NAME

perlcompile - Introduction to the Perl Compiler-Translator

DESCRIPTION

Perl has always had a compiler: your source is compiled into an internal form (a parse tree) which is then optimized before being run. Since version 5.005, Perl has shipped with a module capable of inspecting the optimized parse tree (B), and this has been used to write many useful utilities, including a module that lets you turn your Perl into C source code that can be compiled into a native executable.

The B module provides access to the parse tree, and other modules ("back ends") do things with the tree. Some write it out as bytecode, C source code, or a semi-human-readable text. Another traverses the parse tree to build a cross-reference of which subroutines, formats, and variables are used where. Another checks your code for dubious constructs. Yet another back end dumps the parse tree back out as Perl source, acting as a source code beautifier or deobfuscator.

Because its original purpose was to be a way to produce C code corresponding to a Perl program, and in turn a native executable, the B module and its associated back ends are known as "the compiler", even though they don't really compile anything. Different parts of the compiler are more accurately a "translator", or an "inspector", but people want Perl to have a "compiler option" not an "inspector gadget". What can you do?

This document covers the use of the Perl compiler: which modules it comprises, how to use the most important of the back end modules, what problems there are, and how to work around them.

Layout

The compiler back ends are in the `B::` hierarchy, and the front-end (the module that you, the user of the compiler, will sometimes interact with) is the O module. Some back ends (e.g., `B::C`) have programs (e.g., *perlcc*) to hide the modules' complexity.

Here are the important back ends to know about, with their status expressed as a number from 0 (outline for later implementation) to 10 (if there's a bug in it, we're very surprised):

B::Bytecode

Stores the parse tree in a machine-independent format, suitable for later reloading through the ByteLoader module. Status: 5 (some things work, some things don't, some things are untested).

B::C

Creates a C source file containing code to rebuild the parse tree and resume the interpreter. Status: 6 (many things work adequately, including programs using Tk).

B::CC

Creates a C source file corresponding to the run time code path in the parse tree. This is the closest to a Perl-to-C translator there is, but the code it generates is almost incomprehensible because it translates the parse tree into a giant switch structure that manipulates Perl structures. Eventual goal is to reduce (given sufficient type information in the Perl program) some of the Perl data structure manipulations into manipulations of C-level ints, floats, etc. Status: 5 (some things work, including uncomplicated Tk examples).

B::Lint

Complains if it finds dubious constructs in your source code. Status: 6 (it works adequately, but only has a very limited number of areas that it checks).

B::Deparse

Recreates the Perl source, making an attempt to format it coherently. Status: 8 (it works nicely, but a few obscure things are missing).

B::Xref

Reports on the declaration and use of subroutines and variables. Status: 8 (it works nicely, but still has a few lingering bugs).

Using The Back Ends

The following sections describe how to use the various compiler back ends. They're presented roughly in order of maturity, so that the most stable and proven back ends are described first, and the most experimental and incomplete back ends are described last.

The O module automatically enabled the **-c** flag to Perl, which prevents Perl from executing your code once it has been compiled. This is why all the back ends print:

```
myperlprogram syntax OK
```

before producing any other output.

The Cross Referencing Back End

The cross referencing back end (B::Xref) produces a report on your program, breaking down declarations and uses of subroutines and variables (and formats) by file and subroutine. For instance, here's part of the report from the *pod2man* program that comes with Perl:

```
Subroutine clear_noremap
  Package (lexical)
    $ready_to_print   i1069, 1079
  Package main
    $&                1086
    $.                1086
    $0                1086
    $1                1087
    $2                1085, 1085
    $3                1085, 1085
    $ARGV             1086
    %HTML_Escapes     1085, 1085
```

This shows the variables used in the subroutine `clear_noremap`. The variable `$ready_to_print` is a my() (lexical) variable, **i**ntroduced (first declared with my()) on line 1069, and used on line 1079. The variable `$&` from the main package is used on 1086, and so on.

A line number may be prefixed by a single letter:

i

Lexical variable introduced (declared with my()) for the first time.

&

Subroutine or method call.

s

Subroutine defined.

r

Format defined.

