

NAME

perlop – Perl operators and precedence

SYNOPSIS

Perl operators have the following associativity and precedence, listed from highest precedence to lowest. Note that all operators borrowed from C keep the same precedence relationship with each other, even where C's precedence is slightly screwy. (This makes learning Perl easier for C folks.) With very few exceptions, these all operate on scalar values only, not array values.

```

left      terms and list operators (leftward)
left      ->
nonassoc  ++ --
right     **
right     ! ~ \ and unary + and -
left     = ~ !~
left     * / % x
left     + - .
left     << >>
nonassoc  named unary operators
nonassoc  < > <= >= lt gt le ge
nonassoc  == != <=> eq ne cmp
left     &
left     | ^
left     &&
left     ||
nonassoc  .. ...
right     ?:
right     = += -= *= etc.
left     , =>
nonassoc  list operators (rightward)
right     not
left     and
left     or xor

```

In the following sections, these operators are covered in precedence order.

Many operators can be overloaded for objects. See the *overload* manpage.

DESCRIPTION

Terms and List Operators (Leftward)

A TERM has the highest precedence in Perl. They includes variables, quote and quote-like operators, any expression in parentheses, and any function whose arguments are parenthesized. Actually, there aren't really functions in this sense, just list operators and unary operators behaving as functions because you put parentheses around the arguments. These are all documented in the *perlfunc* manpage.

If any list operator (*print()*, etc.) or any unary operator (*chdir()*, etc.) is followed by a left parenthesis as the next token, the operator and arguments within parentheses are taken to be of highest precedence, just like a normal function call.

In the absence of parentheses, the precedence of list operators such as `print`, `sort`, or `chmod` is either very high or very low depending on whether you are looking at the left side or the right side of the operator. For example, in

```
@ary = (1, 3, sort 4, 2);
print @ary;          # prints 1324
```

the commas on the right of the `sort` are evaluated before the `sort`, but the commas on the left are evaluated after. In other words, list operators tend to gobble up all the arguments that follow them, and then act like a simple TERM with regard to the preceding expression. Note that you have to be careful with parentheses:

```
# These evaluate exit before doing the print:
print($foo, exit); # Obviously not what you want.
print $foo, exit;  # Nor is this.

# These do the print before evaluating exit:
(print $foo), exit; # This is what you want.
print($foo), exit; # Or this.
print ($foo), exit; # Or even this.
```

Also note that

```
print ($foo & 255) + 1, "\n";
```

probably doesn't do what you expect at first glance. See the section on *Named Unary Operators* for more discussion of this.

Also parsed as terms are the `do { }` and `eval { }` constructs, as well as subroutine and method calls, and the anonymous constructors `[]` and `{ }`.

See also the section on *Quote and Quote-like Operators* toward the end of this section, as well as the section on *I/O Operators*.

The Arrow Operator

Just as in C and C++, “`->`” is an infix dereference operator. If the right side is either a `[. . .]` or `{ . . . }` subscript, then the left side must be either a hard or symbolic reference to an array or hash (or a location capable of holding a hard reference, if it's an lvalue (assignable)). See the *perlref* manpage.

Otherwise, the right side is a method name or a simple scalar variable containing the method name, and the left side must either be an object (a blessed reference) or a class name (that is, a package name). See the *perlobj* manpage.

Auto-increment and Auto-decrement

“`++`” and “`--`” work as in C. That is, if placed before a variable, they increment or decrement the variable before returning the value, and if placed after, increment or decrement the variable after returning the value.

The auto-increment operator has a little extra builtin magic to it. If you increment a variable that is numeric, or that has ever been used in a numeric context, you get a normal increment. If, however, the variable has been used in only string contexts since it was set, and has a value that is not the empty string and matches the pattern `/^[a-zA-Z]*[0-9]*$/`, the increment is done as a string, preserving each character within its range, with carry:

```

print ++($foo = '99');      # prints '100'
print ++($foo = 'a0');     # prints 'a1'
print ++($foo = 'Az');     # prints 'Ba'
print ++($foo = 'zz');     # prints 'aaa'

```

The auto-decrement operator is not magical.

Exponentiation

Binary “**” is the exponentiation operator. Note that it binds even more tightly than unary minus, so $-2**4$ is $-(2**4)$, not $(-2)**4$. (This is implemented using C’s *pow*(3) function, which actually works on doubles internally.)

Symbolic Unary Operators

Unary “!” performs logical negation, i.e., “not”. See also *not* for a lower precedence version of this.

Unary “-” performs arithmetic negation if the operand is numeric. If the operand is an identifier, a string consisting of a minus sign concatenated with the identifier is returned. Otherwise, if the string starts with a plus or minus, a string starting with the opposite sign is returned. One effect of these rules is that *-bareword* is equivalent to “-bareword”.

Unary “~” performs bitwise negation, i.e., 1’s complement. For example, $0666 \&\sim 027$ is 0640 . (See also the section on *Integer Arithmetic* and the section on *Bitwise String Operators*.)

Unary “+” has no effect whatsoever, even on strings. It is useful syntactically for separating a function name from a parenthesized expression that would otherwise be interpreted as the complete list of function arguments. (See examples above under the section on *Terms and List Operators (Leftward)*.)

Unary “\” creates a reference to whatever follows it. See the *perlref* manpage. Do not confuse this behavior with the behavior of backslash within a string, although both forms do convey the notion of protecting the next thing from interpretation.

Binding Operators

Binary “=~” binds a scalar expression to a pattern match. Certain operations search or modify the string $\$_$ by default. This operator makes that kind of operation work on some other string. The right argument is a search pattern, substitution, or transliteration. The left argument is what is supposed to be searched, substituted, or transliterated instead of the default $\$_$. The return value indicates the success of the operation. (If the right argument is an expression rather than a search pattern, substitution, or transliteration, it is interpreted as a search pattern at run time. This can be less efficient than an explicit search, because the pattern must be compiled every time the expression is evaluated.)

Binary “!~” is just like “=~” except the return value is negated in the logical sense.

Multiplicative Operators

Binary “*” multiplies two numbers.

Binary “/” divides two numbers.

Binary “%” computes the modulus of two numbers. Given integer operands \$a and \$b: If \$b is positive, then \$a % \$b is \$a minus the largest multiple of \$b that is not greater than \$a. If \$b is negative, then \$a % \$b is \$a minus the smallest multiple of \$b that is not less than \$a (i.e. the result will be less than or equal to zero). Note that when use integer is in scope, “%” give you direct access to the modulus operator as implemented by your C compiler. This operator is not as well defined for negative operands, but it will execute faster.

Binary “x” is the repetition operator. In scalar context, it returns a string consisting of the left operand repeated the number of times specified by the right operand. In list context, if the left operand is a list in parentheses, it repeats the list.

```
print '-' x 80;                # print row of dashes

print "\t" x ($tab/8), ' ' x ($tab%8);    # tab over

@ones = (1) x 80;             # a list of 80 1's
@ones = (5) x @ones;         # set all elements to 5
```

Additive Operators

Binary “+” returns the sum of two numbers.

Binary “-” returns the difference of two numbers.

Binary “.” concatenates two strings.

Shift Operators

Binary “<<” returns the value of its left argument shifted left by the number of bits specified by the right argument. Arguments should be integers. (See also the section on *Integer Arithmetic*.)

