```
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: Thu 07 Sep 2023 07:13:05 PM EDT
6| # Version 0.0.1
```

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 three tokens for convenience.

```
    7| #    We need to declare them with C<my> because we
    8| #    need to use them in a subroutine later. #TODO explain why.
    9|
   10|     my token rest-of-line {    \N* [\n | $]  }
   11|     my token ws-till-EOL  {    \h* [\n | $]  }
   12|     my token blank-line   { ^^ <ws-till-EOL> }
```

The Grammar

Our file will exclusively consist of `Pod` or `Code` sections, and nothing else. The `Code` sections are of two types, a) code that is woven into the documentation, and b) code that is not woven into the documentation. The `TOP` token clearly indicates this.

```
13| #use Grammar::Tracer;
14| grammar Semi::Literate is export {
15|     token TOP {
16|         [
17|           | <non-woven-code>
18|           | <pod>
19|           | <woven-code>
20|         ]*
21|     } # end of token TOP
```

The Pod6 delimiters

According to the documentation,

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

So let's define those tokens.

The `begin-pod` token

```
22|     token begin-pod {
23|         ^^ <.ws> '=' begin <.ws> pod
```

Most programming applications do not focus on the structure of the executable file, which is not meant to be easily read by humans. Our tangle would replace all the Pod6 blocks with a single `\n`. That can clump code together that is easier read if there were one or more blank lines.

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`. [1]

```
24|         [ <.ws> $<num-blank-lines>=(\d+) ]?  # an optional number to specify the
25|                                             # number of blank lines to replace the
26|                                             # C<Pod> blocks when tangling.
```

```
27|         <ws-till-EOL>
28|     } # end of token begin
```

The `end-pod` token

The `end-pod` token is much simpler.

```
29|     token end-pod { ^^ <.ws> '=' end <.ws> pod <ws-till-EOL> }
```

---

[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.

```
30|     token pod {
31|         <begin-pod>
32|             [<pod> | <plain-line>]*
33|         <end-pod>
34|     } # end of token pod
```

The `Code` tokens

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 below for why some code would not be included in the woven code.

Woven sections

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

```
35|     token woven-code { <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.

```
36|     token non-woven-code {
37|         [
38|           | <one-line-no-weave>
39|           | <delimited-no-weave>
40|         ]+
41|     } # end of token non-woven
```

One line of code

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

```
42|     token one-line-no-weave {
43|         ^^ \N*?
44|         '#' <.ws> 'no-weave-this-line'
45|         <ws-till-EOL>
46|     } # end of token one-line-no-weave
```

Delimited blocks of code

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

```
47|     token begin-no-weave {
48|         ^^ <.ws>                    # optional leading whitespace
49|         '#' <.ws> 'begin-no-weave'  # the delimiter itself (# begin-no-weave)
```

```
50|         <ws-till-EOL>               # optional trailing whitespace or comment
51|     } # end of token <begin-no-weave>
52|
53|     token end-no-weave {
54|         ^^ <.ws>                    # optional leading whitespace
55|         '#' <.ws> 'end-no-weave'    # the delimiter itself (#end-no-weave)
56|         <ws-till-EOL>               # optional trailing whitespace or comment
57|     } # end of token <end--no-weave>
58|
59|     token delimited-no-weave {
60|         <begin-no-weave>
61|             <plain-line>*
62|         <end-no-weave>
63|     } # end of token delimited-no-weave
```

The `plain-line` token

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

```
64|     token plain-line {
65|         :my $*EXCEPTION = False;
66|         [
67|           ||  <begin-pod>         { $*EXCEPTION = True }
68|           ||  <end-pod>           { $*EXCEPTION = True }
69|           ||  <begin-no-weave>    { $*EXCEPTION = True }
70|           ||  <end-no-weave>      { $*EXCEPTION = True }
71|           ||  <one-line-no-weave> { $*EXCEPTION = True }
72|           || $<plain-line> = [^^ <rest-of-line>]
73|         ]
74|         <?{ !$*EXCEPTION }>
75|     } # end of token plain-line
```

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

