NAME

perlop - Perl operators and precedence

DESCRIPTION

Operator Precedence and Associativity

Operator precedence and associativity work in Perl more or less like they do in mathematics.

Operator precedence means some operators are evaluated before others. For example, in `2 + 4 * 5`, the multiplication has higher precedence so `4 * 5` is evaluated first yielding `2 + 20 == 22` and not `6 * 5 == 30`.

Operator associativity defines what happens if a sequence of the same operators is used one after another: whether the evaluator will evaluate the left operations first or the right. For example, in `8 - 4 - 2`, subtraction is left associative so Perl evaluates the expression left to right. `8 - 4` is evaluated first making the expression `4 - 2 == 2` and not `8 - 2 == 6`.

Perl operators have the following associativity and precedence, listed from highest precedence to lowest. 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. goto last next redo dump
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 *overload*.

Terms and List Operators (Leftward)

A TERM has the highest precedence in Perl. They include 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 *perlfunc*.

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 arguments that follow, and then act like a simple TERM with regard to the preceding expression. 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. The parentheses enclose the argument list for `print` which is evaluated (printing the result of `$foo & 255`). Then one is added to the return value of `print` (usually 1). The result is something like this:

```
1 + 1, "\n";    # Obviously not what you meant.
```

To do what you meant properly, you must write:

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

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

The Arrow Operator

"`->`" is an infix dereference operator, just as it is in C and C++. If the right side is either a `[...]`, `{...}`, or a `(...)` subscript, then the left side must be either a hard or symbolic reference to an array, a hash, or a subroutine respectively. (Or technically speaking, a location capable of holding a hard reference, if it's an array or hash reference being used for assignment.) See *perlreftut* and *perlref*.

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

Auto-increment and Auto-decrement

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

```
$i = 0;  $j = 0;
print $i++;  # prints 0
print ++$j;  # prints 1
```

Note that just as in C, Perl doesn't define **when** the variable is incremented or decremented. You just know it will be done sometime before or after the value is returned. This also means that modifying a variable twice in the same statement will lead to undefined behavior. Avoid statements like:

```
$i = $i ++;
print ++ $i + $i ++;
```

Perl will not guarantee what the result of the above statements is.

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]*\z/`, 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"
```

`undef` is always treated as numeric, and in particular is changed to `0` before incrementing (so that a post-increment of an undef value will return `0` rather than `undef`).

The auto-decrement operator is not magical.

Exponentiation

Binary "**" is the exponentiation operator. 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, that is, "not". See also `not` for a lower precedence version of this.

Unary "-" performs arithmetic negation if the operand is numeric, including any string that looks like a number. 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 the string "-bareword". If, however, the string begins with a non-alphabetic character (excluding "+" or "-"), Perl will attempt to convert the string to a numeric and the arithmetic negation is performed. If the string cannot be cleanly converted to a numeric, Perl will give the warning **Argument "the string" isn't numeric in negation (-) at ...**.

Unary "~" performs bitwise negation, that is, 1's complement. For example, `0666 & ~027` is 0640. (See also *Integer Arithmetic* and *Bitwise String Operators*.) Note that the width of the result is platform-dependent: ~0 is 32 bits wide on a 32-bit platform, but 64 bits wide on a 64-bit platform, so if you are expecting a certain bit width, remember to use the "&" operator to mask off the excess bits.

When complementing strings, if all characters have ordinal values under 256, then their complements will, also. But if they do not, all characters will be in either 32- or 64-bit complements, depending on your architecture. So for example, `~"\x{3B1}"` is `"\x{FFFF_FC4E}"` on 32-bit machines and `"\x{FFFF_FFFF_FFFF_FC4E}"` on 64-bit machines.

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 *Terms and List Operators (Leftward)*.)

Unary "\" creates a reference to whatever follows it. See *perlreftut* and *perlref*. 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 interpolation.

Binding Operators

Binary "=~" binds a scalar expression to a pattern match. Certain operations search or modify the string `$_` by default. This operator makes that kind of operation work on some other string. The right argument is a search pattern, substitution, or transliteration. The left argument is what is supposed to be searched, substituted, or transliterated instead of the default `$_`. When used in scalar context, the return value generally indicates the success of the operation. The exceptions are substitution (s///) and transliteration (y///) with the `/r` (non-destructive) option, which cause the **r**eturn value to be the result of the substitution. Behavior in list context depends on the particular operator. See *Regexp Quote-Like Operators* for details and *perlretut* for examples using these operators.

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. Note that this means that its contents will be interpolated twice, so

```
'\\' =~ q'\\';
```

is not ok, as the regex engine will end up trying to compile the pattern `\`, which it will consider a syntax error.

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

Binary "!~" with a non-destructive substitution (s///r) or transliteration (y///r) is a syntax error.

Multiplicative Operators

Binary "*" multiplies two numbers.

Binary "/" divides two numbers.

Binary "%" is the modulo operator, which computes the division remainder of its first argument with respect to its second argument. Given integer operands `$a` and `$b`: If `$b` is positive, then `$a % $b` is `$a` minus the largest multiple of `$b` less than or equal to `$a`. If `$b` is negative, then `$a % $b` is `$a` minus the smallest multiple of `$b` that is not less than `$a` (that is, the result will be less than or equal to zero). If the operands `$a` and `$b` are floating point values and the absolute value of `$b` (that is `abs($b)`) is less than `(UV_MAX + 1)`, only the integer portion of `$a` and `$b` will be used in the operation (Note: here `UV_MAX` means the maximum of the unsigned integer type). If the absolute value of the right operand (`abs($b)`) is greater than or equal to `(UV_MAX + 1)`, "%" computes the floating-point remainder `$r` in the equation `($r = $a - $i*$b)` where `$i` is a certain integer that makes `$r` have the same sign as the right operand `$b` (**not** as the left operand `$a` like C function `fmod()`) and the absolute value less than that of `$b`. Note that when `use integer` is in scope, "%" gives you direct access to the modulo 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 or if the left operand is not enclosed in parentheses, 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 enclosed in parentheses or is a list formed by `qw/STRING/`, it repeats the list. If the right operand is zero or negative, it returns an empty string or

an empty list, depending on the context.

```
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 *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 *Integer Arithmetic*.)

Note that both << and >> in Perl are implemented directly using << and >> in C. If `use integer` (see *Integer Arithmetic*) is in force then signed C integers are used, else unsigned C integers are used. Either way, the implementation isn't going to generate results larger than the size of the integer type Perl was built with (32 bits or 64 bits).

The result of overflowing the range of the integers is undefined because it is undefined also in C. In other words, using 32-bit integers, `1 << 32` is undefined. Shifting by a negative number of bits is also undefined.

If you get tired of being subject to your platform's native integers, the `use bigint` pragma neatly sidesteps the issue altogether:

```
print 20 << 20;  # 20971520
print 20 << 40;  # 5120 on 32-bit machines,
                 # 21990232555520 on 64-bit machines
use bigint;
print 20 << 100; # 25353012004564588029934064107520
```

Named Unary Operators

The various named unary operators are treated as functions with one argument, with optional parentheses.

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. For example, because named unary operators are higher precedence than `||`:

```
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 named operators:

```
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)
```

Regarding precedence, the filetest operators, like `-f`, `-M`, etc. are treated like named unary operators, but they don't follow this functional parenthesis rule. That means, for example, that `-f($file).".bak"` is equivalent to `-f "$file.bak"`.

See also *Terms and List Operators (Leftward)*.

Relational Operators

Perl operators that return true or false generally return values that can be safely used as numbers. For example, the relational operators in this section and the equality operators in the next one return `1` for true and a special version of the defined empty string, `""`, which counts as a zero but is exempt from warnings about improper numeric conversions, just as `"0 but true"` is.

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. If your platform supports NaNs (not-a-numbers) as numeric values, using them with "<=>" returns undef. NaN is not "<", "==", ">", "<=" or ">=" anything (even NaN), so those 5 return false. NaN != NaN returns true, as does NaN != anything else. If your platform doesn't support NaNs then NaN is just a string with numeric value 0.

