Literate Programming, MD

How to Treat and Prevent Software Project Mess

James Taylor

Literate Programming, MD

How to Treat and Prevent Software Project Mess

James Taylor

This book is for sale at http://leanpub.com/literate-programming-md

This version was published on 2017-01-08

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help James Taylor by spreading the word about this book on Twitter!

The suggested hashtag for this book is #litpromd.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#litpromd

Contents

Introduction . . . . . **i**

I The Basics 1

1. What is literate programming? **2**

2. LitPro **4**

3. Semi-literate programming **6**

4. Symphonic Programming **8**
4.1 Syntax 10
4.2 Functions or reordering 15

5. Artistic programming **16**
5.1 Web 20

II Cookbook 24

III Web Projects 27

IV Reference 30

V The Literate Programs 33

Introduction

This is a book that specifically covers the author's tool for doing literate programming. The tool is freely available via npm[1], with source code available on github[2].

The tool implements a version of Don Knuth's idea of literate programming. The basic idea is to write code for being read and maintained rather than for the need of the computer. One can reorder code, cut up code into different snippets, promote comments to a top level, and do arbitrary transformations to the input text before outputting the code.

This book covers both the idea and the tool. Most of it is about what this specific tool does.

Part I, which is The Basics, covers the 80% of needs and uses. It is a fairly simple and straightforward concept. The freedom to break code up arbitrarily is quite nice and this part explains how to go about doing this.

Part II details more about what I term "artistic programming". This is very much concerned with arbitrary actions and transformations of the compilation phase. At its core, it takes input text, does some stuff to it, and gives an output that is then arranged into the final code. The transformation is the artistic part, the assembling of small snippets is more of a symphonic part.

Transformations opens a wide world of possibilities and Part II delves into various scenarios and how to deal with them. This is a bit of a cookbook approach.

Part III focuses on this as a tool for the web environment. The web stack is a mess of different technologies, optimizations, and many files. The tool can handle all of this and this is the place to explain it in detail.

Part IV is a reference section. Every piece of syntax is documented, from the format to the commands, directives, and subcommands.

Part V is not included, but links are provided to PDFs of the various literate programs that make up this tool. Yes, the tool compiles itself. It lightly discusses the structure of the tools.

A note on terminology. I use the term `compile` because this transforms source code from one thing to another. Often it will be `transpiling`, that is, transforming from one language to another on the same level of abstraction, but there is nothing inherent about the process that limits us to transpiling. You could set it up to generate machine code, if you really want to. You can read a little more about compiling vs. transpiling[3] if you like.

[1] https://www.npmjs.com/package/litpro

[2] https://github.com/jostylr/litpro

[3] https://www.stevefenton.co.uk/2012/11/compiling-vs-transpiling/

I The Basics

What are we trying to do and how do we do it? This should be enough to get you going.

What is literate programming?

A brief description of the philosophy and history of this mode of programming.

LitPro

A description of the tool, hopefully enough to get started with it.

Semi-literate programming

The first step towards literate programming, this is where comments become elevated to be at least as important, if not more important, than the code. Essentially, comments are the default text mode and code must be initiated.

Symphonic programming

Humans do not think in the same way computers process. Respecting the order of human thought is what symphonic programming is about. We cut up the code in many pieces, scatter it around and then the program acts as a conductor to put it all together in a harmonious symphony of beautiful code.

Artistic programming

This is the full power of LitPro released. The bits of code can be processed in arbitrary ways. We are not bound by the dictates of the language and can craft the code in whatever elegant manner we wish to craft it. With great power comes the possibilty of making a mess. This is painting code on a canvas with the blending of the bits and pieces in ways that are very powerful.

1. What is literate programming?

Literate programming is a way of organizing code developed in the late '70s by Donald Knuth. It allows one to interweave documentation and code snippets laid out in a way agnostic to the underlying language needs. That is, it fits the human need, not the computer need, for how something is organized and explained.

As a first approximation, literate programming allows one to explain the purpose of the code, how it fits in with the larger program, what this specific bit is supposed to do, and some non-obvious maneuvers in the code. The details of how, of course, can be left to well-written code, but the hope is that the prose and section headings allow one to navigate around to the interesting bits.

This approximation is the elevation of comments out of code and into the primary view with code descending into a secondary viewpoint.

This is semi-literate programming and is the most often implemented view, being the easiest and fitting with the name "literate".

At the next level of understanding, it allows one to sculpt the code to fit the human mind. The most interesting bits can be placed first. Or the overall structure can be laid out first. Whatever makes more sense, it can be done that way. Much like how a story may be laid out in non-chronological order for maximum impact of the human reading it, so too does literate programming enable the organization of code for maximal human understanding by shedding the dependency on what the computer requires for organization.