Binary “>>” returns the value of its left argument shifted right by the number of bits specified by the right argument. Arguments should be integers. (See also the section on *Integer Arithmetic*.)

Named Unary Operators

The various named unary operators are treated as functions with one argument, with optional parentheses. These include the filetest operators, like -f, -M, etc. See the *perlfunc* manpage.

If any list operator (*print()*, etc.) or any unary operator (*chdir()*, etc.) is followed by a left parenthesis as the next token, the operator and arguments within parentheses are taken to be of highest precedence, just like a normal function call. Examples:

```
chdir $foo    || die;           # (chdir $foo) || die
chdir($foo)  || die;           # (chdir $foo) || die
chdir ($foo) || die;           # (chdir $foo) || die
chdir +($foo) || die;          # (chdir $foo) || die
```

but, because * is higher precedence than ||:

```

chdir $foo * 20;      # chdir ($foo * 20)
chdir($foo) * 20;    # (chdir $foo) * 20
chdir ($foo) * 20;   # (chdir $foo) * 20
chdir +($foo) * 20;  # chdir ($foo * 20)

rand 10 * 20;        # rand (10 * 20)
rand(10) * 20;       # (rand 10) * 20
rand (10) * 20;      # (rand 10) * 20
rand +(10) * 20;     # rand (10 * 20)

```

See also the section on *Terms and List Operators (Leftward)*.

Relational Operators

Binary “<” returns true if the left argument is numerically less than the right argument.

Binary “>” returns true if the left argument is numerically greater than the right argument.

Binary “<=” returns true if the left argument is numerically less than or equal to the right argument.

Binary “>=” returns true if the left argument is numerically greater than or equal to the right argument.

Binary “lt” returns true if the left argument is stringwise less than the right argument.

Binary “gt” returns true if the left argument is stringwise greater than the right argument.

Binary “le” returns true if the left argument is stringwise less than or equal to the right argument.

Binary “ge” returns true if the left argument is stringwise greater than or equal to the right argument.

Equality Operators

Binary “==” returns true if the left argument is numerically equal to the right argument.

Binary “!=” returns true if the left argument is numerically not equal to the right argument.

Binary “<=>” returns -1, 0, or 1 depending on whether the left argument is numerically less than, equal to, or greater than the right argument.

Binary “eq” returns true if the left argument is stringwise equal to the right argument.

Binary “ne” returns true if the left argument is stringwise not equal to the right argument.

Binary “cmp” returns -1, 0, or 1 depending on whether the left argument is stringwise less than, equal to, or greater than the right argument.

“lt”, “le”, “ge”, “gt” and “cmp” use the collation (sort) order specified by the current locale if use `locale` is in effect. See the *perllocale* manpage.

Bitwise And

Binary “&” returns its operators ANDed together bit by bit. (See also the section on *Integer Arithmetic* and the section on *Bitwise String Operators*.)

Bitwise Or and Exclusive Or

Binary “|” returns its operators ORed together bit by bit. (See also the section on *Integer Arithmetic* and the section on *Bitwise String Operators*.)

Binary “^” returns its operators XORed together bit by bit. (See also the section on *Integer Arithmetic* and the section on *Bitwise String Operators*.)

C–style Logical And

Binary “&&” performs a short-circuit logical AND operation. That is, if the left operand is false, the right operand is not even evaluated. Scalar or list context propagates down to the right operand if it is evaluated.

C–style Logical Or

Binary “||” performs a short-circuit logical OR operation. That is, if the left operand is true, the right operand is not even evaluated. Scalar or list context propagates down to the right operand if it is evaluated.

The || and && operators differ from C’s in that, rather than returning 0 or 1, they return the last value evaluated. Thus, a reasonably portable way to find out the home directory (assuming it’s not “0”) might be:

```
$home = $ENV{'HOME'} || $ENV{'LOGDIR'} ||
        (getpwuid($<))[7] || die "You're homeless!\n";
```

In particular, this means that you shouldn’t use this for selecting between two aggregates for assignment:

```
@a = @b || @c;           # this is wrong
@a = scalar(@b) || @c;   # really meant this
@a = @b ? @b : @c;       # this works fine, though
```

As more readable alternatives to && and || when used for control flow, Perl provides and and or operators (see below). The short-circuit behavior is identical. The precedence of “and” and “or” is much lower, however, so that you can safely use them after a list operator without the need for parentheses:

```
unlink "alpha", "beta", "gamma"
or gripe(), next LINE;
```

With the C–style operators that would have been written like this:

```
unlink("alpha", "beta", "gamma")
|| (gripe(), next LINE);
```

Use “or” for assignment is unlikely to do what you want; see below.

Range Operators

Binary “..” is the range operator, which is really two different operators depending on the context. In list context, it returns an array of values counting (by ones) from the left value to the right value. This is useful for writing `foreach (1..10)` loops and for doing slice operations on arrays. In the current implementation, no temporary array is created when the range operator is used as the expression in `foreach` loops, but older versions of Perl might burn a lot of memory when you write something like this:

```
for (1 .. 1_000_000) {
    # code
}
```

In scalar context, “..” returns a boolean value. The operator is bistable, like a flip-flop, and emulates the line-range (comma) operator of **sed**, **awk**, and various editors. Each “..” operator maintains its own boolean state. It is false as long as its left operand is false. Once the left operand is true, the range operator stays true until the right operand is true, *AFTER* which the range operator becomes false again. (It doesn’t become false till the next time the range operator is evaluated. It can test the right operand and become false on the same evaluation it became true (as in **awk**), but it still returns true once. If you don’t want it to test the right operand till the next evaluation (as in **sed**), use three dots (“...”) instead of two.) The right operand is not evaluated while the operator is in the “false” state, and the left operand is not evaluated while the operator is in the “true” state. The precedence is a little lower than `||` and `&&`. The value returned is either the empty string for false, or a sequence number (beginning with 1) for true. The sequence number is reset for each range encountered. The final sequence number in a range has the string “E0” appended to it, which doesn’t affect its numeric value, but gives you something to search for if you want to exclude the endpoint. You can exclude the beginning point by waiting for the sequence number to be greater than 1. If either operand of scalar “..” is a constant expression, that operand is implicitly compared to the `$.` variable, the current line number. Examples:

As a scalar operator:

```
if (101 .. 200) { print; } # print 2nd hundred lines
next line if (1 .. /^$/); # skip header lines
s/^/> / if (/^$/ .. eof()); # quote body

# parse mail messages
while (<>) {
    $in_header = 1 .. /^$/;
    $in_body   = /^$/ .. eof();
    # do something based on those
} continue {
    close ARGV if eof;          # reset $. each file
}
```

As a list operator:

```
for (101 .. 200) { print; } # print $_ 100 times
@foo = @foo[0 .. $#foo];    # an expensive no-op
@foo = @foo[$#foo-4 .. $#foo]; # slice last 5 items
```

The range operator (in list context) makes use of the magical auto-increment algorithm if the operands are strings. You can say

```
@alphabet = ('A' .. 'Z');
```

to get all the letters of the alphabet, or

```
$hexdigit = (0 .. 9, 'a' .. 'f')[$num & 15];
```

to get a hexadecimal digit, or

```
@z2 = ('01' .. '31'); print $z2[$mday];
```

to get dates with leading zeros. If the final value specified is not in the sequence that the magical increment would produce, the sequence goes until the next value would be longer than the final value specified.