```
$ perl -le '$a = "NaN"; print "No NaN support here" if $a == $a'
$ perl -le '$a = "NaN"; print "NaN support here" if $a != $a'
```

(Note that the *bigint*, *bigrat*, and *bignum* pragmas all support "NaN".)

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.

Binary "~~" does a smartmatch between its arguments. Smart matching is described in the next section.

"lt", "le", "ge", "gt" and "cmp" use the collation (sort) order specified by the current locale if a legacy `use locale` (but not `use locale ':not_characters'`) is in effect. See *perllocale*. Do not mix these with Unicode, only with legacy binary encodings. The standard *Unicode::Collate* and *Unicode::Collate::Locale* modules offer much more powerful solutions to collation issues.

Smartmatch Operator

First available in Perl 5.10.1 (the 5.10.0 version behaved differently), binary ~~ does a "smartmatch" between its arguments. This is mostly used implicitly in the `when` construct described in *perlsyn*, although not all `when` clauses call the smartmatch operator. Unique among all of Perl's operators, the smartmatch operator can recurse.

It is also unique in that all other Perl operators impose a context (usually string or numeric context) on their operands, autoconverting those operands to those imposed contexts. In contrast, smartmatch *infers* contexts from the actual types of its operands and uses that type information to select a suitable comparison mechanism.

The ~~ operator compares its operands "polymorphically", determining how to compare them according to their actual types (numeric, string, array, hash, etc.) Like the equality operators with which it shares the same precedence, ~~ returns 1 for true and "" for false. It is often best read aloud as "in", "inside of", or "is contained in", because the left operand is often looked for *inside* the right operand. That makes the order of the operands to the smartmatch operand often opposite that of the regular match operator. In other words, the "smaller" thing is usually placed in the left operand and the larger one in the right.

The behavior of a smartmatch depends on what type of things its arguments are, as determined by the following table. The first row of the table whose types apply determines the smartmatch behavior. Because what actually happens is mostly determined by the type of the second operand, the table is sorted on the right operand instead of on the left.

```
Left      Right      Description and pseudocode
===============================================================
Any       undef      check whether Any is undefined
               like: !defined Any

Any       Object     invoke ~~ overloading on Object, or die

Right operand is an ARRAY:

Left      Right      Description and pseudocode
===============================================================
ARRAY1    ARRAY2     recurse on paired elements of ARRAY1 and ARRAY2[2]
               like: (ARRAY1[0] ~~ ARRAY2[0])
                       && (ARRAY1[1] ~~ ARRAY2[1]) && ...
HASH      ARRAY      any ARRAY elements exist as HASH keys
               like: grep { exists HASH->{$_} } ARRAY
Regexp    ARRAY      any ARRAY elements pattern match Regexp
               like: grep { /Regexp/ } ARRAY
undef     ARRAY      undef in ARRAY
               like: grep { !defined } ARRAY
Any       ARRAY      smartmatch each ARRAY element[3]
               like: grep { Any ~~ $_ } ARRAY

Right operand is a HASH:
```

```
Left      Right      Description and pseudocode
================================================================
HASH1     HASH2      all same keys in both HASHes
               like: keys HASH1 ==
                        grep { exists HASH2->{$_} } keys HASH1
ARRAY     HASH       any ARRAY elements exist as HASH keys
               like: grep { exists HASH->{$_} } ARRAY
Regexp    HASH       any HASH keys pattern match Regexp
               like: grep { /Regexp/ } keys HASH
undef     HASH       always false (undef can't be a key)
               like: 0 == 1
Any       HASH       HASH key existence
               like: exists HASH->{Any}

Right operand is CODE:

Left      Right      Description and pseudocode
================================================================
ARRAY     CODE       sub returns true on all ARRAY elements[1]
               like: !grep { !CODE->($_) } ARRAY
HASH      CODE       sub returns true on all HASH keys[1]
               like: !grep { !CODE->($_) } keys HASH
Any       CODE       sub passed Any returns true
               like: CODE->(Any)
```

Right operand is a Regexp:

```
Left      Right      Description and pseudocode
================================================================
ARRAY     Regexp     any ARRAY elements match Regexp
               like: grep { /Regexp/ } ARRAY
HASH      Regexp     any HASH keys match Regexp
               like: grep { /Regexp/ } keys HASH
Any       Regexp     pattern match
               like: Any =~ /Regexp/

Other:

Left      Right      Description and pseudocode
================================================================
Object    Any        invoke ~~ overloading on Object,
                     or fall back to...

Any       Num        numeric equality
                like: Any == Num
Num       nummy[4]    numeric equality
                like: Num == nummy
undef     Any        check whether undefined
                like: !defined(Any)
Any       Any        string equality
                like: Any eq Any
```

Notes:

1. Empty hashes or arrays match.
2. That is, each element smartmatches the element of the same index in the other array.[3]
3. If a circular reference is found, fall back to referential equality.
4. Either an actual number, or a string that looks like one.

The smartmatch implicitly dereferences any non-blessed hash or array reference, so the *HASH* and *ARRAY* entries apply in those cases. For blessed references, the *Object* entries apply. Smartmatches involving hashes only consider hash keys, never hash values.

The "like" code entry is not always an exact rendition. For example, the smartmatch operator short-circuits whenever possible, but `grep` does not. Also, `grep` in scalar context returns the number of matches, but ~~ returns only true or false.

Unlike most operators, the smartmatch operator knows to treat `undef` specially:

```
use v5.10.1;
@array = (1, 2, 3, undef, 4, 5);
say "some elements undefined" if undef ~~ @array;
```

Each operand is considered in a modified scalar context, the modification being that array and hash variables are passed by reference to the operator, which implicitly dereferences them. Both elements of each pair are the same:

```
use v5.10.1;

my %hash = (red    => 1, blue   => 2, green  => 3,
            orange => 4, yellow => 5, purple => 6,
            black  => 7, grey   => 8, white  => 9);

my @array = qw(red blue green);

say "some array elements in hash keys" if  @array ~~  %hash;
say "some array elements in hash keys" if \@array ~~ \%hash;

say "red in array" if "red" ~~  @array;
say "red in array" if "red" ~~ \@array;

say "some keys end in e" if /e$/ ~~  %hash;
say "some keys end in e" if /e$/ ~~ \%hash;
```

Two arrays smartmatch if each element in the first array smartmatches (that is, is "in") the corresponding element in the second array, recursively.

```
use v5.10.1;
my @little = qw(red blue green);
my @bigger = ("red", "blue", [ "orange", "green" ] );
if (@little ~~ @bigger) {  # true!
    say "little is contained in bigger";
}
```

Because the smartmatch operator recurses on nested arrays, this will still report that "red" is in the array.

```
use v5.10.1;
my @array = qw(red blue green);
```

```
    my $nested_array = [[[[[[[ @array ]]]]]]];
    say "red in array" if "red" ~~ $nested_array;
```

If two arrays smartmatch each other, then they are deep copies of each others' values, as this example reports:

```
    use v5.12.0;
    my @a = (0, 1, 2, [3, [4, 5], 6], 7);
    my @b = (0, 1, 2, [3, [4, 5], 6], 7);

    if (@a ~~ @b && @b ~~ @a) {
        say "a and b are deep copies of each other";
    }
    elsif (@a ~~ @b) {
        say "a smartmatches in b";
    }
    elsif (@b ~~ @a) {
        say "b smartmatches in a";
    }
    else {
        say "a and b don't smartmatch each other at all";
    }
```

If you were to set `$b[3] = 4`, then instead of reporting that "a and b are deep copies of each other", it now reports that "b smartmatches in a". That because the corresponding position in @a contains an array that (eventually) has a 4 in it.

Smartmatching one hash against another reports whether both contain the same keys, no more and no less. This could be used to see whether two records have the same field names, without caring what values those fields might have. For example:

```
    use v5.10.1;
    sub make_dogtag {
        state $REQUIRED_FIELDS = { name=>1, rank=>1, serial_num=>1 };

        my ($class, $init_fields) = @_;

        die "Must supply (only) name, rank, and serial number"
            unless $init_fields ~~ $REQUIRED_FIELDS;

        ...
    }
```

or, if other non-required fields are allowed, use ARRAY ~~ HASH:

```
    use v5.10.1;
    sub make_dogtag {
        state $REQUIRED_FIELDS = { name=>1, rank=>1, serial_num=>1 };

        my ($class, $init_fields) = @_;

        die "Must supply (at least) name, rank, and serial number"
            unless [keys %{$init_fields}] ~~ $REQUIRED_FIELDS;
```

```
    ...
}
```

The smartmatch operator is most often used as the implicit operator of a `when` clause. See the section on "Switch Statements" in *perlsyn*.

Smartmatching of Objects

To avoid relying on an object's underlying representation, if the smartmatch's right operand is an object that doesn't overload ~~, it raises the exception "`Smartmatching a non-overloaded object breaks encapsulation`". That's because one has no business digging around to see whether something is "in" an object. These are all illegal on objects without a ~~ overload:

```
 %hash ~~ $object
    42 ~~ $object
"fred" ~~ $object
```

However, you can change the way an object is smartmatched by overloading the ~~ operator. This is allowed to extend the usual smartmatch semantics. For objects that do have an ~~ overload, see *overload*.

Using an object as the left operand is allowed, although not very useful. Smartmatching rules take precedence over overloading, so even if the object in the left operand has smartmatch overloading, this will be ignored. A left operand that is a non-overloaded object falls back on a string or numeric comparison of whatever the `ref` operator returns. That means that

```
$object ~~ X
```

does *not* invoke the overload method with *X* as an argument. Instead the above table is consulted as normal, and based on the type of *X*, overloading may or may not be invoked. For simple strings or numbers, in becomes equivalent to this:

```
$object ~~ $number          ref($object) == $number
$object ~~ $string          ref($object) eq $string
```

For example, this reports that the handle smells IOish (but please don't really do this!):

