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 mostly
serves to separate tokens, unlike languages like Python where it is an important part of the syntax.

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 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:

```perl
my $a;
if ($a) {}
```

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

```perl
my $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. Typically all the declarations are
put at the beginning or the end of the script. However, if you're using lexically-scoped private variables
created with `my()`, 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:

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

Note that myname() functions as a list operator, not as a unary operator; so be careful to use `or`
instead of `||` in this case. However, if you were to declare the subroutine as `sub myname ($)`,
then `myname` would function as a unary operator, so either `or` or `||` would work.

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** 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. (A semicolon is still encouraged if the block takes up more than one line, because you may eventually add another line.) Note that there are some operators like eval () and do () that look like compound statements, but aren't (they're just TERMS in an expression), and thus need an explicit termination if 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.

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:

```perl
if EXPR
unless EXPR
while EXPR
until EXPR
foreach LIST
```

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

```perl
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 (i.e., if the condition is false).

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

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

```perl
print "Hello $_:!\n" foreach 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):

```perl
# 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 deprecated do-SUBROUTINE statement), in which case the block executes once before the conditional is evaluated. This is so that you can write loops like:

```perl
do {
```
$line = <STDIN>;
...
} until $line eq ".\n";

See "do" in perfunc. 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) or around it (for last) to do that sort of thing. For next, just double the braces:

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

For last, you have to be more elaborate:

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

NOTE: The behaviour of a my statement modified with a statement modifier conditional or loop construct (e.g. 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.

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 ... else BLOCK
    LABEL while (EXPR) BLOCK
    LABEL while (EXPR) BLOCK continue BLOCK
    LABEL for (EXPR; EXPR; EXPR) BLOCK
    LABEL foreach VAR (LIST) BLOCK
    LABEL foreach VAR (LIST) BLOCK continue BLOCK
    LABEL BLOCK continue BLOCK

Note that, unlike C and Pascal, these are 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) or die "Can’t open $FOO: $!"; # FOO or bust!
    open(FOO)? ‘hi mom’: 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.

The while statement executes the block as long as the expression is true (does not evaluate to the null string " " or 0 or "0"). 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.

Loop Control

The next command starts the next iteration of the loop:

```perl
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:

```perl
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.

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

which is Perl short-hand for the more explicitly written version:

```perl
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 `?pat?` one-time matches:

```perl
# inspired by :1,$g/fred/s//WILMA/
while (<>)
  ?(fred)? && s//WILMA $1 WILMA/;
?{barney)? && s//BETTY $1 BETTY/;
?{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.

The 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.

```perl
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 and Switch Statements*.

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:

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

is the same as this:

```perl
$i = 1;
while ($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).

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.

```perl
$on_a_tty = -t STDIN && -t STDOUT;
```
Using (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.

```perl
for (prompt(); <STDIN>; prompt()) {
  # do something
}
```

**Foreach Loops**

The foreach loop iterates over a normal list value and sets the 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 localisation occurs only in a foreach loop.

The foreach keyword is actually a synonym for the for keyword, so you can use foreach for readability or for brevity. (Or because the Bourne shell is more familiar to you than csh, so writing for comes more naturally.) 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.

Examples:

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

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

for $count (10,9,8,7,6,5,4,3,2,1,'BOOM') {
  print $count, "\n"; sleep(1);
}

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

foreach $item (split(/:|
/; $ENV{TERMCAP})) {
  print "$item\n";
}
```

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

```perl
for (my $i = 0; $i < @ary1; $i++) {
  for (my $j = 0; $j < @ary2; $j++) {
    if ($ary1[$i] > $ary2[$j]) {
```
Whereas here's how a Perl programmer more comfortable with the idiom might do it:

```perl
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.

### Basic BLOCKs and Switch Statements

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 is particularly nice for doing case structures.

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

There is no official `switch` statement in Perl, because there are already several ways to write the equivalent.

However, starting from Perl 5.8 to get switch and case one can use the Switch extension and say:

```perl
use Switch;
```

after which one has switch and case. It is not as fast as it could be because it's not really part of the language (it's done using source filters) but it is available, and it's very flexible.

