NAME

Pod::Parser - base class for creating POD filters and translators

SYNOPSIS

```
use Pod::Parser;

package MyParser;
@ISA = qw(Pod::Parser);

sub command {
    my ($parser, $command, $paragraph, $line_num) = @_;
    ## Interpret the command and its text; sample actions might be:
    if ($command eq 'head1') { ... }
    elsif ($command eq 'head2') { ... }
    ## ... other commands and their actions
    my $out_fh = $parser->output_handle();
    my $expansion = $parser->interpolate($paragraph, $line_num);
    print $out_fh $expansion;
}

sub verbatim {
    my ($parser, $paragraph, $line_num) = @_;
    ## Format verbatim paragraph; sample actions might be:
    my $out_fh = $parser->output_handle();
    print $out_fh $paragraph;
}

sub textblock {
    my ($parser, $paragraph, $line_num) = @_;
    ## Translate/Format this block of text; sample actions might be:
    my $out_fh = $parser->output_handle();
    my $expansion = $parser->interpolate($paragraph, $line_num);
    print $out_fh $expansion;
}

sub interior_sequence {
    my ($parser, $seq_command, $seq_argument) = @_;
    ## Expand an interior sequence; sample actions might be:
    return "*$seq_argument*"     if ($seq_command eq 'B');
    return "`$seq_argument'"     if ($seq_command eq 'C');
    return "_${seq_argument}_'"  if ($seq_command eq 'I');
    ## ... other sequence commands and their resulting text
}

package main;

## Create a parser object and have it parse file whose name was
## given on the command-line (use STDIN if no files were given).
$parser = new MyParser();
$parser->parse_from_filehandle(\*STDIN)  if (@ARGV == 0);
for (@ARGV) { $parser->parse_from_file($_); }
```

REQUIRES

perl5.005, Pod::InputObjects, Exporter, Symbol, Carp

EXPORTS

Nothing.

DESCRIPTION

Pod::Parser is a base class for creating POD filters and translators. It handles most of the effort involved with parsing the POD sections from an input stream, leaving subclasses free to be concerned only with performing the actual translation of text.

Pod::Parser parses PODs, and makes method calls to handle the various components of the POD. Subclasses of **Pod::Parser** override these methods to translate the POD into whatever output format they desire.

QUICK OVERVIEW

To create a POD filter for translating POD documentation into some other format, you create a subclass of **Pod::Parser** which typically overrides just the base class implementation for the following methods:

- **command()**
- **verbatim()**
- **textblock()**
- **interior_sequence()**

You may also want to override the **begin_input()** and **end_input()** methods for your subclass (to perform any needed per-file and/or per-document initialization or cleanup).

If you need to perform any preprocesssing of input before it is parsed you may want to override one or more of **preprocess_line()** and/or **preprocess_paragraph()**.

Sometimes it may be necessary to make more than one pass over the input files. If this is the case you have several options. You can make the first pass using **Pod::Parser** and override your methods to store the intermediate results in memory somewhere for the **end_pod()** method to process. You could use **Pod::Parser** for several passes with an appropriate state variable to control the operation for each pass. If your input source can't be reset to start at the beginning, you can store it in some other structure as a string or an array and have that structure implement a **getline()** method (which is all that **parse_from_filehandle()** uses to read input).

Feel free to add any member data fields you need to keep track of things like current font, indentation, horizontal or vertical position, or whatever else you like. Be sure to read *PRIVATE METHODS AND DATA* to avoid name collisions.

For the most part, the **Pod::Parser** base class should be able to do most of the input parsing for you and leave you free to worry about how to intepret the commands and translate the result.

Note that all we have described here in this quick overview is the simplest most straightforward use of **Pod::Parser** to do stream-based parsing. It is also possible to use the **Pod::Parser::parse_text** function to do more sophisticated tree-based parsing. See *TREE-BASED PARSING*.

