```
 1| #! /usr/bin/env raku
 2|
 3| # Get the Pod vs. Code structure of a Raku/Pod6 file.
 4| # © 2023 Shimon Bollinger. All rights reserved.
 5| # Last modified: Sat 02 Sep 2023 09:42:13 PM EDT
 6| # Version 0.0.1
 7|
 8| # no-weave
 9| # always use the latest version of Raku
10| use v6.*;
11| use PrettyDump;
12| use Data::Dump::Tree;
13| # end-no-weave
14|
```

A grammar to parse a file into `Pod` and `Code` sections.

INTRODUCTION

I want to create a semi-literate Raku source file with the extension `.sl`. Then, I will *weave* it to generate a readable file in formats like Markdown, PDF, HTML, and more. Additionally, I will *tangle* it to create source code without any Pod6.

To do this, I need to divide the file into `Pod` and `Code` sections by parsing it. For this purpose, I will create a dedicated Grammar.

Convenient tokens

Let's create two tokens for convenience.

```
15|
16| #    We need to declare them with C<my> because we
17| #    need to use them in a subroutine later. #TODO explain why.
18|
19|     my token rest-of-line {    \N* [\n | $]  }
20|     my token ws-till-EOL  {    \h* [\n | $]  }
21|     my token blank-line   { ^^ <ws-till-EOL> }
22|
```

The Grammar

Our file will exclusively consist of `Pod` or `Code` sections, and nothing else. The `TOP` token clearly indicates this.

```
   23|
   24| #use Grammar::Tracer;
   25| grammar Semi::Literate is export {
   26|     token TOP {   [ <pod> | <code> ]* }
   27|
   28|
```

The Pod6 delimiters

According to the documentation,

Every Pod6 document has to begin with =begin pod 1 and end with =end pod.

So let's define those tokens.

The `begin` token

```
   29|
   30|     my token begin {
   31|         ^^ \h* \= begin <.ws> pod
   32|
```

Most programming applications do not focus on the structure of the executable file, which is not meant to be easily read by humans.

However, we can provide the option for users to specify the number of empty lines that should replace a `pod` block. To do this, simply add a number at the end of the `=begin` directive. For example, `=begin pod 2`. [1]

```
   33|
   34|         [ \h* $<num-blank-lines>=(\d+) ]?  # an optional number to specify the
   35|                                            # number of blank lines to replace the
   36|                                            # C<Pod> blocks when tangling.
```

The remainder of the `begin` directive can only be whitespace.

```
   37|
   38|         <ws-till-EOL>
   39|     } # end of my token begin
   40|
```

The `end` token

The `end` token is much simpler.

```
   41|
   42|     my token end { ^^ \h* \= end <.ws> pod <ws-till-EOL> }
   43|
```

---

[1] This is non-standard Pod6 and will not compile until woven!

The `Pod` token

Within the delimiters, all lines are considered documentation. We will refer to these lines as `plain-lines`. Additionally, it is possible to have nested `Pod` sections. This allows for a hierarchical organization of documentation, allowing for more structured and detailed explanations.

It is also permissible for the block to be empty. Therefore, we will use the 'zero-or-more' quantifier on the lines of documentation, allowing for the possibility of having no lines in the block.

```
44|
45|     token pod {
46|         <begin>
47|             [<pod> | <plain-line>]*
48|         <end>
49|     } # end of token pod
50|
```

The `Code` token

The `Code` sections are similarly easily defined. There are two types of `Code` sections, depending on whether they will appear in the woven code. See no-weave for why some code would not be included in the woven code.

```
51|
52|     token code {
53|         | <woven>
54|         | <non-woven>
55|     } # end of token code
```

Woven sections

These sections are trivially defined. They are just one or more `plain-lines`.

```
56|
57|     token woven { <plain-line>+ }
```

Non-woven sections

Sometimes there will be code you do not want woven into the document, such as boilerplate code like `use v6.d;`. You have two options to mark such code. By individual lines or by delimited blocks of code.

```
58|
59|     token non-woven {
60|         | <one-line-no-weave>+
61|         | <delimited-no-weave>+
62|     } # end of token non-woven
```

One line of code

Simply append `#no-weave` at the end of the line!

```
63|
64|     token one-line-no-weave {
65|         ^^ \N*
66|         '#' <.ws> 'no-weave'
67|         <.ws> <rest-of-line>
68|     } # end of token one-line-no-weave
```

```
   69|
```