Just as humans may disagree on rather a story is told well so to does this increase the potential conflicts between reader and author of code. It becomes dependent on the author to try and think in the perspective of the reader as to how to present it well enough to the expected audience. This is not an easy task.

This is the level of arbitrary ordering of snippets. We can go even further and have arbitrary input and output files. That is, multiple literate program files can combine to make a single or multiple output files. This allows us to not only reorganize code within a single file, but organize the code across the project.

We could, for example, use a literate program in web development in which some html widget, with its css and javascript, all live in a single literate program file but upon compiling, those pieces go to there separate various destinations. We could also choose to do the opposite such as having javascript, say, in its own literate program, but then inject it (or a small, crucial subset of it) into the html file for performance reasons. We fit the organization of the project to the demands of the mental view of the project.

Related to this is the idea of a central project management file. Compile that literate program and it triggers the compilation of the entire project. But more to the point, it would be the entry point, a sort

of table-of-contents with explanations, for the whole project. Instead of being limited to expressive directory and file names, one can use arbitrary comments to reveal a clear insight into the whole structure of the project.

We could call this literate projecting. Or perhaps symphonic programming, thinking of it as conducting all the disparate pieces of code threaded together in a harmonious fashion.

The third level is complete and arbitrary control of the code. At this stage, we can run snippets through any process we want. We can write a JSON description of an object and then translate that into a language specific object. We can have bits of markdown or pug and compile it into the same HTML document. We can create our own little mini-language for just that one bit of the project to reduce the amount of code written and have the literate program create the final rendered version.

We can also take almost common bits of code and programmatically change them as needed. This is a middle ground between copy-and-paste versus creating functions to implement common code.

This third level is where the bulk of the effort of litpro was spent. It is not hard to do the first two levels, particularly if syntax is not too worried about. But the third opens an entire world of asynchronous text processing with arbitrary tools. Here convenience is extremely important and, as anyone knows, the price of convenience is complexity.

This level is perhaps described as artistic programming. It allows for an unconstrained canvas to paint the code.

As we go up the levels, each one is increasingly more complex and potentially dangerous for the undisciplined. But it is also a big opportunity to learn discipline. There is nothing quite like making a mess to teach one discipline in coding.

2. LitPro

Here we briefly explain the tool that this book describes in detail.

The idea of litpro is to write literate programs in markdown. This is not a new idea. Markdown is a wonderfully simple format where paragraphs are normal text and indented blocks (or fenced off blocks) are treated as code.

Markdown has wide support and, in particular, GitHub supports viewing markdown quite nicely.

Litpro uses the CommonMark[1] variant. This is a highly specified version with edge cases laid out in detail. Anything it considers a heading or a code block is what Litpro considers it to be. Some of the syntax choices have been constrained by following this format, but for the most part, these constraints have been most welcomed.

Litpro is written in JavaScript. The command-line tool runs using node.js[2] and can be installed using `npm install -g litpro`. This is good for explorations and initializations. Use it by using the command `litpro filename`.

Any serious use of the tool for the project should have litpro, strictly versioned, as a development dependency in the package.json file and refer to it that way. When installed as a dependency, the command can be located from the top directory as `node_modules/.bin/litpro`

To initiate a package.json file and put litpro as a dev dependency try

```
npm init
npm install litpro --save-dev
```

You can hook up scripts in the package.json file to make it easier to run. There are also configuration options and a file that we will get into eventually. We will also detail setting up a watch system to continuously compile and test the development.

Litpro is free and open source. Issue request are most welcome and you can find its repository at litpro[3]

The command-line interface is a thin wrapper around a library which is platform agnostic (hopefully). The hope is to one day have a browser based version and the decoupling of the library from the command-line will be most useful to that end. The library can be found at literate-programming-lib[4]

[1] http://commonmark.org/

[2] https://nodejs.org/en/

[3] https://github.com/jostylr/litpro

[4] https://github.com/jostylr/literate-programming-lib

There are also some plugins available. The convention is to call them `litpro-...` For example `litpro-jshint` implements a jshint command for applying that library to JavaScript code in order to find likely errors.

But there is often very little that needs to be done to use an external node module. Thus, there is not a need to have very many plugins. Instead, we can write the code needed out in the configuration file.

There is one other version which is called literate-programming[5] This was the original client, but it has been completely replaced with litpro powering it. It does differ from litpro in that it comes with "batteries included" for web development. While it is more than impossible to include much of what is used in web development, this does attempt to include some of those modules found most useful by the author. At the present time, they include a markdown compiler, jshint, some beautifiers and some minifiers, pug (formerly jade), and postcss.

Much of the inspiration for this tool comes from web development. The number of different languages and tools involved in web development coupled with best practices makes web development quite difficult. The hope is that this tool makes that easier.

