NAME

ePerl − Embedded Perl 5 Language

SYNOPSIS

eperl [**−d** *name*=*value*] [**−D** *name*=*value*] [**−B** *begin_delimiter*] [**−E** *end_delimiter*] [**−i**] [**−m** *mode*] [**−o** *outputfile*] [**−k**] [**−I** *directory*] [**−P**] [**−C**] [**−L**] [**−x**] [**−T**] [**−w**] [**−c**] [*inputfile*]

eperl −r|**−l**|**−v**|**−V**

DESCRIPTION

Abstract

ePerl interprets a text file sprinkled with Perl 5 program statements by evaluating the Perl 5 code while copying the plain text data verbatim. It can operate in various ways: As a stand-alone Unix filter or integrated Perl 5 module for general file generation tasks and as a powerful Webserver scripting language for dynamic HTML page programming.

Introduction

The **eperl** program is the *Embedded Perl 5 Language* interpreter. This really is a full-featured Perl 5 interpreter, but with a different calling environment and source file layout than the default Perl interpreter (**perl**). It is designed for general text file generation with the philosophy of *embedding* the Perl 5 program code into the data instead of the usual way where you embed the data into a Perl 5 program (usually by quoting the data and using them via `print` statements). So, instead of writing a plain Perl script like

```
#!/usr/bin/perl
print "foo bar\n";
print "baz quux\n";
for ($i = 0; $i < 10; $i++) { print "foo #${i}\n"; }
print "foo bar\n";
print "baz quux\n";
```

you can write it now as an ePerl script:

```
#!/usr/bin/eperl
foo bar
baz quux
<: for ($i = 0; $i < 10; $i++) { print "foo #${i}\n"; } :>
foo bar
baz quux
```

Although the ePerl variant has a different source file layout, the semantic is the same, i.e. both scripts create exactly the same resulting data on `STDOUT`.

Intent

ePerl is simply glue code which combines the programming power of the Perl 5 interpreter library with an embedding trick: it converts the source file into a valid Perl script which then gets *entirely* evaluated by only one internal instance of the Perl 5 interpreter. To achieve this, ePerl translates all plain code into (escaped) Perl 5 strings placed into *print* constructs while passing through all embedded native Perl 5 code. This amounts to the same operation as one would do when writing a plain Perl generation script.

Due to the nature of such sprinkled code, ePerl is really the better approach when the generated text contains really more static than dynamic data. Or in other words: *Use ePerl if you want to keep the most of the generated text data in plain format while just programming some sprinkled stuff.* Do not use it when generating pure dynamic data. There it brings no advantage to the ordinary program code of a plain Perl script. So, the static part should be at least 60% or the advantage becomes a disadvantage.

ePerl in its origin was actually designed for an extreme situation: as a webserver scripting-language for on-the-fly HTML page generation. Here you have the typical case that usually 90% of the data consists of pure static HTML tags and plain text while just the remaining 10% are programming constructs which dynamically generate more markup code. This is the reason why ePerl beside its standard Unix filtering runtime-mode also supports the CGI/1.1 and NPH−CGI/1.1 interfaces.

Embedded Perl Syntax

Practically you can put any valid Perl constructs inside the ePerl blocks the used Perl 5 interpreter library can evaluate. But there are some important points you should always remember and never forget when using ePerl:

1. Delimiters are always discarded.
Trivially to say, but should be mentioned at least once. The ePerl block delimiters are always discarded and are only necessary for ePerl to recognize the embedded Perl constructs. They are never copied to the final output.

2. Generated content has to go to `STDOUT`.
Although you can define subroutines, calculate some data, etc. inside ePerl blocks, only data which is explicitly written to the `STDOUT` filehandle is expanded. In other words: When an ePerl block does not generate content on `STDOUT`, it is entirely replaced by an empty string in the final output. But when content is generated it is put at the point of the ePerl block in the final output. Usually content is generated via pure `print` constructs which implicitly use `STDOUT` when no filehandle is given.

