copyright notice from chapter 2.

35a ⟨./fangle[1](), lang=awk⟩ ≡ 35b▽

```
#! /usr/bin/awk -f
# ⟨gpl3-copyright 4a⟩
```

We also use code from Arnold Robbins public domain getopt (1993 revision) defined in 81a, and naturally want to attribute this appropriately.

35b ⟨./fangle[2]() ⇑35a, lang=⟩ +≡ △35a 35c▽

```
# NOTE: Arnold Robbins public domain getopt for awk is also used:
⟨getopt.awk-header 79a⟩
⟨getopt.awk-getopt() 79c⟩

```

And include the following chunks (which are explained further on) to make up the program:

35c ⟨./fangle[3]() ⇑35a, lang=⟩ +≡ △35b 40a▷

```
⟨helper-functions 36d⟩
⟨mode-tracker 59a⟩
⟨parse_chunk_args 42a⟩
⟨chunk-storage-functions 77b⟩
⟨output_chunk_names() 71d⟩
⟨output_chunks() 71e⟩
⟨write_chunk() 72a⟩
⟨expand_chunk_args() 42b⟩

⟨begin 69d⟩
⟨recognize-chunk 61a⟩
⟨end 71c⟩
```

7.1 AWK tricks

The portable way to erase an array in awk is to split the empty string, so we define a fangle macro that can split an array, like this:

35d ⟨awk-delete-array[1](ARRAY), lang=awk⟩ ≡

```
split("", ⟨ARRAY⟩);
```

For debugging it is sometimes convenient to be able to dump the contents of an array to `stderr`, and so this macro is also useful.

35e ⟨dump-array[1](ARRAY), lang=awk⟩ ≡

```
print "\nDump: ⟨ARRAY⟩\n--------\n" > "/dev/stderr";
for (_x in ⟨ARRAY⟩) {
```

35e ⟨dump-array[1](ARRAY), lang=awk⟩ ≡

```
  print _x "=" ⟨ARRAY⟩[_x] "\n" > "/dev/stderr";
}
print "========\n" > "/dev/stderr";
```

7.2 Catching errors

Fatal errors are issued with the error function:

36a ⟨error()[1](), lang=awk⟩ ≡ 36b▽

```
function error(message)
{
  print "ERROR: " FILENAME ":" FNR " " message > "/dev/stderr";
  exit 1;
}
```

and likewise for non-fatal warnings:

36b ⟨error()[2]() ⇑36a, lang=awk⟩ +≡ △36a 36c▽

```
function warning(message)
{
  print "WARNING: " FILENAME ":" FNR " " message > "/dev/stderr";
  warnings++;
}
```

and debug output too:

36c ⟨error()[3]() ⇑36a, lang=awk⟩ +≡ △36b

```
function debug_log(message)
{
  print "DEBUG: " FILENAME ":" FNR " " message > "/dev/stderr";
}
```

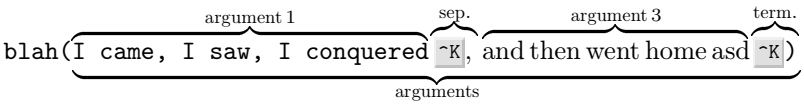
To do: append=helper-functions

36d ⟨helper-functions[1](), lang=⟩ ≡

```
⟨error()  36a⟩
```

Chapter 8
TEXMACS args

TEXMACS functions with arguments[1] appear like this:



Arguments commence after the opening parenthesis. The first argument runs up till the next `^K`.

If the following character is a `,` then another argument follows. If the next character after the `,` is a space character, then it is also eaten. The fangle stylesheet emits `^K` `,` `space` as separators, but the fangle untangler will forgive a missing space.

If the following character is `)` then this is a terminator and there are no more arguments.

37a ⟨constants[1](), lang=⟩ ≡ 77a▷

```
ARG_SEPARATOR=sprintf("%c", 11);
```

To process the `text` in this fashion, we split the string on `^K`

37b ⟨get_chunk_args[1](), lang=⟩ ≡

```
function get_texmacs_chunk_args(text, args,   a, done) {
  split(text, args, ARG_SEPARATOR);

  done=0
  for (a=1; (a in args); a++) if (a>1) {
    if (args[a] == "" || substr(args[a], 1, 1) == ")") done=1;
    if (done) {
      delete args[a];
      break;
    }

    if (substr(args[a], 1, 2) == ", ") args[a]=substr(args[a], 3);
    else if (substr(args[a], 1, 1) == ",") args[a]=substr(args[a], 2);
  }
}
```

1. or function declarations with parameters

Chapter 9
LaTeX and lstlistings

To do: Split LyX and TeXmacs parts

For LyX and LaTeX, the `lstlistings` package is used to format the lines of code chunks. You may recal from chapter XXX that arguments to a chunk definition are pure LaTeX code. This means that fangle needs to be able to parse LaTeX a little.

LaTeX arguments to `lstlistings` macros are a comma seperated list of key-value pairs, and values containing commas are enclosed in `{` braces `}` (which is to be expected for LaTeX).

A sample expressions is:

```
name=thomas, params={a, b}, something, something-else
```

but we see that this is just a simpler form of this expression:

```
name=freddie, foo={bar=baz, quux={quirk, a=fleeg}}, etc
```

We may consider that we need a function that can parse such LaTeX expressions and assign the values to an AWK associated array, perhaps using a recursive parser into a multi-dimensional hash[1], resulting in:

key	value
a[name]	freddie
a[foo, bar]	baz
a[foo, quux, quirk]	
a[foo, quux, a]	fleeg
a[etc]	

Yet, also, on reflection it seems that sometimes such nesting is not desirable, as the braces are also used to delimit values that contain commas — we may consider that

```
name={williamson, freddie}
```

should assign `williamson, freddie` to `name`.

In fact we are not so interested in the detail so as to be bothered by this, which turns out to be a good thing for two reasons. Firstly TeX has a malleable parser with no strict syntax, and secondly whether or not `williamson` and `freddie` should count as two items will be context dependant anyway.

We need to parse this latex for only one reason; which is that we are extending lstlistings to add some additional arguments which will be used to express chunk parameters and other chunk options.

9.1 Additional lstlstings parameters

Further on we define a `\Chunk` LaTeX macro whose arguments will consist of a the chunk name, optionally followed by a comma and then a comma separated list of arguments. In fact we will just need to prefix `name=` to the arguments to in order to create valid lstlistings arguments.

1. as AWK doesn't have nested-hash support

There will be other arguments supported too;

params.
As an extension to many literate-programming styles, fangle permits code chunks to take parameters and thus operate somewhat like C pre-processor macros, or like C++ templates. Chunk parameters are declared with a chunk argument called params, which holds a semi-colon separated list of parameters, like this:

```
achunk,language=C,params=name;address
```

addto.
a named chunk that this chunk is to be included into. This saves the effort of having to declare another listing of the named chunk merely to include this one.

Function get_chunk_args() will accept two paramters, text being the text to parse, and values being an array to receive the parsed values as described above. The optional parameter path is used during recursion to build up the multi-dimensional array path.

40a ⟨./fangle[4]() ⇑35a, lang=⟩ +≡ ◁35c

```
19 ⟨get_chunk_args()  40b⟩
```

40b ⟨get_chunk_args()[1](), lang=⟩ ≡ 40c▽

```
1  function get_tex_chunk_args(text, values,
2    # optional parameters
3    path, # hierarchical precursors
4    # local vars
5    a, name)
```

The strategy is to parse the name, and then look for a value. If the value begins with a brace {, then we recurse and consume as much of the text as necessary, returning the remaining text when we encounter a leading close-brace }. This being the strategy — and executed in a loop — we realise that we must first look for the closing brace (perhaps preceded by white space) in order to terminate the recursion, and returning remaining text.

40c ⟨get_chunk_args()[2]() ⇑40b, lang=⟩ +≡ △40b

```
6  {
7    split("", values);
8    while(length(text)) {
9      if (match(text, "^ *}(.*)", a)) {
10       return a[1];
11     }
12     ⟨parse-chunk-args  40d⟩
13   }
14   return text;
15 }
```

We can see that the text could be inspected with this regex:

40d ⟨parse-chunk-args[1](), lang=⟩ ≡ 41a▷

```
1  if (! match(text, " *([^,=]*[^,= ]) *(([,=]) *(([^,}]*) *,* *(.*))|)$", a)) {
2    return text;
3  }
```

and that a will have the following values:

a[n]	assigned text
1	freddie
2	=freddie, foo={bar=baz, quux={quirk, a=fleeg}}, etc
3	=
4	freddie, foo={bar=baz, quux={quirk, a=fleeg}}, etc
5	freddie
6	, foo={bar=baz, quux={quirk, a=fleeg}}, etc

`a[3]` will be either `=` or `,` and signify whether the option named in `a[1]` has a value or not (respectively).

If the option does have a value, then if the expression `substr(a[4],1,1)` returns a brace `{` it will signify that we need to recurse:

41a ⟨parse-chunk-args[2]() ⇑40d, lang=⟩ +≡ ◁40d

```
name=a[1];
if (a[3] == "=") {
  if (substr(a[4],1,1) == "{") {
    text = get_tex_chunk_args(substr(a[4],2), values, path name SUBSEP);
  } else {
    values[path name]=a[5];
    text = a[6];
  }
} else {
  values[path name]="";
  text = a[2];
}
```

We can test this function like this:

41b ⟨gca-test.awk[1](), lang=⟩ ≡

```
⟨get_chunk_args()  40b⟩
BEGIN {
  SUBSEP=".";

  print get_tex_chunk_args("name=freddie, foo={bar=baz, quux={quirk, a=fleeg}}, etc", a);
  for (b in a) {
    print "a[" b "] => " a[b];
  }
}
```

which should give this output:

41c ⟨gca-test.awk-results[1](), lang=⟩ ≡

```
a[foo.quux.quirk] =>
a[foo.quux.a] => fleeg
a[foo.bar] => baz
a[etc] =>
a[name] => freddie
```

9.2 Parsing chunk arguments

Arguments to paramterized chunks are expressed in round brackets as a comma separated list of optional arguments. For example, a chunk that is defined with:

`\Chunk{achunk, params=name ; address}`

could be invoked as:

`\chunkref{achunk}(John Jones, jones@example.com)`

An argument list may be as simple as in `\chunkref{pull}(thing, otherthing)` or as complex as:

`\chunkref{pull}(things[x, y], get_other_things(a, "(all)"))`

— which for all it's commas and quotes and parenthesis represents only two parameters: `things[x, y]` and `get_other_things(a, "(all)")`.

If we simply split parameter list on commas, then the comma in `things[x,y]` would split into two seperate arguments: `things[x` and `y]`— neither of which make sense on their own.

One way to prevent this would be by refusing to split text between matching delimiters, such as `[`, `]`, `(`, `)`, `{`, `}` and most likely also `"`, `"` and `'`, `'`. Of course this also makes it impossible to pass such mis-matched code fragments as parameters, but I think that it would be hard for readers to cope with authors who would pass such code unbalanced fragments as chunk parameters[2].

Unfortunately, the full set of matching delimiters may vary from language to language. In certain C++ template contexts, `<` and `>` would count as delimiters, and yet in other contexts they would not.

This puts me in the unfortunate position of having to parse-somewhat all programming languages without knowing what they are!

However, if this universal mode-tracking is possible, then parsing the arguments would be trivial. Such a mode tracker is described in chapter 10 and used here with simplicity.

42a ⟨parse_chunk_args[1](), lang=⟩ ≡

```
function parse_chunk_args(language, text, values, mode,
  # local vars
  c, context, rest)
{
  ⟨new-mode-tracker(context, language, mode) 55a⟩
  rest = mode_tracker(context, text, values);
  # extract values
  for(c=1; c <= context[0, "values"]; c++) {
    values[c] = context[0, "values", c];
  }
  return rest;
}
```

9.3 Expanding parameters in the text

Within the body of the chunk, the parameters are referred to with: `${name}` and `${address}`. There is a strong case that a LaTeX style notation should be used, like `\param{name}` which would be expressed in the listing as `=<\param{name}>` and be rendered as ⟨name⟩. Such notation would make me go blind, but I do intend to adopt it.

We therefore need a function `expand_chunk_args` which will take a block of text, a list of permitted parameters, and the arguments which must substitute for the parameters.

Here we split the text on `${` which means that all parts except the first will begin with a parameter name which will be terminated by `}`. The split function will consume the literal `${` in each case.

42b ⟨expand_chunk_args()[1](), lang=⟩ ≡

```
function expand_chunk_args(text, params, args,
  p, text_array, next_text, v, t, l)
{
  if (split(text, text_array, "\\${")) {
    ⟨substitute-chunk-args  43a⟩
  }

  return text;
}
```

First, we produce an associative array of substitution values indexed by parameter names. This will serve as a cache, allowing us to look up the replacement values as we extract each name.

2. I know that I couldn't cope with users doing such things, and although the GPL3 license prevents me from actually forbidding anyone from trying, if they want it to work they'll have to write the code themselves and not expect any support from me.

43a ⟨substitute-chunk-args[1](), lang=⟩ ≡ 43b▽

```
for(p in params) {
  v[params[p]]=args[p];
}
```

We accumulate substituted text in the variable text. As the first part of the split function is the part before the delimiter — which is `${` in our case — this part will never contain a parameter reference, so we assign this directly to the result kept in `$text`.

43b ⟨substitute-chunk-args[2]() ⇑43a, lang=⟩ +≡ △43a 43c▽

```
text=text_array[1];
```

We then iterate over the remaining values in the array, and substitute each reference for it's argument.

43c ⟨substitute-chunk-args[3]() ⇑43a, lang=⟩ +≡ △43b

```
for(t=2; t in text_array; t++) {
  ⟨substitute-chunk-arg  43d⟩
}
```

After the split on `${` a valid parameter reference will consist of valid parameter name terminated by a close-brace `}`. A valid character name begins with the underscore or a letter, and may contain letters, digits or underscores.

A valid looking reference that is not actually the name of a parameter will be and not substituted. This is good because there is nothing to substitute anyway, and it avoids clashes when writing code for languages where `${...}` is a valid construct — such constructs will not be interfered with unless the parameter name also matches.

