NAME

perlsyn - Perl syntax

DESCRIPTION

A Perl program consists of a sequence of declarations and statements which run from the top to the bottom. Loops, subroutines, and other control structures allow you to jump around within the code.

Perl is a **free-form** language: you can format and indent it however you like. Whitespace serves mostly to separate tokens, unlike languages like Python where it is an important part of the syntax, or Fortran where it is immaterial.

Many of Perl's syntactic elements are **optional**. Rather than requiring you to put parentheses around every function call and declare every variable, you can often leave such explicit elements off and Perl will figure out what you meant. This is known as **Do What I Mean**, abbreviated **DWIM**. It allows programmers to be **lazy** and to code in a style with which they are comfortable.

Perl **borrows syntax** and concepts from many languages: awk, sed, C, Bourne Shell, Smalltalk, Lisp and even English. Other languages have borrowed syntax from Perl, particularly its regular expression extensions. So if you have programmed in another language you will see familiar pieces in Perl. They often work the same, but see perltrap for information about how they differ.

Declarations

The only things you need to declare in Perl are report formats and subroutines (and sometimes not even subroutines). A scalar variable holds the undefined value (`undef`) until it has been assigned a defined value, which is anything other than `undef`. When used as a number, `undef` is treated as `0`; when used as a string, it is treated as the empty string, `""`; and when used as a reference that isn't being assigned to, it is treated as an error. If you enable warnings, you'll be notified of an uninitialized value whenever you treat `undef` as a string or a number. Well, usually. Boolean contexts, such as:

```
if ($a) {}
```

are exempt from warnings (because they care about truth rather than definedness). Operators such as `++`, `--`, `+=`, `-=`, and `.=`, that operate on undefined variables such as:

```
undef $a;
$a++;
```

are also always exempt from such warnings.

A declaration can be put anywhere a statement can, but has no effect on the execution of the primary sequence of statements: declarations all take effect at compile time. All declarations are typically put at the beginning or the end of the script. However, if you're using lexically-scoped private variables created with `my()`, `state()`, or `our()`, you'll have to make sure your format or subroutine definition is within the same block scope as the my if you expect to be able to access those private variables.

Declaring a subroutine allows a subroutine name to be used as if it were a list operator from that point forward in the program. You can declare a subroutine without defining it by saying `sub name`, thus:

```
sub myname;
$me = myname $0 or die "can't get myname";
```

A bare declaration like that declares the function to be a list operator, not a unary operator, so you have to be careful to use parentheses (or `or` instead of `||`.) The `||` operator binds too tightly to use after list operators; it becomes part of the last element. You can always use parentheses around the list operators arguments to turn the list operator back into something that behaves more like a function call. Alternatively, you can use the prototype `($)` to turn the subroutine into a unary operator:

```
sub myname ($);
$me = myname $0 || die "can't get myname";
```

That now parses as you'd expect, but you still ought to get in the habit of using parentheses in that situation. For more on prototypes, see perlsub.

Subroutines declarations can also be loaded up with the `require` statement or both loaded and imported

into your namespace with a `use` statement. See perlmod(1) for details on this.

A statement sequence may contain declarations of lexically-scoped variables, but apart from declaring a variable name, the declaration acts like an ordinary statement, and is elaborated within the sequence of statements as if it were an ordinary statement. That means it actually has both compile-time and run-time effects.

Comments

Text from a `"#"` character until the end of the line is a comment, and is ignored. Exceptions include `"#"` inside a string or regular expression.

Simple Statements

The only kind of simple statement is an expression evaluated for its side-effects. Every simple statement must be terminated with a semicolon, unless it is the final statement in a block, in which case the semicolon is optional. But put the semicolon in anyway if the block takes up more than one line, because you may eventually add another line. Note that there are operators like `eval {}`, `sub {}`, and `do {}` that *look* like compound statements, but aren't — they're just TERMs in an expression — and thus need an explicit termination when used as the last item in a statement.

Truth and Falsehood

The number 0, the strings `'0'` and `""`, the empty list `()`, and `undef` are all false in a boolean context. All other values are true. Negation of a true value by `!` or `not` returns a special false value. When evaluated as a string it is treated as `""`, but as a number, it is treated as 0. Most Perl operators that return true or false behave this way.

Statement Modifiers

Any simple statement may optionally be followed by a *SINGLE* modifier, just before the terminating semicolon (or block ending). The possible modifiers are:

```
if EXPR
unless EXPR
while EXPR
until EXPR
for LIST
foreach LIST
when EXPR
```

The `EXPR` following the modifier is referred to as the "condition". Its truth or falsehood determines how the modifier will behave.