PARSING OPTIONS

A *parse-option* is simply a named option of **Pod::Parser** with a value that corresponds to a certain specified behavior. These various behaviors of **Pod::Parser** may be enabled/disabled by setting or unsetting one or more *parse-options* using the **parseopts()** method. The set of currently accepted parse-options is as follows:

-want_nonPODs (default: unset)

Normally (by default) **Pod::Parser** will only provide access to the POD sections of the input. Input paragraphs that are not part of the POD-format documentation are not made available to the caller (not even using **preprocess_paragraph()**). Setting this option to a non-empty, non-zero value will allow **preprocess_paragraph()** to see non-POD sections of the input as well as POD sections. The **cutting()** method can be used to determine if the corresponding paragraph is a POD paragraph, or some other input paragraph.

-process_cut_cmd (default: unset)

Normally (by default) **Pod::Parser** handles the `=cut` POD directive by itself and does not pass it on to the caller for processing. Setting this option to a non-empty, non-zero value will cause **Pod::Parser** to pass the `=cut` directive to the caller just like any other POD command (and hence it may be processed by the **command()** method).

Pod::Parser will still interpret the `=cut` directive to mean that "cutting mode" has been (re)entered, but the caller will get a chance to capture the actual `=cut` paragraph itself for whatever purpose it desires.

-warnings (default: unset)

Normally (by default) **Pod::Parser** recognizes a bare minimum of pod syntax errors and warnings and issues diagnostic messages for errors, but not for warnings. (Use **Pod::Checker** to do more thorough checking of POD syntax.) Setting this option to a non-empty, non-zero value will cause **Pod::Parser** to issue diagnostics for the few warnings it recognizes as well as the errors.

Please see *parseopts()* for a complete description of the interface for the setting and unsetting of parse-options.

RECOMMENDED SUBROUTINE/METHOD OVERRIDES

Pod::Parser provides several methods which most subclasses will probably want to override. These methods are as follows:

command()

```
$parser->command($cmd,$text,$line_num,$pod_para);
```

This method should be overridden by subclasses to take the appropriate action when a POD command paragraph (denoted by a line beginning with "=") is encountered. When such a POD directive is seen in the input, this method is called and is passed:

`$cmd`

the name of the command for this POD paragraph

`$text`

the paragraph text for the given POD paragraph command.

`$line_num`

the line-number of the beginning of the paragraph

`$pod_para`

a reference to a `Pod::Paragraph` object which contains further information about the paragraph command (see *Pod::InputObjects* for details).

Note that this method *is* called for `=pod` paragraphs.

The base class implementation of this method simply treats the raw POD command as normal block of paragraph text (invoking the **textblock()** method with the command paragraph).

verbatim()

```
$parser->verbatim($text,$line_num,$pod_para);
```

This method may be overridden by subclasses to take the appropriate action when a block of verbatim text is encountered. It is passed the following parameters:

`$text`

the block of text for the verbatim paragraph

`$line_num`

the line-number of the beginning of the paragraph

`$pod_para`

a reference to a `Pod::Paragraph` object which contains further information about the paragraph (see *Pod::InputObjects* for details).

The base class implementation of this method simply prints the textblock (unmodified) to the output filehandle.

textblock()

```
$parser->textblock($text,$line_num,$pod_para);
```

This method may be overridden by subclasses to take the appropriate action when a normal block of POD text is encountered (although the base class method will usually do what you want). It is passed the following parameters:

`$text`

the block of text for the a POD paragraph

`$line_num`

the line-number of the beginning of the paragraph

`$pod_para`

a reference to a `Pod::Paragraph` object which contains further information about the paragraph (see *Pod::InputObjects* for details).

In order to process interior sequences, subclasses implementations of this method will probably want to invoke either **interpolate()** or **parse_text()**, passing it the text block `$text`, and the corresponding line number in `$line_num`, and then perform any desired processing upon the returned result.