In addition to the above BLOCK construct, you could write

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

(That's actually not as strange as it looks once you realize that you can use loop control "operators" within an expression. That's just the binary comma operator in scalar context. See "Comma Operator"
or

```perl
SWITCH: {
    /^abc/ && do {
        $abc = 1;
        last SWITCH;
    };
    /^def/ && do {
        $def = 1;
        last SWITCH;
    };
    /^xyz/ && do {
        $xyz = 1;
        last SWITCH;
    };
    $nothing = 1;
}
```

or formatted so it stands out more as a "proper" switch statement:

```perl
SWITCH: {
    /^abc/   && do {
        $abc = 1;
        last SWITCH;
    };
    /^def/   && do {
        $def = 1;
        last SWITCH;
    };
    /^xyz/   && do {
        $xyz = 1;
        last SWITCH;
    };
    $nothing = 1;
}
```

or

```perl
SWITCH: {
    /^abc/ and $abc = 1, last SWITCH;
    /^def/ and $def = 1, last SWITCH;
    /^xyz/ and $xyz = 1, last SWITCH;
    $nothing = 1;
}
```

or even, horrors,

```perl
if (/^abc/)
{ $abc = 1 }
elsif (/^def/)
{ $def = 1 }
elsif (/^xyz/)
{ $xyz = 1 }
else
{ $nothing = 1 }
```

A common idiom for a switch statement is to use foreach's aliasing to make a temporary assignment to $_. for convenient matching:

```perl
SWITCH: for ($where) {
    /In Card Names/   && do { push @flags, '-e'; last; }
    /Anywhere/        && do { push @flags, '-h'; last; }
    ...
```
Another interesting approach to a switch statement is arrange for a do block to return the proper value:

```perl
$amode = do {
  if  ($flag & O_RDONLY) { "r" } # XXX: isn't this 0?
  elsif ($flag & O_WRONLY) { ($flag & O_APPEND) ? "a" : "w" }
  elsif ($flag & O_RDWR) {
    if ($flag & O_CREAT) { "w+" }
    else { ($flag & O_APPEND) ? "a+" : "r+" }
  }
  else { ($flag & O_APPEND) ? "a+" : "r+" }
};
```

Or

```perl
print do {
  ($flags & O_WRONLY) ? "write-only" :
  ($flags & O_RDWR) ? "read-write" :
    "read-only";
};
```

Or if you are certain that all the && clauses are true, you can use something like this, which "switches" on the value of the HTTP_USER_AGENT environment variable.

```perl
#!/usr/bin/perl
# pick out jargon file page based on browser
$dir = 'http://www.wins.uva.nl/~mes/jargon';
for ($ENV{HTTP_USER_AGENT}) {
  $page = /Mac/ && 'm/Macintrash.html'
    || /Win(dows )?NT/ && 'e/evilandrude.html'
    || /Win|MSIE|WebTV/ && 'm/MicroslothWindows.html'
    || /Linux/ && 'l/Linux.html'
    || /HP-UX/ && 'h/HP-SUX.html'
    || /SunOS/ && 's/ScumOS.html'
    || 'a/AppendixB.html';
  print "Location: $dir/$page\015\012\015\012";
}
```

That kind of switch statement only works when you know the && clauses will be true. If you don't, the previous ?: example should be used.

You might also consider writing a hash of subroutine references instead of synthesizing a switch statement.

**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:

```perl
    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.

**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

```perl
=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

```perl
=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.

```perl
@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 the same as for most C preprocessors: it matches the regular expression

```
/^\s* \s+ \d+ \s*\s*.(?:\d+)?\s*$
```

with `$1` being the line number for the next line, and `$3` being the optional filename (specified with or without quotes).

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
% perl
# line 345 "goop"
eval "\n\n#line 345 "\n\nline 345 "goop"
die 'foo'
print $@;
__END__
foo at goop line 345.
```

```perl
% perl
# line 200 "bzzzt"
eval qq[
#line 2001 "

die 'foo';
__END__
foo at bzzzt line 201.
% perl
# line 200 "bzzzt"
eval qq[
#line 200 "foo bar"

die 'foo';
print $@;
__END__
foo at foo bar line 200.
```

```perl
% perl
# line 200 "bzzzt"
eval qq[
#line 200 "bzzzt"

die 'foo';
print $@;
__END__
foo at bzzzt line 200.
% perl
# line 200 "bzzzt"
eval qq[
#line 2001 ""

die 'foo';
print $@;
__END__
foo at - line 2001.
% perl
# line 200 "bzzzt"
eval qq[
#line 200 "foo bar"

die 'foo';
print $@;
__END__
foo at foo bar line 200.
% perl
# line 200 "bzzzt"
eval qq[
#line 2001 ""

die 'foo';
print $@;
__END__
foo at - line 2001.
% perl
# line 200 "bzzzt"
eval qq[
#line 200 "foo bar"

die 'foo';
print $@;
__END__
foo at foo bar line 200.
% perl
# line 200 "bzzzt"
eval qq[
#line 2001 ""

die 'foo';
print $@;
__END__
foo at - line 2001.
```