3. Generated content on `STDERR` always leads to an error.
Whenever content is generated on the `STDERR` filehandle, ePerl displays an error (including the STDERR content). Use this to exit on errors while passing errors from ePerl blocks to the calling environment.

4. Last semicolon.
Because of point 6 (below) and the fact that most of the users don't have the internal ePerl block translations in mind, ePerl is smart about the last semicolon. Usually every Perl block has to end with the semicolon of the last command.

```
<: cmd; ...; cmd; :>
```

But when the last semicolon is missing it is automatically added by ePerl, i.e.

```
<: cmd; ...; cmd :>
```

is also correct syntax. But sometimes it is necessary to force ePerl *not* to add the semicolon. Then you can add a _ (underscore) as the last non-whitespace character in the block to force ePerl to leave the final semicolon. Use this for constructs like the following

```
<: if (...) { _:>
foo
<: } else { _:>
bar
<: } :>
```

where you want to spread a Perl directive over more ePerl blocks.

5. Shorthand for `print`-only blocks.
Because most of the time ePerl is used just to interpolate variables, e.g.

```
<: print $VARIABLE; :>
```

it is useful to provide a shortcut for this kind of constructs. So ePerl provides a shortcut via the character =. When it immediately (no whitespaces allowed here) follows the begin delimiter of an ePerl block, a `print` statement is implicitly generated, i.e. the above block is equivalent to

```
<:=$VARIABLE:>
```

6. Special end-of-line discard command for ePerl blocks.
ePerl provides a special discard command named `//` which discards all data up to and including the following newline character when directly followed an end block delimiter. Usually when you write

```
foo
<: $x = 1; :>
quux
```

the result is

```
foo

quux
```

because ePerl always preserves code around ePerl blocks, even just newlines. But when you write

```
foo
<: $x = 1; :>//
quux
```

the result is

```
foo
quux
```

7. Restrictions in parsing.

Perl is a rich language, but a horrible one to parse. Perhaps you've heard "Only *perl* can parse *Perl*". The implication of this is that ePerl never tries to parse the ePerl blocks itself. It entirely relies on the Perl interpreter library, because it is the only instance which can do this without errors. But the problem is that ePerl at least has to recognize the begin and end positions of those ePerl blocks.

There are two ways: It can either look for the end delimiter while parsing, but at least recognize quoted strings (where the end delimiter gets treated as pure data). Or it can just move forward to the next end delimiter and say that it can not occur inside Perl constructs. In ePerl 2.0 the latter was used, while in ePerl 2.1 the former was taken because a lot of users wanted it this way while using bad end delimiters like >. But actually the author has again revised its opinion in ePerl 2.2 and decided to finally use latter approach. Because while the first one allows more trivial delimiters (which itself is not a really good idea), it fails when constructs like `m|"[^"]+"|` etc. are used inside ePerl blocks. And it is easier to escape end delimiters inside Perl constructs (for instance via backslashes in quoted strings) than rewrite complex Perl constructs to use even numbers of quotes.

So, whenever your end delimiter also occurs inside Perl constructs you have to some-how escape it.

8. HTML entity conversion.

Because one of ePerl's usage is as a server-side scripting-language for HTML pages, there is a common problem in conjunction with HTML editors. They cannot know ePerl blocks, so when you enter those blocks inside the editors they usually encode some characters with the corresponding HTML entities. The problem is that this encoding leads to invalid Perl code. ePerl provides the **−C** option (q.v.) for decoding these entities to CP−1252, which is automatically turned on in CGI modes.