43d ⟨substitute-chunk-arg[1](), lang=⟩ ≡

```
if (match(text_array[t], "^([a-zA-Z_][a-zA-Z0-9_]*)}", l) &&
    l[1] in v)
{
  text = text v[l[1]] substr(text_array[t], length(l[1])+2);
} else {
  text = text "${" text_array[t];
}
```

Chapter 10
Language Modes & Quoting

`lstlistings` and `fangle` both recognize source languages, and perform some basic parsing and syntax highlighting in the rendered document[1]. `lstlistings` can detect strings and comments within a language definition and perform suitable rendering, such as italics for comments, and visible-spaces within strings.

Fangle similarly can recognize strings, and comments, etc, within a language, so that any chunks included with `\chunkref{a-chunk}` or ⟨a-chunk ?⟩ can be suitably escape or quoted.

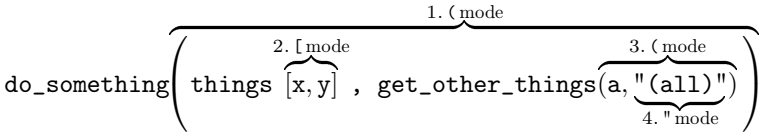
10.1 Modes to keep code together

As an example, the C language has a few parse modes, which affect the interpretation of characters.

One parse mode is the string mode. The string mode is commenced by an un-escaped quotation mark `"` and terminated by the same. Within the string mode, only one additional mode can be commenced, it is the backslash mode `\`, which is always terminated by the following character.

Another mode is `[` which is terminated by a `]` (unless it occurs in a string).

Consider this fragment of C code:



Mode nesting prevents the close parenthesis in the quoted string (part 4) from terminating the parenthesis mode (part 3).

Each language has a set of modes, the default mode being the null mode. Each mode can lead to other modes.

10.2 Modes affect included chunks

For instance, consider this chunk with `language=perl`:

45a ⟨test:example-perl[1](), lang=perl⟩ ≡

```
1 print "hello world $0\n";
```

If it were included in a chunk with `language=sh`, like this:

45b ⟨test:example-sh[1](), lang=sh⟩ ≡

```
1 perl -e "⟨test:example-perl  45a⟩"
```

1. although lstlisting supports many more languages

we might want fangle would to generate output like this:

46a ⟨test:example-sh.result[1](), lang=sh⟩ ≡

```
1 perl -e "print \"hello world \$0\\n\";"
```

See that the double quote ", back-slash \ and $ have been quoted with a back-slash to protect them from shell interpretation.

If that were then included in a chunk with language=make, like this:

46b ⟨test:example-makefile[1](), lang=make⟩ ≡

```
1 target: pre-req
2 ↦       ⟨test:example-sh  45b⟩
```

We would need the output to look like this — note the $$ as the single $ has been makefile-quoted with another $.

46c ⟨test:example-makefile.result[1](), lang=make⟩ ≡

```
1 target: pre-req
2 ↦       perl -e "print \"hello world \$$0\\n\";"
```

10.3 Modes operation

In order to make this work, we must define a mode-tracker supporting each language, that can detect the various quoting modes, and provide a transformation that may be applied to any included text so that included text will be interpreted correctly after any interpolation that it may be subject to at run-time.

For example, the sed transformation for text to be inserted into shell double-quoted strings would be something like:

```
s/\\/\\\\/g;s/$/\\$/g;s/"/\\"/g;
```

which would protect \ $ "

The mode tracker must also nested mode-changes, as in this shell example:

```
echo "hello ‘id ...‘"
                  ↑
```

Any shell special characters inserted at the point marked ↑ would need to be escaped if their plain-text meaning is to be preserved, including ‘ | * among others. The set of characters that need escaping in the back-ticks ‘ is not the same as the set that need escaing in the double-quotes ". However, in shell syntax, a " at the point marked ↑ does not close the leading " and so would not need additional escaping because of the nesting of the two modes.

To do: MAYBE Escaping need not occur if the format and mode of the included chunk matches that of the including chunk.

As each chunk is output a new mode tracker for that language is initialized in it's normal state. As text is output for that chunk the output mode is tracked. When a new chunk is included, a transformation appropriate to that mode is selected and pushed onto a stack of transformations. Any text to be output is passed through this stack of transformations.

It remains to consider if the chunk-include function should return it's generated text so that the caller can apply any transformations (and formatting), or if it should apply the stack of transformations itself.

Note that the transformed included text should have the property of not being able to change the mode in the current chunk.

To do: Note chunk parameters should probably also be transformed

10.4 Quoting scenarios

10.4.1 Direct quoting

He we give examples of various quoting scenarios and discuss what the expected outcome might be and how this could be obtained.

47a ⟨test:q:1[1](), lang=sh⟩ ≡

```
echo "$(⟨test:q:1-inc  47b⟩)"
```

47b ⟨test:q:1-inc[1](), lang=sh⟩ ≡

```
echo "hello"
```

Should this examples produce `echo "$(echo "hello")"` or `echo "\$(echo \"hello\")"` ?

This depends on what the author intended, but we must provde a way to express that intent.

We might argue that as both chunks have `lang=sh` the intent must have been to quote the included chunk — but consider that this might be shell script that writes shell script.

If ⟨test:q:1-inc 47b⟩ had `lang=text` then it certainly would have been right to quote it, which leads us to ask: in what ways can we reduce quoting if lang of the included chunk is compatible with the lang of the including chunk?

If we take a completely nested approach then even though `$(` mode might do no quoting of it's own, `"` mode will still do it's own quoting. We need a model where the nested `$(` mode will prevent `"` from quoting.

This leads rise to the *tunneling* feature. In bash, the `$(` gives rise to a new top-level parsing scenario, so we need to enter the *null* mode, and also ignore any quoting and then undo-this when the `$(` mode is terminated by the corresponding close `)`.

We shall say that tunneling is when a mode in a language ignores other modes in the same language and arrives back at an earlier *null* mode of the same language.

In example ⟨test:q:1 47a⟩ above, the nesting of modes is: *null*, `"`, `$(`

When mode `$(` is commenced, the stack of nest modes will be traversed. If the *null* mode can be found in the same language, without the language varying, then a tunnel will be established so that the intervening modes, `"` in this case, can be skipped when the modes are enumerated to quote the texted being emitted.

In such a case, the correct result would be:

47c ⟨test:q:1.result[1](), lang=sh⟩ ≡

```
echo "$(echo "hello")"
```

10.5 Language Mode Definitions

All modes definitions are stored in a single multi-dimensional hash. The first index is the language, and the second index is the mode-identifier. The third indexes hold properties: terminators, and optionally, submodes, delimiters, and tunnel targets.

A useful set of mode definitions for a nameless general C-type language is shown here. (Don't be confused by the double backslash escaping needed in awk. One set of escaping is for the string, and the second set of escaping is for the regex).

To do: TODO: Add =<\mode{}> command which will allow us to signify that a string is regex and thus fangle will quote it for us.

Submodes are entered by the characters `" ’ { ( [ /*`

48a ⟨common-mode-definitions[1](language), lang=⟩ ≡ 48b▽

```
modes[⟨language⟩, "",  "submodes"]="\\\\|\"|’|{|\\(|\\[";
```

In the default mode, a comma surrounded by un-important white space is a delimiter of language items[2].

48b ⟨common-mode-definitions[2](language) ⇑48a, lang=⟩ +≡ △48a 48d▽

```
modes[⟨language⟩, "",  "delimiters"]=" *, *";
```

and should pass this test: To do: Why do the tests run in ?(? mode and not ?? mode

48c ⟨test:mode-definitions[1](), lang=⟩ ≡ 49g▷

```
parse_chunk_args("c-like", "1,2,3", a, "");
if (a[1] != "1") e++;
if (a[2] != "2") e++;
if (a[3] != "3") e++;
if (length(a) != 3) e++;
⟨pca-test.awk:summary  59c⟩

parse_chunk_args("c-like", "joe, red", a, "");
if (a[1] != "joe") e++;
if (a[2] != "red") e++;
if (length(a) != 2) e++;
⟨pca-test.awk:summary  59c⟩

parse_chunk_args("c-like", "${colour}", a, "");
if (a[1] != "${colour}") e++;
if (length(a) != 1) e++;
⟨pca-test.awk:summary  59c⟩
```

Nested modes are identified by a backslash, a double or single quote, various bracket styles or a `/*` comment.

For each of these sub-modes modes we must also identify at a mode terminator, and any sub-modes or delimiters that may be entered[3].

10.5.1 Backslash

The backslash mode has no submodes or delimiters, and is terminated by any character. Note that we are not so much interested in evaluating or interpolating content as we are in delineating content. It is no matter that a double backslash (\\) may represent a single backslash while a backslash-newline may represent white space, but it does matter that the newline in a backslash newline should not be able to terminate a C pre-processor statement; and so the newline will be consumed by the backslash however it is to be interpreted.

48d ⟨common-mode-definitions[3](language) ⇑48a, lang=⟩ +≡ △48b 49f▷

```
modes[⟨language⟩, "\\", "terminators"]=".";
```

2. whatever a *language item* might be

3. Because we are using the sub-mode characters as the mode identifier it means we can't currently have a mode character dependant on it's context; i.e. `{` can't behave differently when it is inside `[`.

10.5.2 Strings

Common languages support two kinds of strings quoting, double quotes and single quotes.

In a string we have one special mode, which is the backslash. This may escape an embedded quote and prevent us thinking that it should terminate the string.

49a ⟨mode:common-string[1](language, quote), lang=⟩ ≡ 49b▽

```
modes[⟨language⟩, ⟨quote⟩, "submodes"]="\\\\";
```

Otherwise, the string will be terminated by the same character that commenced it.

49b ⟨mode:common-string[2](language, quote) ⇑49a, lang=⟩ +≡ △49a 49c▽

```
modes[⟨language⟩, ⟨quote⟩, "terminators"]=⟨quote⟩;
```

In C type languages, certain escape sequences exist in strings. We need to define mechanism to enclode any chunks included in this mode using those escape sequences. These are expressed in two parts, s meaning search, and r meaning replace.

The first substitution is to replace a backslash with a double backslash. We do this first as other substitutions may introduce a backslash which we would not then want to escape again here.

Note: Backslashes need double-escaping in the search pattern but not in the replacement string, hence we are replacing a literal \ with a literal \\.

49c ⟨mode:common-string[3](language, quote) ⇑49a, lang=⟩ +≡ △49b 49d▽

```
escapes[⟨language⟩, ⟨quote⟩, ++escapes[⟨language⟩, ⟨quote⟩], "s"]="\\\\";
escapes[⟨language⟩, ⟨quote⟩,   escapes[⟨language⟩, ⟨quote⟩], "r"]="\\\\";
```

If the quote character occurs in the text, it should be preceded by a backslash, otherwise it would terminate the string unexpectedly.

49d ⟨mode:common-string[4](language, quote) ⇑49a, lang=⟩ +≡ △49c 49e▽

```
escapes[⟨language⟩, ⟨quote⟩, ++escapes[⟨language⟩, ⟨quote⟩], "s"]=⟨quote⟩;
escapes[⟨language⟩, ⟨quote⟩,   escapes[⟨language⟩, ⟨quote⟩], "r"]="\\" ⟨quote⟩;
```

Any newlines in the string, must be replaced by \n.

49e ⟨mode:common-string[5](language, quote) ⇑49a, lang=⟩ +≡ △49d

```
escapes[⟨language⟩, ⟨quote⟩, ++escapes[⟨language⟩, ⟨quote⟩], "s"]="\n";
escapes[⟨language⟩, ⟨quote⟩,   escapes[⟨language⟩, ⟨quote⟩], "r"]="\\n";
```

For the common modes, we define this string handling for double and single quotes.

49f ⟨common-mode-definitions[4](language) ⇑48a, lang=⟩ +≡ ◁48d 50b▷

```
⟨mode:common-string(⟨language⟩, "\"") 49a⟩
⟨mode:common-string(⟨language⟩, "'") 49a⟩
```

Working strings should pass this test:

49g ⟨test:mode-definitions[2]() ⇑48c, lang=⟩ +≡ ◁48c 54b▷

```
parse_chunk_args("c-like", "say \"I said, \\\"Hello, how are you\\\".\", for me", a, "");
if (a[1] != "say \"I said, \\\"Hello, how are you\\\".\"") e++;
if (a[2] != "for me") e++;
if (length(a) != 2) e++;
⟨pca-test.awk:summary  59c⟩
```

10.5.3 Parentheses, Braces and Brackets

Where quotes are closed by the same character, parentheses, brackets and braces are closed by an alternate character.

50a ⟨mode:common-brackets[1](language, open, close), lang=⟩ ≡

```
modes[⟨language⟩, ⟨open⟩,  "submodes" ]="\\\\|\"|{|\\(|\\[|'|/\\*";
modes[⟨language⟩, ⟨open⟩,  "delimiters"]=" *, *";
modes[⟨language⟩, ⟨open⟩,  "terminators"]=⟨close⟩;
```

Note that the open is NOT a regex but the close token IS.
To do: When we can quote regex we won't have to put the slashes in here

50b ⟨common-mode-definitions[5](language) ⇑48a, lang=⟩ +≡ ◁49f

```
⟨mode:common-brackets(⟨language⟩, "{", "}") 50a⟩
⟨mode:common-brackets(⟨language⟩, "[", "\\]") 50a⟩
⟨mode:common-brackets(⟨language⟩, "(", "\\)") 50a⟩
```

10.5.4 Customizing Standard Modes

50c ⟨mode:add-submode[1](language, mode, submode), lang=⟩ ≡

```
modes[⟨language⟩, ⟨mode⟩, "submodes"] = modes[⟨language⟩, ⟨mode⟩, "submodes"] "|" ⟨submode⟩;
```

50d ⟨mode:add-escapes[1](language, mode, search, replace), lang=⟩ ≡

```
escapes[⟨language⟩, ⟨mode⟩, ++escapes[⟨language⟩, ⟨mode⟩], "s"]=⟨search⟩;
escapes[⟨language⟩, ⟨mode⟩,   escapes[⟨language⟩, ⟨mode⟩], "r"]=⟨replace⟩;
```

10.5.5 Comments

We can define `/* comment */` style comments and `//comment` style comments to be added to any language:

50e ⟨mode:multi-line-comments[1](language), lang=⟩ ≡

```
⟨mode:add-submode(⟨language⟩, "", "/\\*") 50c⟩
modes[⟨language⟩, "/*", "terminators"]="\\*/";
```

50f ⟨mode:single-line-slash-comments[1](language), lang=⟩ ≡

```
⟨mode:add-submode(⟨language⟩, "", "//") 50c⟩
modes[⟨language⟩, "//", "terminators"]="\n";
⟨mode:add-escapes(⟨language⟩, "//", "\n", "\n//") 50d⟩
```

We can also define `# comment` style comments (as used in awk and shell scripts) in a similar manner.