Conditional Operator

Ternary “?:” is the conditional operator, just as in C. It works much like an if-then-else. If the argument before the ? is true, the argument before the : is returned, otherwise the argument after the : is returned. For example:

```
printf "I have %d dog%s.\n", $n,  
      ($n == 1) ? '' : "s";
```

Scalar or list context propagates downward into the 2nd or 3rd argument, whichever is selected.

```
$a = $ok ? $b : $c; # get a scalar  
@a = $ok ? @b : @c; # get an array  
$a = $ok ? @b : @c; # oops, that's just a count!
```

The operator may be assigned to if both the 2nd and 3rd arguments are legal lvalues (meaning that you can assign to them):

```
($a_or_b ? $a : $b) = $c;
```

This is not necessarily guaranteed to contribute to the readability of your program.

Because this operator produces an assignable result, using assignments without parentheses will get you in trouble. For example, this:

```
$a % 2 ? $a += 10 : $a += 2
```

Really means this:

```
(( $a % 2 ) ? ( $a += 10 ) : $a) += 2
```

Rather than this:

```
($a % 2) ? ($a += 10) : ($a += 2)
```

Assignment Operators

“=” is the ordinary assignment operator.

Assignment operators work as in C. That is,

```
$a += 2;
```

is equivalent to

```
$a = $a + 2;
```

although without duplicating any side effects that dereferencing the lvalue might trigger, such as from *tie()*. Other assignment operators work similarly. The following are recognized:

**=	+=	*=	&=	<<=	&&=
	-=	/=	=	>>=	=
	.=	%=	^=		
		x=			

Note that while these are grouped by family, they all have the precedence of assignment.

Unlike in C, the assignment operator produces a valid lvalue. Modifying an assignment is equivalent to doing the assignment and then modifying the variable that was assigned to. This is useful for modifying a copy of something, like this:

```
($tmp = $global) =~ tr [A-Z] [a-z];
```

Likewise,

```
($a += 2) *= 3;
```

is equivalent to

```
$a += 2;
$a *= 3;
```

Comma Operator

Binary “,” is the comma operator. In scalar context it evaluates its left argument, throws that value away, then evaluates its right argument and returns that value. This is just like C’s comma operator.

In list context, it’s just the list argument separator, and inserts both its arguments into the list.

The => digraph is mostly just a synonym for the comma operator. It’s useful for documenting arguments that come in pairs. As of release 5.001, it also forces any word to the left of it to be interpreted as a string.

List Operators (Rightward)

On the right side of a list operator, it has very low precedence, such that it controls all comma-separated expressions found there. The only operators with lower precedence are the logical operators “and”, “or”, and “not”, which may be used to evaluate calls to list operators without the need for extra parentheses:

```
open HANDLE, "filename"
    or die "Can't open: $!\n";
```

See also discussion of list operators in the section on *Terms and List Operators (Leftward)*.

Logical Not

Unary “not” returns the logical negation of the expression to its right. It’s the equivalent of “!” except for the very low precedence.

Logical And

Binary “and” returns the logical conjunction of the two surrounding expressions. It’s equivalent to && except for the very low precedence. This means that it short-circuits: i.e., the right expression is evaluated only if the left expression is true.

Logical or and Exclusive Or

Binary “or” returns the logical disjunction of the two surrounding expressions. It’s equivalent to || except for the very low precedence. This makes it useful for control flow

```
print FH $data                or die "Can't write to FH: $!";
```

This means that it short-circuits: i.e., the right expression is evaluated only if the left expression is false. Due to its precedence, you should probably avoid using this for assignment, only for control flow.

```
$a = $b or $c;                # bug: this is wrong
($a = $b) or $c;             # really means this
$a = $b || $c;               # better written this way
```

However, when it’s a list context assignment and you’re trying to use “||” for control flow, you probably need “or” so that the assignment takes higher precedence.

```
@info = stat($file) || die;   # oops, scalar sense of stat!
@info = stat($file) or die;   # better, now @info gets its due
```

Then again, you could always use parentheses.

Binary “xor” returns the exclusive-OR of the two surrounding expressions. It cannot short circuit, of course.

C Operators Missing From Perl

Here is what C has that Perl doesn’t:

- unary &** Address-of operator. (But see the “\” operator for taking a reference.)
- unary *** Dereference-address operator. (Perl’s prefix dereferencing operators are typed: \$, @, %, and &.)
- (TYPE)** Type casting operator.

Quote and Quote-like Operators

While we usually think of quotes as literal values, in Perl they function as operators, providing various kinds of interpolating and pattern matching capabilities. Perl provides customary quote characters for these behaviors, but also provides a way for you to choose your quote character for any of them. In the following table, a { } represents any pair of delimiters you choose. Non-bracketing delimiters use the same character fore and aft, but the 4 sorts of brackets (round, angle, square, curly) will all nest.

Customary	Generic	Meaning	Interpolates
' '	q{ }	Literal	no
" "	qq{ }	Literal	yes
` `	qx{ }	Command	yes (unless ' ' is deli
	qw{ }	Word list	no
//	m{ }	Pattern match	yes
	qr{ }	Pattern	yes
	s{ }{ }	Substitution	yes
	tr{ }{ }	Transliteration	no (but see below)

Note that there can be whitespace between the operator and the quoting characters, except when # is being used as the quoting character. q#foo# is parsed as being the string foo, while q #foo# is the operator q followed by a comment. Its argument will be taken from the next line. This allows you to write:

```
s {foo} # Replace foo
   {bar} # with bar.
```

For constructs that do interpolation, variables beginning with “\$” or “@” are interpolated, as are the following sequences. Within a transliteration, the first ten of these sequences may be used.

\t	tab	(HT, TAB)
\n	newline	(NL)
\r	return	(CR)
\f	form feed	(FF)
\b	backspace	(BS)
\a	alarm (bell)	(BEL)
\e	escape	(ESC)
\033	octal char	
\x1b	hex char	
\c[control char	

<code>\l</code>	lowercase next char
<code>\u</code>	uppercase next char
<code>\L</code>	lowercase till <code>\E</code>
<code>\U</code>	uppercase till <code>\E</code>
<code>\E</code>	end case modification
<code>\Q</code>	quote non-word characters till <code>\E</code>

If use `locale` is in effect, the case map used by `\l`, `\L`, `\u` and `\U` is taken from the current locale. See the *perllocale* manpage.

All systems use the virtual "`\n`" to represent a line terminator, called a "newline". There is no such thing as an unvarying, physical newline character. It is an illusion that the operating system, device drivers, C libraries, and Perl all conspire to preserve. Not all systems read "`\r`" as ASCII CR and "`\n`" as ASCII LF. For example, on a Mac, these are reversed, and on systems without line terminator, printing "`\n`" may emit no actual data. In general, use "`\n`" when you mean a "newline" for your system, but use the literal ASCII when you need an exact character. For example, most networking protocols expect and prefer a CR+LF ("`\012\015`" or "`\cJ\cM`") for line terminators, and although they often accept just "`\012`", they seldom tolerate just "`\015`". If you get in the habit of using "`\n`" for networking, you may be burned some day.

You cannot include a literal `$` or `@` within a `\Q` sequence. An unescaped `$` or `@` interpolates the corresponding variable, while escaping will cause the literal string `\$` to be inserted. You'll need to write something like `m/\Quser\E@\Qhost/`.