```
use IO::Handle;
my $fh = IO::Handle->new();
if ($fh ~~ /\bIO\b/) {
    say "handle smells IOish";
}
```

That's because it treats `$fh` as a string like `"IO::Handle=GLOB(0x8039e0)"`, then pattern matches against that.

Bitwise And

Binary "&" returns its operands ANDed together bit by bit. (See also *Integer Arithmetic* and *Bitwise String Operators*.)

Note that "&" has lower priority than relational operators, so for example the parentheses are essential in a test like

```
print "Even\n" if ($x & 1) == 0;
```

Bitwise Or and Exclusive Or

Binary "|" returns its operands ORed together bit by bit. (See also *Integer Arithmetic* and *Bitwise String Operators*.)

Binary "^" returns its operands XORed together bit by bit. (See also *Integer Arithmetic* and *Bitwise String Operators*.)

Note that "|" and "^" have lower priority than relational operators, so for example the brackets are essential in a test like

```
print "false\n" if (8 | 2) != 10;
```

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.

Logical Defined-Or

Although it has no direct equivalent in C, Perl's `//` operator is related to its C-style or. In fact, it's exactly the same as `||`, except that it tests the left hand side's definedness instead of its truth. Thus, `EXPR1 // EXPR2` returns the value of `EXPR1` if it's defined, otherwise, the value of `EXPR2` is returned. (`EXPR1` is evaluated in scalar context, `EXPR2` in the context of `//` itself). Usually, this is the same result as `defined(EXPR1) ? EXPR1 : EXPR2` (except that the ternary-operator form can be used as a lvalue, while `EXPR1 // EXPR2` cannot). This is very useful for providing default values for variables. If you actually want to test if at least one of `$a` and `$b` is defined, use `defined($a // $b)`.

The `||`, `//` and `&&` operators return the last value evaluated (unlike C's `||` and `&&`, which return 0 or 1). Thus, a reasonably portable way to find out the home directory 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 alternatives to `&&` and `||` when used for control flow, Perl provides the `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);
```

It would be even more readable to write that this way:

```
unless(unlink("alpha", "beta", "gamma")) {
    gripe();
    next LINE;
}
```

Using "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 a list of values counting (up by ones) from the left value to the right value. If the left value is greater than the right value then it returns the empty list. The range operator 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
}
```

The range operator also works on strings, using the magical auto-increment, see below.

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, even across calls to a subroutine that contains it. 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 until the next evaluation, as in **sed**, just use three dots ("...") instead of two. In all other regards, "..." behaves just like ".." does.

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 considered true if it is equal (==) to the current input line number (the `$.` variable).

To be pedantic, the comparison is actually `int(EXPR) == int(EXPR)`, but that is only an issue if you use a floating point expression; when implicitly using `$.` as described in the previous paragraph, the comparison is `int(EXPR) == int($.)` which is only an issue when `$.` is set to a floating point value and you are not reading from a file. Furthermore, `"span" .. "spat"` or `2.18 .. 3.14` will not do what you want in scalar context because each of the operands are evaluated using their integer representation.

Examples:

As a scalar operator:

```
if (101 .. 200) { print; } # print 2nd hundred lines, short for
                           #  if ($. == 101 .. $. == 200) { print; }

next LINE if (1 .. /^$/);  # skip header lines, short for
                           #   next LINE if ($. == 1 .. /^$/);
                           # (typically in a loop labeled LINE)

s/^/> / if (/^$/ .. eof());  # quote body

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

Here's a simple example to illustrate the difference between the two range operators:

```
@lines = ("   - Foo",
          "01 - Bar",
          "1  - Baz",
          "   - Quux");

foreach (@lines) {
    if (/0/ .. /1/) {
        print "$_\n";
    }
}
```

This program will print only the line containing "Bar". If the range operator is changed to `...`, it will also print the "Baz" line.

And now some examples 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 normal letters of the English 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.

If the initial value specified isn't part of a magical increment sequence (that is, a non-empty string matching `/^[a-zA-Z]*[0-9]*\z/`), only the initial value will be returned. So the following will only return an alpha:

```
use charnames "greek";
my @greek_small =  ("\N{alpha}" .. "\N{omega}");
```

To get the 25 traditional lowercase Greek letters, including both sigmas, you could use this instead:

```
use charnames "greek";
my @greek_small =  map { chr } ( ord("\N{alpha}")
                                   ..
                                 ord("\N{omega}")
                               );
```

However, because there are *many* other lowercase Greek characters than just those, to match lowercase Greek characters in a regular expression, you would use the pattern `/(?:(?=\p{Greek})\p{Lower})+/`.

Because each operand is evaluated in integer form, `2.18 .. 3.14` will return two elements in list context.

```
@list = (2.18 .. 3.14); # same as @list = (2 .. 3);
```

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;
```

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)
```

That should probably be written more simply as:

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

Although these are grouped by family, they all have the precedence of assignment.

Unlike in C, the scalar 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/13579/24680/;
```

Although as of 5.14, that can be also be accomplished this way:

```
use v5.14;
$tmp = ($global =~  tr/13579/24680/r);
```

Likewise,

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

is equivalent to

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

Similarly, a list assignment in list context produces the list of lvalues assigned to, and a list assignment in scalar context returns the number of elements produced by the expression on the right hand side of the assignment.

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. These arguments are also evaluated from left to right.

The => operator is a synonym for the comma except that it causes a word on its left to be interpreted as a string if it begins with a letter or underscore and is composed only of letters, digits and underscores. This includes operands that might otherwise be interpreted as operators, constants, single number v-strings or function calls. If in doubt about this behavior, the left operand can be quoted explicitly.

Otherwise, the => operator behaves exactly as the comma operator or list argument separator, according to context.

For example:

```
use constant FOO => "something";

my %h = ( FOO => 23 );
```

is equivalent to:

```
my %h = ("FOO", 23);
```

It is *NOT*:

```
my %h = ("something", 23);
```

The => operator is helpful in documenting the correspondence between keys and values in hashes, and other paired elements in lists.

```
%hash = ( $key => $value );
login( $username => $password );
```

The special quoting behavior ignores precedence, and hence may apply to *part* of the left operand:

```
print time.shift => "bbb";
```

That example prints something like "1314363215shiftbbb", because the => implicitly quotes the `shift` immediately on its left, ignoring the fact that `time.shift` is the entire left operand.

List Operators (Rightward)

On the right side of a list operator, the comma 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 parentheses:

```
open HANDLE, "< :utf8", "filename" or die "Can't open: $!\n";
```

However, some people find that code harder to read than writing it with parentheses:

```
open(HANDLE, "< :utf8", "filename") or die "Can't open: $!\n";
```

in which case you might as well just use the more customary "||" operator:

```
open(HANDLE, "< :utf8", "filename") || die "Can't open: $!\n";
```

See also discussion of list operators in *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: 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: the right expression is evaluated only if the left expression is false. Due to its precedence, you must be careful to avoid using it as replacement for the `||` operator. It usually works out better for flow control than in assignments:

```
$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).

There is no low precedence operator for defined-OR.

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.

```
    Customary  Generic        Meaning      Interpolates
 ''  q{}       Literal    no
 "" qq{}       Literal    yes
 `` qx{}       Command    yes*
  qw{}      Word list    no
 //  m{}    Pattern match   yes*
  qr{}       Pattern    yes*
   s{}{}     Substitution   yes*
  tr{}{}   Transliteration   no (but see below)
   y{}{}   Transliteration   no (but see below)
       <<EOF                 here-doc           yes*

 * unless the delimiter is ''.
```

Non-bracketing delimiters use the same character fore and aft, but the four sorts of ASCII brackets (round, angle, square, curly) all nest, which means that

```
    q{foo{bar}baz}
```

is the same as

```
    'foo{bar}baz'
```

Note, however, that this does not always work for quoting Perl code:

```
    $s = q{ if($a eq "}") ... }; # WRONG
```

is a syntax error. The `Text::Balanced` module (standard as of v5.8, and from CPAN before then) is able to do this properly.

