Getting Started with Fangle

by Sam Liddicott
sam@liddicott.com

Abstract

This document explains how to use fangle and is a companion to **Fangle** which explains how fangle works.

Of course one does not need to know how Fangle works in order to use it, and one may find it easier to understand how it works when one knows how it is used.

Because of this it is probably better to read **Getting Started with Fangle** before reading **Fangle**.

This document is not intended to cover what *literate programming* is, or what its advantages are. It is assumed that the reader will have some knowledge of this. This document covers how to use fangle for literate programming, assuming that the user has at least some theoretical knowledge of what this entails.

This document includes getting and installing fangle, starting a new simple fangle project (with TeXmacs, LyX, LaTeX, and plain text) and then making use of **Makefile.inc** (from the **Fangle** book) for larger projects and for specific sub-modules of existing Make based projects.

This document should have enough detail to help someone who is un-familiar with TeXmacs or LyX to become acquainted with their use for literate programming, but is not intended to guide the reader in making particularly effective use of these editors.

It is assumed that the reader will already have a functioning installation of TeXmacs, LyX, LaTeX or whatever document preparation system they intend to employ.

Table of contents

I Getting and Installing Fangle

1 Getting Fangle

The latest release of Fangle can be downloaded as a gzip'd tar file from the git repository at http://git.savannah.gnu.org/cgit/fangle.git/snapshot/latest.tar.gz

You can checkout the entire git repository read-only by cloning either git://git.sv.gnu.org/fangle.git or http://git.savannah.gnu.org/r/fangle.git

Users with a Savannah.gnu.org login can also clone ssh://git.sv.gnu.org/srv/git/fangle.git which will also give commit access to project members.

2 Installing Fangle

There is no `make install`[1] so you will need to copy files to the correct places, as described here.

To do: Make install

2.1 For personal use

2.1.1 The fangle untangler

`fangle` itself needs copying to where personal programs are kept. This could just be the git checkout directory or the place where you un-tar'd lateIf you have noweb installed then yst.tar.gz

1. there should be, but I'm writing this document partly to find out what the obstacles to adoption are

I keep my personal programs in a private `.local/bin` directory which I keep in my path.

```
mkdir -p $HOME/.local/bin
cp fangle $HOME/.local/bin
```

If you don't have this folder in your path (and you use bash) you could add it like this:

```
echo 'export PATH=$PATH:$HOME/.local/bin' >> $HOME/.bashrc
```

and if you don't want to have to login again, also set the path for the current session:

```
export PATH=$PATH:$HOME/.local/bin
```

2.1.2 The TeXmacs stylesheet

If you are using TeXmacs, then `fangle.ts` needs copying to your private TeXmacs packages folder:

```
cp fangle.ts $HOME/.TeXmacs/packages/
```

2.1.3 The LyX stylesheet

If you are using LyX, then `fangle.module` needs copying to your private LyX modules folder:

```
cp fangle.module $HOME/.lyx/modules/
```

You will also need to have Norman Ramsey's NOWEB stylesheet installed.

2.1.4 The TeX stylesheet

To do: Still needs ripping off out of the .module maybe

You will also need to have Norman Ramsey's noweb stylesheet installed.

2.2 For system-wide use

2.2.1 The fangle untangler

/usr/local/bin

`fangle` can be copied to `/usr/local/bin`

```
sudo cp fangle /usr/local/bin
```

/opt

you could extract the entire package to `/opt/fangle` but might want to add `/opt/fangle` to the system-wide path. You could do that like this

```
echo 'PATH=$PATH:/opt/fangle' >> /etc/profile.d/fangle.sh
echo export PATH >> /etc/profile.d/fangle.sh
```

2.2.2 The TeXmacs stylesheet

If you are using TeXmacs then you will need to install `fangle.ts` into the TeXmacs system-wide package folder. This might be `/usr/share/texmacs/TeXmacs/packages/` but may vary across installations.

```
cp fangle.ts /usr/share/texmacs/TeXmacs/packages/
```

2.2.3 The LyX stylesheet

If you are using LyX, then you will need to install `fangle.module` into the LyX system-wide modules folder. This might be `/usr/share/lyx/` but may vary across installations

```
cp fangle.module /usr/share/lyx/modules/
```

You will also need to have Norman Ramsey's NOWEB stylesheet installed.