Patterns are subject to an additional level of interpretation as a regular expression. This is done as a second pass, after variables are interpolated, so that regular expressions may be incorporated into the pattern from the variables. If this is not what you want, use `\Q` to interpolate a variable literally.

Apart from the above, there are no multiple levels of interpolation. In particular, contrary to the expectations of shell programmers, back-quotes do *NOT* interpolate within double quotes, nor do single quotes impede evaluation of variables when used within double quotes.

Regex Quote-Like Operators

Here are the quote-like operators that apply to pattern matching and related activities.

Most of this section is related to use of regular expressions from Perl. Such a use may be considered from two points of view: Perl handles a a string and a "pattern" to RE (regular expression) engine to match, RE engine finds (or does not find) the match, and Perl uses the findings of RE engine for its operation, possibly asking the engine for other matches.

RE engine has no idea what Perl is going to do with what it finds, similarly, the rest of Perl has no idea what a particular regular expression means to RE engine. This creates a clean separation, and in this section we discuss matching from Perl point of view only. The other point of view may be found in the *perlre* manpage.

?PATTERN?

This is just like the `/pattern/` search, except that it matches only once between calls to the `reset()` operator. This is a useful optimization when you want to see only the first occurrence of something in each file of a set of files, for instance. Only `??` patterns local to the current package are reset.

```

while (<>) {
    if (?^$?) {
        # blank line between header and body
    }
} continue {
    reset if eof; # clear ?? status for next file
}

```

This usage is vaguely deprecated, and may be removed in some future version of Perl.

m/PATTERN/cgimosx

/PATTERN/cgimosx

Searches a string for a pattern match, and in scalar context returns true (1) or false ("). If no string is specified via the =~ or !~ operator, the \$_ string is searched. (The string specified with =~ need not be an lvalue—it may be the result of an expression evaluation, but remember the =~ binds rather tightly.) See also the *perlre* manpage. See the *perllocale* manpage for discussion of additional considerations that apply when use locale is in effect.

Options are:

```

c  Do not reset search position on a failed match when /g is
g  Match globally, i.e., find all occurrences.
i  Do case-insensitive pattern matching.
m  Treat string as multiple lines.
o  Compile pattern only once.
s  Treat string as single line.
x  Use extended regular expressions.

```

If “/” is the delimiter then the initial m is optional. With the m you can use any pair of non-alphanumeric, non-whitespace characters as delimiters (if single quotes are used, no interpretation is done on the replacement string. Unlike Perl 4, Perl 5 treats backticks as normal delimiters; the replacement text is not evaluated as a command). This is particularly useful for matching Unix path names that contain “/”, to avoid LTS (leaning toothpick syndrome). If “?” is the delimiter, then the match-only-once rule of ?PATTERN? applies.

PATTERN may contain variables, which will be interpolated (and the pattern recompiled) every time the pattern search is evaluated. (Note that \$) and \$| might not be interpolated because they look like end-of-string tests.) If you want such a pattern to be compiled only once, add a /o after the trailing delimiter. This avoids expensive run-time recompilations, and is useful when the value you are interpolating won’t change over the life of the script. However, mentioning /o constitutes a promise that you won’t change the variables in the pattern. If you change them, Perl won’t even notice.

If the PATTERN evaluates to the empty string, the last *successfully* matched regular expression is used instead.

If the /g option is not used, m// in a list context returns a list consisting of the subexpressions matched by the parentheses in the pattern, i.e., (\$1, \$2, \$3...). (Note that here \$1 etc. are also set, and that this differs from Perl 4’s

behavior.) When there are no parentheses in the pattern, the return value is the list (1) for success. With or without parentheses, an empty list is returned upon failure.

Examples:

```
open(TTY, '/dev/tty');
<TTY> =~ /^y/i && foo();    # do foo if desired

if (/Version: *([0-9.]*)/) { $version = $1; }

next if m#^/usr/spool/uucp#;

# poor man's grep
$arg = shift;
while (<>) {
    print if /$arg/o;        # compile only once
}

if (($F1, $F2, $Etc) = ($foo =~ /^(\S+)\s+(\S+)\s*(.*)/))
```

This last example splits `$foo` into the first two words and the remainder of the line, and assigns those three fields to `$F1`, `$F2`, and `$Etc`. The conditional is true if any variables were assigned, i.e., if the pattern matched.

The `/g` modifier specifies global pattern matching—that is, matching as many times as possible within the string. How it behaves depends on the context. In list context, it returns a list of all the substrings matched by all the parentheses in the regular expression. If there are no parentheses, it returns a list of all the matched strings, as if there were parentheses around the whole pattern.

In scalar context, each execution of `m/g` finds the next match, returning `TRUE` if it matches, and `FALSE` if there is no further match. The position after the last match can be read or set using the `pos()` function; see the `pos` entry in the *perlfunc* manpage. A failed match normally resets the search position to the beginning of the string, but you can avoid that by adding the `/c` modifier (e.g. `m/gc`). Modifying the target string also resets the search position.

You can intermix `m/g` matches with `m/\G.../g`, where `\G` is a zero-width assertion that matches the exact position where the previous `m/g`, if any, left off. The `\G` assertion is not supported without the `/g` modifier; currently, without `/g`, `\G` behaves just like `\A`, but that's accidental and may change in the future.

Examples:

```
# list context
($one,$five,$fifteen) = ('uptime' =~ /(\d+\.\d+)/g);
```

```

# scalar context
$/ = ""; $* = 1; # $* deprecated in modern perls
while (defined($paragraph = <>)) {
    while ($paragraph =~ /[a-z](['"])*[.!?]+(['"])*\s/g) {
        $sentences++;
    }
}
print "$sentences\n";

# using m//gc with \G
$_ = "ppooqppqq";
while ($i++ < 2) {
    print "1: '";
    print $1 while /(o)/gc; print "', pos=", pos, "\n";
    print "2: '";
    print $1 if /\G(q)/gc; print "', pos=", pos, "\n";
    print "3: '";
    print $1 while /(p)/gc; print "', pos=", pos, "\n";
}

```

The last example should print:

```

1: 'oo', pos=4
2: 'q', pos=5
3: 'pp', pos=7
1: '', pos=7
2: 'q', pos=8
3: '', pos=8

```

A useful idiom for lex-like scanners is `/\G.../gc`. You can combine several regexps like this to process a string part-by-part, doing different actions depending on which regexp matched. Each regexp tries to match where the previous one leaves off.

```

$_ = <<'EOL';
$url = new URI::URL "http://www/"; die if $url eq "xXx";
EOL
LOOP:
{
    print(" digits"), redo LOOP if /\G\d+\b[.,;]?\s*/gc
    print(" lowercase"), redo LOOP if /\G[a-z]+\b[.,;]?\s*/gc
    print(" UPPERCASE"), redo LOOP if /\G[A-Z]+\b[.,;]?\s*/gc
    print(" Capitalized"), redo LOOP if /\G[A-Z][a-z]+\b[.,;]?\s*/gc
    print(" MiXeD"), redo LOOP if /\G[A-Za-z]+\b[.,;]?\s*/gc
    print(" alphanumeric"), redo LOOP if /\G[A-Za-z0-9]+\b[.,;]?\s*/gc
    print(" line-noise"), redo LOOP if /\G[^\A-Za-z0-9]+\s*/gc
    print ". That's all!\n";
}

```

Here is the output (split into several lines):

```

line-noise lowercase line-noise lowercase UPPERCASE line-noise
UPPERCASE line-noise lowercase line-noise lowercase line-noise
lowercase lowercase line-noise lowercase lowercase line-noise
MiXeD line-noise. That's all!

