```
#! /usr/bin/env raku

# Get the Pod vs. Code structure of a Raku/Pod6 file.
# © 2023 Shimon Bollinger. All rights reserved.
# Last modified: Sun 10 Sep 2023 02:08:28 PM EDT
# Version 0.0.1

# begin-no-weave
# always use the latest version of Raku
use v6.*;
use PrettyDump;
use Data::Dump::Tree;
#end-no-weave
=begin pod
=comment 1


=TITLE A grammar to parse a file into C<Pod> and C<Code> sections.

=head1 INTRODUCTION

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

=head2 Convenient tokens

Let's create some tokens for convenience.

=end pod
#TODO Put these into a Role
    my token hws            {    <!ww>\h*       } # Horizontal White Space
    my token leading-ws     { ^^ <hws>          } # Whitespace at start of line
    my token optional-chars {    \N*?           }
    my token rest-of-line   {    \N*   [\n | $] } #no-weave-this-line
    my token ws-till-EOL    {    <hws> [\n | $] } #no-weave-this-line
    my token blank-line     { ^^ <ws-till-EOL>  } #no-weave-this-line
=begin pod
=comment 2
To do this, I need to divide the file into C<Pod> and C<Code> sections by parsing
it. For this purpose, I will create a dedicated Grammar.


=head1 The Grammar

=end pod
#use Grammar::Tracer;
grammar Semi::Literate is export {
=begin pod

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

=end pod
    token TOP {
        [
          || <pod>
          || <woven-code>
          || <non-woven-code>
        ]*
    } # end of token TOP
=begin pod
=comment 1

=head2 The Pod6 delimiters

According to the L<documentation|https://docs.raku.org/language/pod>,

=begin defn

    Every Pod6 document has to begin with =begin pod and end with =end pod.
=end defn
So let's define those tokens.
=head3 The C<begin-pod> token

=end pod
    token begin-pod {
        <leading-ws>
        '=' begin <hws> pod
        <ws-till-EOL>
    } # end of token begin-pod
=begin pod
=comment 1

=head3 The C<end-pod> token

The C<end-pod> token is much simpler.

=end pod
    token end-pod { <leading-ws> '=' end <hws> pod <ws-till-EOL> }
=begin pod
=comment 1

=head3 Replacing Pod6 sections with blank lines

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 C<\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 C<pod> block. To do this, simply add a Pod6 comment
immediately after the C<=begin  pod> statement.  The comment can say anything
you like, but must end with a digit specifying the number of blank lines with
which to replace the Pod6 section.

=end pod
    token num-blank-line-comment {
        <leading-ws>
        '=' comment
        <optional-chars>
        $<num-blank-lines> = (\d+)?
        <ws-till-EOL>
    } # end of token num-blank-line-comment
=begin pod
=comment 1

=head2 The C<Pod> token

Within the delimiters, all lines are considered documentation. We will refer to
these lines as C<plain-lines>. Additionally, it is possible to have nested
C<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.

=end pod
    token pod {
        <begin-pod>
        <num-blank-line-comment>?
            [<pod> || <plain-line>]*
        <end-pod>
    } # end of token pod
=begin pod
=comment 1

=head2 The C<Code> tokens

The C<Code> sections are similarly easily defined.  There are two types of
C<Code> sections, depending on whether they will appear in the woven code. See
L<below> for why some code would not be included in the woven
code.

=head3 Woven sections

These sections are trivially defined.
They are just one or more C<plain-line>s.

=end pod
    token woven-code  {
        [
            || <comment> { note $/.Str }
            || <plain-line>
        ]+
    } # end of token woven-code

    regex comment {
        $<x>=(<leading-ws> \N*?) # optional code
        '#'                      # comment marker
        <-[#]>*                  # the actual comment
        <ws-till-EOL>
    } # end of my regex comment
=begin pod
=comment 1

=head3 Non-woven sections

Sometimes there will be code you do not want woven into the document, such
as boilerplate code like C<use v6.d;>.  You have two options to mark such
```

```
code.  By individual lines or by delimited blocks of code.
=end pod
    token non-woven-code {
        [
          || <one-line-no-weave>
          || <delimited-no-weave>
        ]+
    } # end of token non-woven
=begin pod
=comment 1

=head4 One line of code

Simply append C<# begin-no-weave> at the end of the line!

=end pod
    token one-line-no-weave {
        ^^ \N*?
        '#' <hws> 'no-weave-this-line'
        <ws-till-EOL>
    } # end of token one-line-no-weave
=begin pod
=comment 1



=head4 Delimited blocks of code

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

=end pod
    token begin-no-weave {
        <leading-ws>                    # optional leading whitespace
        '#' <hws> 'begin-no-weave'  # the delimiter itself (# begin-no-weave)
        <ws-till-EOL>               # optional trailing whitespace or comment
    } # end of token <begin-no-weave>

    token end-no-weave {
        <leading-ws>                    # optional leading whitespace
        '#' <hws> 'end-no-weave'    # the delimiter itself (#end-no-weave)
        <ws-till-EOL>               # optional trailing whitespace or comment
    } # end of token <end--no-weave>

    token delimited-no-weave {
        <begin-no-weave>
            <plain-line>*
        <end-no-weave>
    } # end of token delimited-no-weave

    token code-comments {
            <leading-ws>
            '#' <rest-of-line>
        <!{ / <begin-no-weave> | <end-no-weave> / }>
    } # end of token code-comments
=begin pod
=comment 1