The most useful option the cross referencer has is to save the report to a separate file. For instance, to save the report on *myperlprogram* to the file *report*:

```
$ perl -MO=Xref,-oreport myperlprogram
```

The Decompiling Back End

The Deparse back end turns your Perl source back into Perl source. It can reformat along the way, making it useful as a de-obfuscator. The most basic way to use it is:

```
$ perl -MO=Deparse myperlprogram
```

You'll notice immediately that Perl has no idea of how to paragraph your code. You'll have to separate chunks of code from each other with newlines by hand. However, watch what it will do with one-liners:

```
$ perl -MO=Deparse -e '$op=shift||die "usage: $0
code [...]";chomp(@ARGV=<>)unless@ARGV; for(@ARGV){$was=$_;eval$op;
die$@ if$@; rename$was,$_ unless$was eq $_}'
-e syntax OK
$op = shift @ARGV || die("usage: $0 code [...]");
chomp(@ARGV = <ARGV>) unless @ARGV;
foreach $_ (@ARGV) {
    $was = $_;
    eval $op;
    die $@ if $@;
    rename $was, $_ unless $was eq $_;
}
```

The decompiler has several options for the code it generates. For instance, you can set the size of each indent from 4 (as above) to 2 with:

```
$ perl -MO=Deparse,-si2 myperlprogram
```

The **-p** option adds parentheses where normally they are omitted:

```
$ perl -MO=Deparse -e 'print "Hello, world\n"'
-e syntax OK
print "Hello, world\n";
$ perl -MO=Deparse,-p -e 'print "Hello, world\n"'
-e syntax OK
print("Hello, world\n");
```

See *B::Deparse* for more information on the formatting options.

The Lint Back End

The lint back end (B::Lint) inspects programs for poor style. One programmer's bad style is another programmer's useful tool, so options let you select what is complained about.

To run the style checker across your source code:

```
$ perl -MO=Lint myperlprogram
```

To disable context checks and undefined subroutines:

```
$ perl -MO=Lint,-context,-undefined-subs myperlprogram
```

See *B::Lint* for information on the options.

The Simple C Back End

This module saves the internal compiled state of your Perl program to a C source file, which can be turned into a native executable for that particular platform using a C compiler. The resulting program links against the Perl interpreter library, so it will not save you disk space (unless you build Perl with a

shared library) or program size. It may, however, save you startup time.

The `perlcc` tool generates such executables by default.

```
perlcc myperlprogram.pl
```

The Bytecode Back End

This back end is only useful if you also have a way to load and execute the bytecode that it produces. The ByteLoader module provides this functionality.

To turn a Perl program into executable byte code, you can use `perlcc` with the `-B` switch:

```
perlcc -B myperlprogram.pl
```

The byte code is machine independent, so once you have a compiled module or program, it is as portable as Perl source (assuming that the user of the module or program has a modern-enough Perl interpreter to decode the byte code).

See **B::Bytecode** for information on options to control the optimization and nature of the code generated by the Bytecode module.

The Optimized C Back End

The optimized C back end will turn your Perl program's run time code-path into an equivalent (but optimized) C program that manipulates the Perl data structures directly. The program will still link against the Perl interpreter library, to allow for eval(), `s///e`, `require`, etc.

The `perlcc` tool generates such executables when using the -O switch. To compile a Perl program (ending in `.pl` or `.p`):

```
perlcc -O myperlprogram.pl
```

To produce a shared library from a Perl module (ending in `.pm`):

```
perlcc -O Myperlmodule.pm
```

For more information, see *perlcc* and *B::CC*.

Module List for the Compiler Suite

B

This module is the introspective ("reflective" in Java terms) module, which allows a Perl program to inspect its innards. The back end modules all use this module to gain access to the compiled parse tree. You, the user of a back end module, will not need to interact with B.

O

This module is the front-end to the compiler's back ends. Normally called something like this:

```
$ perl -MO=Deparse myperlprogram
```

This is like saying `use O 'Deparse'` in your Perl program.

B::Asmdata

This module is used by the B::Assembler module, which is in turn used by the B::Bytecode module, which stores a parse-tree as bytecode for later loading. It's not a back end itself, but rather a component of a back end.

B::Assembler

This module turns a parse-tree into data suitable for storing and later decoding back into a

parse-tree. It's not a back end itself, but rather a component of a back end. It's used by the *assemble* program that produces bytecode.