To do: I'm having to use # for hash and ¯extbackslash{} for and have hacky work-arounds in the parser for now

50g ⟨mode:add-hash-comments[1](language), lang=⟩ ≡

```
⟨mode:add-submode(⟨language⟩, "", "#") 50c⟩
modes[⟨language⟩, "#", "terminators"]="\n";
⟨mode:add-escapes(⟨language⟩, "#", "\n", "\n#") 50d⟩
```

In C, the # denotes pre-processor directives which can be multi-line

51a ⟨mode:add-hash-defines[1](language), lang=⟩ ≡

```
⟨mode:add-submode(⟨language⟩, "", "#") 50c⟩
modes[⟨language⟩, "#", "submodes" ]="\\\\";
modes[⟨language⟩, "#", "terminators"]="\n";
⟨mode:add-escapes(⟨language⟩, "#", "\n", "\\\\\n") 50d⟩
```

51b ⟨mode:quote-dollar-escape[1](language, quote), lang=⟩ ≡

```
escapes[⟨language⟩, ⟨quote⟩, ++escapes[⟨language⟩, ⟨quote⟩], "s"]="\\$";
escapes[⟨language⟩, ⟨quote⟩,   escapes[⟨language⟩, ⟨quote⟩], "r"]="\\$";
```

We can add these definitions to various languages

51c ⟨mode-definitions[1](), lang=⟩ ≡ 52b▷

```
⟨common-mode-definitions("c-like") 48a⟩

⟨common-mode-definitions("c") 48a⟩
⟨mode:multi-line-comments("c") 50e⟩
⟨mode:single-line-slash-comments("c") 50f⟩
⟨mode:add-hash-defines("c") 51a⟩

⟨common-mode-definitions("awk") 48a⟩
⟨mode:add-hash-comments("awk") 50g⟩
⟨mode:add-naked-regex("awk") 52a⟩
```

The awk definitions should allow a comment block like this:

51d ⟨test:comment-quote[1](), lang=awk⟩ ≡

```
# Comment: ⟨test:comment-text  51e⟩
```

51e ⟨test:comment-text[1](), lang=⟩ ≡

```
Now is the time for
the quick brown fox to bring lemonade
to the party
```

to come out like this:

51f ⟨test:comment-quote:result[1](), lang=⟩ ≡

```
# Comment: Now is the time for
#the quick brown fox to bring lemonade
#to the party
```

The C definition for such a block should have it come out like this:

51g ⟨test:comment-quote:C-result[1](), lang=⟩ ≡

```
# Comment: Now is the time for\
the quick brown fox to bring lemonade\
to the party
```

10.5.6 Regex

This pattern is incomplete, but meant to detect naked regular expressions in awk and perl; e.g. `/.*$/`, however required capabilities are not present.

Current it only detects regexes anchored with ^ as used in fangle.

For full regex support, modes need to be named not after their starting character, but some other more fully qualified name.

52a ⟨mode:add-naked-regex[1](language), lang=⟩ ≡

```
⟨mode:add-submode(⟨language⟩, "", "/\\^") 50c⟩
modes[⟨language⟩, "/^", "terminators"]="/";
```

10.5.7 Perl

52b ⟨mode-definitions[2]() ⇑51c, lang=⟩ +≡ ◁51c 52c▽

```
⟨common-mode-definitions("perl") 48a⟩
⟨mode:multi-line-comments("perl") 50e⟩
⟨mode:add-hash-comments("perl") 50g⟩
```

Still need to add add `s/`, submode `/`, terminate both with `//`. This is likely to be impossible as perl regexes can contain perl.

10.5.8 sh

Shell single-quote strings are different to other strings and have no escape characters. The only special character is the single quote `'` which always closes the string. Therefore we cannot use ⟨common-mode-definitions("sh") 48a⟩ but we will invoke most of it's definition apart from single-quote strings.

52c ⟨mode-definitions[3]() ⇑51c, lang=awk⟩ +≡ △52b 54a▷

```
modes["sh", "",  "submodes"]="\\\\|\"|'|{|\\(|\\[|\\$\\(";
modes["sh", "\\", "terminators"]=".";

modes["sh", "\"", "submodes"]="\\\\|\\$\\(";
modes["sh", "\"", "terminators"]="\"";
escapes["sh", "\"", ++escapes["sh", "\""], "s"]="\\\\";
escapes["sh", "\"",   escapes["sh", "\""], "r"]="\\\\";
escapes["sh", "\"", ++escapes["sh", "\""], "s"]="\"";
escapes["sh", "\"",   escapes["sh", "\""], "r"]="\\" "\"";
escapes["sh", "\"", ++escapes["sh", "\""], "s"]="\n";
escapes["sh", "\"",   escapes["sh", "\""], "r"]="\\n";

modes["sh", "'", "terminators"]="'";
escapes["sh", "'", ++escapes["sh", "'"], "s"]="'";
escapes["sh", "'",   escapes["sh", "'"], "r"]="'\\'" "'";
⟨mode:common-brackets("sh", "$(", "\\)") 50a⟩
⟨mode:add-tunnel("sh", "$(", "") 52d⟩
⟨mode:common-brackets("sh", "{", "}") 50a⟩
⟨mode:common-brackets("sh", "[", "\\]") 50a⟩
⟨mode:common-brackets("sh", "(", "\\)") 50a⟩
⟨mode:add-hash-comments("sh") 50g⟩
⟨mode:quote-dollar-escape("sh", "\"") 51b⟩
```

The definition of add-tunnel is:

52d ⟨mode:add-tunnel[1](language, mode, tunnel), lang=⟩ ≡

```
escapes[⟨language⟩, ⟨mode⟩, ++escapes[⟨language⟩, ⟨mode⟩], "tunnel"]=⟨tunnel⟩;
```

10.5.9 Make

For makefiles, we currently recognize 2 modes: the *null* mode and ↦ mode, which is tabbed mode and contains the makefile recipie. In the *null* mode the only escape is `$` which must be converted to `$$`.

Tabbed mode is harder to manage, as the GNU Make Manual says in the section on splitting lines[4]. There is no way to escape a multi-line text that occurs as part of a makefile recipe.

Despite this sad fact, if the newline's in the shell script all occur at points of top-level shell syntax, then we could replace them with `;\n↦` and largely get the right effect.

53a ⟨test:make:1[1](), lang=make⟩ ≡

```
all:
↦	echo making
↦	⟨test:make:1-inc($@) 53b⟩
```

53b ⟨test:make:1-inc[1](target), lang=sh⟩ ≡

```
if test "⟨target⟩" = "all"
then echo yes, all
else echo not all
fi
```

The two chunks about could reasonably produce this:

53c ⟨test:make:1.result[1](), lang=make⟩ ≡

```
all:
↦	echo making
↦	if test "$@" = "all" ;\
↦	then echo yes, all ;\
↦	else echo not all ;\
↦	fi
```

But will more likely produce this:

53d ⟨test:make:1.result-actual[1](), lang=make⟩ ≡

```
all:
↦	echo making
↦	if test "$$@" = "all" ;\
↦	 then echo yes, all ;\
↦	 else echo not all ;\
↦	 fi
```

The chunk argument `$@` has been quoted (which would have been fine if we were passing the name of a shell variable), and the other shell lines are (harmlessly) indented by 1 space as part of fangle indent-matching which should have taken into account the expanded tab size, and should generally take into account the expanded prefix of the line whose indent it is trying to match, but which in this case we want to have no effect at all!

To do: The $@ was passed from a make fragment. In what cases should it be converted to $$@? Do we need to track the language of sources of arguments?

A more ugly work-around until this problem can be solved would be to use this notation:

53e ⟨test:make:2[1](), lang=make⟩ ≡

```
all:
↦	echo making
↦	ARG="$@"; ⟨test:make:1-inc($ARG) 53b⟩
```

which produces this output which is more useful (because it works):

53f ⟨test:make:2.result[1](), lang=make⟩ ≡

```
all:
↦	echo making test
↦	ARG="$@"; if test "$$ARG" = "all" ;\
↦	            then echo yes, all ;\
↦	            else echo not all ;\
↦	            fi
```

4. http://www.gnu.org/s/hello/manual/make/Splitting-Lines.html

If, however, the shell fragment contained strings with literal newline characters then there would be no easy way to escape these and preserve the value of the string.

A different style of makefile construction might be used — the recipe could be stored in a target specific variable[5] which contains the recipe with a more normal escape mechanism.

54a ⟨mode-definitions[4]() ⇑51c, lang=awk⟩ +≡ ◁52c

```
modes["make", "",  "submodes"]="↦      ";
escapes["make", "", ++escapes["make", ""], "s"]="\\$";
escapes["make", "", escapes["make", ""], "r"]="$$";
modes["make", "↦      ", "terminators"]="\\n";
escapes["make", "↦      ", ++escapes["make", "↦      "], "s"]="\\n";
escapes["make", "↦      ", escapes["make", "↦      "], "r"]=" ;\\n↦      ";
```

Note also that the tab character is hard-wired into the pattern, and that the make variable `.RECIPEPREFIX` might change this to something else.

10.6 Some tests

Also, the parser must return any spare text at the end that has not been processed due to a mode terminator being found.

54b ⟨test:mode-definitions[3]() ⇑48c, lang=⟩ +≡ ◁49g 54c▽

```
rest = parse_chunk_args("c-like", "1, 2, 3) spare", a, "(");
if (a[1] != 1) e++;
if (a[2] != 2) e++;
if (a[3] != 3) e++;
if (length(a) != 3) e++;
if (rest != " spare") e++;
⟨pca-test.awk:summary  59c⟩
```

We must also be able to parse the example given earlier.

54c ⟨test:mode-definitions[4]() ⇑48c, lang=⟩ +≡ △54b

```
parse_chunk_args("c-like", "things[x, y], get_other_things(a, \"(all)\"), 99", a, "(");
if (a[1] != "things[x, y]") e++;
if (a[2] != "get_other_things(a, \"(all)\")") e++;
if (a[3] != "99") e++;
if (length(a) != 3) e++;
⟨pca-test.awk:summary  59c⟩
```

10.7 A non-recursive mode tracker

10.7.1 Constructor

The mode tracker holds its state in a stack based on a numerically indexed hash. This function, when passed an empty hash, will intialize it.

54d ⟨new_mode_tracker()[1](), lang=⟩ ≡

```
function new_mode_tracker(context, language, mode) {
  context[""] = 0;
```

5. http://www.gnu.org/s/hello/manual/make/Target_002dspecific.html

54d ⟨new_mode_tracker()[1](), lang=⟩ ≡

```
  context[0, "language"] = language;
  context[0, "mode"] = mode;
}
```

Because awk functions cannot return an array, we must create the array first and pass it in, so we have a fangle macro to do this:

55a ⟨new-mode-tracker[1](context, language, mode), lang=awk⟩ ≡

```
⟨awk-delete-array(context) 35d⟩
new_mode_tracker(⟨context⟩, ⟨language⟩, ⟨mode⟩);
```

10.7.2 Management

And for tracking modes, we dispatch to a mode-tracker action based on the current language

55b ⟨mode_tracker[1](), lang=awk⟩ ≡ 55c▽

```
function push_mode_tracker(context, language, mode,
  # local vars
  top)
{
  if (! ("" in context)) {
    ⟨new-mode-tracker(context, language, mode) 55a⟩
    return;
  } else {
    top = context[""];
    if (context[top, "language"] == language && mode=="") mode = context[top, "mode"];
    old_top = top;
    top++;
    context[top, "language"] = language;
    context[top, "mode"] = mode;
    context[""] = top;
  }
  return old_top;
}
```

55c ⟨mode_tracker[2]() ⇑55b, lang=⟩ +≡ △55b 55d▽

```
function dump_mode_tracker(context,
  c, d)
{
  for(c=0; c <= context[""]; c++) {
    printf(" %2d   %s:%s\n", c, context[c, "language"], context[c, "mode"]) > "/dev/stderr";
    for(d=1; ( (c, "values", d) in context); d++) {
      printf("   %2d %s\n", d, context[c, "values", d]) > "/dev/stderr";
    }
  }
}
```

55d ⟨mode_tracker[3]() ⇑55b, lang=⟩ +≡ △55c 60a▷

```
function pop_mode_tracker(context, context_origin)
{
  if ( (context_origin) && ("" in context) && context[""] != (1+context_origin)) return 0;
  context[""] = context_origin;
  return 1;
}
```

This implies that any chunk must be syntactically whole; for instance, this is fine:

55e ⟨test:whole-chunk[1](), lang=⟩ ≡

```
if (1) {
```

55e ⟨test:whole-chunk[1](), lang=⟩ ≡

```
  ⟨test:say-hello  56a⟩
}
```

56a ⟨test:say-hello[1](), lang=⟩ ≡

```
print "hello";
```

But this is not fine; the chunk ⟨test:hidden-else 56c⟩ is not properly cromulent.