```

```
=head3 The C<plain-line> token

The C<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 L<Regex Boolean Condition
Check|https://docs.raku.org/language/regexes\#Regex_Boolean_condition_check>.


=end pod
    token plain-line {
        :my $*EXCEPTION = False;
        [
          ||  <begin-pod>         { $*EXCEPTION = True }
          ||  <end-pod>           { $*EXCEPTION = True }
          ||  <begin-no-weave>    { $*EXCEPTION = True }
          ||  <end-no-weave>      { $*EXCEPTION = True }
          ||  <one-line-no-weave> { $*EXCEPTION = True }
          || $<plain-line> = [^^ <rest-of-line>]
        ]
        <?{ !$*EXCEPTION }>
    } # end of token plain-line
=begin pod
=comment 1

And that concludes the grammar for separating C<Pod> from C<Code>!

=end pod
} # end of grammar Semi::Literate
=begin pod
=comment 2

=head1 The Tangle subroutine

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


=end pod
#TODO multi sub to accept Str & IO::PatGh
sub tangle (
=begin pod

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 C<MAIN>.
=end pod
    Str $input-file!,
=begin pod

The subroutine will return a C<Str>, which will be a working Raku program.
=end pod
        --> Str ) is export {
=begin pod
=comment 1

First we will get the entire Semi-Literate C<.sl> file...
=end pod
    my Str $source = $input-file.IO.slurp;
```

```
=begin pod
=comment 1
=head2 Clean the source

=head3 Remove unnecessary blank lines

Very often the C<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:

=begin code :lang<raku>

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

=end code
=end pod
=begin pod
=comment 1


So we'll remove the blank lines immediately outside the beginning and end of
the Pod6 sections.
=end pod
    my Str $cleaned-source = $source;
    $cleaned-source ~~ s:g{\=end (\N*)\n+} =   "\=end$0\n";
    $cleaned-source ~~ s:g{\n+\=begin (<hws> pod) [<hws> \d]?} = "\n\=begin$0";
=begin pod
=comment 1
=head2 The interesting stuff

We parse it using the C<Semi::Literate> grammar
and obtain a list of submatches (that's what the C<caps> method does) ...
=end pod
    my Pair @submatches = Semi::Literate.parse($cleaned-source).caps;
=begin pod
=comment 1

...and iterate through the submatches and keep only the C<code> sections...
=end pod
#    note "submatches.elems: {@submatches.elems}";
    my Str $raku-code = @submatches.map( {
#        note .key;
        when .key eq 'woven-code'|'non-woven-code' {
            .value;
        }
=begin pod
=comment 1
=head3 Replace Pod6 sections with blank lines

=end pod
        when .key eq 'pod' {
            my $num-blank-lines =
                .value.hash<num-blank-line-comment><num-blank-lines>;
            "\n" x $num-blank-lines with $num-blank-lines;
        }

        # begin-no-weave
        default { die "Tangle: should never get here. .key == {.key}" }
        #end-no-weave
=begin pod
=comment 1

... and we will join all the code sections together...
=end pod
    } # end of my Str $raku-code = @submatches.map(
    ).join;
=begin pod
=comment 1
=head3 Remove the I<no-weave> delimiters

=end pod
    $raku-code ~~ s:g{ <leading-ws> '#' <hws> 'begin-no-weave'     <rest-of-line> }
        = '';
    $raku-code ~~ s:g{ <leading-ws> '#' <hws> 'no-weave-this-line' <rest-of-line> }
        = "$0\n";
    $raku-code ~~ s:g{ <leading-ws> '#' <hws> 'end-no-weave'       <rest-of-line> }
        = '';
=begin pod
=comment 1
=head3 remove blank lines at the end

=end pod
    $raku-code ~~ s{\n  <blank-line>* $ } = '';
=begin pod
=comment 1

And that's the end of the C<tangle> subroutine!
=end pod
    return $raku-code;
} # end of sub tangle (
=begin pod
=comment 2

=head1 The Weave subroutine

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

=end pod
sub weave (
=begin pod
=comment 1
=head2 The parameters of Weave

C<sub weave> will have several parameters.
=head3 C<$input-file>

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

```
C<MAIN>.