`if` executes the statement once *if* and only if the condition is true. `unless` is the opposite, it executes the statement *unless* the condition is true (that is, if the condition is false).

```
print "Basset hounds got long ears" if length $ear >= 10;
go_outside() and play() unless $is_raining;
```

The `for(each)` modifier is an iterator: it executes the statement once for each item in the LIST (with `$_` aliased to each item in turn).

```
print "Hello $_!\n" for qw(world Dolly nurse);
```

`while` repeats the statement *while* the condition is true. `until` does the opposite, it repeats the statement *until* the condition is true (or while the condition is false):

```
# Both of these count from 0 to 10.
print $i++ while $i <= 10;
print $j++ until $j > 10;
```

The `while` and `until` modifiers have the usual "`while` loop" semantics (conditional evaluated first), except when applied to a `do`-BLOCK (or to the Perl4 `do`-SUBROUTINE statement), in which case the block executes once before the conditional is evaluated.

This is so that you can write loops like:

```
do {
$line = <STDIN>;
...
} until !defined($line) || $line eq ".\n"
```

See ‘‘do’’ in perlfunc. Note also that the loop control statements described later will *NOT* work in this construct, because modifiers don’t take loop labels. Sorry. You can always put another block inside of it (for `next`/`redo`) or around it (for `last`) to do that sort of thing.

For `next` or `redo`, just double the braces:

```
do {{
next if $x == $y;
# do something here
}} until $x++ > $z;
```

For `last`, you have to be more elaborate and put braces around it:

```
{
do {
last if $x == $y**2;
# do something here
} while $x++ <= $z;
}
```

If you need both `next` and `last`, you have to do both and also use a loop label:

```
LOOP: {
do {{
next if $x == $y;
last LOOP if $x == $y**2;
# do something here
}} until $x++ > $z;
}
```

NOTE: The behaviour of a `my`, `state`, or `our` modified with a statement modifier conditional or loop construct (for example, `my $x if ...`) is **undefined**. The value of the `my` variable may be `undef`, any previously assigned value, or possibly anything else. Don’t rely on it. Future versions of perl might do something different from the version of perl you try it out on. Here be dragons.

The `when` modifier is an experimental feature that first appeared in Perl 5.14. To use it, you should include a `use v5.14` declaration. (Technically, it requires only the `switch` feature, but that aspect of it was not available before 5.14.) Operative only from within a `foreach` loop or a `given` block, it executes the statement only if the smartmatch `$_ ~~` *EXPR* is true. If the statement executes, it is followed by a `next` from inside a `foreach` and `break` from inside a `given`.

Under the current implementation, the `foreach` loop can be anywhere within the `when` modifier’s dynamic scope, but must be within the `given` block’s lexical scope. This restricted may be relaxed in a future release. See ‘‘Switch Statements’’ below.

Compound Statements

In Perl, a sequence of statements that defines a scope is called a block. Sometimes a block is delimited by the file containing it (in the case of a required file, or the program as a whole), and sometimes a block is delimited by the extent of a string (in the case of an eval).

But generally, a block is delimited by curly brackets, also known as braces. We will call this syntactic construct a BLOCK.

The following compound statements may be used to control flow:

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ...
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK

unless (EXPR) BLOCK
unless (EXPR) BLOCK else BLOCK
unless (EXPR) BLOCK elsif (EXPR) BLOCK ...
unless (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK

given (EXPR) BLOCK

LABEL while (EXPR) BLOCK
LABEL while (EXPR) BLOCK continue BLOCK

LABEL until (EXPR) BLOCK
LABEL until (EXPR) BLOCK continue BLOCK

LABEL for (EXPR; EXPR; EXPR) BLOCK
LABEL for VAR (LIST) BLOCK
LABEL for VAR (LIST) BLOCK continue BLOCK

LABEL foreach (EXPR; EXPR; EXPR) BLOCK
LABEL foreach VAR (LIST) BLOCK
LABEL foreach VAR (LIST) BLOCK continue BLOCK

LABEL BLOCK
LABEL BLOCK continue BLOCK

PHASE BLOCK
```

The experimental `given` statement is *not automatically enabled*; see ‘‘Switch Statements’’ below for how to do so, and the attendant caveats.

Unlike in C and Pascal, in Perl these are all defined in terms of BLOCKs, not statements. This means that the curly brackets are *required*--no dangling statements allowed. If you want to write conditionals without curly brackets, there are several other ways to do it. The following all do the same thing:

```
if (!open(FOO)) { die "Can't open $FOO: $!" }
die "Can't open $FOO: $!" unless open(FOO);
open(FOO) || die "Can't open $FOO: $!";
open(FOO) ? () : die "Can't open $FOO: $!";
# a bit exotic, that last one
```

The `if` statement is straightforward. Because BLOCKs are always bounded by curly brackets, there is never any ambiguity about which `if` an `else` goes with. If you use `unless` in place of `if`, the sense of the test is reversed. Like `if`, `unless` can be followed by `else`. `unless` can even be followed by one or more `elsif` statements, though you may want to think twice before using that particular language construct, as everyone reading your code will have to think at least twice before they can understand what’s going on.

The `while` statement executes the block as long as the expression is true. The `until` statement executes the block as long as the expression is false. The LABEL is optional, and if present, consists of an identifier followed by a colon. The LABEL identifies the loop for the loop control statements `next`, `last`, and `redo`. If the LABEL is omitted, the loop control statement refers to the innermost enclosing loop. This may include dynamically looking back your call-stack at run time to find the LABEL. Such desperate behavior triggers a warning if you use the `use warnings` pragma or the **-w** flag.

If there is a `continue` BLOCK, it is always executed just before the conditional is about to be evaluated again. Thus it can be used to increment a loop variable, even when the loop has been continued via the `next` statement.

When a block is preceding by a compilation phase keyword such as `BEGIN`, `END`, `INIT`, `CHECK`, or `UNITCHECK`, then the block will run only during the corresponding phase of execution. See perlmod(1) for more details.

Extension modules can also hook into the Perl parser to define new kinds of compound statements. These are introduced by a keyword which the extension recognizes, and the syntax following the keyword is defined entirely by the extension. If you are an implementor, see ''PL_keyword_plugin'' in perlapi(1) for the mechanism. If you are using such a module, see the module's documentation for details of the syntax that it defines.

Loop Control

The `next` command starts the next iteration of the loop:

```
LINE: while (<STDIN>) {
next LINE if /^#/; # discard comments
...
}
```

The `last` command immediately exits the loop in question. The `continue` block, if any, is not executed:

```
LINE: while (<STDIN>) {
last LINE if /^$/; # exit when done with header
...
}
```

The `redo` command restarts the loop block without evaluating the conditional again. The `continue` block, if any, is *not* executed. This command is normally used by programs that want to lie to themselves about what was just input.

For example, when processing a file like */etc/termcap*. If your input lines might end in backslashes to indicate continuation, you want to skip ahead and get the next record.

```
while (<>) {
chomp;
if (s/\\$//) {
$_ .= <>;
redo unless eof();
}
# now process $_
}
```

which is Perl shorthand for the more explicitly written version:

```
LINE: while (defined($line = <ARGV>)) {
chomp($line);
if ($line =~ s/\\$//) {
$line .= <ARGV>;
redo LINE unless eof(); # not eof(ARGV)!
}
# now process $line
}
```

Note that if there were a `continue` block on the above code, it would get executed only on lines discarded by the regex (since redo skips the continue block). A continue block is often used to reset line counters or `m?pat?` one-time matches:

```
# inspired by :1,$g/fred/s//WILMA/
while (<>) {
m?(fred)? && s//WILMA $1 WILMA/;
m?(barney)? && s//BETTY $1 BETTY/;
m?(homer)? && s//MARGE $1 MARGE/;
} continue {
print "$ARGV $.: $_";
close ARGV if eof; # reset $.
reset if eof; # reset ?pat?
}
```

If the word `while` is replaced by the word `until`, the sense of the test is reversed, but the conditional is still tested before the first iteration.

Loop control statements don't work in an `if` or `unless`, since they aren't loops. You can double the braces to make them such, though.

```
if (/pattern/) {{
last if /fred/;
next if /barney/; # same effect as "last",
# but doesn't document as well
# do something here
}}
```

This is caused by the fact that a block by itself acts as a loop that executes once, see ''Basic BLOCKs''.

The form `while/if BLOCK BLOCK`, available in Perl 4, is no longer available. Replace any occurrence of `if BLOCK` by `if (do BLOCK)`.

For Loops

Perl's C-style `for` loop works like the corresponding `while` loop; that means that this:

```
for ($i = 1; $i < 10; $i++) {
...
}
```

is the same as this:

```
$i = 1;
while ($i < 10) {
...
} continue {
$i++;
}
```

There is one minor difference: if variables are declared with `my` in the initialization section of the `for`, the lexical scope of those variables is exactly the `for` loop (the body of the loop and the control sections).

As a special case, if the test in the `for` loop (or the corresponding `while` loop) is empty, it is treated as true. That is, both

```
for (;;) {
...
}
```

and

```
while () {
...
}
```

are treated as infinite loops.

Besides the normal array index looping, `for` can lend itself to many other interesting applications. Here's

one that avoids the problem you get into if you explicitly test for end-of-file on an interactive file descriptor causing your program to appear to hang.