56b ⟨test:partial-chunk[1](), lang=⟩ ≡

```
if (1) {
  ⟨test:hidden-else  56c⟩
}
```

56c ⟨test:hidden-else[1](), lang=⟩ ≡

```
  print "I'm fine";
} else {
  print "I'm not";
```

These tests will check for correct behaviour:

56d ⟨test:cromulence[1](), lang=⟩ ≡

```
echo Cromulence test
passtest $FANGLE -Rtest:whole-chunk $TXT_SRC &>/dev/null || ( echo "Whole chunk failed" && exit 1 )
failtest $FANGLE -Rtest:partial-chunk $TXT_SRC &>/dev/null || ( echo "Partial chunk failed" && exit 1
)
```

10.7.3 Tracker

We must avoid recursion as a language construct because we intend to employ mode-tracking to track language mode of emitted code, and the code is emitted from a function which is itself recursive, so instead we implement psuedo-recursion using our own stack based on a hash.

56e ⟨mode_tracker()[1](), lang=awk⟩ ≡ 56f▽

```
function mode_tracker(context, text, values,
  # optional parameters
  # local vars
  mode, submodes, language,
  cindex, c, a, part, item, name, result, new_values, new_mode,
  delimiters, terminators)
{
```

We could be re-commencing with a valid context, so we need to setup the state according to the last context.

56f ⟨mode_tracker()[2]() ⇑56e, lang=⟩ +≡ △56e 57b▷

```
  cindex = context[""] + 0;
  mode = context[cindex, "mode"];
  language = context[cindex, "language" ];
```

First we construct a single large regex combining the possible sub-modes for the current mode along with the terminators for the current mode.

56g ⟨parse_chunk_args-reset-modes[1](), lang=⟩ ≡ 57a▷

```
  submodes=modes[language, mode, "submodes"];

```

56g ⟨parse_chunk_args-reset-modes[1](), lang=⟩ ≡ 57a▷

```
  if ((language, mode, "delimiters") in modes) {
    delimiters = modes[language, mode, "delimiters"];
    if (length(submodes)>0) submodes = submodes "|";
    submodes=submodes delimiters;
  } else delimiters="";
  if ((language, mode, "terminators") in modes) {
    terminators = modes[language, mode, "terminators"];
    if (length(submodes)>0) submodes = submodes "|";
    submodes=submodes terminators;
  } else terminators="";
```

If we don't find anything to match on — probably because the language is not supported — then we return the entire text without matching anything.

57a ⟨parse_chunk_args-reset-modes[2]() ⇑56g, lang=⟩ +≡ ◁56g

```
 if (! length(submodes)) return text;
```

57b ⟨mode_tracker()[3]() ⇑56e, lang=⟩ +≡ ◁56f 57c▽

```
⟨parse_chunk_args-reset-modes  56g⟩
```

We then iterate the text (until there is none left) looking for sub-modes or terminators in the regex.

57c ⟨mode_tracker()[4]() ⇑56e, lang=⟩ +≡ △57b 57d▽

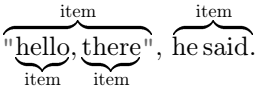
```
  while((cindex >= 0) && length(text)) {
    if (match(text, "(" submodes ")", a)) {
```

A bug that creeps in regularly during development is bad regexes of zero length which result in an infinite loop (as no text is consumed), so I catch that right away with this test.

57d ⟨mode_tracker()[5]() ⇑56e, lang=⟩ +≡ △57c 57e▽

```
      if (RLENGTH<1) {
        error(sprintf("Internal error, matched zero length submode, should be impossible - likely
regex computation error\n" \
                "Language=%s\nmode=%s\nmatch=%s\n", language, mode, submodes));
      }
```

part is defined as the text up to the sub-mode or terminator, and this is appended to item — which is the current text being gathered. If a mode has a delimiter, then item is reset each time a delimiter is found.



57e ⟨mode_tracker()[6]() ⇑56e, lang=⟩ +≡ △57d 57f▽

```
      part = substr(text, 1, RSTART -1);
      item = item part;
```

We must now determine what was matched. If it was a terminator, then we must restore the previous mode.

57f ⟨mode_tracker()[7]() ⇑56e, lang=⟩ +≡ △57e 58a▷

```
      if (match(a[1], "^" terminators "$")) {
#printf("%2d EXIT  MODE [%s] by [%s] [%s]\n", cindex, mode, a[1], text) > "/dev/stderr"
        context[cindex, "values", ++context[cindex, "values"]] = item;
        delete context[cindex];
        context[""] = --cindex;
        if (cindex>=0) {
          mode = context[cindex, "mode"];
```

57f ⟨mode_tracker()[7]() ⇑56e, lang=⟩ +≡ △57e 58a▷

```
27            language = context[cindex, "language"];
28            ⟨parse_chunk_args-reset-modes  56g⟩
29          }
30          item = item a[1];
31          text = substr(text, 1 + length(part) + length(a[1]));
32        }
```

If a delimiter was matched, then we must store the current item in the parsed values array, and reset the item.

58a ⟨mode_tracker()[8]() ⇑56e, lang=⟩ +≡ ◁57f 58b▽

```
33        else if (match(a[1], "^" delimiters "$")) {
34          if (cindex==0) {
35            context[cindex, "values", ++context[cindex, "values"]] = item;
36            item = "";
37          } else {
38            item = item a[1];
39          }
40          text = substr(text, 1 + length(part) + length(a[1]));
41        }
```

otherwise, if a new submode is detected (all submodes have terminators), we must create a nested parse context until we find the terminator for this mode.

58b ⟨mode_tracker()[9]() ⇑56e, lang=⟩ +≡ △58a 58c▽

```
42   else if ((language, a[1], "terminators") in modes) {
43          #check if new_mode is defined
44          item = item a[1];
45  #printf("%2d ENTER MODE [%s] in [%s]\n", cindex, a[1], text) > "/dev/stderr"
46          text = substr(text, 1 + length(part) + length(a[1]));
47          context[""] = ++cindex;
48          context[cindex, "mode"] = a[1];
49          context[cindex, "language"] = language;
50          mode = a[1];
51          ⟨parse_chunk_args-reset-modes  56g⟩
52        } else {
53          error(sprintf("Submode '%s' set unknown mode in text: %s\nLanguage %s Mode %s\n", a[1], text,
    language, mode));
54          text = substr(text, 1 + length(part) + length(a[1]));
55        }
56      }
```

In the final case, we parsed to the end of the string. If the string was entire, then we should have no nested mode context, but if the string was just a fragment we may have a mode context which must be preserved for the next fragment. Todo: Consideration ought to be given if sub-mode strings are split over two fragments.

58c ⟨mode_tracker()[10]() ⇑56e, lang=⟩ +≡ △58b

```
57  else {
58        context[cindex, "values", ++context[cindex, "values"]] = item text;
59        text = "";
60        item = "";
61      }
62    }
63
64    context["item"] = item;
65
66    if (length(item)) context[cindex, "values", ++context[cindex, "values"]] = item;
67    return text;
68  }
```

10.7.3.1 One happy chunk

All the mode tracker chunks are referred to here:

59a ⟨mode-tracker[1](), lang=⟩ ≡

```
⟨new_mode_tracker()  54d⟩
⟨mode_tracker()  56e⟩
```

10.7.3.2 Tests

We can test this function like this:

59b ⟨pca-test.awk[1](), lang=awk⟩ ≡

```
⟨error()  36a⟩
⟨mode-tracker  59a⟩
⟨parse_chunk_args()  ?⟩
BEGIN {
  SUBSEP=".";
  ⟨mode-definitions  51c⟩

  ⟨test:mode-definitions  48c⟩
}
```

59c ⟨pca-test.awk:summary[1](), lang=awk⟩ ≡

```
if (e) {
  printf "Failed " e
  for (b in a) {
    print "a[" b "] => " a[b];
  }
} else {
  print "Passed"
}
split("", a);
e=0;
```

which should give this output:

59d ⟨pca-test.awk-results[1](), lang=⟩ ≡

```
a[foo.quux.quirk] =>
a[foo.quux.a] => fleeg
a[foo.bar] => baz
a[etc] =>
a[name] => freddie
```

10.8 Escaping and Quoting

For the time being and to get around TEXMACS inability to export a `TAB` character, the right arrow ↦ whose UTF-8 sequence is ...

To do: complete

Another special character is used, the left-arrow ↤ with UTF-8 sequence 0xE2 0x86 0xA4 is used to strip any preceding white space as a way of un-tabbing and removing indent that has been applied — this is important for bash here documents, and the like. It's a filthy hack.

To do: remove the hack

60a ⟨mode_tracker[4]() ⇑55b, lang=⟩ +≡ ◁55d 60b▽

```
function untab(text) {
  gsub("[[:space:]]*\xE2\x86\xA4","", text);
  return text;
}
```

Each nested mode can optionally define a set of transforms to be applied to any text that is included from another language.

This code can perform transforms from index c downwards.

60b ⟨mode_tracker[5]() ⇑55b, lang=awk⟩ +≡ △60a 58c▷

```
function transform_escape(context, text, top,
  c, cp, cpl, s, r)
{
  for(c = top; c >= 0; c--) {
    if ( (context[c, "language"], context[c, "mode"]) in escapes) {
      cpl = escapes[context[c, "language"], context[c, "mode"]];
      for (cp = 1; cp <= cpl; cp ++) {
        s = escapes[context[c, "language"], context[c, "mode"], cp, "s"];
        r = escapes[context[c, "language"], context[c, "mode"], cp, "r"];
        if (length(s)) {
          gsub(s, r, text);
        }
        if ( (context[c, "language"], context[c, "mode"], cp, "t") in escapes ) {
          quotes[src, "t"] = escapes[context[c, "language"], context[c, "mode"], cp, "t"];
        }
      }
    }
  }
  return text;
}
function dump_escaper(quotes, r, cc) {
  for(cc=1; cc<=c; cc++) {
    printf("%2d s[%s] r[%s]\n", cc, quotes[cc, "s"], quotes[cc, "r"]) > "/dev/stderr"
  }
}
```

60c ⟨test:escapes[1](), lang=sh⟩ ≡

```
echo escapes test
passtest $FANGLE -Rtest:comment-quote $TXT_SRC &>/dev/null || ( echo "Comment-quote failed" && exit 1
)
```

Chapter 11
Recognizing Chunks

Fangle recognizes noweb chunks, but as we also want better LaTeX integration we will recognize any of these:

- notangle chunks matching the pattern `^<<.*?>>=`
- chunks beginning with `\begin{lstlistings}`, possibly with `\Chunk{...}` on the previous line
- an older form I have used, beginning with `\begin{Chunk}[options]` — also more suitable for plain LaTeX users[1].

11.1 Chunk start

The variable chunking is used to signify that we are processing a code chunk and not document. In such a state, input lines will be assigned to the current chunk; otherwise they are ignored.

11.1.1 TeXmacs

We don't handle TeXmacs files natively yet, but rather instead emit unicode character sequences to mark up the text-export file which we do process.

These hacks detect the unicode character sequences and retro-fit in the old TeX parsing.

We convert ↦ into a tab character.

61a ⟨recognize-chunk[1](), lang=⟩ ≡ 61b▽

```
#/\n/ {
#  gsub("\n*$","");
#  gsub("\n", " ");
#}
#===
/\xE2\x86\xA6/ {
  gsub("\\xE2\\x86\\xA6", "\x09");
}
```

TeXmacs back-tick handling is obscure, and a cut-n-paste back-tick from a shell window comes out as a unicode sequence[2] that is fixed-up here.

61b ⟨recognize-chunk[2]() ⇑61a, lang=⟩ +≡ △61a 62a▷

```

/\xE2\x80\x98/ {
```

1. Is there such a thing as plain LaTeX?
2. that won't export to html, except as a NULL character (literal 0x00)

61b ⟨recognize-chunk[2]() ⇑61a, lang=⟩ +≡ △61a 62a▷

```
  gsub("\\xE2\\x80\\x98", "‘");
}
```

In the TEXMACS output, the start of a chunk will appear like this:

```
5b<example-chunk^K[1](arg1,^K arg2^K^K), lang=C> ≡
```

We detect the the start of a TEXMACS chunk by detecting the ≡ symbol which occurs near the end of the line. We obtain the chunk name, the chunk parameters, and the chunk language.

62a ⟨recognize-chunk[3]() ⇑61a, lang=⟩ +≡ ◁61b 62b▽

```

/\xE2\x89\xA1/ {
  if (match($0, "^ *([^[ ]* |)<([^[ ]*)\\[[0-9]*\\][(](.*)[)].*, lang=([^ ]*)>", line)) {
    next_chunk_name=line[2];
    get_texmacs_chunk_args(line[3], next_chunk_params);
    gsub(ARG_SEPARATOR ",? ?", ";", line[3]);
    params = "params=" line[3];
    if ((line[4])) {
      params = params ",language=" line[4]
    }
    get_tex_chunk_args(params, next_chunk_opts);
    new_chunk(next_chunk_name, next_chunk_opts, next_chunk_params);
    texmacs_chunking = 1;
  } else {
    # warning(sprintf("Unexpected chunk match: %s\n", $_))
  }
  next;
}
```

11.1.2 lstlistings

Our current scheme is to recognize the new lstlisting chunks, but these may be preceded by a `\Chunk` command which in LYX is a more convenient way to pass the chunk name to the `\begin{lstlistings}` command, and a more visible way to specify other `lstset` settings.

The arguments to the `\Chunk` command are a name, and then a comma-seperated list of key-value pairs after the manner of `\lstset`. (In fact within the LATEX `\Chunk` macro (section 16.2.1) the text `name=` is prefixed to the argument which is then literally passed to `\lstset`).

62b ⟨recognize-chunk[4]() ⇑61a, lang=awk⟩ +≡ △62a 62c▽

```
/^\\Chunk{/ {
  if (match($0, "^\\\\Chunk{ *([^ ,}]*),?(.*)}", line)) {
    next_chunk_name = line[1];
    get_tex_chunk_args(line[2], next_chunk_opts);
  }
  next;
}
```

We also make a basic attempt to parse the name out of the `\lstlistings[name=chunk-name]` text, otherwise we fall back to the name found in the previous chunk command. This attempt is very basic and doesn't support commas or spaces or square brackets as part of the chunkname. We also recognize `\begin{Chunk}` which is convenient for some users[3].