Runtime Modes

ePerl can operate in three different runtime modes:

Stand-alone Unix filter mode

This is the default operation mode when used as a generation tool from the Unix shell or as a batch-processing tool from within other programs or scripts:

```
$ eperl [options] - < inputfile > outputfile
$ eperl [options] inputfile > outputfile
$ eperl [options] -o outputfile - < inputfile
$ eperl [options] -o outputfile inputfile
```

As you can see, ePerl can be used in any combination of STDIO and external files. Additionally there are two interesting variants of using this mode. First, you can put ePerl in the shebang to implicitly select it as the interpreter for your script, similar to the way you are used to with the plain Perl interpreter:

```
#!/usr/bin/eperl [options]
foo
<: print "bar"; :>
quux
```

Second, you can use ePerl in conjunction with the shell *here-document* technique from within your shell programs:

```
#!/bin/sh
...
eperl [options] - <<EOS
foo
<: print "quux"; :>
quux
EOS
...
```

If you need to generate shell or other scripts with ePerl, i.e. you need a shebang line in the output of ePerl, you have to add a shebang line containing e.g. `#!/usr/bin/eperl` first, because ePerl will strip the first line from the input if it is a shebang line. For example:

```
#!/usr/bin/eperl
#!/bin/sh
echo <: print "quux"; :>
```

will result in the following output:

```
#!/bin/sh
echo quux
```

Alternatively you can add a preprocessor comment in the first line:

```
#c This is a comment to preserve the shebang line in the following line
#!/bin/sh
echo <: print "quux"; :>
```

And finally you can use ePerl directly from within Perl programs by the use of the **Parse::ePerl** (3) package (assuming that you have installed this also):

```
#!/path/to/perl
...
use Parse::ePerl;
...
$script = <<EOT;
foo
<: print "quux"; :>
quux
EOT
...
$result = Parse::ePerl::Expand({
    Script => $script,
    Result => \$result,
});
...
print $result;
...
```

See **Parse::ePerl** (3pm) for more details.

CGI/1.1 compliant interface mode

This is the runtime mode where ePerl uses the CGI/1.1 interface of a webserver when used as a server-side scripting language. ePerl enters this mode automatically when the CGI/1.1 environment variable `PATH_TRANSLATED` is set and its or the scripts filename does *not* begin with the NPH prefix "*nph−*". In this runtime mode it prefixes the resulting data with HTTP/1.0 (default) or HTTP/1.1 (if identified by the webserver) compliant response header lines.

ePerl also recognizes HTTP header lines at the beginning of the script's generated data, for instance you can generate your own HTTP headers with

```
<? $url = "..";
   print "Location: $url\n";
   print "URI: $url\n\n"; !>
<html>
...
```

But notice that while you can output arbitrary headers, most webservers restrict the headers which are accepted via the CGI/1.1 interface. Usually you can provide only a few specific HTTP headers like `Location` or `Status`. If you need more control you have to use the NPH−CGI/1.1 interface mode.

The default HTTP status is "200 OK". If your script's output starts with an HTTP status line (`HTTP/1.0 123 Description` (or `/1.1`)), that line is used instead.

Additionally ePerl provides a useful feature in this mode: It can switch its UID/GID to the owner of the script if the set-UID bit is set (see *Security*).

There are two commonly known ways of using this CGI/1.1 interface mode on the Web. First, you can use it to explicitly transform plain HTML files into CGI/1.1 scripts with a shebang (see above). For an Apache webserver, just put the following line as the first line of the file:

```
#!/usr/bin/eperl -mc
```

Then rename the script from *file.html* to *file.cgi* and set mark it executable:

```
$ mv file.html file.cgi
$ chmod a+rx file.cgi
```

Now make sure that Apache accepts *file.cgi* as a CGI program by enabling CGI support for the directory where *file.cgi* resides. For this add the line

```
Options +ExecCGI
```

to the *.htaccess* file in this directory. Finally make sure that Apache really recognizes the extension *.cgi*. Perhaps you additionally have to add the following line to your *httpd.conf* file:

```
AddHandler cgi-script .cgi
```

Now you can use *file.cgi* instead of *file.html* and take advantage of the achieved programming capability by bristling *file.cgi* with your Perl blocks (or the transformation into a CGI script would have been useless).