2.2.4 The TEX stylesheet

To do: Still needs ripping off out of the .module maybe

You will also need to have Norman Ramsey's noweb stylesheet installed.

II Authoring with Fangle

Fangle has editor style-sheets for TEXMACS and LYX to aid document editing.

Fangle can untangle[2] sources from text files produced by TEXMACS's verbatim export, from TEX files generated by LYX, from plain hand-edited LATEX or TEX files, and from plain text files that adhere to certain conventions (either hand-written or generated from other document editors).

This part will show how to start a simple project for TEXMACS, LYX, LATEX/TEX and plain text.

The instructions cover more than mere use of the fangle style-sheet. Literate programming is more than just pretty-looks or a bound booklet — it is a mind-set. Good titles, author information, abstracts, good structure and good narrative are essential to stop the whole thing being a good-looking waste of time.

3 TEXMACS

This section does not assume a large degree of familiarity with TEXMACS, but you should have spent at least a few minutes figuring out how to use it.

3.1 Load fangle style-sheet

1. Start TEXMACS with a new document.

2. Work around a dumb bug in Fangle[3].

   From the menu: Tools→Execute→Evaluate scheme expression... and type: `(define-group enumerate-tag nf-chunk)`

   Sadly you will need to do this each time you start TEXMACS but lucky for you it will remember the last command you ran.

3. Choose an appropriate document style:

   From the menu: Document→Style→article

   For small informal projects I usually choose *article*, and for longer more formal projects I usually choose a *book*.

2. *untangling* is the historical term referring to the extraction or generation of source code from the documentation

3. And if you can work out what the fix is to get fangle.ts to execute this command, please let me know!

4. Add the fangle package:

   From the menu: Document→Add package→fangle

   If the *fangle* package isn't listed, then update your styles selection with:

   Tools→Update→Styles and then try again

5. Optionally, (if you prefer this style):

   Document→View→Create preamble (or Document→View→Show preamble) and insert this:

   `<assign|par-first|0fn><assign|par-par-sep|0.5fn>`

   and then: Document→View→Show all parts

3.2 Save the document

Save the document, and call it `hello-world.tm`

From the menu: File→Save

3.3 Sandard document parts

3.3.1 Insert your title

Insert→Title→Insert title

1. Type the name of your document: `L` `i` `t` `e` `r` `a` `t` `e` `space` `E` `x` `a` `m` `p` `l` `e`
2. Press `enter` and then type your name.
3. Press `enter` and then type your email address.
4. Press `→` to leave the title block

3.3.2 Insert your abstract

Insert→Title→Abstract

The abstract should explain what the document is about and help the reader discover if the document is relevant to them. It should not contain explanations that the document contains but it should explain what it is that the document contains.

See the abstract to this document for a fair example.

After you have entered the abstract, press `→` to leave the abstract block

3.3.3 Insert a table of contents

Insert→Automatic→Table of contents

3.3.4 Start a new section (or chapter)

Insert→Section→Section (or Insert→Section→Chapter) and type the name of the chapter:

`H` `e` `l` `l` `o` `space` `W` `o` `r` `l` `d` `enter`

The first chapter will generally illustrate the problem to be solved and explain how the book is to be used to understand and provide the solution.

3.4 Talk about your code

Before you insert a chunk of code, you introduce it.

Usually you will have introduced some aspect of the main problem that the program as a whole will solve, and will then outline the solution that this chunk will provide.