```
$on_a_tty = -t STDIN && -t STDOUT;
sub prompt { print "yes? " if $on_a_tty }
for ( prompt(); <STDIN>; prompt() ) {
# do something
}
```

Using `readline` (or the operator form, `<EXPR>`) as the conditional of a `for` loop is shorthand for the following. This behaviour is the same as a `while` loop conditional.

```
for ( prompt(); defined( $_ = <STDIN> ); prompt() ) {
# do something
}
```

Foreach Loops

The `foreach` loop iterates over a normal list value and sets the scalar variable VAR to be each element of the list in turn. If the variable is preceded with the keyword `my`, then it is lexically scoped, and is therefore visible only within the loop. Otherwise, the variable is implicitly local to the loop and regains its former value upon exiting the loop. If the variable was previously declared with `my`, it uses that variable instead of the global one, but it's still localized to the loop. This implicit localization occurs *only* in a `foreach` loop.

The `foreach` keyword is actually a synonym for the `for` keyword, so you can use either. If VAR is omitted, `$_` is set to each value.

If any element of LIST is an lvalue, you can modify it by modifying VAR inside the loop. Conversely, if any element of LIST is NOT an lvalue, any attempt to modify that element will fail. In other words, the `foreach` loop index variable is an implicit alias for each item in the list that you're looping over.

If any part of LIST is an array, `foreach` will get very confused if you add or remove elements within the loop body, for example with `splice`. So don't do that.

`foreach` probably won't do what you expect if VAR is a tied or other special variable. Don't do that either.

As of Perl 5.22, there is an experimental variant of this loop that accepts a variable preceded by a backslash for VAR, in which case the items in the LIST must be references. The backslashed variable will become an alias to each referenced item in the LIST, which must be of the correct type. The variable needn't be a scalar in this case, and the backslash may be followed by `my`. To use this form, you must enable the `refaliasing` feature via `use feature`. (See feature. See also "Assigning to References" in perlref.)

Examples:

```
for (@ary) { s/foo/bar/ }

for my $elem (@elements) {
$elem *= 2;
}

for $count (reverse(1..10), "BOOM") {
print $count, "\n";
sleep(1)
}

for (1..15) { print "Merry Christmas\n"; }

foreach $item (split(/:[\\\n:]*/, $ENV{TERMCAP})) {
print "Item: $item\n";
}
```

```
use feature "refaliasing";
no warnings "experimental::refaliasing";
foreach \my %hash (@array_of_hash_references) {
# do something which each %hash
}
```

Here's how a C programmer might code up a particular algorithm in Perl:

```
for (my $i = 0; $i < @ary1; $i++) {
for (my $j = 0; $j < @ary2; $j++) {
if ($ary1[$i] > $ary2[$j]) {
last; # can't go to outer :-(
}
$ary1[$i] += $ary2[$j];
}
# this is where that last takes me
}
```

Whereas here's how a Perl programmer more comfortable with the idiom might do it:

```
OUTER: for my $wid (@ary1) {
INNER: for my $jet (@ary2) {
next OUTER if $wid > $jet;
$wid += $jet;
}
}
```

See how much easier this is? It's cleaner, safer, and faster. It's cleaner because it's less noisy. It's safer because if code gets added between the inner and outer loops later on, the new code won't be accidentally executed. The `next` explicitly iterates the other loop rather than merely terminating the inner one. And it's faster because Perl executes a `foreach` statement more rapidly than it would the equivalent `for` loop.

Perceptive Perl hackers may have noticed that a `for` loop has a return value, and that this value can be captured by wrapping the loop in a `do` block. The reward for this discovery is this cautionary advice: The return value of a `for` loop is unspecified and may change without notice. Do not rely on it.

Basic BLOCKs

A BLOCK by itself (labeled or not) is semantically equivalent to a loop that executes once. Thus you can use any of the loop control statements in it to leave or restart the block. (Note that this is *NOT* true in `eval{}`, `sub{}`, or contrary to popular belief `do{}` blocks, which do *NOT* count as loops.) The `continue` block is optional.

The BLOCK construct can be used to emulate case structures.

```
SWITCH: {
if (/^abc/) { $abc = 1; last SWITCH; }
if (/^def/) { $def = 1; last SWITCH; }
if (/^xyz/) { $xyz = 1; last SWITCH; }
$nothing = 1;
}
```

You'll also find that `foreach` loop used to create a topicalizer and a switch:

```
SWITCH:
for ($var) {
if (/^abc/) { $abc = 1; last SWITCH; }
if (/^def/) { $def = 1; last SWITCH; }
if (/^xyz/) { $xyz = 1; last SWITCH; }
$nothing = 1;
}
```

Such constructs are quite frequently used, both because older versions of Perl had no official `switch`

statement, and also because the new version described immediately below remains experimental and can sometimes be confusing.

Switch Statements

Starting from Perl 5.10.1 (well, 5.10.0, but it didn't work right), you can say