```

q/STRING/**'STRING'**

A single-quoted, literal string. A backslash represents a backslash unless followed by the delimiter or another backslash, in which case the delimiter or backslash is interpolated.

```

$foo = q!I said, "You said, 'She said it.'!";
$bar = q('This is it.');
```

\$baz = '\n'; # a two-character string

qq/STRING/**"STRING"**

A double-quoted, interpolated string.

```

$_ .= qq
  (** The previous line contains the naughty word "$1".\n)
    if /(tcl|rexx|python)/; # :-)
$baz = "\n"; # a one-character string

```

qr/STRING/imosx

A string which is (possibly) interpolated and then compiled as a regular expression. The result may be used as a pattern in a match

```

$re = qr/$pattern/;
$string =~ /foo${re}bar/; # can be interpolated in other pa
$string =~ $re; # or used standalone

```

Options are:

```

i Do case-insensitive pattern matching.
m Treat string as multiple lines.
o Compile pattern only once.
s Treat string as single line.
x Use extended regular expressions.

```

The benefit from this is that the pattern is precompiled into an internal representation, and does not need to be recompiled every time a match is attempted. This makes it very efficient to do something like:

```

foreach $pattern (@pattern_list) {
    my $re = qr/$pattern/;
    foreach $line (@lines) {
        if($line =~ /$re/) {
            do_something($line);
        }
    }
}

```

See the *perltre* manpage for additional information on valid syntax for `STRING`, and for a detailed look at the semantics of regular expressions.

qx/STRING/

‘STRING’

A string which is (possibly) interpolated and then executed as a system command with `/bin/sh` or its equivalent. Shell wildcards, pipes, and redirections will be honored. The collected standard output of the command is returned; standard error is unaffected. In scalar context, it comes back as a single (potentially multi-line) string. In list context, returns a list of lines (however you’ve defined lines with `$/` or `$INPUT_RECORD_SEPARATOR`).

Because backticks do not affect standard error, use shell file descriptor syntax (assuming the shell supports this) if you care to address this. To capture a command’s `STDERR` and `STDOUT` together:

```
$output = `cmd 2>&1`;
```

To capture a command’s `STDOUT` but discard its `STDERR`:

```
$output = `cmd 2>/dev/null`;
```

To capture a command’s `STDERR` but discard its `STDOUT` (ordering is important here):

```
$output = `cmd 2>&1 1>/dev/null`;
```

To exchange a command’s `STDOUT` and `STDERR` in order to capture the `STDERR` but leave its `STDOUT` to come out the old `STDERR`:

```
$output = `cmd 3>&1 1>&2 2>&3 3>&-`;
```

To read both a command’s `STDOUT` and its `STDERR` separately, it’s easiest and safest to redirect them separately to files, and then read from those files when the program is done:

```
system("program args 1>/tmp/program.stdout 2>/tmp/program.stderr");
```

Using single-quote as a delimiter protects the command from Perl’s double-quote interpolation, passing it on to the shell instead:

```

$perl_info = qx(ps $$);           # that’s Perl’s $$
$shell_info = qx'ps $$';         # that’s the new shell’s $$

```

Note that how the string gets evaluated is entirely subject to the command interpreter on your system. On most platforms, you will have to protect shell metacharacters if you want them treated literally. This is in practice difficult

to do, as it's unclear how to escape which characters. See the *perlsec* man-page for a clean and safe example of a manual *fork()* and *exec()* to emulate backticks safely.

On some platforms (notably DOS-like ones), the shell may not be capable of dealing with multiline commands, so putting newlines in the string may not get you what you want. You may be able to evaluate multiple commands in a single line by separating them with the command separator character, if your shell supports that (e.g. *;* on many Unix shells; *&* on the Windows NT cmd shell).

Beware that some command shells may place restrictions on the length of the command line. You must ensure your strings don't exceed this limit after any necessary interpolations. See the platform-specific release notes for more details about your particular environment.

Using this operator can lead to programs that are difficult to port, because the shell commands called vary between systems, and may in fact not be present at all. As one example, the *type* command under the POSIX shell is very different from the *type* command under DOS. That doesn't mean you should go out of your way to avoid backticks when they're the right way to get something done. Perl was made to be a glue language, and one of the things it glues together is commands. Just understand what you're getting yourself into.

See the section on *I/O Operators* for more discussion.

qw/STRING/

Returns a list of the words extracted out of *STRING*, using embedded whitespace as the word delimiters. It is exactly equivalent to

```
split(' ', q/STRING/);
```

This equivalency means that if used in scalar context, you'll get *split*'s (unfortunate) scalar context behavior, complete with mysterious warnings.

Some frequently seen examples:

```
use POSIX qw( setlocale localeconv )
@EXPORT = qw( foo bar baz );
```

A common mistake is to try to separate the words with comma or to put comments into a multi-line *qw-string*. For this reason the *-w* switch produce warnings if the *STRING* contains the “,” or the “#” character.

s/PATTERN/REPLACEMENT/egimosx

Searches a string for a pattern, and if found, replaces that pattern with the replacement text and returns the number of substitutions made. Otherwise it returns false (specifically, the empty string).

If no string is specified via the *=~* or *!~* operator, the *\$_* variable is searched and modified. (The string specified with *=~* must be scalar variable, an array element, a hash element, or an assignment to one of those, i.e., an lvalue.)

If the delimiter chosen is single quote, no variable interpolation is done on either the *PATTERN* or the *REPLACEMENT*. Otherwise, if the *PATTERN* contains a *\$* that looks like a variable rather than an end-of-string test, the

variable will be interpolated into the pattern at run-time. If you want the pattern compiled only once the first time the variable is interpolated, use the `/o` option. If the pattern evaluates to the empty string, the last successfully executed regular expression is used instead. See the *perlre* manpage for further explanation on these. See the *perllocale* manpage for discussion of additional considerations that apply when use `locale` is in effect.

Options are:

- e Evaluate the right side as an expression.
- g Replace globally, i.e., all occurrences.
- i Do case-insensitive pattern matching.
- m Treat string as multiple lines.
- o Compile pattern only once.
- s Treat string as single line.
- x Use extended regular expressions.

Any non-alphanumeric, non-whitespace delimiter may replace the slashes. If single quotes are used, no interpretation is done on the replacement string (the `/e` modifier overrides this, however). Unlike Perl 4, Perl 5 treats backticks as normal delimiters; the replacement text is not evaluated as a command. If the PATTERN is delimited by bracketing quotes, the REPLACEMENT has its own pair of quotes, which may or may not be bracketing quotes, e.g., `s(foo)(bar)` or `s<foo>/bar/`. A `/e` will cause the replacement portion to be interpreted as a full-fledged Perl expression and *eval()*ed right then and there. It is, however, syntax checked at compile-time.