The base class implementation of this method simply prints the text block as it occurred in the input stream).

interior_sequence()

```
$parser->interior_sequence($seq_cmd,$seq_arg,$pod_seq);
```

This method should be overridden by subclasses to take the appropriate action when an interior sequence is encountered. An interior sequence is an embedded command within a block of text which appears as a command name (usually a single uppercase character) followed immediately by a string of text which is enclosed in angle brackets. This method is passed the sequence command `$seq_cmd` and the corresponding text `$seq_arg`. It is invoked by the **interpolate()** method for each interior sequence that occurs in the string that it is passed. It should return the desired text string to be used in place of the interior sequence. The `$pod_seq` argument is a reference to a `Pod::InteriorSequence` object which contains further information about the interior sequence. Please see *Pod::InputObjects* for details if you need to access this additional information.

Subclass implementations of this method may wish to invoke the **nested()** method of `$pod_seq` to see if it is nested inside some other interior-sequence (and if so, which kind).

The base class implementation of the **interior_sequence()** method simply returns the raw text of the interior sequence (as it occurred in the input) to the caller.

OPTIONAL SUBROUTINE/METHOD OVERRIDES

Pod::Parser provides several methods which subclasses may want to override to perform any special pre/post-processing. These methods do *not* have to be overridden, but it may be useful for subclasses to take advantage of them.

new()

```
my $parser = Pod::Parser->new();
```

This is the constructor for **Pod::Parser** and its subclasses. You *do not* need to override this method! It is capable of constructing subclass objects as well as base class objects, provided you use any of the following constructor invocation styles:

```
my $parser1 = MyParser->new();
my $parser2 = new MyParser();
my $parser3 = $parser2->new();
```

where `MyParser` is some subclass of **Pod::Parser**.

Using the syntax `MyParser::new()` to invoke the constructor is *not* recommended, but if you insist on being able to do this, then the subclass *will* need to override the **new()** constructor method. If you do override the constructor, you *must* be sure to invoke the **initialize()** method of the newly blessed object.

Using any of the above invocations, the first argument to the constructor is always the corresponding package name (or object reference). No other arguments are required, but if desired, an associative array (or hash-table) my be passed to the **new()** constructor, as in:

```
my $parser1 = MyParser->new( MYDATA => $value1, MOREDATA => $value2 );
my $parser2 = new MyParser( -myflag => 1 );
```

All arguments passed to the **new()** constructor will be treated as key/value pairs in a hash-table. The newly constructed object will be initialized by copying the contents of the given hash-table (which may have been empty). The **new()** constructor for this class and all of its subclasses returns a blessed reference to the initialized object (hash-table).

initialize()

```
$parser->initialize();
```

This method performs any necessary object initialization. It takes no arguments (other than the object instance of course, which is typically copied to a local variable named `$self`). If subclasses override this method then they *must* be sure to invoke `$self->SUPER::initialize()`.

begin_pod()

```
$parser->begin_pod();
```

This method is invoked at the beginning of processing for each POD document that is encountered in the input. Subclasses should override this method to perform any per-document initialization.

begin_input()

```
$parser->begin_input();
```

This method is invoked by **parse_from_filehandle()** immediately *before* processing input from a filehandle. The base class implementation does nothing, however, subclasses may override it to perform any per-file initializations.

Note that if multiple files are parsed for a single POD document (perhaps the result of some future `=include` directive) this method is invoked for every file that is parsed. If you wish to perform certain initializations once per document, then you should use **begin_pod()**.

end_input()

```
$parser->end_input();
```

This method is invoked by **parse_from_filehandle()** immediately *after* processing input from a filehandle. The base class implementation does nothing, however, subclasses may override it to perform any per-file cleanup actions.