While targeted at web development, it is by no means the only use. It is language agnostic and can be used for doing any type of text creation. It takes in markdown files and produces text files. Some languages need much more from an environment (such as code completion) which is orthogonal to this issue.

One possible solution to this is to use named language types for fenced code blocks and, if the editor is made aware of it, using that to do code completion, at least of the canonical language features.

[5] https://github.com/jostylr/literate-programming

3. Semi-literate programming

Here we explain the simplest use of litpro: elevating comments to readable paragraphs.

The idea here is quite simple. Write paragraphs for comments. Then use either indented blocks or code fences to write code blocks.

Here is a simple example:

````
# Saying Hi

We want to write a little javascript command that says hi, waits one second,
and then says bye. We will save it in [teens.js](# "save:")

Greetings are great, right?

    console.log("hi");

Timer is in milliseconds, so 1000. We'll call the function `bye` defined
elsewhere

    setTimeout(bye, 1000);

And now let's define the bye function. For no apparent reason, we use a named
fence block. Hey, maybe we'll get some syntax highlighting!

```js
function bye () {
    console.log("bye");
}
```

Hey, we're all done!
````

With that text saved in a file, we can run litpro on it and it should produce the file `teens.js` consisting of

```
console.log("hi");
setTimeout(bye, 1000);
function bye () {
    console.log("bye");
}
```

This uses the save directive which is, at its simplest, `[filename](# "save:")` where filename is where to save the file. The hash symbol says to use the current section; we'll see later how to reference other sections.

4. Symphonic Programming

Here we want to explore reordering code. And to discuss it, let's first talk about this.

```
a, b ....
if (crazy) {
    // lots of lines
    c = a;
} else if (slightlyCrazy) {
    // fewer lines
    c = b;
} else {
    // very few lines
    c = d;
}
```

So we have a conditional with some complicated bodies. And maybe we want to move those complications elsewhere to make it easier to understand the conditional flow. Typically, we would only have functions to do that role:

```
a, b, ...
if (crazy) {
    c = lots(a, b);
} else if (slightlyCrazy) {
    c = fewer(a, b);
} else {
    c = few(a, b);
}

function lots (a, b) {
    // lots of lines
    return a;
}

function fewer(a,b) {
    // fewer lines
    return b;
}
```

```
function few(a,b) {
    // very few lines
    return d;
}
```

That works. But notice that we have introduced functions solely for the convenience of human reading. It adds in the complexity of new scopes and makes it harder to deal with error conditions effectively. It also means we never get to see the compiled code in its context.

In contrast, we can do the following with litpro.

```
a, b ....
if (crazy) {
    _"lots"
    c = a;
} else if (slightlyCrazy) {
    _"fewer"
    c = b;
} else {
    _"few"
    c = d;
}

## Lots

Here we ...

    //lots of lines

## Fewer

And with less code we do

    // fewer lines

## Few

    // very few lines
```

This will compile to the first block. It allows us to see in the literate program the outline of that section and see the other bits separated. But if we want to see all the context together as the computer sees it, we can do that as well. We also do not lose the surrounding scope or placement in the flow. These are powerful tools at times.

4.1 Syntax

The syntax for different blocks, at its simplest, is a header block making a new section and then referencing that section with `_"section name"` in a code block. For example,

````
Some text

## Awesome details-jack

We have some awesome code

    Yay

 but what?

 ```
 yeah

 ```

 ## Importing

 Here we put in the awesome

    So jack says

    _"awesome details-jack"

    And we get what we want
````

The above snippet will create

```
    So jack says

    Yay
    yeah


    And we get what we want
```

Note that for blank lines, if you want trailing or leading newline from a code block, you need to use the fenced blocks.

Some symbols are allowed in the heading. Given how it is used in the syntax, pipes are not allowed and quotes can lead to conflicts (obviously `## "Quote"` has problems with `_""quote""` but should be fine with `_'"Quote"'`

This syntax works pretty well, but there are a couple of more tricks to learn. In particular, colons are not allowed in headings either as they have been co-opted for minor blocks, discussed below.

The available markdown header syntaxes are those that convert into h1 to h4 headers, namely `#`, `##`, `###`, `####` or either of the underline heading syntax. Those headers become references to the code in their blocks. h5 and h6 headers are reserved for something else which is discussed later.

References should be unique. If you reuse the same reference name (the same h1 to h4 header twice in the same file), then it will concatenate the blocks. h5 and h6 headers can be repeated without them being combined, as will be discussed later.

One can use the same reference as many times as one likes.

One can use any of the quotes (`"`, `'`, `` ` ``) to start a substitution, just use the same quote to end.

Minor blocks

A minor block is designed for little snippets that seem too minor to create another section heading, but that one wants to move out of the way anyway. In a markdown toc, these will not appear nor do they need to be unique between different sections though they should be unique within one section.