```
76| } # end of grammar Semi::Literate
```

The Tangle subroutine

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

```
77| #TODO multi sub to accept Str & IO::PatGh
78| sub tangle (
```

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

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

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

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

```
81|     my Str $source = $input-file.IO.slurp;
```

Clean the source

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
```

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

```
82|     $source ~~ s:g/\=end (\N*)\n+/\=end$0\n/;
83|     $source ~~ s:g/\n+\=begin    /\n\=begin/;
```

The interesting stuff

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

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

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

```
85| #    note "submatches.elems: {@submatches.elems}";
86|     my Str $raku-code = @submatches.map( {
87| #        note .key;
88|         when .key eq 'woven-code'|'non-woven-code' {
89|             .value;
90|         }
```

Replace Pod6 sections with blank lines

```
   91|         when .key eq 'pod' {
   92|             my $num-blank-lines = .value.hash<begin-pod><num-blank-lines>;
   93|             "\n" x $num-blank-lines with $num-blank-lines;
   94|         }
```

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

```
   95|     } # end of my Str $raku-code = @submatches.map(
   96|     ).join;
```

Remove the *no-weave* delimiters

```
   97|     $source ~~ s:g{ ^^ \h*   '#' <.ws>     'begin-no-weave' <rest-of-line> } = '';
   98|     $source ~~ s:g{ ^^ (.*?) '#' <.ws>     'begin-no-weave' <rest-of-line> } = "$0\n";
   99|     $source ~~ s:g{ ^^ \h*   '#' <.ws> 'end-no-weave' <rest-of-line> } = '';
```

remove blank lines at the end

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

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

```
  101|     return $raku-code;
  102| } # end of sub tangle (
```

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.

```
103| sub weave (
```

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`.

```
104|     Str $input-file!;
```

`$format`

The output of the weave can (currently) be Markdown, Text, or HTML. It defaults to Markdown. The variable is case-insensitive, so 'markdown' also works.

```
105|     Str :f(:$format) is copy = 'markdown';
106|         #= The output format for the woven file.
```

`$line-numbers`

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

```
107|     Bool :l(:$line-numbers)  = True;
108|         #= Should line numbers be added to the embeded code?
```

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

```
110|     my UInt $line-number = 1;
```

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

Remove blank lines at the begining and end of the code

EXPLAIN THIS!

```
112|     my Str $cleaned-source = $source;
113|     $cleaned-source ~~ s:g{\=end (\N*)\n+} =   "\=end$0\n";
114|     $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) ...

```
115|     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...

```
116| #    note "weave submatches.elems: {@submatches.elems}";
117| #    note "submatches keys: {@submatches».keys}";
118|     my Str $weave = @submatches.map( {
119|         when .key eq 'pod' {
120|             .value
121|         } # end of when .key
```

```
122|         when .key eq 'woven-code' { qq:to/EOCB/; }
123|             \=begin pod
124|             \=begin code :lang<raku>
125|              { my $fmt = ($line-numbers ?? "%3s| " !! '') ~ "%s\n";
126|                 .value
127|                 .lines
128|                 .map($line-numbers
129|                         ?? {"%4s| %s\n".sprintf($line-number++, $_) }
130|                         !! {     "%s\n".sprintf(                $_) }
131|                     )
132|                 .chomp;
133|              }
134|             \=end code
135|             \=end pod
136|             EOCB
137|
138|         when .key eq 'non-woven-code' {
139| #            note 'not-weaving';
140|           ''; # do nothing
141|         } # end of when .key eq 'non-woven'
```

```
142|     } # end of my $weave = Semi::Literate.parse($source).caps.map
143|     ).join;
```

remove blank lines at the end

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

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

```
145|     return $weave
146| } # end of sub weave (
```

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