Please note that if multiple files are parsed for a single POD document (perhaps the result of some kind of `=include` directive) this method is invoked for every file that is parsed. If you wish to perform certain cleanup actions once per document, then you should use **end_pod()**.

end_pod()

```
$parser->end_pod();
```

This method is invoked at the end of processing for each POD document that is encountered in the input. Subclasses should override this method to perform any per-document finalization.

preprocess_line()

```
$textline = $parser->preprocess_line($text, $line_num);
```

This method should be overridden by subclasses that wish to perform any kind of preprocessing for each *line* of input (*before* it has been determined whether or not it is part of a POD paragraph). The parameter `$text` is the input line; and the parameter `$line_num` is the line number of the corresponding text line.

The value returned should correspond to the new text to use in its place. If the empty string or an undefined value is returned then no further processing will be performed for this line.

Please note that the **preprocess_line()** method is invoked *before* the **preprocess_paragraph()** method. After all (possibly preprocessed) lines in a paragraph have been assembled together and it has been determined that the paragraph is part of the POD documentation from one of the selected sections, then **preprocess_paragraph()** is invoked.

The base class implementation of this method returns the given text.

preprocess_paragraph()

```
$textblock = $parser->preprocess_paragraph($text, $line_num);
```

This method should be overridden by subclasses that wish to perform any kind of preprocessing for each block (paragraph) of POD documentation that appears in the input stream. The parameter `$text` is the POD paragraph from the input file; and the parameter `$line_num` is the line number for the beginning of the corresponding paragraph.

The value returned should correspond to the new text to use in its place If the empty string is returned or an undefined value is returned, then the given `$text` is ignored (not processed).

This method is invoked after gathering up all the lines in a paragraph and after determining the cutting state of the paragraph, but before trying to further parse or interpret them. After **preprocess_paragraph()** returns, the current cutting state (which is returned by `$self->cutting()`) is examined. If it evaluates to true then input text (including the given `$text`) is cut (not processed) until the next POD directive is encountered.

Please note that the **preprocess_line()** method is invoked *before* the **preprocess_paragraph()** method. After all (possibly preprocessed) lines in a paragraph have been assembled together and either it has been determined that the paragraph is part of the POD documentation from one of the selected sections or the `-want_nonPODs` option is true, then **preprocess_paragraph()** is invoked.

The base class implementation of this method returns the given text.

METHODS FOR PARSING AND PROCESSING

Pod::Parser provides several methods to process input text. These methods typically won't need to be overridden (and in some cases they can't be overridden), but subclasses may want to invoke them to exploit their functionality.

parse_text()

```
$ptree1 = $parser->parse_text($text, $line_num);
$ptree2 = $parser->parse_text({%opts}, $text, $line_num);
$ptree3 = $parser->parse_text(\%opts, $text, $line_num);
```

This method is useful if you need to perform your own interpolation of interior sequences and can't rely upon **interpolate** to expand them in simple bottom-up order.

The parameter `$text` is a string or block of text to be parsed for interior sequences; and the parameter `$line_num` is the line number corresponding to the beginning of `$text`.

parse_text() will parse the given text into a parse-tree of "nodes." and interior-sequences. Each "node" in the parse tree is either a text-string, or a **Pod::InteriorSequence**. The result returned is a parse-tree of type **Pod::ParseTree**. Please see *Pod::InputObjects* for more information about **Pod::InteriorSequence** and **Pod::ParseTree**.

If desired, an optional hash-ref may be specified as the first argument to customize certain aspects of the parse-tree that is created and returned. The set of recognized option keywords are:

-expand_seq => *code-ref|method-name*

Normally, the parse-tree returned by **parse_text()** will contain an unexpanded `Pod::InteriorSequence` object for each interior-sequence encountered. Specifying **-expand_seq** tells **parse_text()** to "expand" every interior-sequence it sees by invoking the referenced function (or named method of the parser object) and using the return value as the expanded result.

If a subroutine reference was given, it is invoked as:

```
&$code_ref( $parser, $sequence )
```

and if a method-name was given, it is invoked as:

```
$parser->method_name( $sequence )
```

where `$parser` is a reference to the parser object, and `$sequence` is a reference to the interior-sequence object. [*NOTE*: If the **interior_sequence()** method is specified, then it is invoked according to the interface specified in *interior_sequence()*].