```
use feature "switch";
```

to enable an experimental switch feature. This is loosely based on an old version of a Perl 6 proposal, but it no longer resembles the Perl 6 construct. You also get the switch feature whenever you declare that your code prefers to run under a version of Perl that is 5.10 or later. For example:

```
use v5.14;
```

Under the ''switch'' feature, Perl gains the experimental keywords `given`, `when`, `default`, `continue`, and `break`. Starting from Perl 5.16, one can prefix the switch keywords with `CORE::` to access the feature without a `use feature` statement. The keywords `given` and `when` are analogous to `switch` and `case` in other languages, so the code in the previous section could be rewritten as

```
use v5.10.1;
for ($var) {
when (/^abc/) { $abc = 1 }
when (/^def/) { $def = 1 }
when (/^xyz/) { $xyz = 1 }
default { $nothing = 1 }
}
```

The `foreach` is the non-experimental way to set a topicalizer. If you wish to use the highly experimental `given`, that could be written like this:

```
use v5.10.1;
given ($var) {
when (/^abc/) { $abc = 1 }
when (/^def/) { $def = 1 }
when (/^xyz/) { $xyz = 1 }
default { $nothing = 1 }
}
```

As of 5.14, that can also be written this way:

```
use v5.14;
for ($var) {
$abc = 1 when /^abc/;
$def = 1 when /^def/;
$xyz = 1 when /^xyz/;
default { $nothing = 1 }
}
```

Or if you don't care to play it safe, like this:

```
use v5.14;
given ($var) {
$abc = 1 when /^abc/;
$def = 1 when /^def/;
$xyz = 1 when /^xyz/;
default { $nothing = 1 }
}
```

The arguments to `given` and `when` are in scalar context, and `given` assigns the `$_` variable its topic value.

Exactly what the *EXPR* argument to `when` does is hard to describe precisely, but in general, it tries to guess what you want done. Sometimes it is interpreted as `$_ ~~` *EXPR*, and sometimes it is not. It also behaves

differently when lexically enclosed by a `given` block than it does when dynamically enclosed by a `foreach` loop. The rules are far too difficult to understand to be described here. See ‘‘Experimental Details on given and when’’ later on.

Due to an unfortunate bug in how `given` was implemented between Perl 5.10 and 5.16, under those implementations the version of `$_` governed by `given` is merely a lexically scoped copy of the original, not a dynamically scoped alias to the original, as it would be if it were a `foreach` or under both the original and the current Perl 6 language specification. This bug was fixed in Perl 5.18 (and lexicalized `$_` itself was removed in Perl 5.24).

If your code still needs to run on older versions, stick to `foreach` for your topicalizer and you will be less unhappy.

Goto

Although not for the faint of heart, Perl does support a `goto` statement. There are three forms: `goto`-LABEL, `goto`-EXPR, and `goto`-&NAME. A loop’s LABEL is not actually a valid target for a `goto`; it’s just the name of the loop.

The `goto`-LABEL form finds the statement labeled with LABEL and resumes execution there. It may not be used to go into any construct that requires initialization, such as a subroutine or a `foreach` loop. It also can’t be used to go into a construct that is optimized away. It can be used to go almost anywhere else within the dynamic scope, including out of subroutines, but it’s usually better to use some other construct such as `last` or `die`. The author of Perl has never felt the need to use this form of `goto` (in Perl, that is — C is another matter).

The `goto`-EXPR form expects a label name, whose scope will be resolved dynamically. This allows for computed `goto`s per FORTRAN, but isn’t necessarily recommended if you’re optimizing for maintainability:

```
goto(("FOO", "BAR", "GLARCH")[$i]);
```

The `goto`-&NAME form is highly magical, and substitutes a call to the named subroutine for the currently running subroutine. This is used by `AUTOLOAD()` subroutines that wish to load another subroutine and then pretend that the other subroutine had been called in the first place (except that any modifications to `@_` in the current subroutine are propagated to the other subroutine.) After the `goto`, not even `caller()` will be able to tell that this routine was called first.

In almost all cases like this, it’s usually a far, far better idea to use the structured control flow mechanisms of `next`, `last`, or `redo` instead of resorting to a `goto`. For certain applications, the catch and throw pair of `eval{}` and *die()* for exception processing can also be a prudent approach.

The Ellipsis Statement

Beginning in Perl 5.12, Perl accepts an ellipsis, `"..."`, as a placeholder for code that you haven’t implemented yet. This form of ellipsis, the unimplemented statement, should not be confused with the binary flip-flop `...` operator. One is a statement and the other an operator. (Perl doesn’t usually confuse them because usually Perl can tell whether it wants an operator or a statement, but see below for exceptions.)

When Perl 5.12 or later encounters an ellipsis statement, it parses this without error, but if and when you should actually try to execute it, Perl throws an exception with the text `Unimplemented`:

```
use v5.12;
sub unimplemented { ... }
eval { unimplemented() };
if ($@ =~ /^Unimplemented at /) {
say "I found an ellipsis!";
}
```

You can only use the elliptical statement to stand in for a complete statement. These examples of how the ellipsis works:

```
use v5.12;
{ ... }
sub foo { ... }
...;
eval { ... };
sub somemeth {
my $self = shift;
...;
}
$x = do {
my $n;
...;
say "Hurrah!";
$n;
};
```

The elliptical statement cannot stand in for an expression that is part of a larger statement, since the `...` is also the three-dot version of the flip-flop operator (see ‘‘Range Operators’’ in perlop).

These examples of attempts to use an ellipsis are syntax errors:

```
use v5.12;

print ...;
open(my $fh, ">", "/dev/passwd") or ...;
if ($condition && ... ) { say "Howdy" };
```

There are some cases where Perl can’t immediately tell the difference between an expression and a statement. For instance, the syntax for a block and an anonymous hash reference constructor look the same unless there’s something in the braces to give Perl a hint. The ellipsis is a syntax error if Perl doesn’t guess that the `{ ... }` is a block. In that case, it doesn’t think the `...` is an ellipsis because it’s expecting an expression instead of a statement:

```
@transformed = map { ... } @input; # syntax error
```

Inside your block, you can use a `;` before the ellipsis to denote that the `{ ... }` is a block and not a hash reference constructor. Now the ellipsis works:

```
@transformed = map {; ... } @input; # ';' disambiguates
```

Note: Some folks colloquially refer to this bit of punctuation as a ‘‘yada-yada’’ or ‘‘triple-dot’’, but its true name is actually an ellipsis.

PODs: Embedded Documentation

Perl has a mechanism for intermixing documentation with source code. While it’s expecting the beginning of a new statement, if the compiler encounters a line that begins with an equal sign and a word, like this

```
=head1 Here There Be Pods!
```

Then that text and all remaining text up through and including a line beginning with `=cut` will be ignored. The format of the intervening text is described in perlpod.

This allows you to intermix your source code and your documentation text freely, as in

```
=item snazzle($)

The snazzle() function will behave in the most spectacular
form that you can possibly imagine, not even excepting
cybernetic pyrotechnics.

=cut back to the compiler, nuff of this pod stuff!
```

```
sub snazzle($) {
my $thingie = shift;
.........
}
```

Note that pod translators should look at only paragraphs beginning with a pod directive (it makes parsing easier), whereas the compiler actually knows to look for pod escapes even in the middle of a paragraph. This means that the following secret stuff will be ignored by both the compiler and the translators.

```
$a=3;
=secret stuff
warn "Neither POD nor CODE!?"
=cut back
print "got $a\n";
```

You probably shouldn't rely upon the `warn()` being podded out forever. Not all pod translators are well-behaved in this regard, and perhaps the compiler will become pickier.

One may also use pod directives to quickly comment out a section of code.

Plain Old Comments (Not!)

Perl can process line directives, much like the C preprocessor. Using this, one can control Perl's idea of filenames and line numbers in error or warning messages (especially for strings that are processed with `eval()`). The syntax for this mechanism is almost the same as for most C preprocessors: it matches the regular expression

```
# example: '# line 42 "new_filename.plx"'
/^\# \s*
line \s+ (\d+) \s*
(?:\s("?)([^"]+)\g2)? \s*
$/x
```

with `$1` being the line number for the next line, and `$3` being the optional filename (specified with or without quotes). Note that no whitespace may precede the #, unlike modern C preprocessors.

There is a fairly obvious gotcha included with the line directive: Debuggers and profilers will only show the last source line to appear at a particular line number in a given file. Care should be taken not to cause line number collisions in code you'd like to debug later.

Here are some examples that you should be able to type into your command shell:

```
% perl
# line 200 "bzzzt"
# the '#' on the previous line must be the first char on line
die 'foo';
__END__
foo at bzzzt line 201.

% perl
# line 200 "bzzzt"
eval qq[\n#line 2001 ""\ndie 'foo']; print $@;
__END__
foo at - line 2001.

% perl
eval qq[\n#line 200 "foo bar"\ndie 'foo']; print $@;
__END__
foo at foo bar line 200.