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 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.
```

The following escape sequences are available in constructs that interpolate, and in transliterations:

```
    Sequence     Note  Description
    \t                 tab              (HT, TAB)
    \n                 newline          (NL)
    \r                 return           (CR)
    \f                 form feed        (FF)
    \b                 backspace        (BS)
    \a                 alarm (bell)     (BEL)
    \e                 escape           (ESC)
    \x{263A}     [1,8] hex char         (example: SMILEY)
    \x1b         [2,8] restricted range hex char (example: ESC)
    \N{name}     [3]   named Unicode character or character sequence
    \N{U+263D}   [4,8] Unicode character (example: FIRST QUARTER MOON)
    \c[          [5]   control char     (example: chr(27))
```

```
\o{23072}    [6,8]  octal char        (example: SMILEY)
\033         [7,8]  restricted range octal char  (example: ESC)
```

[1]

The result is the character specified by the hexadecimal number between the braces. See *[8]* below for details on which character.

Only hexadecimal digits are valid between the braces. If an invalid character is encountered, a warning will be issued and the invalid character and all subsequent characters (valid or invalid) within the braces will be discarded.

If there are no valid digits between the braces, the generated character is the NULL character (`\x{00}`). However, an explicit empty brace (`\x{}`) will not cause a warning (currently).

[2]

The result is the character specified by the hexadecimal number in the range 0x00 to 0xFF. See *[8]* below for details on which character.

Only hexadecimal digits are valid following `\x`. When `\x` is followed by fewer than two valid digits, any valid digits will be zero-padded. This means that `\x7` will be interpreted as `\x07`, and a lone <\x> will be interpreted as `\x00`. Except at the end of a string, having fewer than two valid digits will result in a warning. Note that although the warning says the illegal character is ignored, it is only ignored as part of the escape and will still be used as the subsequent character in the string. For example:

```
  Original    Result    Warns?
  "\x7"       "\x07"    no
  "\x"        "\x00"    no
  "\x7q"      "\x07q"   yes
  "\xq"       "\x00q"   yes
```

[3]

The result is the Unicode character or character sequence given by *name*. See *charnames*.

[4]

`\N{U+`*hexadecimal number*`}` means the Unicode character whose Unicode code point is *hexadecimal number*.

[5]

The character following `\c` is mapped to some other character as shown in the table:

```
 Sequence   Value
   \c@      chr(0)
   \cA      chr(1)
   \ca      chr(1)
   \cB      chr(2)
   \cb      chr(2)
   ...
   \cZ      chr(26)
   \cz      chr(26)
   \c[      chr(27)
   \c]      chr(29)
   \c^      chr(30)
   \c?      chr(127)
```

In other words, it's the character whose code point has had 64 xor'd with its uppercase. `\c?` is DELETE because `ord("?") ^ 64` is 127, and `\c@` is NULL because the ord of "@" is 64, so xor'ing 64 itself produces 0.

Also, `\c\X` yields `chr(28) . "X"` for any *X*, but cannot come at the end of a string, because the backslash would be parsed as escaping the end quote.

On ASCII platforms, the resulting characters from the list above are the complete set of ASCII controls. This isn't the case on EBCDIC platforms; see *"OPERATOR DIFFERENCES" in perlebcdic* for the complete list of what these sequences mean on both ASCII and EBCDIC platforms.

Use of any other character following the "c" besides those listed above is discouraged, and some are deprecated with the intention of removing those in a later Perl version. What happens for any of these other characters currently though, is that the value is derived by xor'ing with the seventh bit, which is 64.

To get platform independent controls, you can use `\N{...}`.

[6]

The result is the character specified by the octal number between the braces. See *[8]* below for details on which character.

If a character that isn't an octal digit is encountered, a warning is raised, and the value is based on the octal digits before it, discarding it and all following characters up to the closing brace. It is a fatal error if there are no octal digits at all.

[7]

The result is the character specified by the three-digit octal number in the range 000 to 777 (but best to not use above 077, see next paragraph). See *[8]* below for details on which character.

Some contexts allow 2 or even 1 digit, but any usage without exactly three digits, the first being a zero, may give unintended results. (For example, in a regular expression it may be confused with a backreference; see *"Octal escapes" in perlrebackslash*.) Starting in Perl 5.14, you may use `\o{}` instead, which avoids all these problems. Otherwise, it is best to use this construct only for ordinals `\077` and below, remembering to pad to the left with zeros to make three digits. For larger ordinals, either use `\o{}`, or convert to something else, such as to hex and use `\x{}` instead.

Having fewer than 3 digits may lead to a misleading warning message that says that what follows is ignored. For example, `"\128"` in the ASCII character set is equivalent to the two characters `"\n8"`, but the warning `Illegal octal digit '8' ignored` will be thrown. If `"\n8"` is what you want, you can avoid this warning by padding your octal number with `0`'s: `"\0128"`.

[8]

Several constructs above specify a character by a number. That number gives the character's position in the character set encoding (indexed from 0). This is called synonymously its ordinal, code position, or code point. Perl works on platforms that have a native encoding currently of either ASCII/Latin1 or EBCDIC, each of which allow specification of 256 characters. In general, if the number is 255 (0xFF, 0377) or below, Perl interprets this in the platform's native encoding. If the number is 256 (0x100, 0400) or above, Perl interprets it as a Unicode code point and the result is the corresponding Unicode character. For example `\x{50}` and `\o{120}` both are the number 80 in decimal, which is less than 256, so the number is interpreted in the native character set encoding. In ASCII the character in the 80th position (indexed from 0) is the letter "P", and in EBCDIC it is the ampersand symbol "&". `\x{100}` and `\o{400}` are both 256 in decimal, so the number is interpreted as a Unicode code point no matter what the native encoding is. The name of the character in the 256th position (indexed by 0) in Unicode is `LATIN CAPITAL LETTER A WITH MACRON`.

There are a couple of exceptions to the above rule. `\N{U+`*`hex number`*`}` is always interpreted as a Unicode code point, so that `\N{U+0050}` is "P" even on EBCDIC platforms. And if *`use encoding`* is in effect, the number is considered to be in that encoding, and is translated from that into the platform's native encoding if there is a corresponding native

character; otherwise to Unicode.

NOTE: Unlike C and other languages, Perl has no `\v` escape sequence for the vertical tab (VT, which is 11 in both ASCII and EBCDIC), but you may use `\ck` or `\x0b`. (`\v` does have meaning in regular expression patterns in Perl, see *perlre*.)

The following escape sequences are available in constructs that interpolate, but not in transliterations.

```
    \l  lowercase next character only
    \u  titlecase (not uppercase!) next character only
    \L  lowercase all characters till \E or end of string
    \U  uppercase all characters till \E or end of string
    \F  foldcase all characters till \E or end of string
    \Q          quote (disable) pattern metacharacters till \E or
                end of string
    \E  end either case modification or quoted section
  (whichever was last seen)
```

See *"quotemeta" in perlfunc* for the exact definition of characters that are quoted by `\Q`.

`\L`, `\U`, `\F`, and `\Q` can stack, in which case you need one `\E` for each. For example:

```
 say"This \Qquoting \ubusiness \Uhere isn't quite\E done yet,\E is it?";
 This quoting\ Business\ HERE\ ISN\'T\ QUITE\ done\ yet\, is it?
```

If `use locale` is in effect (but not `use locale ':not_characters'`), the case map used by `\l`, `\L`, `\u`, and `\U` is taken from the current locale. See *perllocale*. If Unicode (for example, `\N{}` or code points of 0x100 or beyond) is being used, the case map used by `\l`, `\L`, `\u`, and `\U` is as defined by Unicode. That means that case-mapping a single character can sometimes produce several characters. Under `use locale`, `\F` produces the same results as `\L`.

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 only 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 the ancient Macs (pre-MacOS X) of yesteryear, these used to be reversed, and on systems without line terminator, printing `"\n"` might 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 (`"\015\012"` or `"\cM\cJ"`) 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.

For constructs that do interpolate, variables beginning with "$" or "@" are interpolated. Subscripted variables such as `$a[3]` or `$href->{key}[0]` are also interpolated, as are array and hash slices. But method calls such as `$obj->meth` are not.

Interpolating an array or slice interpolates the elements in order, separated by the value of `$"`, so is equivalent to interpolating `join $", @array`. "Punctuation" arrays such as `@*` are usually interpolated only if the name is enclosed in braces `@{*}`, but the arrays `@_`, `@+`, and `@-` are interpolated even without braces.

For double-quoted strings, the quoting from `\Q` is applied after interpolation and escapes are processed.

```
    "abc\Qfoo\tbar$s\Exyz"
```

is equivalent to

```
"abc" . quotemeta("foo\tbar$s") . "xyz"
```

For the pattern of regex operators (`qr//`, `m//` and `s///`), the quoting from `\Q` is applied after interpolation is processed, but before escapes are processed. This allows the pattern to match literally (except for $ and @). For example, the following matches:

```
'\s\t' =~ /\Q\s\t/
```

Because $ or @ trigger interpolation, you'll need to use something like `/\Quser\E\@\Qhost/` to match them literally.

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 behavior described above, Perl does not expand 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.

Regexp Quote-Like Operators

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

qr/STRING/msixpodual

This operator quotes (and possibly compiles) its *STRING* as a regular expression. *STRING* is interpolated the same way as *PATTERN* in `m/PATTERN/`. If "'" is used as the delimiter, no interpolation is done. Returns a Perl value which may be used instead of the corresponding `/STRING/msixpodual` expression. The returned value is a normalized version of the original pattern. It magically differs from a string containing the same characters: `ref(qr/x/)` returns "Regexp"; however, dereferencing it is not well defined (you currently get the normalized version of the original pattern, but this may change).

For example,

```
$rex = qr/my.STRING/is;
print $rex;                 # prints (?si-xm:my.STRING)
s/$rex/foo/;
```

is equivalent to

