NAME

B::Deparse - Perl compiler backend to produce perl code

SYNOPSIS

perl -MO=Deparse[**,-d**][**,-f***FILE*][**,-p**][**,-q**][**,-l**] [**,-s***LETTERS*][**,-x***LEVEL*] *prog.pl*

DESCRIPTION

B::Deparse is a backend module for the Perl compiler that generates perl source code, based on the internal compiled structure that perl itself creates after parsing a program. The output of B::Deparse won't be exactly the same as the original source, since perl doesn't keep track of comments or whitespace, and there isn't a one-to-one correspondence between perl's syntactical constructions and their compiled form, but it will often be close. When you use the **-p** option, the output also includes parentheses even when they are not required by precedence, which can make it easy to see if perl is parsing your expressions the way you intended.

While B::Deparse goes to some lengths to try to figure out what your original program was doing, some parts of the language can still trip it up; it still fails even on some parts of Perl's own test suite. If you encounter a failure other than the most common ones described in the BUGS section below, you can help contribute to B::Deparse's ongoing development by submitting a bug report with a small example.

OPTIONS

As with all compiler backend options, these must follow directly after the '-MO=Deparse', separated by a comma but not any white space.

-d

Output data values (when they appear as constants) using Data::Dumper. Without this option, B::Deparse will use some simple routines of its own for the same purpose. Currently, Data::Dumper is better for some kinds of data (such as complex structures with sharing and self-reference) while the built-in routines are better for others (such as odd floating-point values).

-f*FILE*

Normally, B::Deparse deparses the main code of a program, and all the subs defined in the same file. To include subs defined in other files, pass the **-f** option with the filename. You can pass the **-f** option several times, to include more than one secondary file. (Most of the time you don't want to use it at all.) You can also use this option to include subs which are defined in the scope of a **#line** directive with two parameters.

-l

Add '#line' declarations to the output based on the line and file locations of the original code.

-p

Print extra parentheses. Without this option, B::Deparse includes parentheses in its output only when they are needed, based on the structure of your program. With **-p**, it uses parentheses (almost) whenever they would be legal. This can be useful if you are used to LISP, or if you want to see how perl parses your input. If you say

```
if ($var & 0x7f == 65) {print "Gimme an A!"}
print ($which ? $a : $b), "\n";
$name = $ENV{USER} or "Bob";
```

`B::Deparse,-p` will print

```
if (($var & 0)) {
    print('Gimme an A!')
};
```

```
    (print(($which ? $a : $b)), '???');
    (($name = $ENV{'USER'}) or '???')
```

which probably isn't what you intended (the `'???'` is a sign that perl optimized away a constant value).

-P

Disable prototype checking. With this option, all function calls are deparsed as if no prototype was defined for them. In other words,

```
    perl -MO=Deparse,-P -e 'sub foo (\@) { 1 } foo @x'
```

will print

```
    sub foo (\@) {
1;
    }
    &foo(\@x);
```

making clear how the parameters are actually passed to `foo`.

-q

Expand double-quoted strings into the corresponding combinations of concatenation, uc, ucfirst, lc, lcfirst, quotemeta, and join. For instance, print

```
    print "Hello, $world, @ladies, \u$gentlemen\E, \u\L$me!";
```

as

```
    print 'Hello, ' . $world . ', ' . join($", @ladies) . ', '
          . ucfirst($gentlemen) . ', ' . ucfirst(lc $me . '!');
```

Note that the expanded form represents the way perl handles such constructions internally -- this option actually turns off the reverse translation that B::Deparse usually does. On the other hand, note that `$x = "$y"` is not the same as `$x = $y`: the former makes the value of $y into a string before doing the assignment.

-s*LETTERS*

Tweak the style of B::Deparse's output. The letters should follow directly after the 's', with no space or punctuation. The following options are available:

C

Cuddle `elsif`, `else`, and `continue` blocks. For example, print

```
    if (...) {
        ...
    } else {
        ...
    }
```

instead of

```
    if (...) {
        ...
    }
    else {
        ...
    }
```

The default is not to cuddle.

i*NUMBER*

Indent lines by multiples of *NUMBER* columns. The default is 4 columns.

