Some people call Perl an "operator-oriented language". 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.

You've seen how Perl manages context through its operators. To understand Perl fully, you must understand how operators interact with their operands.

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, the contexts it enforces, and the syntax it provides.

`perldoc perlop` and `perldoc perlsyn` provide voluminous information about Perl's operators, but the docs assume you're already familiar with a few essential computer science concepts. Fortunately, you'll recognize these ideas from written language and elementary mathematics, even if you've never heard their complicated names before.

Precedence

The *precedence* of an operator governs when Perl should evaluate it in an expression. Perl evaluates the operator with the highest precedence first, then the next highest, all the way to the lowest precedence. Remember basic math? Multiply and divide before you add and subtract. That's precedence. Because the precedence of multiplication is higher than the precedence of addition, in Perl `7 + 7 * 10` evaluates to `77`, not `140`.

Use grouping parentheses to force the evaluation of some operators before others. In `(7 + 7) * 10`, grouping the addition into a single unit forces its evaluation before the multiplication--though Perl wants to perform the multiplication first, it has to evaluate the grouped subexpression into a single value as the multiplication operator's left operand. The result is `140`.

`perldoc perlop` contains a table of precedence. Skim it a few times, but don't bother memorizing it (almost no one does). Spend your time simplifying your code where you can. Then add parentheses where they clarify.

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, not that order of evaluation matters. Exponentiation is right associative, such that `2 ** 3 ** 4` evaluates `3 ** 4` first, then raises 2 to the 81st power. Use parentheses if you write code like this.

If you memorize only the precedence and associativity of the common mathematical operators, you'll be fine. Simplify your code and you won't have to memorize other associativities. If you can't simplify your code (or if you're maintaining code and trying to understand it), use the core `B::Deparse` module to see exactly how Perl handles operator precedence and associativity.

Run `perl -MO=Deparse,-p` on a snippet of code The `-p` flag adds extra grouping parentheses which often clarify evaluation order. Beware that Perl's optimizer will simplify mathematical operations using constant values. If you really need to deparse a complex expression, use named variables instead of constant values, 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 zero or more operands.

Arithmetic operators are binary operators and are usually left associative. This has implications for tie-breaking evaluation order of operands with the same precedence. For example, `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).

Fixity

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 operators denoting the arguments to `say`, not circumfix parentheses grouping an expression to change precedence.

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 operands. *Postfix* operators follow their operands. 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's operators provide value contexts (*value_contexts*) to their operands. To choose the appropriate operator, you must know 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 of addition (+), subtraction (-), multiplication (*), division (/), exponentiation (**), and 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. These infix operators all exhibit *short-circuiting* behavior (*short_circuiting*). The word forms have lower precedence than the 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 opposites of the boolean values 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 uncommon operators are left shift (<<), right shift (>>), bitwise and (&), bitwise or (|), and bitwise xor (^), as well as their in-place variants (<<=, >>=, &=, |=, and ^=).

Special Operators

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 (if it has no numeric component), the operator increments the value's string component 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. When evaluated in list context *with a list as its first operand*, it evaluates to that list repeated the number of times specified by its second operand. When evaluated in list context *with a scalar as its first operand*, 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 repeats and concatenates a string:

```
    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 *only* produce simple, incrementing ranges of integers or strings.

In boolean context, the range operator performs a *flip-flop* operation. 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 quotes any bareword used as its left operand (*hashes*).

The *triple-dot* or *whatever* operator stands in for a single statement. It is nullary and has neither precedence nor associativity. It parses, but when executed it throws an exception with the string `Unimplemented`. This makes a great placeholder in example code you don't expect anyone to execute:

```
    sub some_example {
        # implement this yourself
        ...
    }
```