We will introduce our hello-world chunk by typing:

`T` `h` `e` `space` `t` `y` `p` `i` `c` `a` `l` `space` `h` `e` `l` `l` `o` `space` `w` `o` `r` `l` `d` `space` `p` `r` `o` `g` `r` `a` `m` `space` `l` `o` `o` `k` `s` `space` `l` `i` `k` `e` `space` `t` `h` `i` `s` `:` `enter`

3.5 Insert your first code chunk

Fangle currently has no menus; all commands are entered with a back-slash. This may annoy you, but it is much faster to keep your hands off the mouse.

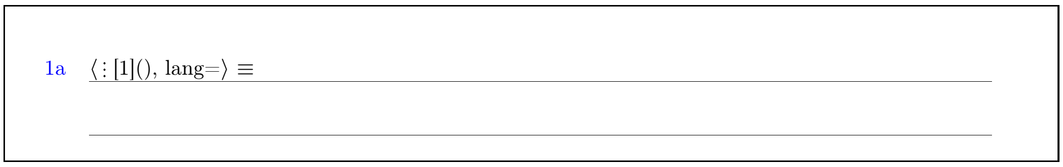
To do: Add some menus bindings

Fangle chunks are (currently) called: `nf-chunk` and are entered like this:

1. type: `\` `n` `f` `-` `c` `h` `u` `n` `k` — it will appear like this: ⟨*nf-chunk*⟩

2. press `enter`

   Depending on your $\mathrm{T_EX_{MACS}}$ environment, you may get either this ⟨nf-chunk|⋮|||⟩ which is the inactive view, or the active view shown below:

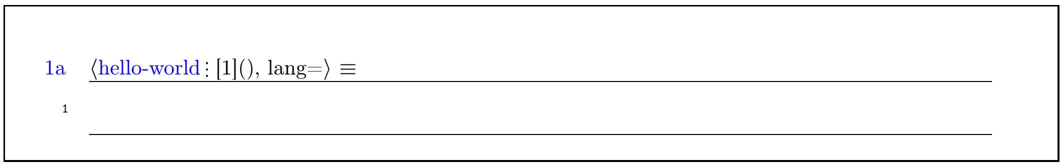
   

   1a ⟨⋮[1](), lang=⟩ ≡

   If the text insertion point (represented by the three vertical dots ⋮) does not appear as shown above, then press `←` so that it does.

3. Type the name of your chunk: `h` `e` `l` `l` `o` `-` `w` `o` `r` `l` `d`

   This will give you either ⟨nf-chunk|hello-world⋮|||⟩ for the inactive view, or the active view shown as below:

   1a ⟨hello-world⋮[1](), lang=⟩ ≡

3.6 Optional chunk parameters

Press `→` to move the text insertion point to the second argument of the chunk.

This is to specify parameters to the code that will be contained in the chunk. Chunks can take optional parameters, and behave somewhat like C macros.

Usually chunks will not have parameters, although parameters can be useful when a chunk is used to express an algorithm (like a sort) or a class of behaviours (like binary tree management). In such cases, a set of parameterized chunks can work like generics or C++ templates.

If chunk has parameters, they must be enclosed in a tuple. When I understand DRD's a bit better this will be done for you, but for now if you want chunk parameters then you create a tuple, otherwise skip to the next step.

3.6.1 Create a tuple

Press `\`. If this comes out as a backslash \ (perhaps red) instead of in angle brackets like this ⟨\⟩ then press `backspace` and enter a command-backslash using the meta key (probably the windows button) by pressing `M-\`.

Once you have the ⟨\⟩, type `t` `u` `p` `l` `e` `enter`.

Type the first chunk argument, and then for additional arguments, `M-→` (windows key and right arrow).

You can type multiple parameters: ⟨nf-chunk|hello-world|⟨tuple|message|language⁝⟩||⟩ or

```
1a  ⟨hello-world[1](message, language⁝), lang=⟩ ≡
```

3.7 Typing code

Press `→` to move the text insertion point to the main code area.