62c ⟨recognize-chunk[5]() ⇑61a, lang=⟩ +≡ △62b 63a▷

```
/^\\begin{lstlisting}|^\\begin{Chunk}/ {
  if (match($0, "}.*[[,] *name= *{? *([^], }]*)", line)) {
    new_chunk(line[1]);
```

3. but not yet supported in the LATEX macros

62c ⟨recognize-chunk[5]() ⇑61a, lang=⟩ +≡ △62b 63a▷

```
  } else {
    new_chunk(next_chunk_name, next_chunk_opts);
  }
  chunking=1;
  next;
}
```

11.2 Chunk Body

11.2.1 TEXMACS

A chunk body in TEXMACS ends with `|_________...` if it is the final chunklet of a chunk, or if there are further chunklets it ends with `|\/\/\/...` which is a depiction of a jagged line of torn paper.

63a ⟨recognize-chunk[6]() ⇑61a, lang=⟩ +≡ ◁62c 63b▽

```
/^ *\|____________*/ && texmacs_chunking {
  active_chunk="";
  texmacs_chunking=0;
  chunking=0;
}
/^ *\|\/\\/ && texmacs_chunking {
  texmacs_chunking=0;
  chunking=0;
  active_chunk="";
}
```

It has been observed that not every line of output when a TEXMACS chunk is active is a line of chunk. This may no longer be true, but we set a variable `texmacs_chunk` if the current line is a chunk line.

Initially we set this to zero...

63b ⟨recognize-chunk[7]() ⇑61a, lang=⟩ +≡ △63a 63c▽

```
texmacs_chunk=0;
```

...and then we look to see if the current line is a chunk line.

TEXMACS lines look like this: `  3 | main() {` so we detect the lines by leading white space, digits, more whiter space and a vertical bar followed by at least once space.

If we find such a line, we remove this line-header and set `texmacs_chunk=1` as well as `chunking=1`

63c ⟨recognize-chunk[8]() ⇑61a, lang=⟩ +≡ △63b 63d▽

```
/^ *[1-9][0-9]* *\| / {
  if (texmacs_chunking) {
    chunking=1;
    texmacs_chunk=1;
    gsub("^ *[1-9][0-9]* *\\| ", "")
  }
}
```

When TEXMACS chunking, lines that commence with `\/` or `__` are not chunk content but visual framing, and are skipped.

63d ⟨recognize-chunk[9]() ⇑61a, lang=⟩ +≡ △63c 64a▷

```
/^ *\.\/\\/ && texmacs_chunking {
  next;
}
```

63d ⟨recognize-chunk[9]() ⇑61a, lang=⟩ +≡ △63c 64a▷

```
/^ *__*$/ && texmacs_chunking {
  next;
}
```

Any other line when TEXMACS chunking is considered to be a line-wrapped line.

64a ⟨recognize-chunk[10]() ⇑61a, lang=⟩ +≡ ◁63d 64b▽

```
texmacs_chunking {
  if (! texmacs_chunk) {
    # must be a texmacs continued line
    chunking=1;
    texmacs_chunk=1;
  }
}
```

This final chunklet seems bogus and probably stops LyX working.

64b ⟨recognize-chunk[11]() ⇑61a, lang=⟩ +≡ △64a 64c▽

```
! texmacs_chunk {
#  texmacs_chunking=0;
  chunking=0;
}
```

11.2.2 Noweb

We recognize notangle style chunks too:

64c ⟨recognize-chunk[12]() ⇑61a, lang=awk⟩ +≡ △64b 64d▽

```
/^[<]<.*[>]>=/ {
  if (match($0, "^[<]<(.*)[>]>= *$", line)) {
    chunking=1;
    notangle_mode=1;
    new_chunk(line[1]);
    next;
  }
}
```

11.3 Chunk end

Likewise, we need to recognize when a chunk ends.

11.3.1 lstlistings

The `e` in `[e]nd{lislisting}` is surrounded by square brackets so that when this document is processed, this chunk doesn't terminate early when the lstlistings package recognizes it's own end-string![4]

64d ⟨recognize-chunk[13]() ⇑61a, lang=⟩ +≡ △64c 65a▷

```
/^\\[e]nd{lstlisting}|^\\[e]nd{Chunk}/ {
  chunking=0;
  active_chunk="";
```

4. This doesn't make sense as the regex is anchored with ^, which this line does not begin with!

64d ⟨recognize-chunk[13]() ⇑61a, lang=⟩ +≡ △64c 65a▷

```
93   next;
94 }
```

11.3.2 noweb

65a ⟨recognize-chunk[14]() ⇑61a, lang=⟩ +≡ ◁64d 65b▽

```
95 /^@ *$/ {
96   chunking=0;
97   active_chunk="";
98 }
```

All other recognizers are only of effect if we are chunking; there's no point in looking at lines if they aren't part of a chunk, so we just ignore them as efficiently as we can.

65b ⟨recognize-chunk[15]() ⇑61a, lang=⟩ +≡ △65a 65c▽

```
99 ! chunking { next; }
```

11.4 Chunk contents

Chunk contents are any lines read while `chunking` is true. Some chunk contents are special in that they refer to other chunks, and will be replaced by the contents of these chunks when the file is generated.

We add the output record separator `ORS` to the line now, because we will set `ORS` to the empty string when we generate the output[5].

65c ⟨recognize-chunk[16]() ⇑61a, lang=⟩ +≡ △65b

```
100 length(active_chunk) {
101   ⟨process-chunk-tabs  65e⟩
102   ⟨process-chunk  66b⟩
103 }
```

If a chunk just consisted of plain text, we could handle the chunk like this:

65d ⟨process-chunk-simple[1](), lang=⟩ ≡

```
1 chunk_line(active_chunk, $0 ORS);
```

but in fact a chunk can include references to other chunks. Chunk includes are traditionally written as `<<chunk-name>>` but we support other variations, some of which are more suitable for particular editing systems.

However, we also process tabs at this point. A tab at input can be replaced by a number of spaces defined by the `tabs` variable, set by the `-T` option. Of course this is poor tab behaviour, we should probably have the option to use proper counted tab-stops and process this on output.

65e ⟨process-chunk-tabs[1](), lang=⟩ ≡

```
1 if (length(tabs)) {
2   gsub("\t", tabs);
3 }
```

5. So that we can partial print lines using `print` instead of `printf`. To do: This does't make sense

11.4.1 lstlistings

If `\lstset{escapeinside={=<}{>}}` is set, then we can use ⟨chunk-name ?⟩ in listings. The sequence `=<` was chosen because:

1. it is a better mnemonic than `<<chunk-name>>` in that the `=` sign signifies equivalence or substitutability.

2. and because `=<` is not valid in C or any language I can think of.

3. and also because lstlistings doesn't like `>>` as an end delimiter for the *texcl* escape, so we must make do with a single `>` which is better complemented by `=<` than by `<<`.

Unfortunately the `=<...>` that we use re-enters a LaTeX parsing mode in which some characters are special, e.g. `# \` and so these cause trouble if used in arguments to `\chunkref`. At some point I must fix the LaTeX command `\chunkref` so that it can accept these literally, but until then, when writing chunkref argumemts that need these characters, I must use the forms `\textbackslash{}` and `\#`; so I also define a hacky chunk `delatex` to be used further on whose purpose it is to remove these from any arguments parsed by fangle.

66a ⟨delatex[1](text), lang=⟩ ≡

```
# FILTHY HACK
gsub("\\\\#", "#", ${text});
gsub("\\\\textbackslash{}", "\\", ${text});
gsub("\\\\\\^", "^", ${text});
```

As each chunk line may contain more than one chunk include, we will split out chunk includes in an iterative fashion[6].

First, as long as the chunk contains a `\chunkref` command we take as much as we can up to the first `\chunkref` command.

TEXMACS text output uses ⟨...⟩ which comes out as unicode sequences `0xC2 0xAB` ... `0xC2 0xBB`. Modern awk will interpret `[^\xC2\xBB]` as a single unicode character if `LANG` is set correctly to the sub-type `UTF-8`, e.g. `LANG=en_GB.UTF-8`, otherwise `[^\xC2\xBB]` will be treated as a two character negated match — but this should not interfere with the function.

66b ⟨process-chunk[1](), lang=⟩ ≡ 66c▽

```
chunk = $0;
indent = 0;
while(match(chunk,"(\xC2\xAB)([^\xC2\xBB]*) [^\xC2\xBB]*\xC2\xBB", line) ||
      match(chunk,
            "([=]<\\\\chunkref{([^}>]*)}(\\(.*\\)|)>|<<([a-zA-Z_][-a-zA-Z0-9_]*)>>)",
            line)\
) {
  chunklet = substr(chunk, 1, RSTART - 1);
```

We keep track of the indent count, by counting the number of literal characters found. We can then preserve this indent on each output line when multi-line chunks are expanded.

We then process this first part literal text, and set the chunk which is still to be processed to be the text after the `\chunkref` command, which we will process next as we continue around the loop.

66c ⟨process-chunk[2]() ⇑66b, lang=⟩ +≡ △66b 67a▷

```
  indent += length(chunklet);
  chunk_line(active_chunk, chunklet);
  chunk = substr(chunk, RSTART + RLENGTH);
```

6. Contrary to our use of split when substituting parameters in chapter ?

We then consider the type of chunk command we have found, whether it is the fangle style command beginning with `=<` the older notangle style beginning with `<<`.

Fangle chunks may have parameters contained within square brackets. These will be matched in `line[3]` and are considered at this stage of processing to be part of the name of the chunk to be included.

67a ⟨process-chunk[3]() ⇑66b, lang=⟩ +≡ ◁66c 67b▽

```
12  if (substr(line[1], 1, 1) == "=") {
13    # chunk name up to }
14        ⟨delatex(line[3]) 66a⟩
15    chunk_include(active_chunk, line[2] line[3], indent);
16  } else if (substr(line[1], 1, 1) == "<") {
17    chunk_include(active_chunk, line[4], indent);
18  } else if (line[1] == "\xC2\xAB") {
19    chunk_include(active_chunk, line[2], indent);
20  } else {
21    error("Unknown chunk fragment: " line[1]);
22  }
```

The loop will continue until there are no more chunkref statements in the text, at which point we process the final part of the chunk.

67b ⟨process-chunk[4]() ⇑66b, lang=⟩ +≡ △67a 67c▽

```
23 }
24 chunk_line(active_chunk, chunk);
```

We add the newline character as a chunklet on it's own, to make it easier to detect new lines and thus manage indentation when processing the output.

67c ⟨process-chunk[5]() ⇑66b, lang=⟩ +≡ △67b

```
25 chunk_line(active_chunk, "\n");
```

We will also permit a chunk-part number to follow in square brackets, so that ⟨`chunk-name[1]` ?⟩ will refer to the first part only. This can make it easy to include a C function prototype in a header file, if the first part of the chunk is just the function prototype without the trailing semi-colon. The header file would include the prototype with the trailing semi-colon, like this:

⟨`chunk-name[1]` ?⟩

This is handled in section 13.1.1

We should perhaps introduce a notion of language specific chunk options; so that perhaps we could specify:

`=<\chunkref{chunk-name[function-declaration]}`

which applies a transform `function-declaration` to the chunk — which in this case would extract a function prototype from a function. To do: Do it

Chapter 12
Processing Options

At the start, first we set the default options.

69a ⟨default-options[1](), lang=⟩ ≡

```
debug=0;
linenos=0;
notangle_mode=0;
root="*";
tabs = "";
```

Then we use getopt the standard way, and null out ARGV afterwards in the normal AWK fashion.

69b ⟨read-options[1](), lang=⟩ ≡

```
Optind = 1    # skip ARGV[0]
while(getopt(ARGC, ARGV, "R:LdT:hr")!=-1) {
  ⟨handle-options 69c⟩
}
for (i=1; i<Optind; i++) { ARGV[i]=""; }
```

This is how we handle our options:

69c ⟨handle-options[1](), lang=⟩ ≡

```
if (Optopt == "R") root = Optarg;
else if (Optopt == "r") root="";
else if (Optopt == "L") linenos = 1;
else if (Optopt == "d") debug = 1;
else if (Optopt == "T") tabs = indent_string(Optarg+0);
else if (Optopt == "h") help();
else if (Optopt == "?") help();
```

We do all of this at the beginning of the program

69d ⟨begin[1](), lang=⟩ ≡

```
BEGIN {
  ⟨constants 37a⟩
  ⟨mode-definitions 51c⟩
  ⟨default-options 69a⟩

  ⟨read-options 69b⟩
}
```

And have a simple help function

69e ⟨help()[1](), lang=⟩ ≡

```
function help() {
  print "Usage:"
  print "  fangle [-L] -R<rootname> [source.tex ...]"
  print "  fangle -r [source.tex ...]"
  print "  If the filename, source.tex is not specified then stdin is used"
  print
  print "-L causes the C statement: #line <lineno> \"filename\"" to be issued"
  print "-R causes the named root to be written to stdout"
  print "-r lists all roots in the file (even those used elsewhere)"
  exit 1;
}
```

Chapter 13
Generating the Output

We generate output by calling output_chunk, or listing the chunk names.

71a ⟨generate-output[1](), lang=⟩ ≡

```
if (length(root)) output_chunk(root);
else output_chunk_names();
```

We also have some other output debugging:

71b ⟨debug-output[1](), lang=⟩ ≡

```
if (debug) {
  print "------ chunk names "
  output_chunk_names();
  print "====== chunks"
  output_chunks();
  print "++++++ debug"
  for (a in chunks) {
    print a "=" chunks[a];
  }
}
```

We do both of these at the end. We also set `ORS=""` because each chunklet is not necessarily a complete line, and we already added `ORS` to each input line in section 11.4.