Alternatively (or even additionally) a webmaster can enable ePerl support in a more seamless way by configuring ePerl as a real implicit server-side scripting language. This is done by assigning a MIME-type to the various valid ePerl file extensions and forcing all files with this MIME-type to be internally processed via the ePerl interpreter. You can accomplish this for Apache by adding the following to your *httpd.conf* file

```
AddType      application/x-httpd-eperl  .phtml .eperl .epl
Action       application/x-httpd-eperl  /internal/cgi/eperl
ScriptAlias  /internal/cgi              /path/to/apache/cgi-bin
```

and creating a copy of the *eperl* program in your CGI-directory:

```
$ cp -p /usr/bin/eperl /path/to/apache/cgi-bin/eperl
```

Now all files with the extensions *.phtml*, *.eperl* and *.epl* are automatically processed by the ePerl interpreter. There is no need for a shebang or any locally-enabled CGI mode.

One final hint: When you want to test your scripts offline, just run them with forced CGI/1.1 mode from your shell. But make sure you prepare all environment variables your script depends on, like `QUERY_STRING` or `PATH_INFO`:

```
$ export QUERY_STRING="key1=value1&key2=value2"
$ eperl -mc file.phtml
```

NPH−CGI/1.1 compliant interface mode

This runtime mode is a special variant of the CGI/1.1 interface mode, because most webservers (e.g. Apache) provide it for special purposes. NPH stands for *Non-Parsed-Header* and is usually used by the webserver when the filename of the CGI program is prefixed with `nph-`. In this mode the webserver does no processing on the HTTP response headers and no buffering of the resulting data, i.e. the CGI program actually has to provide a complete HTTP response itself. The advantage is that the program can generate arbitrary HTTP headers or MIME-encoded multi-block messages.

So, above we have renamed the file to *file.cgi* which restricted us a little bit. When we alternatively rename *file.html* to *nph−file.cgi* and force the NPH−CGI/1.1 interface mode via option **−mn** then this file becomes a NPH−CGI/1.1 compliant program under Apache and other webservers. Now our script *can* provide its own HTTP response (it doesn't *need* to, because ePerl provides a default one if it is absent).

```
#!/path/to/bin/eperl -mn
<? print "HTTP/1.0 200 Ok\n";
   print "X-MyHeader: Foo Bar Quux\n";
   print "Content-type: text/html\n\n";
<html>
...
```

Expectedly, this can be also used with the implicit Server-Side Scripting Language technique. Put

```
AddType      application/x-httpd-eperl  .phtml .eperl .epl
Action       application/x-httpd-eperl  /internal/cgi/nph-eperl
ScriptAlias  /internal/cgi              /path/to/apache/cgi-bin
```

into your *httpd.conf* and run the command

```
$ cp -p /usr/bin/eperl /path/to/apache/cgi-bin/nph-eperl
```

from your shell. *This is the preferred way of using ePerl as a Server-Side Scripting Language, because it provides most flexibility.*

Security

When you are installing ePerl as a CGI/1.1 or NPH−CGI/1.1 compliant program (see above for detailed description of these modes) via

```
$ cp -p /usr/bin/eperl /path/to/apache/cgi-bin/eperl
$ chown root /path/to/apache/cgi-bin/eperl
$ chmod u+s  /path/to/apache/cgi-bin/eperl
```

or

```
$ cp -p /usr/bin/eperl /path/to/apache/cgi-bin/nph-eperl
$ chown root /path/to/apache/cgi-bin/nph-eperl
$ chmod u+s  /path/to/apache/cgi-bin/nph-eperl
```

i.e. with set-UID bit enabled for the **root** user, ePerl can switch to the UID/GID of the *script's owner*. Although this is a very useful feature for script programmers (because one no longer need to make auxiliary files world-readable and temporary files world-writable!), it can be to risky for you when you are paranoid about security of set-UID programs. If so, just don't install ePerl set-UID! This is the reason why ePerl is by default only installed as a stand-alone program which never needs this feature.

