The Manual

This Manual aims to cover all the features of Literate. However, you do not need to read it if you don't want to. syntax is pretty intuitive and you can figure it out on your own by experimenting with the compiler yourself.

Basic usage

Literate is a command line tool which you can use to generate either html, code, or both from a `.lit` file. Once installed, you can use it by typing `lit file.lit`. There many extra options you can provide:

```
--tangle     -t         Only compile code files

--weave      -w         Only compile HTML files

--no-output  -no        Do not generate any output files

--out-dir    -odir DIR  Put the generated files in DIR

--compiler   -c         Report compiler errors (needs @compiler to be defined)
                        (see reporting compiler errors to the correct line)

--linenums   -l    STR  Write line numbers prepended with STR to the output fi
le
                        (see Writing line directives)

--md-compiler COMPILER  Use COMPILER as the markdown compiler instead of the b
uilt-in one
                        (see Using markdown)

--version    -v         Show the version number and compiler information
```

An example program

For a more complete example (and complex) of Literate, see `wc.lit` `(https://github.com/zyedidia/Literate/blob/master/examples/wc.lit)` which is a literate implementation of the word count program found on Unix systems. You can find the woven html here (examples/wc.html).

We'll start out with an example that is fairly simple but shows the important features of Literate.

```
@title Hello world in C

@s Introduction

This is an example hello world C program.
We can define codeblocks with `---`

--- hello.c
@{Includes}

int main() {
    @{Print a string}
    return 0;
}
---

Now we can define the `Includes` codeblock:

--- Includes
#include <stdio.h>
---

Finally, our program needs to print "hello world"

--- Print a string
printf("hello world\n");
---
```

You can read the explanation that follows, or run it on your own machine and understand it on your own. This program is the hello.lit program in the examples directory.

The start of the program

A Literate program will generally begin with 3 statements, although they are all optional. First you should set the code and comment type.

To set the code type use `@code_type type .extension`. This tells Literate what language you will be using. I use `c` and `.c` in the example, but for javascript it would be `javascript` and `.js`.

Next we want to define the comment type using `// %s`. For a multiline comment we could use `/* %s */`.

Now we give our program a title. To do that use the `@title` command.

Adding sections

A Literate program consists of multiple sections, each beginning with the `@s title?` command. Sections may have a title, but it is not required. A section consists of explanation and code. Generally you provide some prose, followed by a piece of code, but really, you can do it in any order you like, with any amounts of prose and code blocks.

Using markdown

When writing prose, you may use Markdown and it will be converted to HTML on compilation. So using `**word**` will make **word** bold etc... You can read a nice description of basic markdown here (https://help.github.com/articles/markdown-basics/) (note that ```` ``` ```` for denoting large code blocks will not work, just use a Literate code block).

The markdown used by Literate is slightly different than normal markdown: underscores do not do anything, and you can directly inline html. If you have to write `<` and `<`, you can use `\<` or `&lt;` and the equivalent for greater than.

If you'd like, you can use a custom markdown compiler by using the `--md-compiler` flag. For example if you want to use pandoc (http://pandoc.org/) for markdown compilation, you can run literate like so:

```
$ lit --md-compiler pandoc file.lit
```

Just make sure that the pandoc command line tool is installed on your machine. Literate just runs the command with the markdown as stdin and captures the stdout. This means that if you want to use a certain pandoc option you can:

```
$ lit --md-compiler 'pandoc -f markdown_strict' file.lit
```

Another possible command you can use while writing prose is the `@{codeblock name}` command. If you write this in prose, Literate will replace this with a reference to the code block name and a link to the section the code block is in.

Writing math equations

Literate supports rendering math equations. It uses the KaTeX (https://khan.github.io/KaTeX/) library to render LaTeX equations. Just put the equation between `$`. If you want the equation to be block level (take up an entire row), use `$$`. If you are using inline equations, make sure that there are no newlines in your equation

Note that if you are viewing the html files offline, the math will still render, but the fonts won't be as nice. If you wish to save the html file with good fonts, you can make a pdf.

Code blocks

To create a code block, you use three dashes, followed by the code block name, and another three dashes to signal the end of the code block. For example:

```
--- Code block
some code here
---
```

A code block may have any name but it must not have any trailing whitespace. In addition, if you name a code block a filename (e.g. `file.c`), this code block will be top-level, and when compiling the Literate file, all the code from this code block will be put in that file.