```
s/my.STRING/foo/is;
```

The result may be used as a subpattern in a match:

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

Since Perl may compile the pattern at the moment of execution of the qr() operator, using qr() may have speed advantages in some situations, notably if the result of qr() is used standalone:

```
    sub match {
my $patterns = shift;
my @compiled = map qr/$_/i, @$patterns;
grep {
    my $success = 0;
    foreach my $pat (@compiled) {
```

```
    $success = 1, last if /$pat/;
       }
       $success;
  } @_;
     }
```

Precompilation of the pattern into an internal representation at the moment of qr() avoids a need to recompile the pattern every time a match `/$pat/` is attempted. (Perl has many other internal optimizations, but none would be triggered in the above example if we did not use qr() operator.)

Options (specified by the following modifiers) are:

```
m Treat string as multiple lines.
s Treat string as single line. (Make . match a newline)
i Do case-insensitive pattern matching.
x Use extended regular expressions.
p When matching preserve a copy of the matched string so
    that ${^PREMATCH}, ${^MATCH}, ${^POSTMATCH} will be
    defined.
o Compile pattern only once.
a   ASCII-restrict: Use ASCII for \d, \s, \w; specifying two
    a's further restricts /i matching so that no ASCII
    character will match a non-ASCII one.
l   Use the locale.
u   Use Unicode rules.
d   Use Unicode or native charset, as in 5.12 and earlier.
```

If a precompiled pattern is embedded in a larger pattern then the effect of "msixpluad" will be propagated appropriately. The effect the "o" modifier has is not propagated, being restricted to those patterns explicitly using it.

The last four modifiers listed above, added in Perl 5.14, control the character set semantics, but `/a` is the only one you are likely to want to specify explicitly; the other three are selected automatically by various pragmas.

See *perlre* for additional information on valid syntax for STRING, and for a detailed look at the semantics of regular expressions. In particular, all modifiers except the largely obsolete `/o` are further explained in *"Modifiers" in perlre*. `/o` is described in the next section.

m/PATTERN/msixpodualgc

/PATTERN/msixpodualgc

Searches a string for a pattern match, and in scalar context returns true if it succeeds, false if it fails. 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 *perlre*.

Options are as described in `qr//` above; in addition, the following match process modifiers are available:

```
g  Match globally, i.e., find all occurrences.
c  Do not reset search position on a failed match when /g is
   in effect.
```

If "/" is the delimiter then the initial `m` is optional. With the `m` you can use any pair of non-whitespace (ASCII) characters as delimiters. This is particularly useful for matching path names that contain "/", to avoid LTS (leaning toothpick syndrome). If "?" is the delimiter, then a match-only-once rule applies, described in `m?PATTERN?` below. If "'" is the delimiter, no interpolation is performed on the PATTERN. When using a

character valid in an identifier, whitespace is required after the `m`.

PATTERN may contain variables, which will be interpolated every time the pattern search is evaluated, except for when the delimiter is a single quote. (Note that `$(`, `$)`, and `$|` are not interpolated because they look like end-of-string tests.) Perl will not recompile the pattern unless an interpolated variable that it contains changes. You can force Perl to skip the test and never recompile by adding a `/o` (which stands for "once") after the trailing delimiter. Once upon a time, Perl would recompile regular expressions unnecessarily, and this modifier was useful to tell it not to do so, in the interests of speed. But now, the only reasons to use `/o` are either:

1 The variables are thousands of characters long and you know that they don't change, and you need to wring out the last little bit of speed by having Perl skip testing for that. (There is a maintenance penalty for doing this, as mentioning `/o` constitutes a promise that you won't change the variables in the pattern. If you do change them, Perl won't even notice.)

2 you want the pattern to use the initial values of the variables regardless of whether they change or not. (But there are saner ways of accomplishing this than using `/o`.)

3 If the pattern contains embedded code, such as

```
use re 'eval';
$code = 'foo(?{ $x })';
/$code/
```

then perl will recompile each time, even though the pattern string hasn't changed, to ensure that the current value of `$x` is seen each time. Use `/o` if you want to avoid this.

The bottom line is that using `/o` is almost never a good idea.

The empty pattern //

If the PATTERN evaluates to the empty string, the last *successfully* matched regular expression is used instead. In this case, only the `g` and `c` flags on the empty pattern are honored; the other flags are taken from the original pattern. If no match has previously succeeded, this will (silently) act instead as a genuine empty pattern (which will always match).

Note that it's possible to confuse Perl into thinking `//` (the empty regex) is really `//` (the defined-or operator). Perl is usually pretty good about this, but some pathological cases might trigger this, such as `$a///` (is that `($a) / (//)` or `$a // /`?) and `print $fh //` (`print $fh(//` or `print($fh //`?). In all of these examples, Perl will assume you meant defined-or. If you meant the empty regex, just use parentheses or spaces to disambiguate, or even prefix the empty regex with an `m` (so `//` becomes `m//`).

Matching in list context

If the `/g` option is not used, `m//` in list context returns a list consisting of the subexpressions matched by the parentheses in the pattern, that is, (`$1`, `$2`, `$3`...) (Note that here `$1` etc. are also set). 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")
   || die "can't access /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 (no longer needed!)
}

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; that is, 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 the substrings matched by any capturing 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 *"pos" in perlfunc*. A failed match normally resets the search position to the beginning of the string, but you can avoid that by adding the `/c` modifier (for example, `m//gc`). Modifying the target string also resets the search position.

\G assertion

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. Without the `/g` modifier, the `\G` assertion still anchors at `pos()` as it was at the start of the operation (see *"pos" in perlfunc*), but the match is of course only attempted once. Using `\G` without `/g` on a target string that has not previously had a `/g` match applied to it is the same as using the `\A` assertion to match the beginning of the string. Note also that, currently, `\G` is only properly supported when anchored at the very beginning of the pattern.

Examples:

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

    # scalar context
    local $/ = "";
    while ($paragraph = <>) {
 while ($paragraph =~ /\p{Ll}['")]*[.!?]+['")]*\s/g) {
     $sentences++;
 }
    }
    say $sentences;
```

Here's another way to check for sentences in a paragraph:

```
 my $sentence_rx = qr{
    (?: (?<= ^ ) | (?<= \s ) )  # after start-of-string or
                                # whitespace
    \p{Lu}                      # capital letter
```

```
    .*?                         # a bunch of anything
    (?<= \S )                   # that ends in non-
                                # whitespace
    (?<! \b [DMS]r  )           # but isn't a common abbr.
    (?<! \b Mrs )
    (?<! \b Sra )
    (?<! \b St  )
    [.?!]                       # followed by a sentence
                                # ender
    (?= $ | \s )                # in front of end-of-string
                                # or whitespace
}sx;
local $/ = "";
while (my $paragraph = <>) {
    say "NEW PARAGRAPH";
    my $count = 0;
    while ($paragraph =~ /($sentence_rx)/g) {
        printf "\tgot sentence %d: <%s>\n", ++$count, $1;
    }
}
```

Here's how to use `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";
    }
    print "Final: '$1', pos=",pos,"\n" if /\G(.)/;
```

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
    Final: 'q', pos=8
```

Notice that the final match matched q instead of p, which a match without the `\G` anchor would have done. Also note that the final match did not update `pos`. `pos` is only updated on a `/g` match. If the final match did indeed match p, it's a good bet that you're running a very old (pre-5.6.0) version of Perl.

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 = URI::URL->new( "http://example.com/" );
    die if $url eq "xXx";
 EOL

 LOOP: {
```

```
    print(" digits"),       redo LOOP if /\G\d+\b[,.;]?\s*/gc;
    print(" lowercase"),    redo LOOP
                                if /\G\p{Ll}+\b[,.;]?\s*/gc;
    print(" UPPERCASE"),    redo LOOP
                                if /\G\p{Lu}+\b[,.;]?\s*/gc;
    print(" Capitalized"),  redo LOOP
                                 if /\G\p{Lu}\p{Ll}+\b[,.;]?\s*/gc;
    print(" MiXeD"),        redo LOOP if /\G\pL+\b[,.;]?\s*/gc;
    print(" alphanumeric"), redo LOOP
                               if /\G[\p{Alpha}\pN]+\b[,.;]?\s*/gc;
    print(" line-noise"),   redo LOOP if /\G\W+/gc;
    print ". That's all!\n";
}
```

Here is the output (split into several lines):

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

m?PATTERN?msixpodualgc

?PATTERN?msixpodualgc

This is just like the `m/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 `m??` patterns local to the current package are reset.

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

Another example switched the first "latin1" encoding it finds to "utf8" in a pod file:

```
    s//utf8/ if m? ^ =encoding \h+ \K latin1 ?x;
```

The match-once behavior is controlled by the match delimiter being `?`; with any other delimiter this is the normal `m//` operator.

For historical reasons, the leading `m` in `m?PATTERN?` is optional, but the resulting `?PATTERN?` syntax is deprecated, will warn on usage and might be removed from a future stable release of Perl (without further notice!).

s/PATTERN/REPLACEMENT/msixpodualgcer

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 the `/r` (non-destructive) option is used then it runs the substitution on a copy of the string and instead of returning the number of substitutions, it returns the copy whether or not a substitution occurred. The original string is never changed when `/r` is used. The copy will always be a plain string, even if the input is an object or a tied variable.

If no string is specified via the =~ or `!~` operator, the `$_` variable is searched and modified. Unless the `/r` option is used, the string specified must be a scalar variable,

an array element, a hash element, or an assignment to one of those; that is, some sort of scalar lvalue.

If the delimiter chosen is a single quote, no 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 *perlre* for further explanation on these.

Options are as with m// with the addition of the following replacement specific options:

```
e Evaluate the right side as an expression.
ee  Evaluate the right side as a string then eval the
    result.
r   Return substitution and leave the original string
    untouched.
```

Any non-whitespace delimiter may replace the slashes. Add space after the `s` when using a character allowed in identifiers. If single quotes are used, no interpretation is done on the replacement string (the `/e` modifier overrides this, however). Note that Perl 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, for example, `s(foo)(bar)` or `s<foo>/bar/`. A `/e` will cause the replacement portion to be treated as a full-fledged Perl expression and evaluated right then and there. It is, however, syntax checked at compile-time. A second e modifier will cause the replacement portion to be `eval`ed before being run as a Perl expression.

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 = $bar) =~ s/this/that/; # copy first, then
                                    # change
($foo = "$bar") =~ s/this/that/; # convert to string,
                                    # copy, then change
$foo = $bar =~ s/this/that/r; # Same as above using /r
$foo = $bar =~ s/this/that/r
            =~ s/that/the other/r; # Chained substitutes
                                    # using /r
@foo = map { s/this/that/r } @bar # /r is very useful in
                                    # maps

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

$_ = 'abc123xyz';
s/\d+/$&*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
```

```
    $_ = 'abc123xyz';
    $a = s/abc/def/r;           # $a is 'def123xyz' and
                                # $_ remains 'abc123xyz'.

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

    # Add one to the value of any numbers in the string
    s/(\d+)/1 + $1/eg;

    # Titlecase words in the last 30 characters only
    substr($str, -30) =~ s/\b(\p{Alpha}+)\b/\u\L$1/g;

    # This will expand any embedded scalar variable
    # (including lexicals) in $_ : First $1 is interpolated
    # to the variable name, and then evaluated
    s/(\$\w+)/$1/eeg;

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

    s/^\s*(.*?)\s*$/$1/; # trim whitespace in $_,
                                # expensively

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

    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 that you might want. Here are two common cases:

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

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

Quote-Like Operators

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 /\b(tcl|java|python)\b/i;      # :-)
    $baz = "\n";  # a one-character string
```

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, or undef if the command failed. In list context, returns a list of lines (however you've defined lines with $/ or $INPUT_RECORD_SEPARATOR), or an empty list if the command failed.

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 to redirect them separately to files, and then read from those files when the program is done:

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

The STDIN filehandle used by the command is inherited from Perl's STDIN. For example:

```
    open(SPLAT, "stuff")   || die "can't open stuff: $!";
    open(STDIN, "<&SPLAT") || die "can't dupe SPLAT: $!";
    print STDOUT `sort`;
```

will print the sorted contents of the file named *"stuff"*.

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 $$';            # that's the new shell's $$
```

How that 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

perlsec 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 (for example, `;` on many Unix shells and `&` on the Windows NT `cmd` shell).

Perl will attempt to flush all files opened for output before starting the child process, but this may not be supported on some platforms (see *perlport*). To be safe, you may need to set `$|` ($AUTOFLUSH in English) or call the `autoflush()` method of `IO::Handle` on any open handles.

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 *I/O Operators* for more discussion.

qw/STRING/

Evaluates to a list of the words extracted out of STRING, using embedded whitespace as the word delimiters. It can be understood as being roughly equivalent to:

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

the differences being that it generates a real list at compile time, and in scalar context it returns the last element in the list. So this expression:

```
qw(foo bar baz)
```

is semantically equivalent to the list:

```
"foo", "bar", "baz"
```

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 `use warnings` pragma and the **-w** switch (that is, the `$^W` variable) produces warnings if the STRING contains the "," or the "#" character.

tr/SEARCHLIST/REPLACEMENTLIST/cdsr

y/SEARCHLIST/REPLACEMENTLIST/cdsr

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.

If the `/r` (non-destructive) option is present, a new copy of the string is made and its characters transliterated, and this copy is returned no matter whether it was modified or not: the original string is always left unchanged. The new copy is always a plain string, even if the input string is an object or a tied variable.

Unless the `/r` option is used, the string specified with =~ must be a scalar variable, an array

element, a hash element, or an assignment to one of those; in other words, an lvalue.

A character range may be specified with a hyphen, so `tr/A-J/0-9/` does the same replacement as `tr/ACEGIBDFHJ/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; for example, `tr[aeiouy][yuoiea]` or `tr(+\-*/)/ABCD/`.

Note that `tr` does **not** do regular expression character classes such as `\d` or `\pL`. The `tr` operator is not equivalent to the tr(1) utility. If you want to map strings between lower/upper cases, see *"lc" in perlfunc* and *"uc" in perlfunc*, and in general consider using the `s` operator if you need regular expressions. The `\U`, `\u`, `\L`, and `\l` string-interpolation escapes on the right side of a substitution operator will perform correct case-mappings, but `tr[a-z][A-Z]` will not (except sometimes on legacy 7-bit data).

Note also that the whole range idea is rather unportable between character sets--and even within character sets they may cause results you probably didn't expect. A sound principle is to use only ranges that begin from and end at either alphabets of equal case (a-e, A-E), or digits (0-4). Anything else is unsafe. If in doubt, spell out the character sets in full.

Options:

```
    c Complement the SEARCHLIST.
    d Delete found but unreplaced characters.
    s Squash duplicate replaced characters.
    r Return the modified string and leave the original string
 untouched.
```

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 ASCII

    $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/;
     $HOST = $host  =~ tr/a-z/A-Z/r;   # same thing

    $HOST = $host =~ tr/a-z/A-Z/r    # chained with s///r
                  =~ s/:/ -p/r;

    tr/a-zA-Z/ /cs;  # change non-alphas to single space
```

```
@stripped = map tr/a-zA-Z/ /csr, @original;
# /r with map

tr [\200-\377]
   [\000-\177];  # wickedly 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.

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 $@;
```

<<EOF

A line-oriented form of quoting is based on the shell "here-document" syntax. Following a << you specify a string to terminate the quoted material, and all lines following the current line down to the terminating string are the value of the item.

The terminating string may be either an identifier (a word), or some quoted text. An unquoted identifier works like double quotes. There may not be a space between the << and the identifier, unless the identifier is explicitly quoted. (If you put a space it will be treated as a null identifier, which is valid, and matches the first empty line.) The terminating string must appear by itself (unquoted and with no surrounding whitespace) on the terminating line.

If the terminating string is quoted, the type of quotes used determine the treatment of the text.

Double Quotes

Double quotes indicate that the text will be interpolated using exactly the same rules as normal double quoted strings.

```
       print <<EOF;
    The price is $Price.
    EOF

       print << "EOF"; # same as above
    The price is $Price.
    EOF
```

Single Quotes

Single quotes indicate the text is to be treated literally with no interpolation of its content. This is similar to single quoted strings except that backslashes have no special meaning, with \\ being treated as two backslashes and not one as they would in every other quoting construct.

Just as in the shell, a backslashed bareword following the << means the same thing as a single-quoted string does:

```
 $cost = <<'VISTA';  # hasta la ...
    That'll be $10 please, ma'am.
    VISTA

 $cost = <<\VISTA;   # Same thing!
```

```
    That'll be $10 please, ma'am.
    VISTA
```

This is the only form of quoting in perl where there is no need to worry about escaping content, something that code generators can and do make good use of.

Backticks

The content of the here doc is treated just as it would be if the string were embedded in backticks. Thus the content is interpolated as though it were double quoted and then executed via the shell, with the results of the execution returned.

```
    print << `EOC`; # execute command and get results
echo hi there
EOC
```

It is possible to stack multiple here-docs in a row:

```
    print <<"foo", <<"bar"; # you can stack them
I said foo.
foo
I said bar.
bar

    myfunc(<< "THIS", 23, <<'THAT');
Here's a line
or two.
THIS
and here's another.
THAT
```

Just don't forget that you have to put a semicolon on the end to finish the statement, as Perl doesn't know you're not going to try to do this:

```
    print <<ABC
179231
ABC
    + 20;
```

If you want to remove the line terminator from your here-docs, use `chomp()`.

```
chomp($string = <<'END');
This is a string.
END
```

If you want your here-docs to be indented with the rest of the code, you'll need to remove leading whitespace from each line manually:

```
($quote = <<'FINIS') =~ s/^\s+//gm;
   The Road goes ever on and on,
   down from the door where it began.
FINIS
```

If you use a here-doc within a delimited construct, such as in `s///eg`, the quoted material must still come on the line following the <<FOO marker, which means it may be inside the delimited construct:

```
s/this/<<E . 'that'
the other
E
 . 'more '/eg;
```

It works this way as of Perl 5.18. Historically, it was inconsistent, and you would have to write

```
s/this/<<E . 'that'
 . 'more '/eg;
the other
E
```

outside of string evals.