A minor block is initiated with either a link whose title starts with a colon or one whose target is completely empty. Let's assume this is in section `bob`.

`[jane](# ":")` would create reference `bob:jane` while `[jack 2]()` would create reference `bob:jack 2`

Within section bob, we can reference it as `":jane"` or `":jack 2"` and outside of bob, we need the full syntax of `"bob:jane"` and `"bob:jack 2"`

We can also do a save directive using section names as such `[whatever](#:jane "save:")` if we are in the bob section or `[whatever](#bob:jane "save:")` if not.

This works quite nicely (for headers, not minors) in terms of being an actual link when viewed on GitHub for on-minor references. For example, `[whatever](#bob-is-cool "save:")` will save the section "Bob is cool" and link to it.

Other literate program files

To use other literate program files, we use the load directive. We then use a double colon syntax to refer to it. Let's assume we have a file called `cool.md`. Then we can load it in a litpro document using `[great](cool.md "load:")` and then reference section `bob:jane` in it using `_"great::bob:jane"` The load syntax has the alias as the linkname (bit in square brackets) but it can be omitted. In either case, the filename itself can be referenced. That is, in the above example we can also use `_"cool.md::jane"`

Full Example

Here we have a full example with the different syntaxes being used. We will load from two files and save to three.

load.md is the main entry point. One would compile it with `litpro -s . load.md` The `-s` says to look for any other literate document to be loaded in the current directory; the default is `src`.

```
# Full HTML

Here we are building an HTML page. We will use a
[widget](load2.md "load:") that comes from another literate program.

    <html>
        <head>
            _"widget::files link"
        </head>
        <body>
            <h2> Widgets for everybody!</h2>
            _"sp-load2.md::html:top"
            <p> snuck something in! </p>
            _"widget::html:bottom"
        </body>
    </html>

[full.html](# "save:")
```

The above code loads up load2.md which has some nice bits in it.

```
# Widget

Here we define a widget. This widget will be amazing!

We will save two files from here

* [widget.js](#js "save:")
* [widget.css](#css-for-widget "save:")

## Files link

This is the loading tags for our pages

    <link rel="stylesheet" href="widget.css" />
```

```
    <script src="widget.js"></script>

## HTML

Here we put the html for our widget. We have a top and a bottom. The top
contains the div and a heading.  The minor block syntax can be either of the
below.

[top]()

    <div class="widget">
        <h2>Click a button</h2>

[bottom](# ":")

    <button>Awesome!</button>
    </div>

## JS

When the button is clicked, we add the big class to the h2 element in the
widget

    document.addEventListener("DOMContentLoaded", function () {
        _":add click"
    });

[add click]()

Here we add a click listener to the button. It should add a class to make the
text big and then one second later remove it.

    var button = document.querySelector(".widget button");
    var h2 = document.querySelector(".widget h2").classList;
    button.addEventListener("click", function () {
        h2.add("big");
        setTimeout(_":remove class", 1000);
    });

[remove class]()

This removes the big class
```

```
    function () {
        h2.remove("big");
    }

## CSS for widget

We want the h2 to have a red background!

    .widget h2 {
        background-color : red;
    }

    .big {
        font-size: 5em;
    }
```

When compiled, we get widget.js:

```
document.addEventListener("DOMContentLoaded", function () {
    var button = document.querySelector(".widget button");
    var h2 = document.querySelector(".widget h2").classList;
    button.addEventListener("click", function () {
        h2.add("big");
        setTimeout(function () {
            h2.remove("big");
        }, 1000);
    });
});
```

widget.css:

```
.widget h2 {
    background-color : red;
}

.big {
    font-size: 5em;
}
```

and full.html:

```
<html>
    <head>
        <link rel="stylesheet" href="widget.css" />
        <script src="widget.js"></script>
    </head>
    <body>
        <h2> Widgets for everybody!</h2>
        <div class="widget">
            <h2>Click a button</h2>
        <p> snuck something in! </p>
        <button>Awesome!</button>
        </div>
    </body>
</html>
```

4.2 Functions or reordering

A common point of debate is whether this reordering is significant. The argument is that in modern languages, the ability to define functions after they have been called allows one to shift the code elsewhere.

While true, this muddies the role of functions a bit. Functions are best used to be repeated bits of code that are called over and over. In old JavaScript, functions are also the only way to have scoped variables, but with the `let` keyword, that has changed. Both are considerations for the computer and are good reasons to use functions.

But to use functions solely for human readable purposes at the expense of increased complexity and cognitive load seems like a poor trade. Reordering using literate programming allows one complete freedom in how to cut up the code without any extra programming complexity. There is no wrestling with scope because of that reordering. There is no increased levels of redirection nor restrictions from redirecting from within the local code. It simply shifts the placement for the sole purpose of humans reading the code.