71c ⟨end[1](), lang=⟩ ≡

```
END {
  ⟨debug-output  71b⟩
  ORS="";
  ⟨generate-output  71a⟩
}
```

We write chunk names like this. If we seem to be running in notangle compatibility mode, then we enclose the name like this `<<name>>` the same way notangle does:

71d ⟨output_chunk_names()[1](), lang=⟩ ≡

```
function output_chunk_names(   c, prefix, suffix)
{
  if (notangle_mode) {
    prefix="<<";
    suffix=">>";
  }
  for (c in chunk_names) {
    print prefix c suffix "\n";
  }
}
```

This function would write out all chunks

71e ⟨output_chunks()[1](), lang=⟩ ≡

```
function output_chunks(  a)
```

71e ⟨output_chunks()[1](), lang=⟩ ≡

```
{
  for (a in chunk_names) {
    output_chunk(a);
  }
}

function output_chunk(chunk) {
  newline = 1;
  lineno_needed = linenos;

  write_chunk(chunk);
}

```

13.1 Assembling the Chunks

`chunk_path` holds a string consisting of the names of all the chunks that resulted in this chunk being output. It should probably also contain the source line numbers at which each inclusion also occured.

We first initialize the mode tracker for this chunk.

72a ⟨write_chunk()[1](), lang=awk⟩ ≡ 72b▽

```
function write_chunk(chunk_name) {
  ⟨awk-delete-array(context) 35d⟩
  return write_chunk_r(chunk_name, context);
}

function write_chunk_r(chunk_name, context, indent, tail,
  # optional vars
  chunk_path, chunk_args,
  # local vars
  context_origin,
  chunk_params, part, max_part, part_line, frag, max_frag, text,
  chunklet, only_part, call_chunk_args, new_context)
{
  if (debug) debug_log("write_chunk_r(" chunk_name ")");
```

13.1.1 Chunk Parts

As mentioned in section ?, a chunk name may contain a part specifier in square brackets, limiting the parts that should be emitted.

72b ⟨write_chunk()[2]() ⇑72a, lang=⟩ +≡ △72a 72c▽

```
  if (match(chunk_name, "^(.*)\\[([0-9]*)\\]$", chunk_name_parts)) {
    chunk_name = chunk_name_parts[1];
    only_part = chunk_name_parts[2];
  }
```

We then create a mode tracker

72c ⟨write_chunk()[3]() ⇑72a, lang=⟩ +≡ △72b 73a▷

```
  context_origin = push_mode_tracker(context, chunks[chunk_name, "language"], "");
```

We extract into `chunk_params` the names of the parameters that this chunk accepts, whose values were (optionally) passed in `chunk_args`.

73a ⟨write_chunk()[4]() ⇑72a, lang=⟩ +≡ ◁72c 73b▽

```
  split(chunks[chunk_name, "params"], chunk_params, " *; *");
```

To assemble a chunk, we write out each part.

73b ⟨write_chunk()[5]() ⇑72a, lang=⟩ +≡ △73a

```
  if (! (chunk_name in chunk_names)) {
    error(sprintf(_"The root module <<%s>> was not defined.\nUsed by: %s",\
                  chunk_name, chunk_path));
  }

  max_part = chunks[chunk_name, "part"];
  for(part = 1; part <= max_part; part++) {
    if (! only_part || part == only_part) {
      ⟨write-part  73c⟩
    }
  }
  if (! pop_mode_tracker(context, context_origin)) {
    dump_mode_tracker(context);
    error(sprintf(_"Module %s did not close context properly.\nUsed by: %s\n", chunk_name,
chunk_path));
  }
}
```

A part can either be a chunklet of lines, or an include of another chunk.

Chunks may also have parameters, specified in LaTeX style with braces after the chunk name — looking like this in the document: chunkname{param1, param2}. Arguments are passed in square brackets: `\chunkref{chunkname}[arg1, arg2]`.

Before we process each part, we check that the source position hasn't changed unexpectedly, so that we can know if we need to output a new file-line directive.

73c ⟨write-part[1](), lang=⟩ ≡

```
⟨check-source-jump  75d⟩

chunklet = chunks[chunk_name, "part", part];
if (chunks[chunk_name, "part", part, "type"] == part_type_chunk) {
  ⟨write-included-chunk  73d⟩
} else if (chunklet SUBSEP "line" in chunks) {
  ⟨write-chunklets  74a⟩
} else {
  # empty last chunklet
}
```

To write an included chunk, we must detect any optional chunk arguments in parenthesis. Then we recurse calling `write_chunk()`.

73d ⟨write-included-chunk[1](), lang=⟩ ≡

```
if (match(chunklet, "^([^\\[\\(]*)\\((.*)\\)$", chunklet_parts)) {
  chunklet = chunklet_parts[1];
# hack
gsub(sprintf("%c",11), "", chunklet);
gsub(sprintf("%c",11), "", chunklet_parts[2]);
  parse_chunk_args("c-like", chunklet_parts[2], call_chunk_args, "(");
  for (c in call_chunk_args) {
    call_chunk_args[c] = expand_chunk_args(call_chunk_args[c], chunk_params, chunk_args);
  }
} else {
  split("", call_chunk_args);
}

write_chunk_r(chunklet, context,
```

73d ⟨write-included-chunk[1](), lang=⟩ ≡

```
            chunks[chunk_name, "part", part, "indent"] indent,
            chunks[chunk_name, "part", part, "tail"],
            chunk_path "\n          " chunk_name,
            call_chunk_args);
```

Before we output a chunklet of lines, we first emit the file and line number if we have one, and if it is safe to do so.

Chunklets are generally broken up by includes, so the start of a chunklet is a good place to do this. Then we output each line of the chunklet.

When it is not safe, such as in the middle of a multi-line macro definition, `lineno_suppressed` is set to true, and in such a case we note that we want to emit the line statement when it is next safe.

74a ⟨write-chunklets[1](), lang=⟩ ≡ 74b▽

```
max_frag = chunks[chunklet, "line"];
for(frag = 1; frag <= max_frag; frag++) {
  ⟨write-file-line  75c⟩
```

We then extract the chunklet text and expand any arguments.

74b ⟨write-chunklets[2]() ⇑74a, lang=⟩ +≡ △74a 74c▽

```

  text = chunks[chunklet, frag];

  /* check params */
  text = expand_chunk_args(text, chunk_params, chunk_args);
```

If the text is a single newline (which we keep separate - see 6) then we increment the line number. In the case where this is the last line of a chunk and it is not a top-level chunk we replace the newline with an empty string — because the chunk that included this chunk will have the newline at the end of the line that included this chunk.

We also note by `newline = 1` that we have started a new line, so that indentation can be managed with the following piece of text.

74c ⟨write-chunklets[3]() ⇑74a, lang=⟩ +≡ △74b 74d▽

```

 if (text == "\n") {
    lineno++;
    if (part == max_part && frag == max_frag && length(chunk_path)) {
      text = "";
      break;
    } else {
      newline = 1;
    }
```

If this text does not represent a newline, but we see that we are the first piece of text on a newline, then we prefix our text with the current indent.

Note 1. `newline` is a global output-state variable, but the `indent` is not.

74d ⟨write-chunklets[4]() ⇑74a, lang=⟩ +≡ △74c 75a▷

```
  } else if (length(text) || length(tail)) {
    if (newline) text = indent text;
    newline = 0;
  }

```

Tail will soon no longer be relevant once mode-detection is in place.

75a ⟨write-chunklets[5]() ⇑74a, lang=⟩ +≡ ◁74d 75b▽

```
  text = text tail;
  mode_tracker(context, text);
  print untab(transform_escape(context, text, context_origin));
```

If a line ends in a backslash — suggesting continuation — then we supress outputting file-line as it would probably break the continued lines.

75b ⟨write-chunklets[6]() ⇑74a, lang=⟩ +≡ △75a

```
  if (linenos) {
    lineno_suppressed = substr(lastline, length(lastline)) == "\\";
  }
}
```

Of course there is no point in actually outputting the source filename and line number (file-line) if they don't say anything new! We only need to emit them if they aren't what is expected, or if we we not able to emit one when they had changed.

75c ⟨write-file-line[1](), lang=⟩ ≡

```
if (newline && lineno_needed && ! lineno_suppressed) {
  filename = a_filename;
  lineno = a_lineno;
  print "#line " lineno " \"" filename "\"\n"
  lineno_needed = 0;
}
```

We check if a new file-line is needed by checking if the source line matches what we (or a compiler) would expect.

75d ⟨check-source-jump[1](), lang=⟩ ≡

```
if (linenos && (chunk_name SUBSEP "part" SUBSEP part SUBSEP "FILENAME" in chunks)) {
  a_filename = chunks[chunk_name, "part", part, "FILENAME"];
  a_lineno = chunks[chunk_name, "part", part, "LINENO"];
  if (a_filename != filename || a_lineno != lineno) {
    lineno_needed++;
  }
}
```

Chapter 14
Storing Chunks

Awk has pretty limited data structures, so we will use two main hashes. Uninterrupted sequences of a chunk will be stored in chunklets and the chunklets used in a chunk will be stored in `chunks`.

77a ⟨constants[2]() ⇑37a, lang=⟩ +≡ ◁37a

```
part_type_chunk=1;
SUBSEP=",";
```

The params mentioned are not chunk parameters for parameterized chunks, as mentioned in 9.2, but the lstlistings style parameters used in the `\Chunk` command[1].

77b ⟨chunk-storage-functions[1](), lang=⟩ ≡ 77c▽

```
function new_chunk(chunk_name, opts, args,
  # local vars
  p, append )
{
  # HACK WHILE WE CHANGE TO ( ) for PARAM CHUNKS
  gsub("\\(\\)$", "", chunk_name);
  if (! (chunk_name in chunk_names)) {
    if (debug) print "New chunk " chunk_name;
    chunk_names[chunk_name];
    for (p in opts) {
      chunks[chunk_name, p] = opts[p];
      if (debug) print "chunks[" chunk_name "," p "] = " opts[p];
    }
    for (p in args) {
      chunks[chunk_name, "params", p] = args[p];
    }
    if ("append" in opts) {
      append=opts["append"];
      if (! (append in chunk_names)) {
        warning("Chunk " chunk_name " is appended to chunk " append " which is not defined yet");
        new_chunk(append);
      }
      chunk_include(append, chunk_name);
      chunk_line(append, ORS);
    }
  }
  active_chunk = chunk_name;
  prime_chunk(chunk_name);
}
```

77c ⟨chunk-storage-functions[2]() ⇑77b, lang=⟩ +≡ △77b 78a▷

```

function prime_chunk(chunk_name)
{
  chunks[chunk_name, "part", ++chunks[chunk_name, "part"] ] = \
         chunk_name SUBSEP "chunklet" SUBSEP "" ++chunks[chunk_name, "chunklet"];
  chunks[chunk_name, "part", chunks[chunk_name, "part"], "FILENAME"] = FILENAME;
  chunks[chunk_name, "part", chunks[chunk_name, "part"], "LINENO"] = FNR + 1;
```

1. The `params` parameter is used to hold the parameters for parameterized chunks

77c ⟨chunk-storage-functions[2]() ⇑77b, lang=⟩ +≡ △77b 78a▷

```
}

function chunk_line(chunk_name, line){
  chunks[chunk_name, "chunklet", chunks[chunk_name, "chunklet"],
         ++chunks[chunk_name, "chunklet", chunks[chunk_name, "chunklet"], "line"]  ] = line;
}

```

Chunk include represents a *chunkref* statement, and stores the requirement to include another chunk. The parameter indent represents the quanity of literal text characters that preceded this *chunkref* statement and therefore by how much additional lines of the included chunk should be indented.

78a ⟨chunk-storage-functions[3]() ⇑77b, lang=⟩ +≡ ◁77c 78b▽

```
function chunk_include(chunk_name, chunk_ref, indent, tail)
{
  chunks[chunk_name, "part", ++chunks[chunk_name, "part"] ] = chunk_ref;
  chunks[chunk_name, "part", chunks[chunk_name, "part"], "type" ] = part_type_chunk;
  chunks[chunk_name, "part", chunks[chunk_name, "part"], "indent" ] = indent_string(indent);
  chunks[chunk_name, "part", chunks[chunk_name, "part"], "tail" ] = tail;
  prime_chunk(chunk_name);
}

```

The indent is calculated by indent_string, which may in future convert some spaces into tab characters. This function works by generating a printf padded format string, like `%22s` for an indent of 22, and then printing an empty string using that format.

78b ⟨chunk-storage-functions[4]() ⇑77b, lang=⟩ +≡ △78a

```
function indent_string(indent) {
  return sprintf("%" indent "s", "");
}
```

Chapter 15
getopt

I use Arnold Robbins public domain getopt (1993 revision). This is probably the same one that is covered in chapter 12 of âĂIJEdition 3 of GAWK: Effective AWK Programming: A User's Guide for GNU AwkâĂİ but as that is licensed under the GNU Free Documentation License, Version 1.3, which conflicts with the GPL3, I can't use it from there (or it's accompanying explanations), so I do my best to explain how it works here.

The getopt.awk header is:

79a ⟨getopt.awk-header[1](), lang=⟩ ≡

```
# getopt.awk --- do C library getopt(3) function in awk
#
# Arnold Robbins, arnold@skeeve.com, Public Domain
#
# Initial version: March, 1991
# Revised: May, 1993

```

The provided explanation is:

79b ⟨getopt.awk-notes[1](), lang=⟩ ≡

```
# External variables:
#    Optind -- index in ARGV of first nonoption argument
#    Optarg -- string value of argument to current option
#    Opterr -- if nonzero, print our own diagnostic
#    Optopt -- current option letter

# Returns:
#    -1     at end of options
#    ?      for unrecognized option
#    <c>    a character representing the current option

# Private Data:
#    _opti  -- index in multi-flag option, e.g., -abc

```

The function follows. The final two parameters, `thisopt` and `i` are local variables and not parameters — as indicated by the multiple spaces preceding them. Awk doesn't care, the multiple spaces are a convention to help us humans.