For those of us who decided that this feature is essential, ePerl tries really hard to make it secure. The following steps have to be successfully passed before ePerl actually switches its UID/GID (in this order):

1. The script has to match the following extensions: *.html*, *.phtml*, *.eperl*, *.ephtml*, *.epl*, *.pl*, *.cgi*
2. The UID of the calling process has to be a valid UID, i.e. it has to be found in **passwd** (5)
3. The UID of the calling process has to match the following users: nobody, root
4. The UID of the script owner has to be a valid UID, i.e. it has to be found in **passwd** (5)
5. The GID of the script group has to be a valid GID, i.e. it has to be found in **group** (5)
6. The script has to stay below the owner's home directory

IF ANY ONE OF THOSE STEPS FAILS, NO UID/GID SWITCHING TAKES PLACE!. Additionally (if `DO_ON_FAILED_STEP` was defined to `STOP_AND_ERROR` in *eperl_security.h*, which is not the default) ePerl can totally stop processing and display its error page. This is for the really paranoid webmasters. Per default when any step failed the UID/GID switching is just disabled, but ePerl goes on with processing. Alternatively you can disable some steps at compile time. See *eperl_security.h.*

For security reasons, if the effective UID is that of root, the effectuve UID/GID is ***always*** *reset to the real UID/GID, regardless of the mode.*

ePerl Preprocessor

ePerl provides its own preprocessor, similar to the C preprocessor, which is either enabled manually via option **–P**, or automatically when ePerl runs in (NPH–)CGI mode. The following directives are supported:

`#include` *`path`*

The contents of *path*, which can be either a relative or absolute path or a fully qualified HTTP URL, are read and preprocessed recursively.

An absolute path is opened directly, relative paths are tried in the working directory and then in directories given by **–I**. An HTTP URL is retrieved via a HTTP/1.0 request on the network, and 301/303 redirects are followed.

While ePerl strictly preserves the line numbers when removing sprinklings to yield the plain Perl format, the preprocessor can't do this for this directive. So, line numbers in error messages will be wrong.

The security implications are obvious: This can run arbitrary code. You probably shouldn't use this if you don't implicitly trust the reply. `#sinclude` is appropriate then.

`#sinclude` *`path`*

This is just like `#include`, but all delimiters are removed. Thus, *path* is reduced to only data, and no code.

`#if` *`expr`*, `#elsif` *`expr`*, `#else`, `#endif`

These implement a C–preprocessor-style `#if`/`#else`/`#endif` construct, but *expr* is a Perl expression evaluated at run-time. These are converted as follows (where *BD*/*ED* are the delimiters):

```
#if expr     → BD if (expr) { _ ED//
#elsif expr → BD } elsif (expr) { _ ED//
#else        → BD } else { _ ED//
#endif       → BD } _ ED//
```

`#c` Comment, discards everything up to and including the newline.

Provided Functionality

You can put really *any* Perl code into the ePerl blocks which are valid to the Perl interpreter ePerl was linked with. ePerl does *not* provide any special functionality inside these ePerl blocks, because Perl is already sophisticated enough ;–)

Because you can use any valid Perl code you can use all available Perl 5 modules, even those which use shared objects. The Comprehensive Perl Archive Network <http://www.perl.com/perl/CPAN> provides packages for use both from within plain Perl scripts *and* ePerl scripts. `use name;` works as-expected.

OPTIONS

–d *name*=*value*

Sets a Perl variable in the package `main` which can be referenced via `$name` or more explicitly via `$main::name`. This is equivalent to adding

```
<? $name = value; !>
```

to the beginning of *inputfile*. This option can occur more than once.