Additionally, quoting rules for the end-of-string identifier are unrelated to Perl's quoting rules. `q()`, `qq()`, and the like are not supported in place of `''` and `""`, and the only interpolation is for backslashing the quoting character:

```
print << "abc\"def";
testing...
abc"def
```

Finally, quoted strings cannot span multiple lines. The general rule is that the identifier must be a string literal. Stick with that, and you should be safe.

Gory details of parsing quoted constructs

When presented with something that might have several different interpretations, Perl uses the **DWIM** (that's "Do What I Mean") principle to pick the most probable interpretation. This strategy is so successful that Perl programmers often do not suspect the ambivalence of what they write. But from time to time, Perl's notions differ substantially from what the author honestly meant.

This section hopes to clarify how Perl handles quoted constructs. Although the most common reason to learn this is to unravel labyrinthine regular expressions, because the initial steps of parsing are the same for all quoting operators, they are all discussed together.

The most important Perl parsing rule is the first one discussed below: when processing a quoted construct, Perl first finds the end of that construct, then interprets its contents. If you understand this rule, you may skip the rest of this section on the first reading. The other rules are likely to contradict the user's expectations much less frequently than this first one.

Some passes discussed below are performed concurrently, but because their results are the same, we consider them individually. For different quoting constructs, Perl performs different numbers of passes, from one to four, but these passes are always performed in the same order.

Finding the end

The first pass is finding the end of the quoted construct, where the information about the delimiters is used in parsing. During this search, text between the starting and ending delimiters is copied to a safe location. The text copied gets delimiter-independent.

If the construct is a here-doc, the ending delimiter is a line that has a terminating string as the content. Therefore `<<EOF` is terminated by `EOF` immediately followed by `"\n"` and starting from the first column of the terminating line. When searching for the terminating line of a here-doc, nothing is skipped. In other words, lines after the here-doc syntax are compared with the terminating string line by line.

For the constructs except here-docs, single characters are used as starting and ending delimiters. If the starting delimiter is an opening punctuation (that is `(`, `[`, `{`, or `<`), the ending delimiter is the corresponding closing punctuation (that is `)`, `]`, `}`, or `>`). If the starting delimiter is an unpaired character like `/` or a closing punctuation, the ending delimiter is same as the starting delimiter. Therefore a `/` terminates a `qq//` construct, while a `]` terminates `qq[]` and `qq]]` constructs.

When searching for single-character delimiters, escaped delimiters and `\\` are skipped. For example, while searching for terminating `/`, combinations of `\\` and `\/` are skipped. If the delimiters are bracketing, nested pairs are also skipped. For example, while searching for closing `]` paired with the opening `[`, combinations of `\\`, `\]`, and `\[` are all skipped, and

nested [and] are skipped as well. However, when backslashes are used as the delimiters (like qq\\ and tr\\\), nothing is skipped. During the search for the end, backslashes that escape delimiters or other backslashes are removed (exactly speaking, they are not copied to the safe location).

For constructs with three-part delimiters (s///, y///, and tr///), the search is repeated once more. If the first delimiter is not an opening punctuation, three delimiters must be same such as s!!! and tr)))), in which case the second delimiter terminates the left part and starts the right part at once. If the left part is delimited by bracketing punctuation (that is (), [], {}, or <>), the right part needs another pair of delimiters such as s(){} and tr[]//. In these cases, whitespace and comments are allowed between both parts, though the comment must follow at least one whitespace character; otherwise a character expected as the start of the comment may be regarded as the starting delimiter of the right part.

During this search no attention is paid to the semantics of the construct. Thus:

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

or:

```
m/
  bar # NOT a comment, this slash / terminated m//!
 /x
```

do not form legal quoted expressions. The quoted part ends on the first " and /, and the rest happens to be a syntax error. Because the slash that terminated m// was followed by a SPACE, the example above is not m//x, but rather m// with no /x modifier. So the embedded # is interpreted as a literal #.

Also no attention is paid to \c\ (multichar control char syntax) during this search. Thus the second \ in qq/\c\/ is interpreted as a part of \/, and the following / is not recognized as a delimiter. Instead, use \034 or \x1c at the end of quoted constructs.

Interpolation

The next step is interpolation in the text obtained, which is now delimiter-independent. There are multiple cases.

<<'EOF'

No interpolation is performed. Note that the combination \\ is left intact, since escaped delimiters are not available for here-docs.

m'', the pattern of s'''

No interpolation is performed at this stage. Any backslashed sequences including \\ are treated at the stage to *parsing regular expressions*.

'', q//, tr''', y''', the replacement of s'''

The only interpolation is removal of \ from pairs of \\. Therefore - in tr''' and y''' is treated literally as a hyphen and no character range is available. \1 in the replacement of s''' does not work as $1.

tr///, y///

No variable interpolation occurs. String modifying combinations for case and quoting such as \Q, \U, and \E are not recognized. The other escape sequences such as \200 and \t and backslashed characters such as \\ and \- are converted to appropriate literals. The character - is treated specially and therefore \- is treated as a literal -.

"", ``, qq//, qx//, <file*glob>, <<"EOF"

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

`. $bar)` internally. The other escape sequences such as `\200` and `\t` and backslashed characters such as `\\` and `\-` are replaced with appropriate expansions.

Let it be stressed that *whatever falls between* `\Q` *and* `\E` is interpolated in the usual way. Something like `"\Q\\E"` has no `\E` inside. Instead, it has `\Q`, `\\`, and `E`, so the result is the same as for `"\\\\E"`. As a general rule, backslashes between `\Q` and `\E` may lead to counterintuitive results. So, `"\Q\t\E"` is converted to `quotemeta("\t")`, which is the same as `"\\\t"` (since TAB is not alphanumeric). Note also that:

```
$str = '\t';
return "\Q$str";
```

may be closer to the conjectural *intention* of the writer of `"\Q\t\E"`.

Interpolated scalars and arrays are converted internally to the `join` and `.` catenation operations. Thus, `"$foo XXX '@arr'"` becomes:

```
$foo . " XXX '" . (join $", @arr) . "'";
```

All operations above are performed simultaneously, left to right.

Because the result of `"\Q STRING \E"` has all metacharacters quoted, there is no way to insert a literal `$` or `@` inside a `\Q\E` pair. If protected by `\`, `$` will be quoted to became `"\\\$"`; if not, it is interpreted as the start of an interpolated scalar.

Note also that the interpolation code needs to make a decision on where the interpolated scalar ends. For instance, whether `"a $b -> {c}"` really means:

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

or:

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

Most of the time, the longest possible text that does not include spaces between components and which contains matching braces or brackets. because the outcome may be determined by voting based on heuristic estimators, the result is not strictly predictable. Fortunately, it's usually correct for ambiguous cases.

the replacement of `s///`

Processing of `\Q`, `\U`, `\u`, `\L`, `\l`, `\F` and interpolation happens as with `qq//` constructs.

It is at this step that `\1` is begrudgingly converted to `$1` in the replacement text of `s///`, in order to correct the incorrigible *sed* hackers who haven't picked up the saner idiom yet. A warning is emitted if the `use warnings` pragma or the **-w** command-line flag (that is, the `$^W` variable) was set.

RE in `?RE?`, `/RE/`, `m/RE/`, `s/RE/foo/`,

Processing of `\Q`, `\U`, `\u`, `\L`, `\l`, `\F`, `\E`, and interpolation happens (almost) as with `qq//` constructs.

Processing of `\N{...}` is also done here, and compiled into an intermediate form for the regex compiler. (This is because, as mentioned below, the regex compilation may be done at execution time, and `\N{...}` is a compile-time construct.)

However any other combinations of `\` followed by a character are not substituted but only skipped, in order to parse them as regular expressions at the following step. As `\c` is skipped at this step, `@` of `\c@` in RE is possibly treated as an array symbol (for example `@foo`), even though the same text in `qq//` gives interpolation of `\c@`.

Code blocks such as `(?{BLOCK})` are handled by temporarily passing control back to the perl parser, in a similar way that an interpolated array subscript expression such as

`"foo$array[1+f("[xyz")]bar"` would be.

Moreover, inside `(?{BLOCK})`, `(?# comment )`, and a `#`-comment in a `//x`-regular expression, no processing is performed whatsoever. This is the first step at which the presence of the `//x` modifier is relevant.

Interpolation in patterns has several quirks: `$|`, `$(`, `$)`, `@+` and `@-` are not interpolated, and constructs `$var[SOMETHING]` are voted (by several different estimators) to be either an array element or `$var` followed by an RE alternative. This is where the notation `${arr[$bar]}` comes handy: `/${arr[0-9]}/` is interpreted as array element `-9`, not as a regular expression from the variable `$arr` followed by a digit, which would be the interpretation of `/$arr[0-9]/`. Since voting among different estimators may occur, the result is not predictable.