If your chunk shows as inactive then this will be visible as the third argument, but you may prefer to activate your chunk at this point. You should be able to do this by pressing `enter` or clicking the icon on the toolbar. Sometimes the icon is absent and pressent enter does nothing — in which case try the ⟨menu|Tools|Update|Styles⟩ and if that doesn't work then I don't know what to do.

The code body is an enumerate style. Press `enter` to insert a new numbered line. (You'll probably want to press `←` `backspace` `→` to delete the blank line that is somehow there. To do: stop that from happening

```
1a  ⟨hello-world[1](message, language), lang=⟩ ≡
    ⁝
```

At this point, start typing code.

When you press `enter`, a new line number will be inserted at the left of the listing. If you press `S-enter` then you can break the line for layout purposes, but it will not be considered a new-line when the code is extracted and leading white-space will be stripped.

```
1a  ⟨hello-world[1](message, language), lang=⟩ ≡
#include stdio.c

main() {
  printf("⁝
```

The listing above is incomplete. Instead of typing the the traditional `hello world!`, we can make use of our chunk arguments. Let's place the value of the argument message at this point.

The command for a chunk argument is `\nf-arg`, but when you press the `\` it will enter a literal \ because the cursor is in a code block. To enter a command-backslash in code block, use the meta key (probably the windows button): `M-\` `nf-arg` and this will produce: ⟨nf-arg|⟩

To enter the name of the argument message, type `message` `→` which will produce ⟨message⟩

Finish typing the code as shown below:

1a ⟨hello-world[1](message, language), lang=⟩ ≡

```
#include stdio.c

main() {
  printf("⟨message⟩\n");
};
```

We've now defined a chunk of code which can potentially produce the famous `hello world!` in any language.

If the chunk were more complicated, we could break off part-way through and provide more explanation, and then insert another chunk *with the same name* to continue the code. In this way a single chunk can be broken across sections and spread across the whole document and still be assembled in order.

Let's define some file-chunks that use this chunk.

3.8 File chunks

By convention, file chunk is just a regular chunk whose name begins with `./` which signifies to build-tools that it should be extracted into a file.

3.8.1 French hello-world

Insert a new sub-section for french:

Insert→Section→Subsection (or Insert→Section→Section) and type the name of the subsection:

`In` `space` `French` `enter`

Then introduce the next code chunk, type: `We` `space` `will` `space` `derive` `space` `the` `space` `french` `space` `hello-world` `space` `program` `space` `like` `space` `this:` `enter`

Then, create a chunk called hello-world.fr.c, by typing: `\nf-chunk` `enter` and then the chunk name `./hello-world.fr.c` `→` `→`

1.1 In French

We will derive the french hello-world program like this:

1b ⟨./hello-world.fr.c[1](), lang=⟩ ≡

```
⋮
```

To include our previous chunk with the `nf-ref` command, type `M-\` `nf-ref` `enter` and then type the name of our previous chunk, `hello-world`

We then move to the arguments part of the `nf-ref`, `→`, and type the argument *Bonjour tout le monde* in a tuple:

`M-\` `t u p l e` `enter` `B o n j o u r` `space` `t o u t` `space` `l e` `space` `m o n d e` `enter`

> **1.1 In French**
>
> We will derive the french hello-world program like this:
>
> 1b ⟨./hello-world.fr.c[1](), lang=⟩ ≡
>
> ```
> ⟨hello-world(Bonjour tout le monde) 1a⟩:
> ```

Note that when there are no arguments to the reference, the parenthesis do not appear, but they appear automatically when there are arguments.