If you add a `+=` after the code block name, the code will be added to that block (the block must already be defined somewhere else). This is useful because sometimes you want to add a piece of code a code block, but it only makes sense to add it later after the definition. For example, if you have a code block for all the constants in your program, you can add them as you use them instead of adding them all at once at the top of the program as you would in a standard programming setting.

The most useful command in a code block is the `@{codeblock name}`. Just like it does in prose, it will reference and link the code block name that you use. However, when generating code from the Literate file, the code that is contained in `codeblock name` will be inserted at this position in the generated source.

Code block modifiers

There are certain codeblock modifiers which you can add in the codeblock name to change the behavior of Literate when compiling the codeblock. One such modifier that we have already seen is the `+=` modifier, but there are a few others. Here is a short list of the codeblock modifiers and what they do:

- `+=`: Adds code to an already defined block
- `:=`: Redefines a codeblock
- `noWeave`: Specifies that the codeblock should not be included in the HTML output (for an example, see the end of `weave.lit` in the source code for Literate)

To include a codeblock modifier, you must put the name of the codeblock, followed by `---` followed by the codeblock name:

```
--- Some code block name --- noWeave
```

With `+=` and `:=` you do not have to include the `---` but you can.

Customizing HTML output with CSS

You may not like what the HTML output from Literate looks like by default. In that case, fear not, you can customize every bit of the page with CSS (after all, it is just HTML). There are three different kinds of customizations you can make:

- You can completely overwrite the default CSS with your own
- You can add your own CSS to the default CSS
- You can overwrite the syntax highlighting theme

You can make each of these customizations with a separate command, and each takes the CSS file which contains your changes:

- `@overwrite_css file.css`
- `@add_css file.css`
- `@colorscheme file.css`

You can find some nice colorschemes here (http://jmblog.github.io/color-themes-for-google-code-prettify/). The default colorscheme is Tomorrow Light.

Saving the HTML as a PDF

The HTML file should look the same on any computer, even when used offline (except for the math fonts which won't look as nice). Even so, if you want to save the HTML file for offline use, or for another reason, you can use your browser's "print to pdf" option. I think Chrome's PDF printer performs the best. Go to `file -> print`. Change the destination to "Save as PDF", and under "More settings" make sure you have "Background graphics" selected.

Change files

First, note that `@include <file.lit>` will include another literate file, as if you had written the contents of the included file in the current file.

The `@change` command is very similar, except that some changes are performed on the other file when it is included.

Here is an example of the syntax:

```
@change examples/wc.lit

This is an example of using the @change command to recreate change files.

We are going to change the title in wc.lit from WC to Word Count.

@replace
@title WC
@with
@title Word Count
@end

Let's also change the comment_type from // %s to /* %s */.

@replace
@comment_type // %s
@with
@comment_type /* %s */
@end

@change_end
```

You can make as many changes as you want between the `@change` and `@change_end` commands.

You should see that the title was changed to Word Count and if you view the code, It should be commented mostly with /* ... */.

Reporting compiler errors to the correct line

It would be nice if when compiling the generated code file, if the program has errors, for the compiler to report the error to the correct line in the lit file. To do this, you will tell which command Literate should use to run the compiler, and Literate will check with the compiler when generating the code file.

To use this feature, you should use the `@compiler <sh-command>` command. In addition you must compile with the `--compiler` flag (otherwise Literate will ignore the command). The `sh-command` should invoke literate itself because using `--compiler` turns off output generation. This is because on larger projects you probably have your own build mechanism and don't want little code and html files being generated whenever you lint. As an example, to check for errors in the hello_world.c file, you could put the following in the file:

```
@compiler lit hello_world.lit && clang hello_world.c
```

On a larger project you might just have

```
@compiler make
```

which will build using Literate and then compile the resulting files. You can see this in the source code for literate.
When the lit file is compiled, this command will be run, and the error output (if any) will be parsed and changed to report the corresponding line numbers in the lit file. Literate can parse clang, because clang is a common compiler. If the compiler you are using is not supported, you can provide an `error_format` that will be used to parse the compiler output. The supported compilers are:

- clang (C, C++)
- gcc (C)
- g++ (C++)
- javac (Java)
- pyflakes (Python)
- jshint (Javascript)
- dmd (D)

For example, the to define the error_format for `clang` you would add this line to your file:

```
@error_format %s:%l:%s:%s: %m
```