The lack 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. Because `/` is equivalent to `\/` inside a regular expression, this does not matter unless the delimiter happens to be character special to the RE engine, such as in `s*foo*bar*`, `m[foo]`, or `?foo?`; or an alphanumeric char, as in:

```
m m ^ a \s* b mmx;
```

In the RE above, which is intentionally obfuscated for illustration, the delimiter is `m`, the modifier is `mx`, and after delimiter-removal the RE is the same as for `m/ ^ a \s* b /mx`. There's more than one reason you're encouraged to restrict your delimiters to non-alphanumeric, non-whitespace choices.

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

parsing regular expressions

Previous steps were performed during the compilation of Perl code, but this one happens at run time, although it may be optimized to be calculated at compile time if appropriate. After preprocessing described above, and possibly after evaluation if concatenation, joining, casing translation, or metaquoting are involved, the resulting *string* is passed to the RE engine for compilation.

Whatever happens in the RE engine might be better discussed in *perlre*, but for the sake of continuity, we shall do so here.

This is another step where the presence of the `//x` modifier is relevant. The RE engine scans the string from left to right and converts it to a finite automaton.

Backslashed characters are either replaced with corresponding literal strings (as with `\{`), or else they generate special nodes in the finite automaton (as with `\b`). Characters special to the RE engine (such as `|`) generate corresponding nodes or groups of nodes. `(?#...)` comments are ignored. All the rest is either converted to literal strings to match, or else is ignored (as is whitespace and `#`-style comments if `//x` is present).

Parsing of the bracketed character class construct, `[...]`, is rather different than the rule used for the rest of the pattern. The terminator of this construct is found using the same rules as for finding the terminator of a `{}`-delimited construct, the only exception being that `]` immediately following `[` is treated as though preceded by a backslash.

The terminator of runtime `(?{...})` is found by temporarily switching control to the perl parser, which should stop at the point where the logically balancing terminating `}` is found.

It is possible to inspect both the string given to RE engine and the resulting finite automaton. See the arguments `debug`/`debugcolor` in the `use re` pragma, as well as Perl's **-Dr** command-line switch documented in *"Command Switches" in perlrun*.

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 without notice. This step is performed over the finite automaton that was generated during the previous pass.

It is at this stage that `split()` silently optimizes `/^/` to mean `/^/m`.

I/O Operators

There are several I/O operators you should know about.

A string enclosed by backticks (grave accents) first undergoes double-quote interpolation. It is then interpreted as an external command, and the output of that command is the value of the backtick string, like in a shell. In scalar context, a single string consisting of all output is returned. In list context, a list of values is returned, one per 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 *perlvar* 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 literal dollar-sign 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 *perlsec* for security concerns.)

In scalar context, evaluating a filehandle in angle brackets yields the next line from that file (the newline, if any, included), or `undef` at end-of-file or on error. When `$/` is set to `undef` (sometimes known as file-slurp mode) and the file is empty, it returns `''` the first time, followed by `undef` subsequently.

Ordinarily you must assign the returned 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` statement (even if disguised as a `for(;;)` loop), the value is automatically assigned to the global variable $_, destroying whatever was there previously. (This may seem like an odd thing to you, but you'll use the construct in almost every Perl script you write.) The $_ variable is not implicitly localized. You'll have to put a `local $_;` before the loop if you want that to happen.

The following lines are equivalent:

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

This also behaves similarly, but assigns to a lexical variable instead of to $_:

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

In these loop constructs, the assigned value (whether assignment is automatic or explicit) is then tested to see whether it is defined. The defined test avoids problems where the line has a string value that would be treated as false by Perl; for example a "" or a "0" with no trailing newline. If you really mean for 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 an explicit `defined` test or comparison elicits a warning if the `use warnings` pragma or the **-w** command-line switch (the `$^W` variable) 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, amongst others. See *perlopentut* and *"open" in perlfunc* for details on this.

If a <FILEHANDLE> is used in a context that is looking for a list, a list comprising all input lines is returned, one line per list element. It's easy to grow to a rather large data space this way, so use with care.

<FILEHANDLE> may also be spelled `readline(*FILEHANDLE)`. See *"readline" in perlfunc*.

The null filehandle <> is special: it can be used to emulate the behavior of **sed** and **awk**, and any other Unix filter program that takes a list of filenames, doing the same to each line of input from all of them. 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 the @ARGV array and put the current filename into the $ARGV variable. 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.)

Since the null filehandle uses the two argument form of *"open" in perlfunc* it interprets special characters, so if you have a script like this:

```
    while (<>) {
        print;
    }
```

and call it with `perl dangerous.pl 'rm -rfv *|'`, it actually opens a pipe, executes the `rm` command and reads `rm`'s output from that pipe. If you want all items in `@ARGV` to be interpreted as file names, you can use the module `ARGV::readonly` from CPAN.

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 though the input were one big happy file. See the example in *"eof" in perlfunc* 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 read input from STDIN.

If what the angle brackets contain is a simple scalar variable (for example, <$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 $hash{key} is not--it's a hash element. Even <$x > (note the extra space) is treated as `glob("$x ")`, not `readline($x)`.

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.) For example:

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

is roughly equivalent to:

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

except that the globbing is actually done internally using the standard `File::Glob` extension. Of course, the shortest way to do the above is:

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

A (file)glob evaluates its (embedded) argument only when it is starting a new list. All values must be read before it will start over. In list context, this isn't important because you automatically get them all anyway. However, in scalar context the operator returns the next value each time it's called, or `undef` when the list has run out. As with filehandle reads, an automatic `defined` is generated when the glob occurs in the test part of a `while`, because legal glob returns (for example, 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 interpolation 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 precomputes the number which that expression represents so that the interpreter won't have to.

No-ops

Perl doesn't officially have a no-op operator, but the bare constants `0` and `1` are special-cased not to produce a warning in void context, so you can for example safely do

```
    1 while foo();
```

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, **|** and **^** ops act as though the shorter operand had additional zero bits on the right, while the **&** op acts as though the longer operand were truncated to the length of the shorter. The granularity for such extension or truncation is one or more bytes.

```
# 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, 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
```

See *"vec" in perlfunc* for information on how to manipulate individual bits in a bit vector.

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 to use integer operations (see *integer* for a detailed explanation) 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. Note that this doesn't mean everything is an integer, merely that Perl will use integer operations for arithmetic, comparison, and bitwise operators. For example, even under `use integer`, if you take the `sqrt(2)`, you'll still get `1.4142135623731` or so.

Used on numbers, the bitwise operators ("&", "|", "^", "~", "<<", and ">>") always produce integral results. (But see also *Bitwise String Operators*.) However, `use integer` still has meaning for them. By default, their results are interpreted as unsigned integers, but 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 two's-complement machines.

Floating-point Arithmetic

While `use integer` provides integer-only arithmetic, there is no analogous mechanism to provide automatic rounding or truncation to a certain number of decimal places. For rounding to a certain number of digits, sprintf() or printf() is usually the easiest route. See *perlfaq4*.

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 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 other mathematical and trigonometric functions. The Math::Complex module (part of the standard perl distribution) defines mathematical functions that work on both the reals and the imaginary 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`, `Math::BigRat`, and `Math::BigFloat` modules, along with the `bignum`, `bigint`, and `bigrat` pragmas, provide variable-precision arithmetic and overloaded operators, although they're currently pretty slow. At the cost of some space and considerable speed, they avoid the normal pitfalls associated with limited-precision representations.

```
use 5.010;
use bigint;  # easy interface to Math::BigInt
$x = 123456789123456789;
say $x * $x;
   +15241578780673678515622620750190521
```

Or with rationals:

```
use 5.010;
use bigrat;
$a = 3/22;
$b = 4/6;
say "a/b is ", $a/$b;
say "a*b is ", $a*$b;
   a/b is 9/44
   a*b is 1/11
```

Several modules let you calculate with (bound only by memory and CPU time) unlimited or fixed precision. There are also some non-standard modules that provide faster implementations via external C libraries.

Here is a short, but incomplete summary:

```
  Math::String           treat string sequences like numbers
  Math::FixedPrecision   calculate with a fixed precision
  Math::Currency         for currency calculations
  Bit::Vector            manipulate bit vectors fast (uses C)
  Math::BigIntFast       Bit::Vector wrapper for big numbers
  Math::Pari             provides access to the Pari C library
  Math::Cephes           uses the external Cephes C library (no
                         big numbers)
  Math::Cephes::Fraction fractions via the Cephes library
  Math::GMP              another one using an external C library
  Math::GMPz             an alternative interface to libgmp's big ints
```

```
    Math::GMPq          an interface to libgmp's fraction numbers
    Math::GMPf          an interface to libgmp's floating point numbers
```

Choose wisely.