–D *name*=*value*

Sets environment variable *name* to *value*, which can be referenced via `$ENV{'`*`NAME`*`'}`. This is equivalent to just running

```
$ name=value eperl ...
```

This option can occur more than once.

−B *begin_delimiter* =item **−E** *end_delimiter*
Set the Perl block begin and end delimiter strings. Default delimiters are `<?` & `!>` for CGI modes and `<:` & `:>` otherwise.

These may be of interest:

`<:` & `:>` (the default ePerl stand-alone filtering mode delimiters)
`<?` & `!>` (the default ePerl CGI interface mode delimiters)
`<script language='ePerl'>` & `</script>` (standard HTML scripting language style)
`<script type="text/eperl">` & `</script>` (forthcoming HTML3.2+ aka Cougar style)
`<eperl>` & `</eperl>` (HTML-like style)
`<!--#eperl code='` & `' -->` (NeoScript and SSI style)
`<?` & `>` (PHP/FI style; but this no longer recommended because it can lead to parsing problems. Should be used only for backward compatibility to old ePerl versions 1.x).

−i Forces the begin and end delimiters to be searched case-insensitively. Use this when you are using delimiters like `<ePerl>`...`</ePerl>` or other more textual ones.

−m *mode*
Forces ePerl to act in a specific runtime mode: stand-alone filter (**−mf**), the CGI/1.1 interface (**−mc**), or the NPH−CGI/1.1 interface (**−mn**).

−o *outputfile*
Write to *outputfile* instead of *STDOUT* (− specifies *STDOUT* explicitly.) This path is relative to the directory containing *inputfile* in the CGI modes.

−k Don't change the working directory. By default, ePerl will change to the directory containing *inputfile*.

−x Output the internally created Perl script to the console (*/dev/tty*) before executing it.

−I *directory*
Specifies a directory which where `#include` and `#sinclude` files are searched, and which is to be added to Perl `@INC`. This option can occur more than once.

−P Enable the special ePerl Preprocessor (see above). This option is enabled for all CGI modes automatically.

−C This enables the HTML entity conversion for ePerl blocks. This option is automatically forced in CGI modes.

The solved problem here is the following: When you use ePerl as a server-side-scripting-language for HTML pages and you edit your ePerl source files via a HTML editor, it's likely that it translates some entered characters into HTML entities, like < to `&lt;`. This leads to invalid Perl code inside ePerl blocks. Using this option, the ePerl parser automatically converts all entities found inside ePerl blocks back to plain CP−1252 characters, so the Perl interpreter again receives valid code blocks.

−L This enables the line continuation character \ (backslash) outside ePerl blocks. With this option you can spread one-line data over more lines. But use with care: This option changes your data (outside ePerl blocks). Usually ePerl really pass through all surrounding data as raw data. With this option the newlines have new semantics.

−T This enabled Perl's *Tainting mode* where the Perl interpreter takes special precautions called "taint checks" to prevent both obvious and subtle traps. See **perlsec** (1) for more details.

−w This enables Warnings where the Perl interpreter produces some lovely diagnostics. See **perldiag** (1) for more details.

−c This only runs a syntax check, like `perl -c`.

−r This copies the ePerl README to *STDOUT*.

−l This copies the ePerl licences to *STDOUT*.

−v This shows ePerl version information to *STDOUT*.

−V **−v** + shows the Perl compilation parameters.

ENVIRONMENT

Used Variables

`PATH_TRANSLATED`
This CGI/1.1 variable is used to determine the source file when ePerl operates as a NPH−CGI/1.1 program under the environment of a webserver.

Provided Variables

`SCRIPT_SRC_PATH`
The absolute pathname of the script. Use this when you want to directly access the script from within itself, for instance to do **stat** (2) and other calls.

`SCRIPT_SRC_PATH_DIR`
The directory part of `SCRIPT_SRC_PATH`. Use this one when you want to directly access other files residing in the same directory as the script, for instance to read config files, etc.

