Some people call Perl an "operator-oriented language". To understand a Perl program, you must understand how its operators interact with their operands.

A Perl *operator* is a series of one or more symbols used as part of the syntax of a language. Each operator operates on zero or more *operands*. Think of an operator as a special sort of function the parser understands and its operands as arguments.

Operator Characteristics

Every operator possesses several important characteristics which govern its behavior: the number of operands on which it operates, its relationship to other operators, and its syntactic possibilities.

`perldoc perlop` and `perldoc perlsyn` provide voluminous information about Perl's operators, but the documentation assumes you're already familiar with some details of how they work. The essential computer science concepts may sound imposing at first, but once you get past their names, they're straightforward. You already understand them implicitly.

Precedence

The *precedence* of an operator governs when Perl should evaluate it in an expression. Evaluation order proceeds from highest to lowest precedence. Because the precedence of multiplication is higher than the precedence of addition, `7 + 7 * 10` evaluates to `77`, not `140`.

To force the evaluation of some operators before others, group their subexpressions in parentheses. In `(7 + 7) * 10`, grouping the addition into a single unit forces its evaluation before the multiplication. The result is `140`.

`perldoc perlop` contains a table of precedence. Read it, understand it, but don't bother memorizing it (almost no one does). Spend your time keeping your expressions simple, and then add parentheses to clarify your intent.

In cases where two operators have the same precedence, other factors such as associativity (*associativity*) and fixity (*fixity*) break the tie.

Associativity

The *associativity* of an operator governs whether it evaluates from left to right or right to left. Addition is left associative, such that `2 + 3 + 4` evaluates `2 + 3` first, then adds 4 to the result. Exponentiation is right associative, such that `2 ** 3 ** 4` evaluates `3 ** 4` first, then raises 2 to the 81st power.

It's worth your time to memorize the precedence and associativity of the common mathematical operators, but again simplicity rules the day. Use parentheses to make your intentions clear.

The core `B::Deparse` module is an invaluable debugging tool. Run `perl -MO=Deparse,-p` on a snippet of code to see exactly how Perl handles operator precedence and associativity. The `-p` flag adds extra grouping parentheses which often clarify evaluation order.

Beware that Perl's optimizer will simplify mathematical operations as given as examples earlier in this section; use variables instead, as in `$x ** $y ** $z`.

Arity

The *arity* of an operator is the number of operands on which it operates. A *nullary* operator operates on zero operands. A *unary* operator operates on one operand. A *binary* operator operates on two operands. A *trinary* operator operates on three operands. A *listary* operator operates on a list of operands. An operator's documentation and examples should make its arity clear.

For example, the arithmetic operators are binary operators, and are usually left associative. `2 + 3 - 4` evaluates `2 + 3` first; addition and subtraction have the same precedence, but they're left associative and binary, so the proper evaluation order applies the leftmost operator (+) to the leftmost two operands (2 and 3) with the leftmost operator (+), then applies the rightmost operator (`-`) to the result of the first operation and the rightmost operand (4).

Perl novices often find confusion between the interaction of listary operators--especially function calls--and nested expressions. Where parentheses usually help, beware of the parsing complexity of:

```
# probably buggy code
say ( 1 + 2 + 3 ) * 4;
```

... which prints the value 6 and (probably) evaluates as a whole to 4 (the return value of `say` multiplied by 4). Perl's parser happily interprets the parentheses as postcircumfix (*fixity*) operators denoting the arguments to `say`, not circumfix parentheses grouping an expression to change precedence.

Fixity

An operator's *fixity* is its position relative to its operands:

* *Infix* operators appear between their operands. Most mathematical operators are infix operators, such as the multiplication operator in `$length * $width`.

* *Prefix* operators precede their operators. *Postfix* operators follow. These operators tend to be unary, such as mathematic negation (`-$x`), boolean negation (`!$y`), and postfix increment (`$z++`).

* *Circumfix* operators surround their operands, as with the anonymous hash constructor (`{ ... }`) and quoting operators (`qq[ ... ]`).

* *Postcircumfix* operators follow certain operands and surround others, as seen in hash and array element access (`$hash{$x}` and `$array[$y]`).

Operator Types

Perl operators provide value contexts (*value_contexts*) to their operands. To choose the appropriate operator, understand the values of the operands you provide as well as the value you expect to receive.