Examples:

```
s/\bgreen\b/mauve/g;           # don't change wintergreen

$path = ~ s|/usr/bin|/usr/local/bin|;

s/Login: $foo/Login: $bar/; # run-time pattern

($foo = $bar) = ~ s/this/that/; # copy first, then change

$count = ($paragraph = ~ s/Mister\b/Mr./g); # get change-count

$_ = 'abc123xyz';
s/\d+/$&*2/e;                 # yields 'abc246xyz'
s/\d+/sprintf("%5d",$&)/e;    # yields 'abc 246xyz'
s/\w/$& x 2/eg;               # yields 'aabbcc 224466xxyyzz'

s/%(.)/$percent{$1}/g;        # change percent escapes; no /e
s/%(.)/$percent{$1} || $&/ge; # expr now, so /e
s/^(\\w+)/&pod($1)/ge;        # use function call

# expand variables in $_, but dynamics only, using
# symbolic dereferencing
s/\\$(\\w+)/${$1}/g;
```

```

# /e's can even nest; this will expand
# any embedded scalar variable (including lexicals) in $_
s/(\$\w+)/$1/eeg;

# Delete (most) C comments.
$program =~ s {
    /\*      # Match the opening delimiter.
    .*?     # Match a minimal number of characters.
    \*/     # Match the closing delimiter.
} [lgsx];

s/^\s*(.*?)\s*$/ $1/;      # trim white space in $_, expensive

for ($variable) {          # trim white space in $variable,
    s/^\s+//;
    s/\s+$//;
}

s/([ ]*) *([ ]*)/$2 $1/;  # reverse 1st two fields

```

Note the use of \$ instead of \ in the last example. Unlike **sed**, we use the `<digit>` form in only the left hand side. Anywhere else it's `<digit>`.

Occasionally, you can't use just a /g to get all the changes to occur. Here are two common cases:

```

# put commas in the right places in an integer
1 while s/(.*\d)(\d\d\d)/$1,$2/g;      # perl4
1 while s/(\d)(\d\d\d)(?!\d)/$1,$2/g;  # perl5

# expand tabs to 8-column spacing
1 while s/\t+/' ' x (length($&)*8 - length($`)%8)/e;

```

tr/SEARCHLIST/REPLACEMENTLIST/cds

y/SEARCHLIST/REPLACEMENTLIST/cds

Transliterates all occurrences of the characters found in the search list with the corresponding character in the replacement list. It returns the number of characters replaced or deleted. If no string is specified via the `=~` or `!~` operator, the `$_` string is transliterated. (The string specified with `=~` must be a scalar variable, an array element, a hash element, or an assignment to one of those, i.e., an lvalue.) A character range may be specified with a hyphen, so `tr/A-J/0-9/` does the same replacement as `tr/ACEG-IBDFHJ/0246813579/`. For **sed** devotees, `y` is provided as a synonym for `tr`. If the SEARCHLIST is delimited by bracketing quotes, the REPLACEMENTLIST has its own pair of quotes, which may or may not be bracketing quotes, e.g., `tr[A-Z][a-z]` or `tr(+\-*)/ABCD/`.

Options:

```

c   Complement the SEARCHLIST.
d   Delete found but unreplaced characters.
s   Squash duplicate replaced characters.

```

If the `/c` modifier is specified, the SEARCHLIST character set is complemented. If the `/d` modifier is specified, any characters specified by SEARCHLIST not found in REPLACEMENTLIST are deleted. (Note that this is slightly more flexible than the behavior of some `tr` programs, which delete anything they find in the SEARCHLIST, period.) If the `/s` modifier is specified, sequences of characters that were transliterated to the same character are squashed down to a single instance of the character.

If the `/d` modifier is used, the REPLACEMENTLIST is always interpreted exactly as specified. Otherwise, if the REPLACEMENTLIST is shorter than the SEARCHLIST, the final character is replicated till it is long enough. If the REPLACEMENTLIST is empty, the SEARCHLIST is replicated. This latter is useful for counting characters in a class or for squashing character sequences in a class.

Examples:

```
$ARGV[1] =~ tr/A-Z/a-z/;      # canonicalize to lower case
$cnt = tr/*/*/;              # count the stars in $_
$cnt = $sky =~ tr/*/*/;      # count the stars in $sky
$cnt = tr/0-9//;             # count the digits in $_
tr/a-zA-Z//s;                # bookkeeper -> bokeper
($HOST = $host) =~ tr/a-z/A-Z/;
tr/a-zA-Z/ /cs;              # change non-alphas to single space
tr [\200-\377]
  [\000-\177];               # delete 8th bit
```

If multiple transliterations are given for a character, only the first one is used:

```
tr/AAA/XYZ/
```

will transliterate any A to X.

Note that because the transliteration table is built at compile time, neither the SEARCHLIST nor the REPLACEMENTLIST are subjected to double quote interpolation. That means that if you want to use variables, you must use an `eval()`:

```
eval "tr/$oldlist/$newlist/";
die "$@" if $@;

eval "tr/$oldlist/$newlist/, 1" or die $@;
```

Gory details of parsing quoted constructs

When presented with something which may have several different interpretations, Perl uses the principle **DWIM** (expanded to Do What I Mean – not what I wrote) to pick up the most probable interpretation of the source. This strategy is so successful that Perl users usually do not suspect ambivalence of what they write. However, time to time Perl's ideas differ from what the author meant.

The target of this section is to clarify the Perl's way of interpreting quoted constructs. The most frequent reason one may have to want to know the details discussed in this section is hairy regular expressions. However, the first steps of parsing are the same for all Perl quoting operators, so here they are discussed together.

Some of the passes discussed below are performed concurrently, but as far as results are the same, we consider them one-by-one. For different quoting constructs Perl performs different number of passes, from one to five, but they are always performed in the same order.

Finding the end

First pass is finding the end of the quoted construct, be it multichar ender "\nEOF\n" of <<EOF construct, / which terminates qq/ construct,] which terminates qq[construct, or > which terminates a fileglob started with <.

When searching for multichar construct no skipping is performed. When searching for one-char non-matching delimiter, such as /, combinations \\
/ are skipped. When searching for one-char matching delimiter, such as], combinations \\
] and \[are skipped, and nested [,] are skipped as well.

For 3-parts constructs, s/// etc. the search is repeated once more.

During this search no attention is paid to the semantic of the construct, thus

```
"$hash{ "$foo/$bar" }"
```

or

```
m/
  bar      # This is not a comment, this slash / terminated m/
/x
```

do not form legal quoted expressions. Note that since the slash which terminated m// was followed by a SPACE, this is not m//x, thus # was interpreted as a literal #.

Removal of backslashes before delimiters

During the second pass the text between the starting delimiter and the ending delimiter is copied to a safe location, and the \ is removed from combinations consisting of \ and *delimiter* (s) (both starting and ending delimiter if they differ).

The removal does not happen for multi-char delimiters.

Note that the combination \\
 is left as it was!

Starting from this step no information about the *delimiter* (s) is used in the parsing.

Interpolation

Next step is interpolation in the obtained delimiter-independent text. There are four different cases.

```
<<'EOF', m'', s'', tr///, y///
```

No interpolation is performed.

```
'', q//
```

The only interpolation is removal of \ from pairs \\.

\Q, \U, \u, \L, \l (possibly paired with \E) are converted to corresponding Perl constructs, thus "\$foo\Qbaz\$bar" is converted to

```
$foo . (quotemeta("baz" . $bar));
```

Other combinations of \ with following chars are substituted with appropriate expansions.

Interpolated scalars and arrays are converted to join and . Perl constructs, thus "@arr" becomes

```
" " . (join $", @arr) . " ";
```

Since all three above steps are performed simultaneously left-to-right, there is no way to insert a literal \$ or @ inside \Q\E pair: it cannot be protected by \, since any \ (except in \E) is interpreted as a literal inside \Q\E, and any \$ is interpreted as starting an interpolated scalar.