`SCRIPT_SRC_PATH_FILE`
The filename part of `SCRIPT_SRC_PATH`. Use this one when you need the basename of the script, for instance for relative self-references through URLs.

`SCRIPT_SRC_URL`
The fully-qualified URL of the script.

`SCRIPT_SRC_URL_DIR`
The directory part of `SCRIPT_SRC_URL`.

`SCRIPT_SRC_URL_FILE`
The filename part of `SCRIPT_SRC_URL`. Same as `SCRIPT_SRC_PATH_FILE`, but provided for consistency.

`SCRIPT_SRC_SIZE`
The filesize of the script, in bytes.

`SCRIPT_SRC_MODIFIED`
The last modification time of the script, in seconds since epoch.

`SCRIPT_SRC_MODIFIED_CTIME`
The last modification time of the script, in **ctime** (3) format (*WDAY MMM DD HH:MM:SS YYYY*).

`SCRIPT_SRC_MODIFIED_ISOTIME`
The last modification time of the script, in German format (*DD-MM-YYYY HH:MM*).

`SCRIPT_SRC_OWNER`
The username of the script owner or `unknown-uid-`*`123`*.

`VERSION_INTERPRETER`
The ePerl identification string.

`VERSION_LANGUAGE`
The identification string of the Perl interpreter.

Provided Built-In Images

The following built-in images can be accessed via URL `/url/to/nph-eperl/`*`NAME`*`.gif`:

`logo.gif`
The standard ePerl logo. Please do not include this one on your website.

`powered.gif`
The "*powered by ePerl 2.2*" logo. Feel free to use this on your website.

AUTHOR

```
Ralf S. Engelschall
rse@engelschall.com
www.engelschall.com
```

SEE ALSO

Parse::ePerl (3).

Web-References:

```
Perl:   perl(1),  http://www.perl.com/
ePerl:  eperl(1), http://sr.ht/˜nabijaczleweli/ossp
Apache: httpd(8), http://www.apache.org/
```

NAME

Parse::ePerl − Perl interface to the ePerl parser

SYNOPSIS

```
  use Parse::ePerl;

  $rc = Parse::ePerl::Preprocess($p);
  $rc = Parse::ePerl::Translate($p);
  $rc = Parse::ePerl::Precompile($p);
  $rc = Parse::ePerl::Evaluate($p);
  $rc = Parse::ePerl::Expand($p);
```

DESCRIPTION

Parse::ePerl is the Perl 5 interface package to the functionality of the ePerl parser (see **eperl**(1) for more details about the stand-alone program). It directly uses the parser code from ePerl to translate a sprinkled script into a plain Perl script and additionally provides functions to precompile such scripts into P−code and evaluate those scripts to a buffer.

All functions are parameterized via a hash reference `$p` which provide the necessary parameters. The result is a return code `$rc` which indicates success (1) or failure (0).

PREPROCESSOR: `$rc` = Parse::ePerl::Preprocess($p)

This is the ePerl preprocessor which expands `#include` directives. See **eperl**(1) for more details.

Possible parameters for `$p`:

Script
  Scalar holding the input script in source format.

Result
  Reference to scalar receiving the resulting script in sprinkled ePerl format.

BeginDelimiter
  Scalar specifying the begin delimiter. Default is `<:`.

EndDelimiter
  Scalar specifying the end delimiter. Default is `:>`.

INC
  A reference to a list specifying include directories. Default is `\@INC`.

TRANSLATION: `$rc` = Parse::ePerl::Translate($p)

This is the actual ePerl parser, i.e. this function converts a sprinkled ePerl-style script (provided in `$p−{`Script}> as a scalar) to a plain Perl script. The resulting script is stored into a buffer provided via a scalar reference in `$p−{`Result}>. The translation is directly done by the original C function **Bristled2Plain()** from ePerl, so the resulting script is exactly the same as with the stand-alone program *eperl*.