It is separation of concerns at its best. We have computer needs and we have human needs. We need to cater to both. Most paradigms try to balance the two needs simultaneously with various costs and trade-offs. This approach allows us to satisfy both constituents with the small price of an extra compile (or transpile) step.

5. Artistic programming

And so we reach the third and final level of our journey. The other two levels are pretty much complete in explaining them and how to use them. What follows for the rest of this tome is all about the third level.

The third level is the shedding of constraints. We take those snippets of code from the second level and we run them through code transformations. This enables us to do just about anything. Even more, we can call out to external programs, feeding the snippets in as standard input and receiving the output as the next form of the snippet.

This is the level of complete artistry. We can write little mini-languages to make efficient input. We can use a wide variety of text transformations of one language to the next all smoothly in the code. We can weave in user-end documentation, tests, sanity checks, compile contexts, and whatever else we can dream of, to make the job of coding, and all of its ancillary tasks, efficient and enjoyable.

The basic command syntax uses pipe syntax. Instead of just a reference, such as `_"name"`, we can pipe the incoming text into a command, or series of commands, such as `_"name | cat awesome, _"dude" | sub A, b"`. This would take the text in the reference `name` and send it through the command `cat` which concatenates stuff together. In particular it would join the incoming text with awesome and the stuff in the `dude` section. Then that resulting text goes through the sub command and all instances of `A` get replaced with `b`. Note that section names can be referenced in the command arguments.

Sub

To start with, let's say we want to create functions for the arithmetic operators. That is, we want to be able to write `add(4, 5)` instead of `4+5`. In JavaScript, we could do

```
var add = function (a, b) {
    return a + b;
}
```

And that would be fine. We could then do that for subtraction, multiplication, and division. But that is a bit repetitive. So instead we can do

```
# Operators

We want to convert the operators to functions.

We start with a basic form using addition

    var OP = function (a, b) {
        return a ?? b;
    }

 When done we will save it in [ops.js](#math-ops "save:")

## Math ops

Let's create the operators

    _"operators| sub OP, add, ??, +"
    _"operators| sub OP, mul, ??, *"
    _"operators| sub OP, div, ??, /"
    _"operators| sub OP, sub, ??, -"
```

And that generates the following in a file called ops.js.

```
var add = function (a, b) {
    return a + b;
}
var mul = function (a, b) {
    return a * b;
}
var div = function (a, b) {
    return a / b;
}
var sub = function (a, b) {
    return a - b;
}
```

This is not that exciting of an example. And there are a number of ways we could have approached this with a simple copy and paste being one of them. Indeed, the four lines of litpro code is copy and pasted (it is also possible to run some custom code to do this more elegantly, but this is a simple introduction).

But you may wonder what is the advantage? The creative effort is about the same level, but now all four are dependent on each other. This may be good, it may not be good. It is good if you later

decide to modify this. For example, you may want to check that the operators are numbers. Or you may want to round the result or cast it into some other form. By modifying the originating function, we get all of them modified at once. That is, the benefit is in the maintenance.

The downside is that they are coupled. So, for example if we want to inject a test for division by zero, we need to add something to facilitate that. One solution is to put in a comment line that gets replaced with what you need.

```
# Operators

We want to convert the operators to functions.

The ?? will be replaced with operator symbols. We check for something to be a
number and then do the computation. This is not really necessary nor perhaps
the best to do it, but there you have it.

    var OP = function (a, b) {
        if (Number.isNumber(a) && Number.isNumber(b) ) {
            //rep
            return a ?? b;
        } else {
            return NaN;
        }
    }

 When done we will save it in [ops2.js](#math-ops "save:")

## Math ops

Let's create the operators

    _"operators| sub OP, add, ??, +"
    _"operators| sub OP, mul, ??, *"
    _"operators| sub OP, div, ??, /, //rep, _"division by zero" "
    _"operators| sub OP, sub, ??, -"

## Division by zero

The division by zero is so bad so let's deal it. Let's be bad mathematical
people and give some values of oo, -oo, and 1

    if (b === 0) {
        if (a > 0) {
```

```
            return Infinity;
        } else if (a < 0) {
            return -Infinity;
        } else {
            return 1;
        }
    }
```

Turning into:

```
var add = function (a, b) {
    if (Number.isNumber(a) && Number.isNumber(b) ) {
        //rep
        return a + b;
    } else {
        return NaN;
    }
}
var mul = function (a, b) {
    if (Number.isNumber(a) && Number.isNumber(b) ) {
        //rep
        return a * b;
    } else {
        return NaN;
    }
}
var div = function (a, b) {
    if (Number.isNumber(a) && Number.isNumber(b) ) {
        if (b === 0) {
            if (a > 0) {
                return Infinity;
            } else if (a < 0) {
                return -Infinity;
            } else {
                return 1;
            }
        }
        return a / b;
    } else {
        return NaN;
    }
}
```

```
var sub = function (a, b) {
    if (Number.isNumber(a) && Number.isNumber(b) ) {
        //rep
        return a - b;
    } else {
        return NaN;
    }
}
```

This could easily become cumbersome. As is generally true with this style of programming, it is a matter of artistic choice as to which is the most useful and clear in any given situation. It is likely to evolve over time just as repeating similar code often evolves into pulling it out to functions. This is just one of the stops along the way.