=end pod
    Str $input-file!;
=begin pod
=comment 1
=head3 C<$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.
=end pod
    Str :f(:$format) is copy = 'markdown';
        #= The output format for the woven file.
=begin pod
=comment 1
=head3 C<$line-numbers>

It can be useful to print line numbers in the code listing.  It currently
defaults to True.
=end pod
    Bool :l(:$line-numbers)  = True;
        #= Should line numbers be added to the embeded code?
=begin pod
C<sub weave> returns a Str.
=end pod
        --> Str ) is export {

    my UInt $line-number = 1;
=begin pod
First we will get the entire C<.sl> file...
=end pod
    my Str $source = $input-file.IO.slurp;
=begin pod
=comment 1
=head3 Remove blank lines at the begining and end of the code

B<EXPLAIN THIS!>

=end pod
    my Str $cleaned-source = $source;
    $cleaned-source ~~ s:g{\=end (\N*)\n+} =   "\=end$0\n";
    $cleaned-source ~~ s:g{\n+\=begin (<hws> pod) [<hws> \d]?} = "\n\=begin$0";
=begin pod
=comment 1

=head2 Interesting stuff

...Next, we parse it using the C<Semi::Literate> grammar
and obtain a list of submatches (that's what the C<caps> method does) ...
=end pod
    my Pair @submatches = Semi::Literate.parse($cleaned-source).caps;
=begin pod
=comment 1

...And now begins the interesting part.  We iterate through the submatches and
insert the C<code> sections into the Pod6...
=end pod
```

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

        when .key eq 'woven-code' {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 # get rid of the last \n
             }
            \=end code
            \=end pod
            EOCB

        when .key eq 'non-woven-code' {
          ''; # do nothing
          #TODO don't insert a newline here.
        } # end of when .key eq 'non-woven-code'

        # begin-no-weave
        default {
            die "Weave: should never get here. .key == {.key}" }
        # end-no-weave
    } # end of my Str $weave = @submatches.map(
    ).join;
=begin pod
=comment 1
=head3 remove blank lines at the end

=end pod
    $weave ~~ s{\n  <blank-line>* $ } = '';
=begin pod
=comment 1

And that's the end of the C<tangle> subroutine!
=end pod
    return $weave
} # end of sub weave (
=begin pod
=comment 1
=head1 NAME

C<Semi::Literate> - A semi-literate way to weave and tangle Raku/Pod6 source code.
=head1 VERSION

This documentation refers to C<Semi-Literate> version 0.0.1

=head1 SYNOPSIS

```

```
=begin code :lang<raku>

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.

=end code
=head1 DESCRIPTION

C<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.)

=head1 BUGS AND LIMITATIONS

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

=head1 AUTHOR

Shimon Bollinger (deoac.bollinger@gmail.com)

=head1 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 L<The Artistic License 2.0|https://opensource.org/licenses/Artistic-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.

=end pod
# begin-no-weave
my %*SUB-MAIN-OPTS =
  :named-anywhere,             # allow named variables at any location
  :bundling,                   # allow bundling of named arguments
#  :coerce-allomorphs-to(Str),  # coerce allomorphic arguments to given type
  :allow-no,                   # allow --no-foo as alternative to --/foo
  :numeric-suffix-as-value,    # allow -j2 as alternative to --j=2
;

#| Run with option '--pod' to see all of the Pod6 objects
multi MAIN(Bool :$pod!) is hidden-from-USAGE {
    for $=pod -> $pod-item {
        for $pod-item.contents -> $pod-block {
            $pod-block.raku.say;
        }
    }
} # end of multi MAIN (:$pod)

#| Run with option '--doc' to generate a document from the Pod6
#| It will be rendered in Text format
#| unless specified with the --format option.  e.g.
#|       --doc --format=HTML
multi MAIN(Bool :$doc!, Str :$format = 'Text') is hidden-from-USAGE {
    run $*EXECUTABLE, "--doc=$format", $*PROGRAM;
} # end of multi MAIN(Bool :$man!)

my $semi-literate-file =
'/Users/jimbollinger/Documents/Development/raku/Projects/Semi-Literate/source/Literate.sl';
multi MAIN(Bool :$testt!) {
    say tangle($semi-literate-file);
} # end of multi MAIN(Bool :$test!)

multi MAIN(Bool :$testw!) {
    say weave($semi-literate-file);
} # end of multi MAIN(Bool :$test!)

#end-no-weave
```