Possible parameters for `$p`:

Script
  Scalar holding the input script in sprinkled format.

Result
  Reference to scalar receiving the resulting script in plain Perl format.

BeginDelimiter
  Scalar specifying the begin delimiter. Default is `<:`.

EndDelimiter
  Scalar specifying the end delimiter. Default is `:>`.

CaseDelimiters
  Boolean flag indicating if the delimiters are case-sensitive (1=default) or case-insensitive (0).

Example: The following code

```
  $script = <<'EOT';
  foo
  <: print "bar"; :>
  quux
  EOT

  Parse::ePerl::Translate({
      Script => $script,
      Result => \$script,
  });
```

translates the script in `$script` to the following plain Perl format:

```
  print "foo\n";
  print "bar"; print "\n";
  print "quux\n";
```

COMPILATION: `$rc` = Parse::ePerl::Precompile($p);

This is an optional step between translation and evaluation where the plain Perl script is compiled from ASCII representation to P–code (the internal Perl bytecode). This step is used in rare cases only, for instance from within for caching purposes.

Possible parameters for `$p`:

Script
: Scalar holding the input script in plain Perl format, usually the result from a previous **Parse::ePerl::Translate** (3) call.

Result
: Reference to scalar receiving the resulting code reference. This code can be later directly used via the `&$var` construct or given to the **Parse::ePerl::Evaluate** (3) function.

Error
: Reference to scalar receiving possible error messages from the compilation (e.g. syntax errors).

Cwd
: Directory to switch to while precompiling the script.

Name
: Name of the script for informal references inside error messages.

Example: The following code

```
  Parse::ePerl::Precompile({
      Script => $script,
      Result => \$script,
  });
```

translates the plain Perl code (see above) in `$script` to a code reference and stores the reference again in `$script`. The code later can be either directly used via `&$script` instead of `eval($script)` or passed to the **Parse::ePerl::Evaluate** (3) function.

EVALUATION: `$rc` = Parse::ePerl::Evaluate($p);

Beside **Parse::ePerl::Translate** (3) this is the second main function of this package. It is intended to evaluate the result of **Parse::ePerl::Translate** (3) in a ePerl-like environment, i.e. this function tries to emulate the runtime environment and behavior of the program *eperl*. This actually means that it changes the current working directory and evaluates the script while capturing data generated on STDOUT/STDERR.

Possible parameters for `$p`:

Script
: Scalar (standard case) or reference to scalar (compiled case) holding the input script in plain Perl format or P–code, usually the result from a previous **Parse::ePerl::Translate** (3) or **Parse::ePerl::Precompile** (3) call.

Result
Reference to scalar receiving the resulting code reference.

Error
Reference to scalar receiving possible error messages from the evaluation (e.g. runtime errors).

ENV
Hash containing the environment for `%ENV` which should be used while evaluating the script.

Cwd
Directory to switch to while evaluating the script.

Name
Name of the script for informal references inside error messages.

Example: The following code

```
$script = <<'EOT';
print "foo\n";
print "bar"; print "\n";
print "quux\n";
EOT

Parse::ePerl::Evaluate({
    Script => $script,
    Result => \$script,
});
```

translates the script in `$script` to the following plain data:

```
foo
bar
quux
```

ONE-STEP EXPANSION: `$rc` = Parse::ePerl::Expand($p);

This function just combines, **Parse::ePerl::Translate** (3) and **Parse::ePerl::Evaluate** (3) into one step. The parameters in `$p` are the union of the possible parameters for both functions. This is intended as a high-level interface for Parse::ePerl.

AUTHOR

```
Ralf S. Engelschall
rse@engelschall.com
www.engelschall.com
```

SEE ALSO

eperl (1)

Web-References:

```
Perl:  perl(1),  http://www.perl.com/
ePerl: eperl(1), http://www.engelschall.com/sw/eperl/
```