5.1 Web

As another example, let's say we are creating a web page. We want to have the content written in markdown, but we want to create the structure in pug[1] (formerly the jade language). Let's presume that the commands to convert them are `md` and `pug`, respectively.

```
# Webpage

We are going to make a web page.

[web.html](#structure "save:|log | jade | compile structure ")

## Structure

We use the HTML5 elements of nav, article and aside in their appropriate
roles. The nav is very lame.

    nav
        .left Click
        .right Clack

    article \_":content |md "

    aside \_"announcements |md "

    script \_"js stuff | jshint "
```

[1] https://pugjs.org/

```
[content]()

Here we have some great body content

    # Best Webs Ever

    We have the best webs **ever**.

    ## Silk

    Yeah, we got spider silk. [Kevlar strength](kevlar.html). Real cool.


### Announcements

Whatever we want to say

    _Open Now!_

## js Stuff

Just some random javascript. The jshint will check it for errors and such.

    var a, b;
    if (a = b) {
        a = 2;
    } else {
        // never reached;
    }

Jshint just loves it when one uses assignment instead of equality testing in a
boolean expression.

Note that jshint still allows it to be built. It will just give a warning on
the command line.
```

This yields

```
<nav>
  <div class="left">Click</div>
  <div class="right">Clack</div>
</nav>
<article><h1>Best Webs Ever</h1>
<p>We have the best webs <strong>ever</strong>.</p>
<h2>Silk</h2>
<p>Yeah, we got spider silk. <a href="kevlar.html">Kevlar strength</a>. Real coo\
l.</p>
</article>
<aside><p><em>Open Now!</em></p>
</aside>
<script>var a, b;
if (a = b) {
    a = 2;
} else {
    // never reached;
}</script>
```

The easiest way to run this is with `literate-programming` which has both markdown and pug included.

But to run this with `litpro`, one needs to use the lprc.js file that defines these commands. There are other methods that we'll discuss later, but this is perhaps the most useful method.

The lprc.js file requires the modules markdown-it and pug. We also include jshint which was done to check the script that we snuck in there.

```
module.exports = function (Folder, args) {

    if (args.file.length === 0) {
        args.file = ["web.md"];
    }


    if (!Folder.prototype.local) {
        Folder.prototype.local = {};
    }

    require('litpro-jshint')(Folder, args);

    var jade = require('pug');
```

```
    Folder.sync("pug" , function (code, args) {
        options = args.join(",").trim();
        if (options) {
            options = JSON.parse(options);
        } else {
            options = {'pretty':true};
        }
        return jade.render(code, options);
    });

    var md = require('markdown-it')({
        html:true,
        linkify:true
    });


    Folder.prototype.local.md = md;

    Folder.sync( "md", function (code, args) {
        return  md.render(code);
    });

};
```

We will discuss a lot of what is going on in this example as time goes on. But notice how we can cut up our assembly of the material, run them through commands, and get a nice compiled output.

The backslashes are escaping the substitution in the first round as mixing pug with compiled HTML content does not lead to good results. So we compile the pug and then when that is ready, we call the compile command to run that through our standard compiler. In this example, we need to include the structure named section as the overarching section.

II Cookbook

This is where we do a variety of specific examples of use. It mainly focuses on examples from the web stack. In particular, the programming is almost entirely JavaScript.

Constants

We can define constants once with explanations and then put them into whatever files need them.

Eval

How to run some JavaScript

Boilerplate

If you have need of boilerplate, then we can do that with some substitutions for templating it as well.

Making a Command

Commands are very useful. This is how to make them in a literate program.

Making a Directive

Inline custom directives are not that useful, but it can be insightful in going towards making plugins.

Project Files

Often there are other files needing managing for a project other than just code. For example, .gitignore, readme, package.json, … These can all live in a single literate programming document that generates these files. It could also be setup to have variables that you define once and then use when needed, such as the version number.

Making a lprc.js file

What constitutes a lprc.js file. This is where directives and commands are ideally added, either directly or from plugins.

Other languages

If you want to write in your own style or language or whatever, it can do that as well. Write in a way comfortable to you and have it compile it into any target language.