Delimited blocks of code

Simply add comments `#no-weave` and `#end-no-weave` before and after the code you want ignored in the formatted document.

```
   70|     token delimited-no-weave {
   71|         <begin-no-weave>
   72|             <plain-line>*?
   73|         <end-no-weave>
   74|     } # end of token delimited-no-weave
   75|
   76|     token begin-no-weave {
   77|         ^^ \h*                      # optional leading whitespace
   78|         '#' <.ws> 'no-weave'        # the delimiter itself (#no-weave)
   79|         <.ws> <rest-of-line>        # optional trailing whitespace
   80|     } # end of token <begin-no-weave>
   81|
   82|     token end-no-weave {
   83|         ^^ \h*                      # optional leading whitespace
   84|         '#' <.ws> 'end-no-weave'    # the delimiter itself (#end-no-weave)
   85|         <.ws> <rest-of-line>        # optional trailing whitespace
   86|     } # end of token <end--no-weave>
   87|
```

The `plain-line` token

The `plain-line` token is, really, any line at all...

```
   88|
   89|     token plain-line {
   90|        $<plain-line> = [^^ <rest-of-line>]
   91|
```

Disallowing the delimiters in a `plain-line`.

... except for one subtlety. They it can't be one of the begin/end delimiters. We can specify that with a Regex Boolean Condition Check.

```
   92|
   93|         <?{ &not-a-delimiter($<plain-line>.Str) }>
   94|     } # end of token plain-line
   95|
```

This function simply checks whether the `plain-line` match object matches either the `begin` or `end` token.

Incidentally, this function is why we had to declare those tokens with the `my` keyword. This function wouldn't work otherwise.

```
   96|
   97|     sub not-a-delimiter (Str $line --> Bool) {
   98|         return not $line ~~ /<begin> | <end>/;
   99|     } # end of sub not-a-delimiter (Match $line --> Bool)
  100|
```

And that concludes the grammar for separating `Pod` from `Code`!

```
  101|
```

```
102| } # end of grammar Semi::Literate
103|
```

The Tangle subroutine

This subroutine will remove all the Pod6 code from a semi-literate file (`.sl`) and keep only the Raku code.

```
104|
105| #TODO multi sub to accept Str & IO::PatGh
106| sub tangle (
107|
```

The subroutine has a single parameter, which is the input filename. The filename is required. Typically, this parameter is obtained from the command line or passed from the subroutine `MAIN`.

```
108|     Str $input-file!,
```

The subroutine will return a `Str`, which will be a working Raku program.

```
109|         --> Str ) is export {
```

First we will get the entire Semi-Literate `.sl` file...

```
110|
111|     my Str $source = $input-file.IO.slurp;
112|
```

Clean the source

Remove the *no-weave* delimiters

```
113|
114|     $source ~~ s:g{ ^^ \h* '#' <.ws>     'no-weave' <rest-of-line> } = '';
115|     $source ~~ s:g{ ^^ \h* '#' <.ws> 'end-no-weave' <rest-of-line> } = '';
116|
```

Remove unnecessary blank lines

Very often the `code` section of the Semi-Literate file will have blank lines that you don't want to see in the tangled working code. For example:

```
                                                # <== unwanted blank lines
                                                # <== unwanted blank lines
  sub foo () {
      { ... }
  } # end of sub foo ()
                                                # <== unwanted blank lines
                                                # <== unwanted blank lines
```

```
117|
```

So we'll remove the blank lines at the beginning and end of the Pod6 sections.

```
118|
119|     $source ~~ s:g/\=end (\N*)\n+/\=end$0\n/;
120|     $source ~~ s:g/\n+\=begin    /\n\=begin/;
121|
```

The interesting stuff

We parse it using the `Semi::Literate` grammar and obtain a list of submatches (that's what the `caps` method does) ...

```
122|
123|     my Pair @submatches = Semi::Literate.parse($source).caps;
124|
```

...and iterate through the submatches and keep only the `code` sections...

```
125|
126|     my Str $raku-code = @submatches.map( {
127|         when .key eq 'code' {
128|             .value;
129|         }
130|
```

Replace Pod6 sections with blank lines

#TODO rewrite Most programming applications do not focus on the structure of the executable file, which is not meant to be easily read by humans.

