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:

```
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:

```
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:

```
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. 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.

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
when EXPR
for LIST
foreach LIST
```

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

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

`when` executes the statement *when* `$_` smart matches `EXPR`, and then either `break`s out if it's enclosed in a `given` scope or skips to the `next` element when it lies directly inside a `for` loop. See also *Switch statements*.

```
given ($something) {
    $abc    = 1 when /^abc/;
    $just_a = 1 when /^a/;
    $other  = 1;
}

    for (@names) {
admin($_)   when [ qw/Alice Bob/ ];
regular($_) when [ qw/Chris David Ellen/ ];
```

```
    }
```

The `foreach` 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" 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):

```
    # 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:

```
    do {
 $line = <STDIN>;
 ...
    } until $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`) 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
unless (EXPR) BLOCK
unless (EXPR) BLOCK else BLOCK
unless (EXPR) BLOCK elsif (EXPR) BLOCK ... else 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 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. 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.

Extension modules can also hook into the Perl parser to define new kinds of compound statement. 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* 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 short-hand 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.

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.

```
    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).

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 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 `foreach` for

readability or `for` 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:

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

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.

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;
    }
```

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

Switch statements

Starting from Perl 5.10, you can say

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

which enables a switch feature that is closely based on the Perl 6 proposal.

The keywords `given` and `when` are analogous to `switch` and `case` in other languages, so the code above could be written as

```
    given($_) {
 when (/^abc/) { $abc = 1; }
 when (/^def/) { $def = 1; }
 when (/^xyz/) { $xyz = 1; }
 default { $nothing = 1; }
    }
```

This construct is very flexible and powerful. For example:

```
    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);
}
    }
```

`given(EXPR)` will assign the value of EXPR to `$_` within the lexical scope of the block, so it's similar to

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

except that the block is automatically broken out of by a successful `when` or an explicit `break`.

Most of the power comes from implicit smart matching:

```
when($foo)
```

is exactly equivalent to

```
when($_ ~~ $foo)
```

Most of the time, `when(EXPR)` is treated as an implicit smart match of `$_`, i.e. `$_ ~~ EXPR`. (See *Smart matching in detail* for more information on smart matching.) But when EXPR is one of the below exceptional cases, it is used directly as a boolean:

- a subroutine or method call

- a regular expression match, i.e. `/REGEX/` or `$foo =~ /REGEX/`, or a negated regular expression match (`!/REGEX/` or `$foo !~ /REGEX/`).

- a comparison such as `$_ < 10` or `$x eq "abc"` (or of course `$_ ~~ $c`)

- `defined(...)`, `exists(...)`, or `eof(...)`

- a negated expression `!(...)` or `not (...)`, or a logical exclusive-or `(...) xor (...)`.

- a filetest operator, with the exception of `-s`, `-M`, `-A`, and `-C`, that return numerical values, not boolean ones.

- the `..` and `...` flip-flop operators.

In those cases the value of EXPR is used directly as a boolean.

Furthermore, Perl inspects the operands of the binary boolean operators to decide whether to use smart matching for each one by applying the above test to the operands:

- If EXPR is `... && ...` or `... and ...`, the test is applied recursively to both operands. If *both* operands pass the test, then the expression is treated as boolean; otherwise, smart matching is used.

- If EXPR is `... || ...`, `... // ...` or `... or ...`, the test is applied recursively to the first operand (which may be a higher-precedence AND operator, for example). If the first operand is to use smart matching, then both operands will do so; if it is not, then the second argument will not be either.

These rules look complicated, but usually they will do what you want. 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 smart matching because only *one* of the operands is a boolean; the other uses smart matching, 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.

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 smart match on `'foo'`). For an alternation like this, an array ref will work, because this will instigate smart matching:

```
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 (e.g. 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's 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:

```
    my $count = 0;
    for (@array) {
when ("foo") { ++$count }
    }
    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 only interested in the first match.

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

Smart matching in detail

The behaviour of a smart match depends on what type of thing its arguments are. The behaviour is determined by the following table: the first row that applies determines the match behaviour (which is thus mostly determined by the type of the right operand). Note that the smart match implicitly dereferences any non-blessed hash or array ref, so the "Hash" and "Array" entries apply in those cases. (For blessed references, the "Object" entries apply.)

Note that the "Matching Code" column is not always an exact rendition. For example, the smart match operator short-circuits whenever possible, but `grep` does not.

```
    $a      $b        Type of Match Implied    Matching Code
    ======  =====     ====================     =============
    Any     undef     undefined                !defined $a


    Any     Object    invokes ~~ overloading on $object, or dies


    Hash    CodeRef   sub truth for each key[1] !grep { !$b->($_) } keys
%$a
    Array   CodeRef   sub truth for each elt[1] !grep { !$b->($_) } @$a
    Any     CodeRef   scalar sub truth          $b->($a)


    Hash    Hash      hash keys identical (every key is found in both
hashes)
    Array   Hash      hash keys intersection    grep { exists $b->{$_} } @$a
    Regex   Hash      hash key grep             grep /$a/, keys %$b
```

```
undef   Hash      always false (undef can't be a key)
Any     Hash      hash entry existence    exists $b->{$a}


Hash    Array     hash keys intersection  grep { exists $a->{$_} } @$b
Array   Array     arrays are comparable[2]
Regex   Array     array grep              grep /$a/, @$b
undef   Array     array contains undef    grep !defined, @$b
Any     Array     match against an array element[3]
                                          grep $a ~~ $_, @$b


Hash    Regex     hash key grep           grep /$b/, keys %$a
Array   Regex     array grep              grep /$b/, @$a
Any     Regex     pattern match           $a =~ /$b/


Object  Any       invokes ~~ overloading on $object, or falls back:
Any     Num       numeric equality        $a == $b
Num     numish[4] numeric equality        $a == $b
undef   Any       undefined               !defined($b)
Any     Any       string equality         $a eq $b
```

```
1 - empty hashes or arrays will match.
2 - that is, each element smart-matches the element of same index in the
    other array. [3]
3 - If a circular reference is found, we fall back to referential
equality.
4 - either a real number, or a string that looks like a number
```

Custom matching via overloading

You can change the way that an object is matched by overloading the ~~ operator. This may alter the usual smart match semantics.

It should be noted that ~~ will refuse to work on objects that don't overload it (in order to avoid relying on the object's underlying structure).

Note also that smart match's matching rules take precedence over overloading, so if $obj has smart match overloading, then

```
$obj ~~ X
```

will not automatically invoke the overload method with X as an argument; instead the table above is consulted as normal, and based in the type of X, overloading may or may not be invoked.

See *overload*.

Differences from Perl 6

The Perl 5 smart match and given/when constructs are not absolutely identical to their Perl 6 analogues. The most visible 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.

The table of smart matches is not identical to that proposed by the Perl 6 specification, mainly due to the differences between Perl 6's and Perl 5's data models.

In Perl 6, `when()` will always do an implicit smart match with its argument, whilst it is convenient in Perl 5 to suppress this implicit smart match in certain situations, as documented above. (The difference is largely because Perl 5 does not, even internally, have a boolean type.)

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.

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.
```