-expand_text => *code-ref|method-name*

Normally, the parse-tree returned by **parse_text()** will contain a text-string for each contiguous sequence of characters outside of an interior-sequence. Specifying **-expand_text** tells

parse_text() to "preprocess" every such text-string it sees by invoking the referenced function (or named method of the parser object) and using the return value as the preprocessed (or "expanded") result. [Note that if the result is an interior-sequence, then it will *not* be expanded as specified by the **-expand_seq** option; Any such recursive expansion needs to be handled by the specified callback routine.]

If a subroutine reference was given, it is invoked as:

```
&$code_ref( $parser, $text, $ptree_node )
```

and if a method-name was given, it is invoked as:

```
$parser->method_name( $text, $ptree_node )
```

where `$parser` is a reference to the parser object, `$text` is the text-string encountered, and `$ptree_node` is a reference to the current node in the parse-tree (usually an interior-sequence object or else the top-level node of the parse-tree).

-expand_ptree => *code-ref|method-name*

Rather than returning a `Pod::ParseTree`, pass the parse-tree as an argument to the referenced subroutine (or named method of the parser object) and return the result instead of the parse-tree object.

If a subroutine reference was given, it is invoked as:

```
&$code_ref( $parser, $ptree )
```

and if a method-name was given, it is invoked as:

```
$parser->method_name( $ptree )
```

where `$parser` is a reference to the parser object, and `$ptree` is a reference to the parse-tree object.

interpolate()

```
$textblock = $parser->interpolate($text, $line_num);
```

This method translates all text (including any embedded interior sequences) in the given text string `$text` and returns the interpolated result. The parameter `$line_num` is the line number corresponding to the beginning of `$text`.

interpolate() merely invokes a private method to recursively expand nested interior sequences in bottom-up order (innermost sequences are expanded first). If there is a need to expand nested sequences in some alternate order, use **parse_text** instead.

parse_paragraph()

```
$parser->parse_paragraph($text, $line_num);
```

This method takes the text of a POD paragraph to be processed, along with its corresponding line number, and invokes the appropriate method (one of **command()**, **verbatim()**, or **textblock()**).

For performance reasons, this method is invoked directly without any dynamic lookup; Hence subclasses may *not* override it!

parse_from_filehandle()

```
$parser->parse_from_filehandle($in_fh,$out_fh);
```

This method takes an input filehandle (which is assumed to already be opened for reading) and reads the entire input stream looking for blocks (paragraphs) of POD documentation to be processed. If no first argument is given the default input filehandle `STDIN` is used.

The `$in_fh` parameter may be any object that provides a **getline()** method to retrieve a single line of input text (hence, an appropriate wrapper object could be used to parse PODs from a single string or an array of strings).

Using `$in_fh->getline()`, input is read line-by-line and assembled into paragraphs or "blocks" (which are separated by lines containing nothing but whitespace). For each block of POD documentation encountered it will invoke a method to parse the given paragraph.

If a second argument is given then it should correspond to a filehandle where output should be sent (otherwise the default output filehandle is `STDOUT` if no output filehandle is currently in use).

NOTE: For performance reasons, this method caches the input stream at the top of the stack in a local variable. Any attempts by clients to change the stack contents during processing when in the midst executing of this method *will not affect* the input stream used by the current invocation of this method.

This method does *not* usually need to be overridden by subclasses.

parse_from_file()

```
$parser->parse_from_file($filename,$outfile);
```

This method takes a filename and does the following:

- opens the input and output files for reading (creating the appropriate filehandles)
- invokes the **parse_from_filehandle()** method passing it the corresponding input and output filehandles.
- closes the input and output files.

If the special input filename "-" or "<&STDIN" is given then the STDIN filehandle is used for input (and no open or close is performed). If no input filename is specified then "-" is implied.