Making a Plugin

Making a plugin leads to good consistent behavior.

Data Entry

Dealing with little bits of data is a difficult problem. Here we look at inline data entry using a quick split style as well as reading in data from an external file. This pertains more to generating written output, then code itself, but it could be needed there as well such as if you want to pull in secrets from a file outside of the repo and put it in the compiled code that is also stored outside of the repo.

Conditionals

We may want to do one thing or another depending on command line options or based on some programmatic condition. We can do that.

Domain Specific Languages

We can write full blown domain specific languages and have them translated at build time. Or you can use standard languages that get transformed, such as markdown into html.

Making a Subcommand

These allow for doing complex actions in arguments. These are different from commands that expect to be a part of a pipe flow.

Linting

We can lint pieces of our code. It would even be possible to lint the pieces in isolation to see what shared variables might pop up.

h5 and h6 Headers

These are headers that generate path-escaped references using the most recent main block reference as the root path. These sections can not only be useful for commonly named parts, but can also have custom behavior automatically acting on the parts.

Testing

Testing should be easy. Here we give a strategy for testing little bits of code (unit testing) as well as testing the combined pieces (integrated testing).

Having fake data specified as well as expected output is also easy.

Documentation

While literate programming is designed to be the documentation for the maintenance programmer, there is still the little issue of user documentation. While that may call for a separate document, it could be included in the same literate program as well using the h5 and h6 mechanism, keeping it with the code.

Debugging

Debugging can be supported by conditional commands.

III Web Projects

The web is a particularly beautiful hot mess of tech stacks.

We first detail various useful parts related to particular pieces of the stack.

HTML

This is a language that covers both the structure of a webpage and the content. Combining the tasks lead to a language that is a bit painful to write out.

Litpro can help by offering boiler plate with subbing in of content. The boiler plate can be written in a structure focused (templating) language, such as pug, and the content can be written in markdown. Litpro can then transform and stitch together it all.

JS

Litpro is written in JavaScript and thus many of the common commands turn out to be quite useful in writing JavaScript.

There are a few very specific ways that can be very helpful.

The first is using JSHint. You can take any block of JS code and feed it into JSHint to get a log output of issues and problems. It is as easy as `js code | jshint`assuming you have installed litpro-jshint or using the full literate-programming.

Another handy command, in all versions, is `js-string`. This takes the text and breaks it by newline and outputs text suitable for creating a string in JS code that returns that text. This enables one to, say, write some HTML for templating and put it in there. In newer versions of JS, one can have multi-line strings, but this works for all versions of JS.

There is also the `ife` command which creates an immediate function execution around the incoming text. This is again less necessary in the newer variants, but it can still be handy for closures or if one does not want to use the new `let` version of `var`.

CSS

This is a language with great potential, but it is hampered by differing browser specs. While that has greatly improved, it is best to let a tool handle the differences. There can also be a lot of repetition at times. Both of these things Litpro can help with.

One can take any incoming text and run it through a preprocessor such as SASS or run it through PostCSS and the wonderful auto-prefixer. The latter is included in literate-programming.

One can also use some snippets or the `caps` command to help ease some of the more verbose parts of the syntax.

One can also do variables with the substitutions. This may help avoid using something like SASS entirely.

Tidy and Minify

Literate-programming includes tools for formatting the above three languages, either to make them pretty (tidy) or to reduce their size (minify).

One can have multiple directories generated say, one for development and one for production. Using the `cd` command, this is often quite convenient.

It is also possible to include some code in one, but not the other. Again, this comes to transforming the blocks of text.

Servers

In addition to the static content served up, one often needs dynamic interactions and thus writing a server and using a database comes up.

The same tools above for writing JavaScript are just as helpful with node. In particular, having development and production splits can be very useful.

There is also the database. For a database, one has the language of the server and the database language. Plus, one may have schemas to guide the development, perhaps even a specific language.

This tool may help with that by offering a mapping between the languages. Unlike the magic tools of ORM, this tool does the conversion and allows one to see the output in context.

AWS

AWS is an amazing land of tools. It is all very piecemeal and run by resource policies. There are a lot of configurations to handle and updating requires a process.

It seems like a perfect place for a text transformation tool to run.

writeweb

This is an attempt by the author to create automatic compiling of literate programs. Due to their insecure nature, this would be something that would need to be installed by an individual (or team).

static site

This is where we show how to setup a static site, one in which the content and individual page styling/js is done in a directory of files that are not literate programs, but a literate program processes them.

That is, we make a static site generator that runs as a literate program.

IV Reference

This part is the technical documentation of literate programming. While it does document itself, in some sense, this is a ready index of syntax, directives, commands, subcommands, and plugins.

