```
1| use v6.*;
2| use PrettyDump;
3| use Data::Dump::Tree;
```

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.

```
    4|      my token rest-of-line {     \N* [\n | $] }
    5|      my token blank-line   { ^^ \h* [\n | $] }
```

The Grammar

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

```
    6| grammar Semi::Literate is export {
    7|     token TOP {   [ <pod> | <code> ]* }
```

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` token

```
    8|     my token begin {
    9|         ^^ \h* \= 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.

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]

```
   10|         [ \h* $<num-blank-lines>=(\d+) ]?
```

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

```
   11|         <rest-of-line>
   12|     }
```

The `end` token

The `end` token is much simpler.

```
   13|     my token end { ^^ \h* \= end <.ws> pod <rest-of-line> }
```

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.

```
   14|     token pod {
   15|         <begin>
   16|             [<pod> | <plain-line>]*
   17|         <end>
   18|     }
```

---

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

The `Code` token

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

```
   19|     token code { <plain-line>+ }
```

The `plain-line` token

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

```
   20|     token plain-line {
   21|        $<plain-line> = [^^ <rest-of-line>]
```

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.

```
   22|          <?{ &not-a-delimiter($<plain-line>.Str) }>
   23|     }
```

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.

```
   24|     sub not-a-delimiter (Str $line --> Bool) {
   25|         return not $line ~~ /<begin> | <end>/;
   26|     }
```

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

```
   27| }
```

The Tangle subroutine

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

```
28| sub tangle (
```

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`.

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

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

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

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

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

Remove the *no-weave* delimiters

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

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:

```
34|     sub foo () {
35|         { ... }
36|     }
```

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

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

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

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

...And now begins the interesting part. We iterate through the submatches and keep only the `code` sections...

```
40|     my Str $raku-code = @submatches.map( {
41|         when .key eq 'code' {
42|             .value;
43|         }
```

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`.

```
44|         when .key eq 'pod' {
45|             my $num-blank-lines = .value.hash<begin><num-blank-lines>;
46|             with $num-blank-lines { "\n" x $num-blank-lines }
47|         }
```

```
48|         default { die 'Should never get here' }
```

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

```
    }
    ).join;
```

remove blank lines at the end

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

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

```
    return $raku-code;
}
```

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.

```
   54| 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`.

```
   55|     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.

```
   56|     Str :f(:$format) is copy = 'markdown';
```

`$line-numbers`

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

```
   57|     Bool :l(:$line-numbers)  = True;
```

`sub weave` returns a Str.

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

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

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

```
   60|     my Str $source = $input-file.IO.slurp;
   61|
   62|     my Str $cleaned-source;
```

Clean the source of items we don't want to see in the formatted document.

Remove code marked as 'no-weave'

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.

Delimited blocks of code

Simply add comments before and after the code you want ignored in the formatted document.

```
    {...
    ...}
```

Remove full comment lines followed by blank lines

```
63|     $source ~~ s:g{ ^^ \h* '#' \N* \n+} = '';
```

Remove EOL comments

```
64|     for $source.split("\n") -> $line {
65|         my $m = $line ~~ m{
66|                 ^^
67|                $<stuff-before-the-comment> = ( \N*? )
68|
69|                 <!after
70|                     ( [
71|                         | \\
72|                         | \" <-[\"]>*
73|                         | \' <-[\']>*
74|                         | \□ <-[\□]>*
75|                     ] )
76|                 >
77|                 "#"
78|
79|
80|                 <!before
81|                       [
82|                         | 'no-weave'
83|                         | 'end-no-weave'
84|                       ]
85|                 >
86|                 \N*
87|                 $$ };
88|
89|         $cleaned-source ~= $m ?? $<stuff-before-the-comment> !! $line;
90|         $cleaned-source ~= "\n";
91|     }
```

Remove blank lines at the begining and end of the code

EXPLAIN THIS!

```
92|     $cleaned-source ~~ s:g{\=end (\N*)\n+} =   "\=end$0\n";
93|     $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) ...

```
94|     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...

```
95|     my Str $weave = @submatches.map( {
96|         when .key eq 'pod' {
```

```
   97|             .value
   98|         }
```

```
   99|         when .key eq 'code' { qq:to/EOCB/; }
  100|             \=begin  pod
  101|             \=begin  code :lang<raku>
  102|              { my $fmt = ($line-numbers ?? "%3s| " !! '') ~ "%s\n";
  103|                 .value
  104|                 .lines
  105|                 .map($line-numbers
  106|                         ?? {"%4s| %s\n".sprintf($line-number++, $_) }
  107|                         !! {     "%s\n".sprintf(                $_) }
  108|                     )
  109|                 .chomp;
  110|              }
  111|             \=end  code
  112|             \=end  pod
  113|             EOCB
  114|
  115|         default { die 'Should never get here.' }
  116|     }
  117|     ).join;
```

remove useless Pod directives

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

remove blank lines at the end

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

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

```
  121|     return $weave
  122| }
```

NAME

`Semi::Literate` - Get the Pod vs Code structure from a Raku/Pod6 file.

VERSION

This documentation refers to `Semi-Literate` version 0.0.1

SYNOPSIS

```
use Semi::Literate;
```

DESCRIPTION

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


```
  123| my %*SUB-MAIN-OPTS =
  124|   :named-anywhere,
  125|   :bundling,
  126|   :allow-no,
  127|   :numeric-suffix-as-value,
  128| ;
  129|
  130| multi MAIN(Bool :$pod!) is hidden-from-USAGE {
  131|     for $=pod -> $pod-item {
  132|         for $pod-item.contents -> $pod-block {
  133|             $pod-block.raku.say;
  134|         }
  135|     }
  136| }
  137|
  138| multi MAIN(Bool :$doc!, Str :$format = 'Text') is hidden-from-USAGE {
  139|     run $*EXECUTABLE, "--doc=$format", $*PROGRAM;
  140| }
  141|
  142| my $semi-literate-file =
'/Users/jimbollinger/Documents/Development/raku/Projects/Semi-Literate/source/Literate.sl';
  143| multi MAIN(Bool :$testt!) {
  144|     say tangle($semi-literate-file);
  145| }
  146|
  147| multi MAIN(Bool :$testw!) {
  148|     say weave($semi-literate-file);
  149| }
  150|
  151|
```