If a second argument is given then it should be the name of the desired output file. If the special output filename "-" or ">&STDOUT" is given then the STDOUT filehandle is used for output (and no open or close is performed). If the special output filename ">&STDERR" is given then the STDERR filehandle is used for output (and no open or close is performed). If no output filehandle is currently in use and no output filename is specified, then "-" is implied. Alternatively, an *IO::String* object is also accepted as an output file handle.

This method does *not* usually need to be overridden by subclasses.

ACCESSOR METHODS

Clients of **Pod::Parser** should use the following methods to access instance data fields:

errorsub()

```
$parser->errorsub("method_name");
$parser->errorsub(\&warn_user);
$parser->errorsub(sub { print STDERR, @_ });
```

Specifies the method or subroutine to use when printing error messages about POD syntax. The supplied method/subroutine *must* return TRUE upon successful printing of the message. If `undef` is given, then the **warn** builtin is used to issue error messages (this is the default behavior).

```
my $errorsub = $parser->errorsub()
my $errmsg = "This is an error message!\n"
(ref $errorsub) and &{$errorsub}($errmsg)
    or (defined $errorsub) and $parser->$errorsub($errmsg)
        or  warn($errmsg);
```

Returns a method name, or else a reference to the user-supplied subroutine used to print error messages. Returns `undef` if the **warn** builtin is used to issue error messages (this is the default behavior).

cutting()

```
$boolean = $parser->cutting();
```

Returns the current `cutting` state: a boolean-valued scalar which evaluates to true if text from the input file is currently being "cut" (meaning it is *not* considered part of the POD document).

```
$parser->cutting($boolean);
```

Sets the current `cutting` state to the given value and returns the result.

parseopts()

When invoked with no additional arguments, **parseopts** returns a hashtable of all the current parsing options.

```
## See if we are parsing non-POD sections as well as POD ones
my %opts = $parser->parseopts();
$opts{'-want_nonPODs}' and print "-want_nonPODs\n";
```

When invoked using a single string, **parseopts** treats the string as the name of a parse-option and returns its corresponding value if it exists (returns `undef` if it doesn't).

```
## Did we ask to see '=cut' paragraphs?
my $want_cut = $parser->parseopts('-process_cut_cmd');
$want_cut and print "-process_cut_cmd\n";
```

When invoked with multiple arguments, **parseopts** treats them as key/value pairs and the specified parse-option names are set to the given values. Any unspecified parse-options are unaffected.

```
## Set them back to the default
$parser->parseopts(-warnings => 0);
```

When passed a single hash-ref, **parseopts** uses that hash to completely reset the existing parse-options, all previous parse-option values are lost.

```
## Reset all options to default
$parser->parseopts( { } );
```

See *PARSING OPTIONS* for more information on the name and meaning of each parse-option currently recognized.

output_file()

```
$fname = $parser->output_file();
```

Returns the name of the output file being written.

output_handle()

```
$fhandle = $parser->output_handle();
```

Returns the output filehandle object.

input_file()

```
$fname = $parser->input_file();
```

Returns the name of the input file being read.

input_handle()

```
$fhandle = $parser->input_handle();
```

Returns the current input filehandle object.

input_streams()

```
$listref = $parser->input_streams();
```

Returns a reference to an array which corresponds to the stack of all the input streams that are currently in the middle of being parsed.

While parsing an input stream, it is possible to invoke **parse_from_file()** or **parse_from_filehandle()** to parse a new input stream and then return to parsing the previous input stream. Each input stream to be parsed is pushed onto the end of this input stack before any of its input is read. The input stream that is currently being parsed is always at the end (or top) of the input stack. When an input stream has been exhausted, it is popped off the end of the input stack.

Each element on this input stack is a reference to `Pod::InputSource` object. Please see *Pod::InputObjects* for more details.

This method might be invoked when printing diagnostic messages, for example, to obtain the name and line number of the all input files that are currently being processed.

top_stream()

```
$hashref = $parser->top_stream();
```

Returns a reference to the hash-table that represents the element that is currently at the top (end) of the input stream stack (see *input_streams()*). The return value will be the `undef` if the input stack is empty.