79c ⟨getopt.awk-getopt()[1](), lang=⟩ ≡ 80a▷

```
function getopt(argc, argv, options,    thisopt, i)
{
    if (length(options) == 0)    # no options given
        return -1
    if (argv[Optind] == "--") {  # all done
        Optind++
        _opti = 0
        return -1
    } else if (argv[Optind] !~ /^-[^: \t\n\f\r\v\b]/) {
        _opti = 0
        return -1
```

```
    }
    if (_opti == 0)
        _opti = 2
    thisopt = substr(argv[Optind], _opti, 1)
    Optopt = thisopt
    i = index(options, thisopt)
    if (i == 0) {
        if (Opterr)
            printf("%c -- invalid option\n",
                                thisopt) > "/dev/stderr"
        if (_opti >= length(argv[Optind])) {
            Optind++
            _opti = 0
        } else
            _opti++
        return "?"
    }
```

At this point, the option has been found and we need to know if it takes any arguments.


```
    if (substr(options, i + 1, 1) == ":") {
        # get option argument
        if (length(substr(argv[Optind], _opti + 1)) > 0)
            Optarg = substr(argv[Optind], _opti + 1)
        else
            Optarg = argv[++Optind]
        _opti = 0
    } else
        Optarg = ""
    if (_opti == 0 || _opti >= length(argv[Optind])) {
        Optind++
        _opti = 0
    } else
        _opti++
    return thisopt
}
```

A test program is built in, too


```
BEGIN {
    Opterr = 1    # default is to diagnose
    Optind = 1    # skip ARGV[0]
    # test program
    if (_getopt_test) {
        while ((_go_c = getopt(ARGC, ARGV, "ab:cd")) != -1)
            printf("c = <%c>, optarg = <%s>\n",
                                    _go_c, Optarg)
        printf("non-option arguments:\n")
        for (; Optind < ARGC; Optind++)
            printf("\tARGV[%d] = <%s>\n",
                                Optind, ARGV[Optind])
    }
}
```

The entire getopt.awk is made out of these chunks in order


```
⟨getopt.awk-header  79a⟩

⟨getopt.awk-notes  79b⟩
⟨getopt.awk-getopt()  79c⟩
```

80c ⟨getopt.awk[1](), lang=⟩ ≡

```
5 ⟨getopt.awk-begin  80b⟩
```

Although we only want the header and function:

81a ⟨getopt[1](), lang=⟩ ≡

```
1 # try: locate getopt.awk for the full original file
2 # as part of your standard awk installation
3 ⟨getopt.awk-header  79a⟩
4
5 ⟨getopt.awk-getopt()  79c⟩
```

Chapter 16
Fangle LaTeX source code

16.1 fangle module

Here we define a LyX `.module` file that makes it convenient to use LyX for writing such literate programs.

This file `./fangle.module` can be installed in your personal `.lyx/layouts` folder. You will need to Tools Reconfigure so that LyX notices it. It adds a new format Chunk, which should precede every listing and contain the chunk name.

83a ⟨./fangle.module[1](), lang=lyx-module⟩ ≡

```
#\DeclareLyXModule{Fangle Literate Listings}
#DescriptionBegin
#  Fangle literate listings allow one to write
#   literate programs after the fashion of noweb, but without having
#   to use noweave to generate the documentation. Instead the listings
#   package is extended in conjunction with the noweb package to implement
#   to code formating directly as latex.
#  The fangle awk script
#DescriptionEnd

⟨gpl3-copyright.hashed  83b⟩

Format 11

AddToPreamble
⟨./fangle.sty  84d⟩
EndPreamble

⟨chunkstyle  84a⟩

⟨chunkref  84c⟩
```

Because LyX modules are not yet a language supported by fangle or lstlistings, we resort to this fake awk chunk below in order to have each line of the GPL3 license commence with a #

83b ⟨gpl3-copyright.hashed[1](), lang=awk⟩ ≡

```
#⟨gpl3-copyright  4a⟩

```

16.1.1 The Chunk style

The purpose of the CHUNK style is to make it easier for LyX users to provide the name to `lstlistings`. Normally this requires right-clicking on the listing, choosing settings, advanced, and then typing `name=chunk-name`. This has the further disadvantage that the name (and other options) are not generally visible during document editing.

The chunk style is defined as a LaTeX command, so that all text on the same line is passed to the `LaTeX` command `Chunk`. This makes it easy to parse using `fangle`, and easy to pass these options on to the listings package. The first word in a chunk section should be the chunk name, and will have `name=` prepended to it. Any other words are accepted arguments to `lstset`.

We set PassThru to 1 because the user is actually entering raw latex.

84a ⟨chunkstyle[1](), lang=⟩ ≡ 84b▽

```
Style Chunk
  LatexType             Command
  LatexName             Chunk
  Margin                First_Dynamic
  LeftMargin            Chunk:xxx
  LabelSep              xx
  LabelType             Static
  LabelString           "Chunk:"
  Align                 Left
  PassThru              1

```

To make the label very visible we choose a larger font coloured red.

84b ⟨chunkstyle[2]() ⇑84a, lang=⟩ +≡ △84a

```
  LabelFont
    Family              Sans
    Size                Large
    Series              Bold
    Shape               Italic
    Color               red
  EndFont
End
```

16.1.2 The chunkref style

We also define the Chunkref style which can be used to express cross references to chunks.

84c ⟨chunkref[1](), lang=⟩ ≡

```
InsetLayout Chunkref
  LyxType               charstyle
  LatexType             Command
  LatexName             chunkref
  PassThru              1
  LabelFont
    Shape               Italic
    Color               red
  EndFont
End
```

16.2 Latex Macros

We require the listings, noweb and xargs packages. As noweb defines it's own `\code` environment, we re-define the one that LyX logical markup module expects here.

84d ⟨./fangle.sty[1](), lang=tex⟩ ≡ 85a▷

```
\usepackage{listings}%
\usepackage{noweb}%
```

84d ⟨./fangle.sty[1]() lang=tex⟩ ≡ 85a▷

```
\usepackage{xargs}%
\renewcommand{\code}[1]{\texttt{#1}}%
```

We also define a `CChunk` macro, for use as: `\begin{CChunk}` which will need renaming to `\begin{Chunk}` when I can do this without clashing with `\Chunk`.

85a ⟨./fangle.sty[2]() ⇑84d lang=⟩ +≡ ◁84d 85b▽

```
\lstnewenvironment{Chunk}{\relax}{\relax}%
```

We also define a suitable `\lstset` of parameters that suit the literate programming style after the fashion of NOWEAVE.

85b ⟨./fangle.sty[3]() ⇑84d lang=⟩ +≡ △85a 85c▽

```
\lstset{numbers=left, stepnumber=5, numbersep=5pt,
        breaklines=false,basicstyle=\ttfamily,
        numberstyle=\tiny, language=C}%
```

We also define a notangle-like mechanism for escaping to LaTeX from the listing, and by which we can refer to other listings. We declare the `=<...>` sequence to contain LaTeX code, and include another like this chunk: ⟨`chunkname` ?⟩. However, because `=<...>` is already defined to contain LaTeX code for this document — this is a fangle document after all — the code fragment below effectively contains the LaTeX code: `}{`. To avoid problems with document generation, I had to declare an lstlistings property: `escapeinside={}` for this listing only; which in LyX was done by right-clicking the listings inset, choosing settings->advanced. Therefore `=<` isn't interpreted literally here, in a listing when the escape sequence is already defined as shown... we need to somehow escape this representation...

85c ⟨./fangle.sty[4]() ⇑84d lang=⟩ +≡ △85b 85d▽

```
\lstset{escapeinside={=<}{>}}%
```

Although our macros will contain the `@` symbol, they will be included in a `\makeatletter` section by LyX; however we keep the commented out `\makeatletter` as a reminder. The listings package likes to centre the titles, but noweb titles are specially formatted and must be left aligned. The simplest way to do this turned out to be by removing the definition of `\lst@maketitle`. This may interact badly if other listings want a regular title or caption. We remember the old maketitle in case we need it.

85d ⟨./fangle.sty[5]() ⇑84d lang=⟩ +≡ △85c 85e▽

```
%\makeatletter
%somehow re-defining maketitle gives us a left-aligned title
%which is extactly what our specially formatted title needs!
\global\let\fangle@lst@maketitle\lst@maketitle%
\global\def\lst@maketitle{}%
```

16.2.1 The chunk command

Our chunk command accepts one argument, and calls `\ltset`. Although `\ltset` will note the name, this is erased when the next `\lstlisting` starts, so we make a note of this in `\lst@chunkname` and restore in in lstlistings Init hook.

85e ⟨./fangle.sty[6]() ⇑84d lang=⟩ +≡ △85d 86a▷

```
\def\Chunk#1{%
  \lstset{title={\fanglecaption},name=#1}%
  \global\edef\lst@chunkname{\lst@intname}%
}%
```

85e ⟨./fangle.sty[6]() ⇑84d, lang=⟩ +≡ △85d 86a▷

```
19 \def\lst@chunkname{\empty}%
```

16.2.1.1 Chunk parameters

Fangle permits parameterized chunks, and requires the paramters to be specified as listings options. The fangle script uses this, and although we don't do anything with these in the LaTeX code right now, we need to stop the listings package complaining.

86a ⟨./fangle.sty[7]() ⇑84d, lang=⟩ +≡ ◁85e 86b▽

```
20 \lst@Key{params}\relax{\def\fangle@chunk@params{#1}}%
```

As it is common to define a chunk which then needs appending to another chunk, and annoying to have to declare a single line chunk to manage the include, we support an append= option.

86b ⟨./fangle.sty[8]() ⇑84d, lang=⟩ +≡ △86a 86c▽

```
21 \lst@Key{append}\relax{\def\fangle@chunk@append{#1}}%
```

16.2.2 The noweb styled caption

We define a public macro `\fanglecaption` which can be set as a regular title. By means of `\protect`, It expands to `\fangle@caption` at the appopriate time when the caption is emitted.

86c ⟨./fangle.sty[9]() ⇑84d, lang=⟩ +≡ △86b 86d▽

```
\def\fanglecaption{\protect\fangle@caption}%
```

22c ⟨some-chunk 19b⟩ ≡ + ◁22b 24d▷

In this example, the current chunk is 22c, and therefore the third chunk on page 22.
It's name is some-chunk.
The first chunk with this name (19b) occurs as the second chunk on page 19.
The previous chunk (22d) with the same name is the second chunk on page 22.
The next chunk (24d) is the fourth chunk on page 24.

Figure 1. Noweb Heading

The general noweb output format compactly identifies the current chunk, and references to the first chunk, and the previous and next chunks that have the same name.

This means that we need to keep a counter for each chunk-name, that we use to count chunks of the same name.

16.2.3 The chunk counter

It would be natural to have a counter for each chunk name, but TeX would soon run out of counters[1], so we have one counter which we save at the end of a chunk and restore at the beginning of a chunk.

86d ⟨./fangle.sty[10]() ⇑84d, lang=⟩ +≡ △86c 87c▷

```
22 \newcounter{fangle@chunkcounter}%
```

1. ...soon did run out of counters and so I had to re-write the LaTeX macros to share a counter as described here.

We construct the name of this variable to store the counter to be the text `lst-chunk-` prefixed onto the chunks own name, and store it in `\chunkcount`.

We save the counter like this:

87a ⟨save-counter[1](), lang=⟩ ≡

```
\global\expandafter\edef\csname \chunkcount\endcsname{\arabic{fangle@chunkcounter}}%
```

and restore the counter like this:

87b ⟨restore-counter[1](), lang=⟩ ≡

```
\setcounter{fangle@chunkcounter}{\csname \chunkcount\endcsname}%
```

If there does not already exist a variable whose name is stored in `\chunkcount`, then we know we are the first chunk with this name, and then define a counter.

Although chunks of the same name share a common counter, they must still be distinguished. We use is the internal name of the listing, suffixed by the counter value. So the first chunk might be `something-1` and the second chunk be `something-2`, etc.

We also calculate the name of the previous chunk if we can (before we increment the chunk counter). If this is the first chunk of that name, then `\prevchunkname` is set to `\relax` which the noweb package will interpret as not existing.

87c ⟨./fangle.sty[11]() ⇑84d, lang=⟩ +≡ ◁86d 87d▽

```
\def\fangle@caption{%
  \edef\chunkcount{lst-chunk-\lst@intname}%
  \@ifundefined{\chunkcount}{%
    \expandafter\gdef\csname \chunkcount\endcsname{0}%
    \setcounter{fangle@chunkcounter}{\csname \chunkcount\endcsname}%
    \let\prevchunkname\relax%
  }{%
    \setcounter{fangle@chunkcounter}{\csname \chunkcount\endcsname}%
    \edef\prevchunkname{\lst@intname-\arabic{fangle@chunkcounter}}%
  }%
```

After incrementing the chunk counter, we then define the name of this chunk, as well as the name of the first chunk.

87d ⟨./fangle.sty[12]() ⇑84d, lang=⟩ +≡ △87c 87e▽

```
  \addtocounter{fangle@chunkcounter}{1}%
  \global\expandafter\edef\csname \chunkcount\endcsname{\arabic{fangle@chunkcounter}}%
  \edef\chunkname{\lst@intname-\arabic{fangle@chunkcounter}}%
  \edef\firstchunkname{\lst@intname-1}%
```

We now need to calculate the name of the next chunk. We do this by temporarily skipping the counter on by one; however there may not actually be another chunk with this name! We detect this by also defining a label for each chunk based on the chunkname. If there is a next chunkname then it will define a label with that name. As labels are persistent, we can at least tell the second time LaTeX is run. If we don't find such a defined label then we define `\nextchunkname` to `\relax`.

87e ⟨./fangle.sty[13]() ⇑84d, lang=⟩ +≡ △87d 88a▷

```
  \addtocounter{fangle@chunkcounter}{1}%
  \edef\nextchunkname{\lst@intname-\arabic{fangle@chunkcounter}}%
  \@ifundefined{r@label-\nextchunkname}{\let\nextchunkname\relax}{}%
```

The noweb package requires that we define a `\sublabel` for every chunk, with a unique name, which is then used to print out it's navigation hints.

We also define a regular label for this chunk, as was mentioned above when we calculated `\nextchunkname`. This requires LaTeX to be run at least twice after new chunk sections are added — but noweb requried that anyway.

88a ⟨./fangle.sty[14]() ⇑84d, lang=⟩ +≡ ◁87e 88b▽

```
  \sublabel{\chunkname}%
% define this label for every chunk instance, so we
% can tell when we are the last chunk of this name
  \label{label-\chunkname}%
```

We also try and add the chunk to the list of listings, but I'm afraid we don't do very well. We want each chunk name listing once, with all of it's references.