Most of this applies to literate-programming core library, but there are directives and commands that may only be applicable in the command client. Those are marked in their own section, either with Litpro (thin and fat client applicable) or Literate-Programming (just the fat client)

Syntax

Here we describe the syntax as formally as possible.

Commonmark

This is the markdown specification and parser that we use to determine when something is a header, a code block or link. The rest of the syntax is not relevant for our purposes though it may be of relevance to readers.

Code blocks

Our main concern is to transform and string together code blocks. A code block is something that essentially is indented by 4 spaces or is fenced off (3 backticks before and after).

Minor Blocks

A minor block is a block that is being referenced by two parts: a header block name and, after a colon, the minor block's name.

References and Substitutions

References are what we use to refer to the chunks of code, or rather, what we call the names of those chunks.

Substitutions are the syntax saying that something else should go there. Only substitutions and their escaped versions cause a change in the code chunks.

Substitutions can have null references. The point of these would be the pipes and commands.

References can occur before the first pipe or as part of a command or as part of a directive.

Pipes

These initiate commands that transform the incoming text.

Directive Syntax

This is a link syntax with the title text starting with a word followed by a colon. Each part of the link may or may not have relevance to the directive.

Command Syntax

No parentheses required for command arguments.

Subcommand Syntax

These are functions in the arguments of a command. These require parentheses.

Directives

Directives are generally intended for using text in a certain way, such as saving it to a file or defining a command with it. There is also the eval syntax which will act immediately upon being encountered.

Commands

Much of the heart of the transformations are commands and there are a great many commands that come by default.

Subcommands

These help in getting the right kind of arguments into a command. One can create arrays, objects, booleans, numbers, etc., as well as manipulate such structures.

Command-line

The command line programs have flags that we explain here.

Plugins

Plugins are a way to have commonly used function easily packaged up and used.

lprc.js

This is a file that gets executed upon startup. It is the place to load up plugins, define commands, directives, and defaults.

Compile Events

Here we detail some of the parts of the program that one may want to access for a variety of reasons. Further details can be read in the literate source code, mainly in that of literate-programming-lib[2].

[2] https://github.com/jostylr/literate-programming-lib

V The Literate Programs

Here we give descriptions and links to all the various pieces of the literate program family. They are all written in a literate fashion and the source code is freely available at their repositories, listed below.

All of this relies on node.js[3].

To understand these tools, the most interesting repositories are those for the underlying library, literate-programming-lib, and the event library, event-when.

litpro

This is our main tool. It is a command-line tool. It can be installed globally or locally into a repository. Use `npm install litpro -g` to install it globally.

- github[4] is where the code lives
- npm[5] is the package.

literate-programming

This is the fat command-line client. It can do everything that litpro can do, but it also comes bundled with pug, markdown-it, postcss, tidy, minifiers. This is a "batteries loaded" tool for web development.

- github[6] is where the code lives
- npm[7] is the package.

This can also be installed globally with `npm install literate-programming -g`. The command is then `literate-programming`.

[3] https://nodejs.org/en/
[4] https://github.com/jostylr/litpro
[5] https://www.npmjs.com/package/litpro
[6] https://github.com/jostylr/literate-programming
[7] https://www.npmjs.com/package/literate-programming

litpro-jshint

If you all you need is jshint, you might want to install the module litpro-jshint instead of the full one.

- github[8] is where the code lives
- npm[9] is the package.

literate-programming-cli

This is an intermediate project between the library and litpro. This is where most of the command line client code is written.

- github[10] is where the code lives
- npm[11] is the package.

literate-programming-cli-test

This is our test framework for the command line client. Basically, we have a directory with the files to process, a canonical directory showing what we expect to be in the build directory.

- github[12] is where the code lives
- npm[13] is the package.

literate-progamming-lib

This is the place to read the literate program construction in detail. Start with project.md and migrate to the other files in the structure. Parsing and stitching are the two main algorithmic pages.

- github[14] is where the code lives
- npm[15] is the package.

[8] https://github.com/jostylr/literate-programming
[9] https://www.npmjs.com/package/literate-programming
[10] https://github.com/jostylr/literate-programming-cli
[11] https://www.npmjs.com/package/literate-programming-cli
[12] https://github.com/jostylr/literate-programming-cli-test
[13] https://www.npmjs.com/package/literate-programming-cli-test
[14] https://github.com/jostylr/literate-programming-lib
[15] https://www.npmjs.com/package/literate-programming-lib

event-when

This is an event library that allows one to wait for multiple events with ease. We use it, for example, to assemble a variety of substitutions into a single code block. The return of that code block must wait for when the events of the other blocks being subbed in have fired.

- github[16] is where the code lives
- npm[17] is the package.

[16] https://github.com/jostylr/event-when
[17] https://www.npmjs.com/package/event-when