Note also that the interpolating code needs to make decision where the interpolated scalar ends, say, whether "a \$b -> {c}" means

```
"a " . $b . " -> {c}";
```

or

```
"a " . $b -> {c};
```

Most the time the decision is to take the longest possible text which does not include spaces between components and contains matching braces/brackets.

```
?RE?, /RE/, m/RE/, s/RE/fooo/,
```

Processing of \Q, \U, \u, \L, \l and interpolation happens (almost) as with qq// constructs, but *the substitution of * followed by other chars is not performed! Moreover, inside (?{BLOCK}) no processing is performed at all.

Interpolation has several quirks: \$|, \$(and \$) are not interpolated, and constructs \$var[SOMETHING] are *voted* (by several different estimators) to be an array element or \$var followed by a RE alternative. This is the place where the notation \${arr[\$bar]} comes handy:

/\${arr[0-9]}/ is interpreted as an array element -9, not as a regular expression from variable \$arr followed by a digit, which is the interpretation of /\$arr[0-9]/.

Note that absence of processing of \\ creates specific restrictions on the post-processed text: if the delimiter is /, one cannot get the combination \/ into the result of this step: / will finish the regular expression, \/ will be stripped to / on the previous step, and \/ will be left as is. Since / is

equivalent to `\/` inside a regular expression, this does not matter unless the delimiter is special character for the RE engine, as in `s*foo*bar*`, `m[foo]`, or `?foo?`.

This step is the last one for all the constructs except regular expressions, which are processed further.

Interpolation of regular expressions

All the previous steps were performed during the compilation of Perl code, this one happens in run time (though it may be optimized to be calculated at compile time if appropriate). After all the preprocessing performed above (and possibly after evaluation if catenation, joining, up/down-casing and `quotemeta()` are involved) the resulting *string* is passed to RE engine for compilation.

Whatever happens in the RE engine is better be discussed in the *perlre* manpage, but for the sake of continuity let us do it here.

This is the first step where presence of the `/x` switch is relevant. The RE engine scans the string left-to-right, and converts it to a finite automaton.

Backslashed chars are either substituted by corresponding literal strings, or generate special nodes of the finite automaton. Characters which are special to the RE engine generate corresponding nodes. (`?#. . .`) comments are ignored. All the rest is either converted to literal strings to match, or is ignored (as is whitespace and `#`-style comments if `/x` is present).

Note that the parsing of the construct `[. . .]` is performed using absolutely different rules than the rest of the regular expression. Similarly, the `{ . . . }` is only checked for matching braces.

Optimization of regular expressions

This step is listed for completeness only. Since it does not change semantics, details of this step are not documented and are subject to change.

I/O Operators

There are several I/O operators you should know about. A string enclosed by backticks (grave accents) first undergoes variable substitution just like a double quoted string. It is then interpreted as a command, and the output of that command is the value of the pseudo-literal, like in a shell. In scalar context, a single string consisting of all the output is returned. In list context, a list of values is returned, one for each line of output. (You can set `$/` to use a different line terminator.) The command is executed each time the pseudo-literal is evaluated. The status value of the command is returned in `$?` (see the *perlvar* manpage for the interpretation of `$?`). Unlike in **csh**, no translation is done on the return data—newlines remain newlines. Unlike in any of the shells, single quotes do not hide variable names in the command from interpretation. To pass a `$` through to the shell you need to hide it with a backslash. The generalized form of backticks is `qx//`. (Because backticks always undergo shell expansion as well, see the *perlsec* manpage for security concerns.)

Evaluating a filehandle in angle brackets yields the next line from that file (newline, if any, included), or `undef` at end of file. Ordinarily you must assign that value to a variable, but there is one situation where an automatic assignment happens. *If and ONLY if* the input symbol is the only thing inside the conditional of a `while` or `for(; ;)` loop, the value is automatically assigned to the variable `$_`. In these loop constructs, the assigned value (whether assignment is automatic or explicit) is then tested to see if it is defined. The defined test avoids problems where line has a string

value that would be treated as false by perl e.g. "" or "0" with no trailing newline. (This may seem like an odd thing to you, but you'll use the construct in almost every Perl script you write.) Anyway, the following lines are equivalent to each other:

```
while (defined($_ = <STDIN>)) { print; }
while ($_ = <STDIN>) { print; }
while (<STDIN>) { print; }
for (;<STDIN>;) { print; }
print while defined($_ = <STDIN>);
print while ($_ = <STDIN>);
print while <STDIN>;
```

and this also behaves similarly, but avoids the use of \$_ :

```
while (my $line = <STDIN>) { print $line }
```

If you really mean such values to terminate the loop they should be tested for explicitly:

```
while (($_ = <STDIN>) ne '0') { ... }
while (<STDIN>) { last unless $_; ... }
```

In other boolean contexts, *<filehandle>* without explicit `defined` test or comparison will solicit a warning if `-w` is in effect.

The filehandles `STDIN`, `STDOUT`, and `STDERR` are predefined. (The filehandles `stdin`, `stdout`, and `stderr` will also work except in packages, where they would be interpreted as local identifiers rather than global.) Additional filehandles may be created with the *open()* function. See the *open()* entry in the *perlfunc* manpage for details on this.

If a *<FILEHANDLE>* is used in a context that is looking for a list, a list consisting of all the input lines is returned, one line per list element. It's easy to make a *LARGE* data space this way, so use with care.

The null filehandle *<>* is special and can be used to emulate the behavior of **sed** and **awk**. Input from *<>* comes either from standard input, or from each file listed on the command line. Here's how it works: the first time *<>* is evaluated, the `@ARGV` array is checked, and if it is empty, `$ARGV[0]` is set to "-", which when opened gives you standard input. The `@ARGV` array is then processed as a list of filenames. The loop

```
while (<>) {
    ...                               # code for each line
}
```

is equivalent to the following Perl-like pseudo code:

```
unshift(@ARGV, '-') unless @ARGV;
while ($ARGV = shift) {
    open(ARGV, $ARGV);
    while (<ARGV>) {
        ...                               # code for each line
    }
}
```

except that it isn't so cumbersome to say, and will actually work. It really does shift

array `@ARGV` and put the current filename into variable `$ARGV`. It also uses filehandle `ARGV` internally--`<>` is just a synonym for `<ARGV>`, which is magical. (The pseudo code above doesn't work because it treats `<ARGV>` as non-magical.)

You can modify `@ARGV` before the first `<>` as long as the array ends up containing the list of filenames you really want. Line numbers (`$.`) continue as if the input were one big happy file. (But see example under `eof` for how to reset line numbers on each file.)

If you want to set `@ARGV` to your own list of files, go right ahead. This sets `@ARGV` to all plain text files if no `@ARGV` was given:

```
@ARGV = grep { -f && -T } glob('*') unless @ARGV;
```

You can even set them to pipe commands. For example, this automatically filters compressed arguments through **gzip**:

```
@ARGV = map { /\.(gz|z)$/ ? "gzip -dc < $_ |" : $_ } @ARGV;
```

If you want to pass switches into your script, you can use one of the `Getopts` modules or put a loop on the front like this:

```
while ($_ = $ARGV[0], /^-/) {
    shift;
    last if /^--$/;
    if (/^-D(.*)/) { $debug = $1 }
    if (/^-v/)      { $verbose++ }
    # ...          # other switches
}

while (<>) {
    # ...          # code for each line
}
```

The `<>` symbol will return `undef` for end-of-file only once. If you call it again after this it will assume you are processing another `@ARGV` list, and if you haven't set `@ARGV`, will input from `STDIN`.