This method might be used when printing diagnostic messages, for example, to obtain the name and line number of the current input file.

PRIVATE METHODS AND DATA

Pod::Parser makes use of several internal methods and data fields which clients should not need to see or use. For the sake of avoiding name collisions for client data and methods, these methods and fields are briefly discussed here. Determined hackers may obtain further information about them by reading the **Pod::Parser** source code.

Private data fields are stored in the hash-object whose reference is returned by the **new()** constructor for this class. The names of all private methods and data-fields used by **Pod::Parser** begin with a prefix of "_" and match the regular expression `/^_\w+$/`.

_push_input_stream()

```
$hashref = $parser->_push_input_stream($in_fh,$out_fh);
```

This method will push the given input stream on the input stack and perform any necessary beginning-of-document or beginning-of-file processing. The argument `$in_fh` is the input stream filehandle to push, and `$out_fh` is the corresponding output filehandle to use (if it is not given or is undefined, then the current output stream is used, which defaults to standard output if it doesnt exist yet).

The value returned will be reference to the hash-table that represents the new top of the input stream stack. *Please Note* that it is possible for this method to use default values for the input and output file handles. If this happens, you will need to look at the `INPUT` and `OUTPUT` instance data members to determine their new values.

_pop_input_stream()

```
$hashref = $parser->_pop_input_stream();
```

This takes no arguments. It will perform any necessary end-of-file or end-of-document processing and then pop the current input stream from the top of the input stack.

The value returned will be reference to the hash-table that represents the new top of the input stream stack.

TREE-BASED PARSING

If straightforward stream-based parsing wont meet your needs (as is likely the case for tasks such as translating PODs into structured markup languages like HTML and XML) then you may need to take the tree-based approach. Rather than doing everything in one pass and calling the **interpolate()** method to expand sequences into text, it may be desirable to instead create a parse-tree using the **parse_text()** method to return a tree-like structure which may contain an ordered list of children (each of which may be a text-string, or a similar tree-like structure).

Pay special attention to *METHODS FOR PARSING AND PROCESSING* and to the objects described in *Pod::InputObjects*. The former describes the gory details and parameters for how to customize and extend the parsing behavior of **Pod::Parser**. **Pod::InputObjects** provides several objects that may all be used interchangeably as parse-trees. The most obvious one is the **Pod::ParseTree** object. It defines the basic interface and functionality that all things trying to be a POD parse-tree should do. A **Pod::ParseTree** is defined such that each "node" may be a text-string, or a reference to another parse-tree. Each **Pod::Paragraph** object and each **Pod::InteriorSequence** object also supports the basic parse-tree interface.

The **parse_text()** method takes a given paragraph of text, and returns a parse-tree that contains one or more children, each of which may be a text-string, or an InteriorSequence object. There are also callback-options that may be passed to **parse_text()** to customize the way it expands or transforms interior-sequences, as well as the returned result. These callbacks can be used to create a parse-tree with custom-made objects (which may or may not support the parse-tree interface, depending on how you choose to do it).

If you wish to turn an entire POD document into a parse-tree, that process is fairly straightforward. The **parse_text()** method is the key to doing this successfully. Every paragraph-callback (i.e. the polymorphic methods for **command()**, **verbatim()**, and **textblock()** paragraphs) takes a **Pod::Paragraph** object as an argument. Each paragraph object has a **parse_tree()** method that can be used to get or set a corresponding parse-tree. So for each of those paragraph-callback methods, simply call **parse_text()** with the options you desire, and then use the returned parse-tree to assign to the given paragraph object.

