Literate Programming Preprocessor

Will Ware <wware@alum.mit.edu>

1 Literate programming preprocessor

Knuth's original vision for literate programming involved a single source file that could be processed in two different ways to produce compilable code on one pathway and a TeX/LaTeX source doc on the other. This approach diverges from that, using two sources, one of them an unmodified source code file.

There are a few reasons for this divergence. First, the large volume of legacy code that hadn't yet been written in Knuth's day. Second, the large number of very useful tools that now exist and that assume their input is a normal source code file, not some squirrely predecessor to a source code file. Third, modern engineers don't want to learn some new scheme of deriving their source code from some other document format.

So the first source for this scheme is a normal source code file, with no modifications whatsoever on behalf of this approach. There are no extra comments or tags or markup of any sort. Zero impact on source code is a hard requirement of this approach.

Also, the preprocessor should be as agnostic as possible about how documents are processed. LaTeX, Markdown, HTML, and any other format should work with minimal distraction. Currently there are examples for LaTeX and Markdown.

To include a code snippet into a document, you use a sequence of regular expresssions. The last two in the sequence specify the starting and ending line of the snippet and the earlier regexes are steps to get you to the staring line. For instance you get to a particular method in a particular class in a particular file with something like this, where the "@" sign is the first character in the line, signaling the preprocessor that this line is to be preprocessed.

```
@foobar.py:class Foobar/def foo\(self\)/x = 3/print x
```

produces

```
x = 3
print "yup"
print x
```

To generate a PDF from this LaTeX source, run ”make”.