88b ⟨./fangle.sty[15]() ⇑84d, lang=⟩ +≡ △88a 88c▽

```
  \addcontentsline{lol}{lstlisting}{\lst@name~[\protect\subpageref{\chunkname}]}%
```

We then call the noweb output macros in the same way that noweave generates them, except that we don't need to call `\nwstartdeflinemarkup` or `\nwenddeflinemarkup` — and if we do, it messes up the output somewhat.

88c ⟨./fangle.sty[16]() ⇑84d, lang=⟩ +≡ △88b 88d▽

```
  \nwmargintag{%
    {%
      \nwtagstyle{}%
      \subpageref{\chunkname}%
    }%
  }%
%
  \moddef{%
    {\lst@name}%
    {%
      \nwtagstyle{}\/%
      \@ifundefined{fangle@chunk@params}{}{%
        (\fangle@chunk@params)%
      }%
      [\csname \chunkcount\endcsname]~%
      \subpageref{\firstchunkname}%
    }%
    \@ifundefined{fangle@chunk@append}{}{%
    \ifx{}\fangle@chunk@append{x}\else%
        ,~add~to~\fangle@chunk@append%
    \fi%
    }%
\global\def\fangle@chunk@append{}%
\lstset{append=x}%
  }%
%
  \ifx\relax\prevchunkname\endmoddef\else\plusendmoddef\fi%
%  \nwstartdeflinemarkup%
  \nwprevnextdefs{\prevchunkname}{\nextchunkname}%
%  \nwenddeflinemarkup%
}%
```

Originally this was developed as a `listings` aspect, in the Init hook, but it was found easier to affect the title without using a hook — `\lst@AddToHookExe{PreSet}` is still required to set the listings name to the name passed to the `\Chunk` command, though.

88d ⟨./fangle.sty[17]() ⇑84d, lang=⟩ +≡ △88c 89a▷

```
%\lst@BeginAspect{fangle}
%\lst@Key{fangle}{true}[t]{\lstKV@SetIf{#1}{true}}
\lst@AddToHookExe{PreSet}{\global\let\lst@intname\lst@chunkname}
\lst@AddToHook{Init}{}%\fangle@caption}
```

```
%\lst@EndAspect
```

16.2.4 Cross references

We define the `\chunkref` command which makes it easy to generate visual references to different code chunks, e.g.

Macro	Appearance
`\chunkref{preamble}`	
`\chunkref[3]{preamble}`	
`\chunkref{preamble}[arg1, arg2]`	

Chunkref can also be used within a code chunk to include another code chunk. The third optional parameter to chunkref is a comma sepatarated list of arguments, which will replace defined parameters in the chunkref.

Note 1. Darn it, if I have: `=<\chunkref{new-mode-tracker}[{chunks[chunk_name, "language"]},{mode}]>` the inner braces (inside [ ]) cause _ to signify subscript even though we have `lst@ReplaceIn`


```
\def\chunkref@args#1,{%
  \def\arg{#1}%
  \lst@ReplaceIn\arg\lst@filenamerpl%
  \arg%
  \@ifnextchar){\relax}{, \chunkref@args}%
}%
\newcommand\chunkref[2][0]{%
  \@ifnextchar({\chunkref@i{#1}{#2}}{\chunkref@i{#1}{#2}()}%
}%
\def\chunkref@i#1#2(#3){%
  \def\zero{0}%
  \def\chunk{#2}%
  \def\chunkno{#1}%
  \def\chunkargs{#3}%
  \ifx\chunkno\zero%
    \def\chunkname{#2-1}%
  \else%
    \def\chunkname{#2-\chunkno}%
  \fi%
  \let\lst@arg\chunk%
  \lst@ReplaceIn\chunk\lst@filenamerpl%
  \LA{%\moddef{%
    {\chunk}%
    {%
      \nwtagstyle{}\/%
      \ifx\chunkno\zero%
      \else%
      [\chunkno]%
      \fi%
      \ifx\chunkargs\empty%
      \else%
        (\chunkref@args #3,)%
      \fi%
      ~\subpageref{\chunkname}%
    }%
  }%
  \RA%\endmoddef%
}%
```

16.2.5 The end

90a ⟨./fangle.sty[19]() ⇑84d, lang=⟩ +≡ ◁89a

```
119 %
120 %\makeatother
```

Chapter 17
Extracting fangle

17.1 Extracting from Lyx

To extract from LyX, you will need to configure LyX as explained in section ?.

And this lyx-build scrap will extract fangle for me.

91a ⟨lyx-build[2]() ⇑20a, lang=sh⟩ +≡ ◁20a

```
#! /bin/sh
set -x

⟨lyx-build-helper  19b⟩
cd $PROJECT_DIR || exit 1

/usr/local/bin/fangle -R./fangle $TEX_SRC > ./fangle
/usr/local/bin/fangle -R./fangle.module $TEX_SRC > ./fangle.module

export FANGLE=./fangle
export TMP=${TMP:-/tmp}
⟨test:*  95a⟩
```

With a lyx-build-helper

91b ⟨lyx-build-helper[2]() ⇑19b, lang=sh⟩ +≡ ◁19b

```
PROJECT_DIR="$LYX_r"
LYX_SRC="$PROJECT_DIR/${LYX_i%.tex}.lyx"
TEX_DIR="$LYX_p"
TEX_SRC="$TEX_DIR/$LYX_i"
TXT_SRC="$TEX_SRC"
```

17.2 Extracting documentation

91c ⟨./gen-www[1](), lang=⟩ ≡

```
#python -m elyxer --css lyx.css $LYX_SRC | \
#  iconv -c -f utf-8 -t ISO-8859-1//TRANSLIT | \
#  sed 's/UTF-8"\(.\)>/ISO-8859-1"\1>/' > www/docs/fangle.html

python -m elyxer --css lyx.css --iso885915 --html --destdirectory www/docs/fangle.e \
       fangle.lyx > www/docs/fangle.e/fangle.html

( mkdir -p www/docs/fangle && cd www/docs/fangle && \
  lyx -e latex ../../../fangle.lyx && \
  htlatex ../../../fangle.tex "xhtml,fn-in" && \
  sed -i -e 's/<!--l\. [0-9][0-9]* *-->//g' fangle.html
)

```

91c ⟨./gen-www[1](), lang=⟩ ≡

```
( mkdir -p www/docs/literate && cd www/docs/literate && \
  lyx -e latex ../../../literate.lyx && \
  htlatex ../../../literate.tex "xhtml,fn-in" && \
  sed -i -e 's/<!--l\. [0-9][0-9]* *-->$//g' literate.html
)
```

17.3 Extracting from the command line

First you will need the tex output, then you can extract:

92a ⟨lyx-build-manual[1](), lang=sh⟩ ≡

```
lyx -e latex fangle.lyx
fangle -R./fangle fangle.tex > ./fangle
fangle -R./fangle.module fangle.tex > ./fangle.module
```

Part III

Tests

Chapter 18
Tests

95a ⟨test:*[1](), lang=⟩ ≡

```
#! /bin/bash

export SRC="${SRC:-./fangle.tm}"
export FANGLE="${FANGLE:-./fangle}"
export TMP="${TMP:-/tmp}"
export TESTDIR="$TMP/$USER/fangle.tests"
export TXT_SRC="${TXT_SRC:-$TESTDIR/fangle.txt}"

mkdir -p "$TESTDIR"

tm -s -c "$SRC" "$TXT_SRC" -q

⟨test:helpers  95c⟩
run_tests() {
  ⟨test:run-tests  95b⟩
}

# test current fangle
echo Testing current fangle
run_tests

# extract new fangle
echo testing new fangle
$FANGLE -R./fangle "$TXT_SRC" > "$TESTDIR/fangle"
export FANGLE="$TESTDIR/fangle"
run_tests

# Now check that it can extract a fangle that also passes the tests!
echo testing if new fangle can generate itself
$FANGLE -R./fangle "$TXT_SRC" > "$TESTDIR/fangle.new"
passtest diff -bwu "$FANGLE" "$TESTDIR/fangle.new"
export FANGLE="$TESTDIR/fangle.new"
run_tests
```

95b ⟨test:run-tests[1](), lang=sh⟩ ≡

```
# run tests
$FANGLE -Rpca-test.awk $TXT_SRC | awk -f - || exit 1
⟨test:cromulence  56d⟩
⟨test:escapes  60c⟩
⟨test:test-chunk(test:example-sh) 96a⟩
⟨test:test-chunk(test:example-makefile) 96a⟩
⟨test:test-chunk(test:q:1) 96a⟩
⟨test:test-chunk(test:make:1) 96a⟩
⟨test:test-chunk(test:make:2) 96a⟩
⟨test:chunk-params  97e⟩
```

95c ⟨test:helpers[1](), lang=⟩ ≡

```
passtest() {
  if "$@"
  then echo "Passed $TEST"
```

95c ⟨test:helpers[1](), lang=⟩ ≡

```
  else echo "Failed $TEST"
       return 1
  fi
}

failtest() {
  if ! "$@"
  then echo "Passed $TEST"
  else echo "Failed $TEST"
       return 1
  fi
}
```

This chunk will render a named chunk and compare it to another rendered nameed chunk

96a ⟨test:test-chunk[1](chunk), lang=sh⟩ ≡

```
⟨test:test-chunk-result(⟨chunk⟩, ⟨chunk⟩result) 96b⟩
```

96b ⟨test:test-chunk-result[1](chunk, result), lang=sh⟩ ≡

```
TEST="⟨result⟩" passtest diff -u --label "⟨chunk⟩" <( $FANGLE -R⟨chunk⟩ $TXT_SRC ) \
                                 --label "⟨result⟩" <( $FANGLE -R⟨result⟩ $TXT_SRC )
```

Chapter 19
Chunk Parameters

19.1 LyX

97a ⟨test:lyx:chunk-params:sub[1](THING, colour), lang=⟩ ≡

```
I see a ${THING},
a ${THING} of colour ${colour},
and looking closer =<\chunkref{test:lyx:chunk-params:sub:sub}(${colour})>
```

97b ⟨test:lyx:chunk-params:sub:sub[1](colour), lang=⟩ ≡

```
a funny shade of ${colour}
```

97c ⟨test:lyx:chunk-params:text[1](), lang=⟩ ≡

```
What do you see? "=<\chunkref{test:lyx:chunk-params:sub}(joe, red)>"
Well, fancy!
```

Should generate output:

97d ⟨test:lyx:chunk-params:result[1](), lang=⟩ ≡

```
What do you see? "I see a joe,
                  a joe of colour red,
                  and looking closer a funny shade of red"
Well, fancy!
```

And this chunk will perform the test:

97e ⟨test:chunk-params[1](), lang=⟩ ≡ 98b▷

```
⟨test:test-chunk-result(test:lyx:chunk-params:text, test:lyx:chunk-params:result) 96b⟩ || exit 1
```

19.2 TeXmacs

97f ⟨test:chunk-params:sub[1](THING, colour), lang=⟩ ≡

```
I see a ⟨THING⟩,
a ⟨THING⟩ of colour ⟨colour⟩,
and looking closer ⟨test:chunk-params:sub:sub(colour) 97g⟩
```

97g ⟨test:chunk-params:sub:sub[1](colour), lang=⟩ ≡

```
a funny shade of ⟨colour⟩
```

97h ⟨test:chunk-params:text[1](), lang=⟩ ≡ 96a▷

```
What do you see? "⟨test:chunk-params:sub(joe, red) 97f⟩"
```

97h ⟨test:chunk-params:text[1](), lang=⟩ ≡ 96a▷

```
Well, fancy!
```

Should generate output:

98a ⟨test:chunk-params:result[1](), lang=⟩ ≡

```
What do you see? "I see a joe,
                  a joe of colour red,
                  and looking closer a funny shade of red"
Well, fancy!
```

And this chunk will perform the test:

98b ⟨test:chunk-params[2]() ⇑97e, lang=⟩ +≡ ◁97e

```
⟨test:test-chunk-result(test:chunk-params:text, test:chunk-params:result) 96b⟩ || exit 1
```

Chapter 20
Compile-log-lyx

99a ⟨Chunk:./compile-log-lyx[1](), lang=sh⟩ ≡

```
#! /bin/sh
# can't use gtkdialog -i, cos it uses the "source" command which ubuntu sh doesn't have

main() {
  errors="/tmp/compile.log.$$"
#  if grep '^[^ ]*:\( In \|[0-9][0-9]*: [^ ]*:\)' > $errors
if grep '^[^ ]*(\([0-9][0-9]*\)) *: *\(error\|warning\)' > $errors
  then
    sed -i -e 's/^[^ ]*[/\\]\([^/\\]*\)(\([ 0-9][ 0-9]*\)) *: */\1:\2|\2|/' $errors
    COMPILE_DIALOG='
 <vbox>
  <text>
    <label>Compiler errors:</label>
  </text>
  <tree exported_column="0">
    <variable>LINE</variable>
    <height>400</height><width>800</width>
    <label>File | Line | Message</label>
    <action>'". $SELF ; "'lyxgoto $LINE</action>
    <input>'"cat $errors"'</input>
  </tree>
  <hbox>
   <button><label>Build</label>
     <action>lyxclient -c "LYXCMD:build-program" &</action>
   </button>
   <button ok></button>
  </hbox>
 </vbox>
'
    export COMPILE_DIALOG
    ( gtkdialog --program=COMPILE_DIALOG ; rm $errors ) &
  else
    rm $errors
  fi
}

lyxgoto() {
  file="${LINE%:*}"
  line="${LINE##*:}"
  extraline=`cat $file | head -n $line | tac | sed '/^\\\\begin{lstlisting}/q' | wc -l`
  extraline=`expr $extraline - 1`
  lyxclient -c "LYXCMD:command-sequence server-goto-file-row $file $line ; char-forward ; repeat
$extraline paragraph-down ; paragraph-up-select"
}

SELF="$0"
if test -z "$COMPILE_DIALOG"
then main "$@"
fi
```