That gives you a parse-tree for each paragraph - so now all you need is an ordered list of paragraphs. You can maintain that yourself as a data element in the object/hash. The most straightforward way would be simply to use an array-ref, with the desired set of custom "options" for each invocation of **parse_text**. Let's assume the desired option-set is given by the hash `%options`. Then we might do something like the following:

```
package MyPodParserTree;

@ISA = qw( Pod::Parser );

...
```

```
sub begin_pod {
    my $self = shift;
    $self->{'-paragraphs'} = [];  ## initialize paragraph list
}

sub command {
    my ($parser, $command, $paragraph, $line_num, $pod_para) = @_;
    my $ptree = $parser->parse_text({%options}, $paragraph, ...);
    $pod_para->parse_tree( $ptree );
    push @{ $self->{'-paragraphs'} }, $pod_para;
}

sub verbatim {
    my ($parser, $paragraph, $line_num, $pod_para) = @_;
    push @{ $self->{'-paragraphs'} }, $pod_para;
}

sub textblock {
    my ($parser, $paragraph, $line_num, $pod_para) = @_;
    my $ptree = $parser->parse_text({%options}, $paragraph, ...);
    $pod_para->parse_tree( $ptree );
    push @{ $self->{'-paragraphs'} }, $pod_para;
}

...

package main;
...
my $parser = new MyPodParserTree(...);
$parser->parse_from_file(...);
my $paragraphs_ref = $parser->{'-paragraphs'};
```

Of course, in this module-author's humble opinion, I'd be more inclined to use the existing **Pod::ParseTree** object than a simple array. That way everything in it, paragraphs and sequences, all respond to the same core interface for all parse-tree nodes. The result would look something like:

```
package MyPodParserTree2;

...

sub begin_pod {
    my $self = shift;
    $self->{'-ptree'} = new Pod::ParseTree;  ## initialize parse-tree
}

sub parse_tree {
    ## convenience method to get/set the parse-tree for the entire POD
    (@_ > 1)  and  $_[0]->{'-ptree'} = $_[1];
    return $_[0]->{'-ptree'};
}

sub command {
    my ($parser, $command, $paragraph, $line_num, $pod_para) = @_;
```

```
    my $ptree = $parser->parse_text({<<options>>}, $paragraph, ...);
    $pod_para->parse_tree( $ptree );
    $parser->parse_tree()->append( $pod_para );
}

sub verbatim {
    my ($parser, $paragraph, $line_num, $pod_para) = @_;
    $parser->parse_tree()->append( $pod_para );
}

sub textblock {
    my ($parser, $paragraph, $line_num, $pod_para) = @_;
    my $ptree = $parser->parse_text({<<options>>}, $paragraph, ...);
    $pod_para->parse_tree( $ptree );
    $parser->parse_tree()->append( $pod_para );
}

...

package main;
...
my $parser = new MyPodParserTree2(...);
$parser->parse_from_file(...);
my $ptree = $parser->parse_tree;
...
```

Now you have the entire POD document as one great big parse-tree. You can even use the **-expand_seq** option to **parse_text** to insert whole different kinds of objects. Just don't expect **Pod::Parser** to know what to do with them after that. That will need to be in your code. Or, alternatively, you can insert any object you like so long as it conforms to the **Pod::ParseTree** interface.

One could use this to create subclasses of **Pod::Paragraphs** and **Pod::InteriorSequences** for specific commands (or to create your own custom node-types in the parse-tree) and add some kind of **emit()** method to each custom node/subclass object in the tree. Then all you'd need to do is recursively walk the tree in the desired order, processing the children (most likely from left to right) by formatting them if they are text-strings, or by calling their **emit()** method if they are objects/references.

SEE ALSO

Pod::InputObjects, *Pod::Select*

Pod::InputObjects defines POD input objects corresponding to command paragraphs, parse-trees, and interior-sequences.

Pod::Select is a subclass of **Pod::Parser** which provides the ability to selectively include and/or exclude sections of a POD document from being translated based upon the current heading, subheading, subsubheading, etc.

AUTHOR

Please report bugs using *http://rt.cpan.org*.

Brad Appleton <bradapp@enteract.com>

Based on code for **Pod::Text** written by Tom Christiansen <tchrist@mox.perl.com>