If the string inside the angle brackets is a reference to a scalar variable (e.g., `<$foo>`), then that variable contains the name of the filehandle to input from, or its `typeglob`, or a reference to the same. For example:

```
$fh = \*STDIN;
$line = <$fh>;
```

If what's within the angle brackets is neither a filehandle nor a simple scalar variable containing a filehandle name, `typeglob`, or `typeglob` reference, it is interpreted as a filename pattern to be globbed, and either a list of filenames or the next filename in the list is returned, depending on context. This distinction is determined on syntactic grounds alone. That means `<$x>` is always a readline from an indirect handle, but `<$hash{key}>` is always a glob. That's because `$x` is a simple scalar variable, but `$hash{key}` is not—it's a hash element.

One level of double-quote interpretation is done first, but you can't say `<$foo>` because that's an indirect filehandle as explained in the previous paragraph. (In older versions of Perl, programmers would insert curly brackets to force interpretation as a

filename glob: `<${foo}>`. These days, it's considered cleaner to call the internal function directly as `glob($foo)`, which is probably the right way to have done it in the first place.) Example:

```
while (<*.c>) {
    chmod 0644, $_;
}
```

is equivalent to

```
open(FOO, "echo *.c | tr -s ' \\t\\r\\f' '\\012\\012\\012\\012'|");
while (<FOO>) {
    chop;
    chmod 0644, $_;
}
```

In fact, it's currently implemented that way. (Which means it will not work on filenames with spaces in them unless you have *csh*(1) on your machine.) Of course, the shortest way to do the above is:

```
chmod 0644, <*.c>;
```

Because globbing invokes a shell, it's often faster to call *readdir()* yourself and do your own *grep()* on the filenames. Furthermore, due to its current implementation of using a shell, the *glob()* routine may get "Arg list too long" errors (unless you've installed *tcsh*(1L) as */bin/csh*).

A glob evaluates its (embedded) argument only when it is starting a new list. All values must be read before it will start over. In a list context this isn't important, because you automatically get them all anyway. In scalar context, however, the operator returns the next value each time it is called, or a `undef` value if you've just run out. As for filehandles an automatic `defined` is generated when the glob occurs in the test part of a `while` or `for` – because legal glob returns (e.g. a file called `0`) would otherwise terminate the loop. Again, `undef` is returned only once. So if you're expecting a single value from a glob, it is much better to say

```
($file) = <blurch*>;
```

than

```
$file = <blurch*>;
```

because the latter will alternate between returning a filename and returning `FALSE`.

If you're trying to do variable interpolation, it's definitely better to use the *glob()* function, because the older notation can cause people to become confused with the indirect filehandle notation.

```
@files = glob("$dir/*. [ch]");
@files = glob($files[$i]);
```

Constant Folding

Like C, Perl does a certain amount of expression evaluation at compile time, whenever it determines that all arguments to an operator are static and have no side effects. In particular, string concatenation happens at compile time between literals that don't do variable substitution. Backslash interpretation also happens at compile time. You can say

```
'Now is the time for all' . "\n" .
    'good men to come to.'
```

and this all reduces to one string internally. Likewise, if you say

```
foreach $file (@filenames) {
    if (-s $file > 5 + 100 * 2**16) { }
}
```

the compiler will precompute the number that expression represents so that the interpreter won't have to.

Bitwise String Operators

Bitstrings of any size may be manipulated by the bitwise operators (`~` | `&` `^`).

If the operands to a binary bitwise op are strings of different sizes, **or** and **xor** ops will act as if the shorter operand had additional zero bits on the right, while the **and** op will act as if the longer operand were truncated to the length of the shorter.

```
# ASCII-based examples
print "j p \n" ^ " a h";           # prints "JAPH\n"
print "JA" | " ph\n";             # prints "japh\n"
print "japh\nJunk" & '_____' ;    # prints "JAPH\n"
print 'p N$' ^ " E<H\n";         # prints "Perl\n"
```

If you are intending to manipulate bitstrings, you should be certain that you're supplying bitstrings: If an operand is a number, that will imply a **numeric** bitwise operation. You may explicitly show which type of operation you intend by using `"` or `0+`, as in the examples below.

```
$foo = 150 | 105 ;                 # yields 255 (0x96 | 0x69 is 0xFF)
$foo = '150' | 105 ;              # yields 255
$foo = 150 | '105';               # yields 255
$foo = '150' | '105';             # yields string '155' (under ASCII)

$baz = 0+$foo & 0+$bar;           # both ops explicitly numeric
$biz = "$foo" ^ "$bar";          # both ops explicitly stringy
```

Integer Arithmetic

By default Perl assumes that it must do most of its arithmetic in floating point. But by saying

```
use integer;
```

you may tell the compiler that it's okay to use integer operations from here to the end

of the enclosing BLOCK. An inner BLOCK may countermand this by saying

```
no integer;
```

which lasts until the end of that BLOCK.

The bitwise operators ("&", "|", "^", "~", "<<", and ">>") always produce integral results. (But see also the section on *Bitwise String Operators*.) However, `use integer` still has meaning for them. By default, their results are interpreted as unsigned integers. However, if `use integer` is in effect, their results are interpreted as signed integers. For example, `~0` usually evaluates to a large integral value. However, `use integer; ~0` is `-1` on twos-complement machines.

Floating-point Arithmetic

While `use integer` provides integer-only arithmetic, there is no similar ways to provide rounding or truncation at a certain number of decimal places. For rounding to a certain number of digits, *sprintf()* or *printf()* is usually the easiest route.

Floating-point numbers are only approximations to what a mathematician would call real numbers. There are infinitely more reals than floats, so some corners must be cut. For example:

```
printf "%.20g\n", 123456789123456789;
#          produces 123456789123456784
```

Testing for exact equality of floating-point equality or inequality is not a good idea. Here's a (relatively expensive) work-around to compare whether two floating-point numbers are equal to a particular number of decimal places. See Knuth, volume II, for a more robust treatment of this topic.

```
sub fp_equal {
    my ($X, $Y, $POINTS) = @_;
    my ($tX, $tY);
    $tX = sprintf("%.${POINTS}g", $X);
    $tY = sprintf("%.${POINTS}g", $Y);
    return $tX eq $tY;
}
```

The POSIX module (part of the standard perl distribution) implements *ceil()*, *floor()*, and a number of other mathematical and trigonometric functions. The `Math::Complex` module (part of the standard perl distribution) defines a number of mathematical functions that can also work on real numbers. `Math::Complex` not as efficient as POSIX, but POSIX can't work with complex numbers.

Rounding in financial applications can have serious implications, and the rounding method used should be specified precisely. In these cases, it probably pays not to trust whichever system rounding is being used by Perl, but to instead implement the rounding function you need yourself.

Bigger Numbers

The standard `Math::BigInt` and `Math::BigFloat` modules provide variable precision arithmetic and overloaded operators. At the cost of some space and considerable speed, they avoid the normal pitfalls associated with limited-precision representations.

```
use Math::BigInt;  
$x = Math::BigInt->new('123456789123456789');  
print $x * $x;  
  
# prints +15241578780673678515622620750190521
```