There are several special characters here:

- `%s` means any string of characters that should be ignored
- `%l` means the line number
- `%m` means the error message

Writing line directives

If you don't want to deal with the `@compiler` option and are using a language that supports line directives, you can use the `--linenums` (or `-l`) option which will make Literate output line directives. Here is an example for a C program:

```
$ lit -l '#line' hello.lit
```

The output of this (using the `hello.lit` program from above) will be:

```
#line 20
#include <stdio.h>

#line 11
int main() {
#line 26
    printf("hello world\n");

#line 13
    return 0;
#line 14
}
```

Writing literate "books"

Literate also lets you write Literate "books". These are a collection of lit files that are strung together as chapters. They can use each others' codeblocks. To create a book, you need a book file which specifies which files are included as a sort of table of contents. Here is an example book file for the hangman, hello, and word count programs (The example lit files here can be found in the `Examples/` directory in the github repository).

```
@book
@title Example Book

This is an example book which puts hangman.lit as
chapter 1, hello.lit as chapter 2, and wc.lit as chapter 3.

This section is a short introduction you can include which will be
added to the table of contents file.

[Hangman](hangman.lit)
[Hello World](hello.lit)
[Word Count Program](wc.lit)
```

When you run lit on this file (`$ lit book.lit`) a `_book` directory will be generated containing the html files for each chapter, plus a file named `Example Book_contents.html` (in this case). This file will contain an overview of the book, including the explanation you put in the book file.

Using the Vim plugin

Literate also comes with a Vim plugin to make writing Lit files much easier. You can download it here (https://github.com/zyedidia/literate.vim). The plugin provides all sorts of niceties like syntax highlighting (it correctly syntax highlights the language embedded in code blocks using information from the @code_type), and keybindings to let you jump between code blocks.

When you first open a `.lit` file, vim will syntax highlight commands like @code_type, and @title... correctly, but it will not syntax highlight the embedded code blocks right away. This is because when you opened the empty file, there was no @code_type, so vim was unable to know what language you are using. You can execute `:e` to reload the syntax highlighting (make sure the file is saved).

You can use `<C-]>` to jump to a code block definition from a code block use. If your cursor is on `@{block name}` and you press `<C-]>`, your cursor will jump to the next use/definition.

Index of all Commands

`@code_type lang-name .lang-ext`
`lang-name`: The name of the language you are using
`lang-ext`: The extension of the language you are using. For Python this would be `py`.
The `code_type` command is used for knowing which language you are using. This information is used when creating an index, and for syntax highlighting with the Vim plugin.

`@comment_type /* %s */`
This command tells Literate what the syntax is for comments. Literate automatically adds comments with the section names to the generated code. The above example would be for a language like C.

`@title title`
Sets the title for the page.

`@s title?`
This command creates a new section. The `title` argument is optional.

`$equation$`
Renders `equation` as TeX by using MathJax. Use `$$equation$$` to make the equation a block element (takes up an entire line).

`@{code block name}`
When used in prose, this creates a link to `code block name`.
When used in a code block. This also links to the code block name, but when tangling, the code from that code block, will be inserted at this position.

`Code blocks`

```
--- Block name
code...
---
```

Creates a code block named Block name. If you add `+=` to the end of the name, Literate will append the code in the code block to an existing block named Block name.

`@include <file>`
If the file is a lit file, the lit file will be included as if you had written everything contained in the lit file in the current file.

`@change`

```
@change <file>

Comments here

@replace
Some code
@with
Some other code
@end

More comments

...

@change_end
```

This is the syntax for a change command. This will include `<file>`, but will the changes given. The code between `@replace` and `@with` will be replaced with `@with` and `@end`. You can have as many of these `@replace ... @with .. @end` statements in a change statement as you want.

`@add_css file.css`
This will take the css contained in `file.css` and add it to the default css that Literate uses.

`@overwrite_css file.css`
This will overwrite Literate's default CSS with the CSS contained in `file.css`.

`@colorscheme file.css`
This will replace the default colorscheme (Tomorrow Light) will the css in `file.css`. You can find some nice colorschemes here (http://jmblog.github.io/color-themes-for-google-code-prettify/).

`@compiler <sh-command>`
Defines the command that will be run to check for errors from the language compiler. See this section.

`@error_format format`

If your compiler is not supported, you can provide an error format string to define how Literate should parse the compiler command's output.

`@book` Mark the file as a book file.