B::Bblock

This module is used by the B::CC back end. It walks "basic blocks". A basic block is a series of operations which is known to execute from start to finish, with no possibility of branching or halting.

B::Bytecode

This module is a back end that generates bytecode from a program's parse tree. This bytecode is written to a file, from where it can later be reconstructed back into a parse tree. The goal is to do the expensive program compilation once, save the interpreter's state into a file, and then restore the state from the file when the program is to be executed. See *The Bytecode Back End* for details about usage.

B::C

This module writes out C code corresponding to the parse tree and other interpreter internal structures. You compile the corresponding C file, and get an executable file that will restore the internal structures and the Perl interpreter will begin running the program. See *The Simple C Back End* for details about usage.

B::CC

This module writes out C code corresponding to your program's operations. Unlike the B::C module, which merely stores the interpreter and its state in a C program, the B::CC module makes a C program that does not involve the interpreter. As a consequence, programs translated into C by B::CC can execute faster than normal interpreted programs. See *The Optimized C Back End* for details about usage.

B::Concise

This module prints a concise (but complete) version of the Perl parse tree. Its output is more customizable than the one of B::Terse or B::Debug (and it can emulate them). This module useful for people who are writing their own back end, or who are learning about the Perl internals. It's not useful to the average programmer.

B::Debug

This module dumps the Perl parse tree in verbose detail to STDOUT. It's useful for people who are writing their own back end, or who are learning about the Perl internals. It's not useful to the average programmer.

B::Deparse

This module produces Perl source code from the compiled parse tree. It is useful in debugging and deconstructing other people's code, also as a pretty-printer for your own source. See *The Decompiling Back End* for details about usage.

B::Disassembler

This module turns bytecode back into a parse tree. It's not a back end itself, but rather a component of a back end. It's used by the *disassemble* program that comes with the bytecode.

B::Lint

This module inspects the compiled form of your source code for things which, while some people frown on them, aren't necessarily bad enough to justify a warning. For instance, use of an array in scalar context without explicitly saying `scalar(@array)` is something that Lint can identify. See *The Lint Back End* for details about usage.

B::Showlex

This module prints out the my() variables used in a function or a file. To get a list of the my() variables used in the subroutine mysub() defined in the file myperlprogram:

```
$ perl -MO=Showlex,mysub myperlprogram
```

To get a list of the my() variables used in the file myperlprogram:

```
$ perl -MO=Showlex myperlprogram
```

[BROKEN]

B::Stackobj

This module is used by the B::CC module. It's not a back end itself, but rather a component of a back end.

B::Stash

This module is used by the *perlcc* program, which compiles a module into an executable. B::Stash prints the symbol tables in use by a program, and is used to prevent B::CC from producing C code for the B::* and O modules. It's not a back end itself, but rather a component of a back end.

B::Terse

This module prints the contents of the parse tree, but without as much information as B::Debug. For comparison, `print "Hello, world."` produced 96 lines of output from B::Debug, but only 6 from B::Terse.

This module is useful for people who are writing their own back end, or who are learning about the Perl internals. It's not useful to the average programmer.

B::Xref

This module prints a report on where the variables, subroutines, and formats are defined and used within a program and the modules it loads. See *The Cross Referencing Back End* for details about usage.

KNOWN PROBLEMS

The simple C backend currently only saves typeglobs with alphanumeric names.

The optimized C backend outputs code for more modules than it should (e.g., DirHandle). It also has little hope of properly handling `goto LABEL` outside the running subroutine (`goto &sub` is okay). `goto LABEL` currently does not work at all in this backend. It also creates a huge initialization function that gives C compilers headaches. Splitting the initialization function gives better results. Other problems include: unsigned math does not work correctly; some opcodes are handled incorrectly by default opcode handling mechanism.

BEGIN{} blocks are executed while compiling your code. Any external state that is initialized in BEGIN{}, such as opening files, initiating database connections etc., do not behave properly. To work around this, Perl has an INIT{} block that corresponds to code being executed before your program begins running but after your program has finished being compiled. Execution order: BEGIN{}, (possible save of state through compiler back-end), INIT{}, program runs, END{}.

AUTHOR

This document was originally written by Nathan Torkington, and is now maintained by the perl5-porters mailing list *perl5-porters@perl.org*.