Numeric Operators

Numeric operators impose numeric contexts on their operands. These operators are the standard arithmetic operators such as addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), exponentiation (`**`), modulo (`%`), their in-place variants (`+=`, `-=`, `*=`, `/=`, `**=`, and `%=`), and both postfix and prefix auto-decrement (`--`).

The auto-increment operator has special string behavior (*auto_increment_operator*).

Several comparison operators impose numeric contexts upon their operands. These are numeric equality (==), numeric inequality (`!=`), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), and the sort comparison operator (<=>).

String Operators

String operators impose string contexts on their operands. These operators are positive and negative regular expression binding (=~ and `!~`, respectively), and concatenation (`.`).

Several comparison operators impose string contexts upon their operands. These are string equality (`eq`), string inequality (`ne`), greater than (`gt`), less than (`lt`), greater than or equal to (`ge`), less than or equal to (`le`), and the string sort comparison operator (`cmp`).

Logical Operators

Logical operators impose a boolean context on their operands. These operators are &&, `and`, `||`, and `or`. All are infix and all exhibit *short-circuiting* behavior (*short_circuiting*). The word variants have lower precedence than their punctuation forms.

The defined-or operator, `//`, tests the *definedness* of its operand. Unlike `||` which tests the *truth* of its operand, `//` evaluates to a true value even if its operand evaluates to a numeric zero or the empty string. This is especially useful for setting default parameter values:

```
sub name_pet
{
    my $name = shift // 'Fluffy';
    ...
}
```

The ternary conditional operator (`?:`) takes three operands. It evaluates the first in boolean context and evaluates to the second if the first is true and the third otherwise:

```
my $truthiness = $value ? 'true' : 'false';
```

The prefix `!` and `not` operators return the logical opposite of the boolean value of their operands. `not` is a lower precedence version of `!`.

The `xor` operator is an infix operator which evaluates to the exclusive-or of its operands.

Bitwise Operators

Bitwise operators treat their operands numerically at the bit level. These operations are uncommon. They consist of left shift (<<), right shift (>>), bitwise and (&), bitwise or (`|`), and bitwise xor (^), as well as their in-place variants (<<=, >>=, &=, `|=`, and ^=).

The auto-increment operator has special behavior. When used on a value with a numeric component (*cached_coercions*), the operator increments that numeric component. If the value is obviously a string (and has no numeric component), the operator increments that string value such that a becomes b, zz becomes aaa, and a9 becomes b0.

```
my $num = 1;
my $str = 'a';

$num++;
$str++;
is( $num,   2, 'numeric autoincrement' );
is( $str, 'b', 'string autoincrement'  );

no warnings 'numeric';
$num += $str;
$str++;

is( $num, 2, 'numeric addition with $str'    );
is( $str, 1, '... gives $str a numeric part' );
```

The repetition operator (x) is an infix operator with complex behavior. In list context, when given a list, it evaluates to that list repeated the number of times specified by its second operand. In list context when given a scalar, it produces a string consisting of the string value of its first operand concatenated to itself the number of times specified by its second operand.

In scalar context, the operator always produces a concatenated string repeated appropriately. For example:

```
my @scheherazade = ('nights') x 1001;
my $calendar     =  'nights'  x 1001;
my $cal_length   =  length $calendar;

is( @scheherazade, 1001, 'list repeated' );
is( $cal_length,   1001 * length 'nights',
                   'word repeated' );

my @schenolist   =  'nights'  x 1001;
my $calscalar    = ('nights') x 1001;

is( @schenolist, 1, 'no lvalue list' );
is( length $calscalar,
    1001 * length 'nights', 'word still repeated' );
```

The infix *range* operator (`..`) produces a list of items in list context:

```
my @cards = ( 2 .. 10, 'J', 'Q', 'K', 'A' );
```

It can produce simple, incrementing ranges (both as integers or strings), but it cannot intuit patterns or more complex ranges.

In boolean context, the range operator becomes the *flip-flop* operator. This operator produces a false value until its left operand is true. That value stays true until the right operand is true, after which the value is false again until the left operand is true again. Imagine parsing the text of a formal letter with:

```
while (/Hello, $user/ .. /Sincerely,/)
{
    say "> $_";
}
```

The *comma* operator (`,`) is an infix operator. In scalar context it evaluates its left operand then returns the value produced by evaluating its right operand. In list context, it evaluates both operands in left-to-right order.

The fat comma operator (=>) also automatically quotes any bareword used as its left operand (*hashes*).