3.8.2 German hello-world

And let's create a similar chunk for german. Insert a new sub-section:

Insert→Section→Subsection (or Insert→Section→Section) and type the name of the subsection:

`I n` `space` `G e r m a n` `enter`

Then introduce the next code chunk, type: `W e` `space` `w i l l` `space` `d e r i v e` `space` `t h e` `space` `g e r m a n` `space` `h e l l o - w o r l d` `space` `p r o g r a m` `space` `l i k e` `space` `t h i s :` `enter`

Create a chunk called hello-world.de.c, by typing: `\ n f - c h u n k` `enter` and then the chunk name `. / h e l l o - w o r l d . d e . c` `→` `→`

> **1.2 In German**
>
> We will derive the german hello-world program like this:
>
> 1c ⟨./hello-world.de.c[1](), lang=⟩ ≡
>
> ```
> ⟨hello-world(Hallo welt) 1a⟩:
> ```

3.9 Additional parameters

Astute readers will have noticed that the `hello-world` chunk has two parameters but that our french and german invocations only have one argument. This is not really a problem as the `hello-world` chunk only uses one; but let's change that:

> 1a ⟨hello-world[1](message, language), lang=⟩ ≡
>
> ```
> /* The traditional hello-world program in ⟨language⟩
>  * generated using fangle literate programming macros
>  */
>
> #include stdio.c
>
> main() {
>   printf("⟨message⟩\n");
> }:
> ```

We will now modify our french and german .c files by clicking inside Bonjour tout le monde and pressing `M-→` and then typing: `f r e n c h`

1b ⟨./hello-world.fr.c[1](), lang=⟩ ≡

```
⟨hello-world(Bonjour tout le monde, french) 1a⟩:
```

And doing similarly for the german:

1c ⟨./hello-world.de.c[1](), lang=⟩ ≡

```
⟨hello-world(Hallo welt, german) 1a⟩:
```

3.10 Extracting individual files

Later on, automatic extraction using `Makefile.inc` is shown, but this is how to extract chunks manually from a $\TeX_{\text{MACS}}$ document.

1. Save the `hello-world.tm` document

2. Generate a text file hello-world.txt, either with File→Export|Verbatim or with this command line:

   ```
   texmacs -s -c hello-world.tm hello-world.txt -q
   ```

3. Extract the french and german files:

   ```
   fangle -R./hello-world.fr.c hello-world.txt > hello-world.fr.c
   fangle -R./hello-world.de.c hello-world.txt > hello-world.de.c
   ```

The resultant french file should look like this:

```
#include stdio.c
/* The traditional hell-world program in french
 * generated using literate programming macros
 */
main() {
  printf("Bonjour tout le monde\n");
}
```

3.11 Extracting all files

A list of all the chunks can be obtained with:

```
fangle -r hello-world.txt
```

So we can extract all files like this:

```
texmacs -s -c hello-world.tm hello-world.txt -q &&
fangle -r hello-world.txt | while read file
do fangle -R"$file" hello-world.txt > "$file"
done
```

If you have *noweb* installed then you can use `cpif` to avoid updating files that haven't changed:

```
texmacs -s -c hello-world.tm hello-world.txt -q &&
fangle -r hello-world.txt | while read file
do fangle -R"$file" hello-world.txt | cpif "$file"
done
```

3.12 The completed document

The document you typed might look something like this:

Literate Example

Joe Soap joe@example.com

Abstract

This is a simple example of how to use literate programming templates, using hello-world.

Hello-world is a famous *first program* with a visible side effect.

This example produces hello-world in multiple languages.

Table of Contents

1 Hello World

The typical hello-world program looks something like this:

1a ⟨hello-world[1](message, language), lang=⟩ ≡

```
/* The traditional hello-world program in ⟨language⟩
 * generated using literate programming macros
 */
#include stdio.c

main() {
  printf("⟨message⟩\n");
}
```

1.1 In French

We will derive the french hello-world program like this:

1b ⟨./hello-world.fr.c[1](), lang=⟩ ≡

```
⟨hello-world(Bonjour tout le monde, french) 1a⟩
```

1.2 In German

We will derive the german hello-world program like this:

1c ⟨./hello-world.de.c[1](), lang=⟩ ≡

```
⟨hello-world(Hallo welt, german) 1a⟩
```

Which demonstrates nicely how to use fangle in terms of function, but less so in terms of style.