T

Use tabs for each 8 columns of indent. The default is to use only spaces. For instance, if the style options are **-si4T**, a line that's indented 3 times will be preceded by one tab and four spaces; if the options were **-si8T**, the same line would be preceded by three tabs.

v*STRING***.**

Print *STRING* for the value of a constant that can't be determined because it was optimized away (mnemonic: this happens when a constant is used in **v**oid context). The end of the string is marked by a period. The string should be a valid perl expression, generally a constant. Note that unless it's a number, it probably needs to be quoted, and on a command line quotes need to be protected from the shell. Some conventional values include 0, 1, 42, '', 'foo', and 'Useless use of constant omitted' (which may need to be **-sv"'Useless use of constant omitted'."** or something similar depending on your shell). The default is '???'. If you're using B::Deparse on a module or other file that's require'd, you shouldn't use a value that evaluates to false, since the customary true constant at the end of a module will be in void context when the file is compiled as a main program.

-x*LEVEL*

Expand conventional syntax constructions into equivalent ones that expose their internal operation. *LEVEL* should be a digit, with higher values meaning more expansion. As with **-q**, this actually involves turning off special cases in B::Deparse's normal operations.

If *LEVEL* is at least 3, `for` loops will be translated into equivalent while loops with continue blocks; for instance

```
for ($i = 0; $i < 10; ++$i) {
    print $i;
}
```

turns into

```
$i = 0;
while ($i < 10) {
    print $i;
} continue {
    ++$i
}
```

Note that in a few cases this translation can't be perfectly carried back into the source code -- if the loop's initializer declares a my variable, for instance, it won't have the correct scope outside of the loop.

If *LEVEL* is at least 5, `use` declarations will be translated into `BEGIN` blocks containing calls to `require` and `import`; for instance,

```
use strict 'refs';
```

turns into

```
sub BEGIN {
    require strict;
    do {
        'strict'->import('refs')
    };
```

```
}
```

If *LEVEL* is at least 7, `if` statements will be translated into equivalent expressions using `&&`, `?:` and `do {}`; for instance

```
print 'hi' if $nice;
if ($nice) {
    print 'hi';
}
if ($nice) {
    print 'hi';
} else {
    print 'bye';
}
```

turns into

```
$nice and print 'hi';
$nice and do { print 'hi' };
$nice ? do { print 'hi' } : do { print 'bye' };
```

Long sequences of elsifs will turn into nested ternary operators, which B::Deparse doesn't know how to indent nicely.

USING B::Deparse AS A MODULE

Synopsis

```
use B::Deparse;
$deparse = B::Deparse->new("-p", "-sC");
$body = $deparse->coderef2text(\&func);
eval "sub func $body"; # the inverse operation
```

Description

B::Deparse can also be used on a sub-by-sub basis from other perl programs.

new

```
$deparse = B::Deparse->new(OPTIONS)
```

Create an object to store the state of a deparsing operation and any options. The options are the same as those that can be given on the command line (see *OPTIONS*); options that are separated by commas after **-MO=Deparse** should be given as separate strings. Some options, like **-u**, don't make sense for a single subroutine, so don't pass them.

ambient_pragmas

```
$deparse->ambient_pragmas(strict => 'all', '$[' => $[);
```

The compilation of a subroutine can be affected by a few compiler directives, **pragmas**. These are:

- use strict;
- use warnings;
- Assigning to the special variable $[
- use integer;
- use bytes;

- use utf8;
- use re;

Ordinarily, if you use B::Deparse on a subroutine which has been compiled in the presence of one or more of these pragmas, the output will include statements to turn on the appropriate directives. So if you then compile the code returned by coderef2text, it will behave the same way as the subroutine which you deparsed.

However, you may know that you intend to use the results in a particular context, where some pragmas are already in scope. In this case, you use the **ambient_pragmas** method to describe the assumptions you wish to make.

Not all of the options currently have any useful effect. See *BUGS* for more details.