% perl
```

```
# line 345 "goop"
eval "\n#line " . __LINE__ . ' "' . __FILE__ ."\"\ndie 'foo'";
print $@;
__END__
foo at goop line 345.
```

Experimental Details on given and when

As previously mentioned, the "switch" feature is considered highly experimental; it is subject to change with little notice. In particular, `when` has tricky behaviours that are expected to change to become less tricky in the future. Do not rely upon its current (mis)implementation. Before Perl 5.18, `given` also had tricky behaviours that you should still beware of if your code must run on older versions of Perl.

Here is a longer example of `given`:

```
use feature ":5.10";
given ($foo) {
when (undef) {
say '$foo is undefined';
}
when ("foo") {
say '$foo is the string "foo"';
}
when ([1,3,5,7,9]) {
say '$foo is an odd digit';
continue; # Fall through
}
when ($_ < 100) {
say '$foo is numerically less than 100';
}
when (\&complicated_check) {
say 'a complicated check for $foo is true';
}
default {
die q(I don't know what to do with $foo);
}
}
```

Before Perl 5.18, `given(EXPR)` assigned the value of *EXPR* to merely a lexically scoped ***copy*** (!) of `$_`, not a dynamically scoped alias the way `foreach` does. That made it similar to

```
do { my $_ = EXPR; ... }
```

except that the block was automatically broken out of by a successful `when` or an explicit `break`. Because it was only a copy, and because it was only lexically scoped, not dynamically scoped, you could not do the things with it that you are used to in a `foreach` loop. In particular, it did not work for arbitrary function calls if those functions might try to access `$_`. Best stick to `foreach` for that.

Most of the power comes from the implicit smartmatching that can sometimes apply. Most of the time, `when(EXPR)` is treated as an implicit smartmatch of `$_`, that is, `$_ ~~ EXPR`. (See "Smartmatch Operator" in perlop(1) for more information on smartmatching.) But when *EXPR* is one of the 10 exceptional cases (or things like them) listed below, it is used directly as a boolean.

1. A user-defined subroutine call or a method invocation.
2. A regular expression match in the form of `/REGEX/`, `$foo =~ /REGEX/`, or `$foo =~ EXPR`. Also, a negated regular expression match in the form `!/REGEX/`, `$foo !~ /REGEX/`, or `$foo !~ EXPR`.
3. A smart match that uses an explicit `~~` operator, such as `EXPR ~~ EXPR`.

   NOTE: You will often have to use `$c ~~ $_` because the default case uses `$_ ~~ $c` , which is

frequentlythe opposite of what you want.

4. A boolean comparison operator such as `$_ < 10` or `$x eq "abc"`. The relational operators that this applies to are the six numeric comparisons (`<`, `>`, `<=`, `>=`, `==`, and `!=`), and the six string comparisons (`lt`, `gt`, `le`, `ge`, `eq`, and `ne`).
5. At least the three builtin functions `defined(...)`, `exists(...)`, and `eof(...)`. We might someday add more of these later if we think of them.
6. A negated expression, whether `!(EXPR)` or `not(EXPR)`, or a logical exclusive-or, `(EXPR1) xor (EXPR2)`. The bitwise versions (`~` and `^`) are not included.
7. A filetest operator, with exactly 4 exceptions: `-s`, `-M`, `-A`, and `-C`, as these return numerical values, not boolean ones. The `-z` filetest operator is not included in the exception list.
8. The `..` and `...` flip-flop operators. Note that the `...` flip-flop operator is completely different from the `...` elliptical statement just described.

In those 8 cases above, the value of EXPR is used directly as a boolean, so no smartmatching is done. You may think of `when` as a smartsmartmatch.

Furthermore, Perl inspects the operands of logical operators to decide whether to use smartmatching for each one by applying the above test to the operands:

9. If EXPR is `EXPR1 && EXPR2` or `EXPR1 and EXPR2`, the test is applied *recursively* to both EXPR1 and EXPR2. Only if*both* operands also pass the test, *r ecursively*, will the expression be treated as boolean. Otherwise, smartmatching is used.
10. If EXPR is `EXPR1 || EXPR2`, `EXPR1 // EXPR2`, or `EXPR1 or EXPR2`, the test is applied *recursively* to EXPR1 only (which might itself be a higher-precedence AND operator, for example, and thus subject to the previous rule), not to EXPR2. If EXPR1 is to use smartmatching, then EXPR2 also does so, no matter what EXPR2 contains. But if EXPR2 does not get to use smartmatching, then the second argument will not be either. This is quite different from the `&&` case just described, so be careful.

These rules are complicated, but the goal is for them to do what you want (even if you don't quite understand why they are doing it). For example:

```
when (/^\d+$/ && $_ < 75) { ... }
```

will be treated as a boolean match because the rules say both a regex match and an explicit test on `$_` will be treated as boolean.

Also:

```
when ([qw(foo bar)] && /baz/) { ... }
```

will use smartmatching because only *one* of the operands is a boolean: the other uses smartmatching, and that wins.

Further:

```
when ([qw(foo bar)] || /^baz/) { ... }
```

will use smart matching (only the first operand is considered), whereas

```
when (/^baz/ || [qw(foo bar)]) { ... }
```

will test only the regex, which causes both operands to be treated as boolean. Watch out for this one, then, because an arrayref is always a true value, which makes it effectively redundant. Not a good idea.

Tautologous boolean operators are still going to be optimized away. Don't be tempted to write

```
when ("foo" or "bar") { ... }
```

This will optimize down to `"foo"`, so `"bar"` will never be considered (even though the rules say to use a smartmatch on `"foo"`). For an alternation like this, an array ref will work, because this will instigate smartmatching:

```
when ([qw(foo bar)] { ... }
```

This is somewhat equivalent to the C-style switch statement's fallthrough functionality (not to be confused with *Perl's* fallthrough functionality — see below), wherein the same block is used for several `case` statements.

Another useful shortcut is that, if you use a literal array or hash as the argument to `given`, it is turned into a reference. So `given(@foo)` is the same as `given(\@foo)`, for example.

`default` behaves exactly like `when(1 == 1)`, which is to say that it always matches.

Breaking out

You can use the `break` keyword to break out of the enclosing `given` block. Every `when` block is implicitly ended with a `break`.

Fall-through

You can use the `continue` keyword to fall through from one case to the next:

```
given($foo) {
when (/x/) { say '$foo contains an x'; continue }
when (/y/) { say '$foo contains a y' }
default { say '$foo does not contain a y' }
}
```

Return value

When a `given` statement is also a valid expression (for example, when it's the last statement of a block), it evaluates to:

- An empty list as soon as an explicit `break` is encountered.
- The value of the last evaluated expression of the successful `when`/`default` clause, if there happens to be one.
- The value of the last evaluated expression of the `given` block if no condition is true.

In both last cases, the last expression is evaluated in the context that was applied to the `given` block.

Note that, unlike `if` and `unless`, failed `when` statements always evaluate to an empty list.

```
my $price = do {
given ($item) {
when (["pear", "apple"]) { 1 }
break when "vote"; # My vote cannot be bought
1e10 when /Mona Lisa/;
"unknown";
}
};
```

Currently, `given` blocks can't always be used as proper expressions. This may be addressed in a future version of Perl.

Switching in a loop

Instead of using `given()`, you can use a `foreach()` loop. For example, here's one way to count how many times a particular string occurs in an array:

```
use v5.10.1;
my $count = 0;
for (@array) {
when ("foo") { ++$count }
}
print "\@array contains $count copies of 'foo'\n";
```

Or in a more recent version:

```
use v5.14;
my $count = 0;
for (@array) {
++$count when "foo";
}
print "\@array contains $count copies of 'foo'\n";
```

At the end of all `when` blocks, there is an implicit `next`. You can override that with an explicit `last` if you're interested in only the first match alone.

This doesn't work if you explicitly specify a loop variable, as in `for $item (@array)`. You have to use the default variable `$_`.

Differences from Perl 6

The Perl 5 smartmatch and `given`/`when` constructs are not compatible with their Perl 6 analogues. The most visible difference and least important difference is that, in Perl 5, parentheses are required around the argument to `given()` and `when()` (except when this last one is used as a statement modifier). Parentheses in Perl 6 are always optional in a control construct such as `if()`, `while()`, or `when()`; they can't be made optional in Perl 5 without a great deal of potential confusion, because Perl 5 would parse the expression

```
given $foo {
...
}
```

as though the argument to `given` were an element of the hash `%foo`, interpreting the braces as hash-element syntax.

However, their are many, many other differences. For example, this works in Perl 5:

```
use v5.12;
my @primary = ("red", "blue", "green");

if (@primary ~~ "red") {
say "primary smartmatches red";
}

if ("red" ~~ @primary) {
say "red smartmatches primary";
}

say "that's all, folks!";
```

But it doesn't work at all in Perl 6. Instead, you should use the (parallelizable) `any` operator:

```
if any(@primary) eq "red" {
say "primary smartmatches red";
}

if "red" eq any(@primary) {
say "red smartmatches primary";
}
```

The table of smartmatches in "Smartmatch Operator" in perlop(1) is not identical to that proposed by the Perl 6 specification, mainly due to differences between Perl 6's and Perl 5's data models, but also because the Perl 6 spec has changed since Perl 5 rushed into early adoption.

In Perl 6, `when()` will always do an implicit smartmatch with its argument, while in Perl 5 it is convenient (albeit potentially confusing) to suppress this implicit smartmatch in various rather loosely-defined situations, as roughly outlined above. (The difference is largely because Perl 5 does not have, even internally, a boolean type.)