However, we can provide the option for users to specify the number of empty lines that should replace a `pod` block. To do this, simply add a number at the end of the `=begin` directive. For example, `=begin pod 2` .

```
131|
132|
133|         when .key eq 'pod' {
134|             my $num-blank-lines = .value.hash<begin><num-blank-lines>;
135|             "\n" x $num-blank-lines with $num-blank-lines;
136|         }
137|
138|         #no-weave
139|         default { die 'Should never get here' }
140|         #end-no-weave
```

... and we will join all the code sections together...

```
141|
142|     } # end of my Str $raku-code = @submatches.map(
143|     ).join;
144|
```

remove blank lines at the end

```
145|
146|     $raku-code ~~ s{\n  <blank-line>* $ } = '';
147|
```

And that's the end of the `tangle` subroutine!

```
148|     return $raku-code;
149| } # end of sub tangle (
150|
```

The Weave subroutine

The `Weave` subroutine will *weave* the `.sl` file into a readable Markdown, HTML, or other format. It is a little more complicated than `sub tangle` because it has to include the `code` sections.

```
151|
152| sub weave (
153|
```

The parameters of Weave

`sub weave` will have several parameters.

`$input-file`

The input filename is required. Typically, this parameter is obtained from the command line through a wrapper subroutine `MAIN`.

```
154|
155|     Str $input-file!;
```

`$format`

The output of the weave can (currently) be Markdown, Text, PDF or HTML. (Assuming you have the necessary `Pod::To::X` module installed.) It defaults to Markdown. The variable is case-insensitive, so 'markdown' also works.

```
156|
157|     Str :f(:$format) is copy = 'markdown';
158|         #= The output format for the woven file.
159|
```

`$line-numbers`

It can be useful to print line numbers in the code listing. It defaults to True.

```
160|
161|     Bool :l(:$line-numbers)  = True;
162|         #= Should line numbers be added to the embeded code?
163|
164|
```

`sub weave` returns a Str.

```
165|
166|         --> Str ) is export {
```

The alogrithm

```
167|
168|     my UInt $line-number = 1;
169|
```

First we will get the entire `.sl` file...

```
170|
```

```
171|     my Str $source = $input-file.IO.slurp;
172|
173|     my Str $cleaned-source;
174|
175| $cleaned-source = $source;
176| #=begin pod 1
177| #
178| #=head3 Remove full comment lines followed by blank lines
179| #
180| #=end pod
181| #
182| #    # delete full comment lines
183| #    $source ~~ s:g{ ^^ \h* '#' \N* \n+} = '';
184| #
185| #    # remove Raku comments, unless the '#' is escaped with
186| #    # a backslash or is in a quote. (It doesn't catch all quote
187| #    # constructs...(that's a TODO))
188| #    # And leave the newline.
189| #
190| #=begin pod 1
191| #
192| #=head3 Remove EOL comments
193| #
194| #=end pod
195| #
196| #    for $source.split("\n") -> $line {
197| #        my $m = $line ~~ m{
198| #                ^^
199| #               $<stuff-before-the-comment> = ( \N*? )
200| #
201| #                #TODO make this more robust - allow other delimiters, take into
202| #                #account the Q language, heredocs, nested strings...
203| #                <!after         # make sure the '#' isn't in a string
204| #                    ( [
205| #                        | \\
206| #                        | \" <-[\"]>*
207| #                        | \' <-[\']>*
208| #                        | \□ <-[\□]>*
209| #                    ] )
210| #                >
211| #                "#"
212| #
213| #
214| #                # We need to keep these delimiters.
215| #                # See the section above "Remove code marked as 'no-weave'".
216| #                <!before
217| #                      [
218| #                        | 'no-weave'
219| #                        | 'end-no-weave'
220| #                      ]
221| #                >
222| #                \N*
223| #                $$ };
224| #
225| #        $cleaned-source ~= $m ?? $<stuff-before-the-comment> !! $line;
226| #        $cleaned-source ~= "\n";
227| #    } # end of for $source.split("\n") -> $line
228| #
```

```
#=begin pod 1
#=head3 Remove blank lines at the begining and end of the code
#
#B<EXPLAIN THIS!>
#
#=end pod
#
#    $cleaned-source ~~ s:g{\=end (\N*)\n+} =   "\=end$0\n";
#    $cleaned-source ~~ s:g{\n+\=begin (<.ws> pod) [<.ws> \d]?} = "\n\=begin$0";
#
```