The parameters it accepts are:

strict

Takes a string, possibly containing several values separated by whitespace. The special values "all" and "none" mean what you'd expect.

```
$deparse->ambient_pragmas(strict => 'subs refs');
```

$[

Takes a number, the value of the array base $[.

bytes

utf8

integer

If the value is true, then the appropriate pragma is assumed to be in the ambient scope, otherwise not.

re

Takes a string, possibly containing a whitespace-separated list of values. The values "all" and "none" are special. It's also permissible to pass an array reference here.

```
$deparser->ambient_pragmas(re => 'eval');
```

warnings

Takes a string, possibly containing a whitespace-separated list of values. The values "all" and "none" are special, again. It's also permissible to pass an array reference here.

```
$deparser->ambient_pragmas(warnings => [qw[void io]]);
```

If one of the values is the string "FATAL", then all the warnings in that list will be considered fatal, just as with the **warnings** pragma itself. Should you need to specify that some warnings are fatal, and others are merely enabled, you can pass the **warnings** parameter twice:

```
    $deparser->ambient_pragmas(
 warnings => 'all',
 warnings => [FATAL => qw/void io/],
    );
```

See *perllexwarn* for more information about lexical warnings.

hint_bits

warning_bits

These two parameters are used to specify the ambient pragmas in the format used by the

special variables $^H and ${^WARNING_BITS}.

They exist principally so that you can write code like:

```
    { my ($hint_bits, $warning_bits);
    BEGIN {($hint_bits, $warning_bits) = ($^H, ${^WARNING_BITS})}
    $deparser->ambient_pragmas (
 hint_bits    => $hint_bits,
 warning_bits => $warning_bits,
 '$['         => 0 + $[
    ); }
```

which specifies that the ambient pragmas are exactly those which are in scope at the point of calling.

coderef2text

```
$body = $deparse->coderef2text(\&func)
$body = $deparse->coderef2text(sub ($$) { ... })
```

Return source code for the body of a subroutine (a block, optionally preceded by a prototype in parens), given a reference to the sub. Because a subroutine can have no names, or more than one name, this method doesn't return a complete subroutine definition -- if you want to eval the result, you should prepend "sub subname ", or "sub " for an anonymous function constructor. Unless the sub was defined in the main:: package, the code will include a package declaration.

BUGS

- The only pragmas to be completely supported are: `use warnings`, `use strict 'refs'`, `use bytes`, and `use integer`. (`$[`, which behaves like a pragma, is also supported.)

  Excepting those listed above, we're currently unable to guarantee that B::Deparse will produce a pragma at the correct point in the program. (Specifically, pragmas at the beginning of a block often appear right before the start of the block instead.) Since the effects of pragmas are often lexically scoped, this can mean that the pragma holds sway over a different portion of the program than in the input file.

- In fact, the above is a specific instance of a more general problem: we can't guarantee to produce BEGIN blocks or `use` declarations in exactly the right place. So if you use a module which affects compilation (such as by over-riding keywords, overloading constants or whatever) then the output code might not work as intended.

  This is the most serious outstanding problem, and will require some help from the Perl core to fix.

- If a keyword is over-ridden, and your program explicitly calls the built-in version by using CORE::keyword, the output of B::Deparse will not reflect this. If you run the resulting code, it will call the over-ridden version rather than the built-in one. (Maybe there should be an option to **always** print keyword calls as `CORE::name`.)

- Some constants don't print correctly either with or without **-d**. For instance, neither B::Deparse nor Data::Dumper know how to print dual-valued scalars correctly, as in:

  ```
  use constant E2BIG => ($!=7); $y = E2BIG; print $y, 0+$y;
  ```

- An input file that uses source filtering probably won't be deparsed into runnable code, because it will still include the **use** declaration for the source filtering module, even though the code that is produced is already ordinary Perl which shouldn't be filtered again.

- Optimised away statements are rendered as '???'. This includes statements that have a compile-time side-effect, such as the obscure

```
my $x if 0;
```

which is not, consequently, deparsed correctly.

- There are probably many more bugs on non-ASCII platforms (EBCDIC).

AUTHOR

Stephen McCamant <smcc@CSUA.Berkeley.EDU>, based on an earlier version by Malcolm Beattie <mbeattie@sable.ox.ac.uk>, with contributions from Gisle Aas, James Duncan, Albert Dvornik, Robin Houston, Dave Mitchell, Hugo van der Sanden, Gurusamy Sarathy, Nick Ing-Simmons, and Rafael Garcia-Suarez.