Interesting stuff ...Next, we parse it using the `Semi::Literate` grammar and obtain a list of submatches (that's what the `caps` method does) ...

```

    my Pair @submatches = Semi::Literate.parse($cleaned-source).caps;

```

...And now begins the interesting part. We iterate through the submatches and insert the `code` sections into the Pod6...

```


    my Str $weave = @submatches.map( {
        when .key eq 'pod' {
            .value
        } # end of when .key
```

#TODO

```

        when .key eq 'code' { pd $_; qq:to/EOCB/; }
            \=begin  pod
            \=begin  code :lang<raku>
             { my $fmt = ($line-numbers ?? "%3s| " !! '') ~ "%s\n";
                .value
                .lines
                .map($line-numbers
                        ?? {"%4s| %s\n".sprintf($line-number++, $_) }
                        !! {     "%s\n".sprintf(                $_) }
                    )
                .chomp;
             }
            \=end  code
            \=end  pod
            EOCB

        when .key eq 'non-woven' {
            note "Inside non-woven"
            ; # do nothing
        } # end of when .key eq 'non-woven'

        # no-weave
        default { die 'Should never get here.' }
        # end-no-weave
    } # end of my $weave = Semi::Literate.parse($source).caps.map
    ).join;
```

```
275|
```

remove useless Pod directives

```
276|
277|     $weave ~~ s:g{ \h* \=end   <.ws> pod  <rest-of-line>
278|                     \h* \=begin <.ws> pod <rest-of-line> } = '';
279|
```

remove blank lines at the end

```
280|
281|     $weave ~~ s{\n  <blank-line>* $ } = '';
282|
```

And that's the end of the `tangle` subroutine!

```
283|
284|     return $weave
285| } # end of sub weave (
286|
```

NAME

`Semi::Literate` - A semi-literate way to weave and tangle Raku/Pod6 source code.

VERSION

This documentation refers to `Semi-Literate` version 0.0.1

SYNOPSIS

```
use Semi::Literate;
# Brief but working code example(s) here showing the most common usage(s)

# This section will be as far as many users bother reading
# so make it as educational and exemplary as possible.
```

DESCRIPTION

`Semi::Literate` is based on Daniel Sockwell's Pod::Literate module

A full description of the module and its features. May include numerous subsections (i.e. =head2, =head2, etc.)

BUGS AND LIMITATIONS

There are no known bugs in this module. Patches are welcome.

AUTHOR

Shimon Bollinger (deoac.bollinger@gmail.com)

LICENSE AND COPYRIGHT

© 2023 Shimon Bollinger. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Raku itself. See The Artistic License 2.0.

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.

```
  287|
  288| # no-weave
  289| my %*SUB-MAIN-OPTS =
  290|   :named-anywhere,            # allow named variables at any location
  291|   :bundling,                  # allow bundling of named arguments
  292| #  :coerce-allomorphs-to(Str),  # coerce allomorphic arguments to given type
  293|   :allow-no,                  # allow --no-foo as alternative to --/foo
  294|   :numeric-suffix-as-value,   # allow -j2 as alternative to --j=2
  295| ;
  296|
  297| #| Run with option '--pod' to see all of the Pod6 objects
  298| multi MAIN(Bool :$pod!) is hidden-from-USAGE {
  299|     for $=pod -> $pod-item {
  300|         for $pod-item.contents -> $pod-block {
  301|             $pod-block.raku.say;
  302|         }
  303|     }
  304| } # end of multi MAIN (:$pod)
  305|
  306| #| Run with option '--doc' to generate a document from the Pod6
  307| #| It will be rendered in Text format
  308| #| unless specified with the --format option.  e.g.
  309| #|       --doc --format=HTML
  310| multi MAIN(Bool :$doc!, Str :$format = 'Text') is hidden-from-USAGE {
  311|     run $*EXECUTABLE, "--doc=$format", $*PROGRAM;
  312| } # end of multi MAIN(Bool :$man!)
  313|
  314| my $semi-literate-file =
'/Users/jimbollinger/Documents/Development/raku/Projects/Semi-Literate/source/Literate.sl';
  315| multi MAIN(Bool :$testt!) {
  316|     say tangle($semi-literate-file);
  317| } # end of multi MAIN(Bool :$test!)
  318|
  319| multi MAIN(Bool :$testw!) {
  320|     say weave($semi-literate-file);
  321| } # end of multi MAIN(Bool :$test!)
  322|
  323| #end-no-weave
```