Operators

An accurate, if irreverent, description of Perl is an "operator-oriented language". The interaction of operators with their operands gives Perl its expressivity and power. Understanding Perl requires understanding its operators and how they behave. For the sake of this discussion, a working definition of 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*; this definition is circular, as an operand is a value on which an operator operates.

The most accurate definition of operators is "What's in `perlop`", but even that leaves out some operators in `perlsyn` and includes built-in functions. Don't get too attached to a single definition.

Operator Characteristics

Both `perldoc perlop` and `perldoc perlsyn` provide voluminous information about the behavior of Perl's operators. Even so, what they *don't* explain is more important to their understanding. The documentation assumes you have a familiarity with several concepts in language design. These concepts may sound imposing at first, but they're straightforward to understand.

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.

Precedence

The *precedence* of an operator helps determine when Perl should evaluate it in an expression. Evaluation order proceeds from highest to lowest precedence. For example, because the precedence of multiplication is higher than the precedence of addition, `7 + 7 * 10` evaluates to `77`, not `140`. You may force the evaluation of some operators before others by grouping their subexpressions in parentheses; `(7 + 7) * 10` *does* evaluate to `140`, as the addition operation becomes a single unit which must evaluate fully before multiplication can occur.

In case of a tie--where two operators have the same precedence--other factors (*fixity* and *associativity*) break the tie.

`perldoc perlop` contains a table of precedence. Almost no one has this table memorized. The best way to manage precedence is to keep your expressions simple. The second best way is to use parentheses to clarify precedence in complex expressions. If you find yourself drowning in a sea of parentheses, see the first rule again.

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` performs `3 ** 4` first, then raises 2 to the 81st power.

Simplifying complex expressions and using parentheses to demonstrate your intent is more important than memorizing associativity tables. Even so, memorizing the associativity of the mathematic operators is worthwhile.

The core `B::Deparse` module can rewrite snippets of code to demonstrate 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 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.

There's no single good rule for determining the arity of an operator, other than the fact that most operate on two, many, or one operands. The operator's documentation should make this 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).

One common source of confusion for Perl novices is the interaction of listary operators (especially function calls) with nested expressions. Using grouping parentheses to clarify your intent, yet watch out for confusion in code such as:

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

... as Perl 5 happily interprets the parentheses as postcircumfix (*fixity*) operators denoting the arguments to `say`, not circumfix parentheses grouping an expression to change precedence. In other words, the code prints the value 6 and evaluates to the return value of `say` multiplied by 4.

Fixity

The *fixity* of an operator is its position relative to its operands. The mathematic operators tend to be *infix* operators, where they appear between their operands. Other operators are *prefix*, where they appear before their operands; these tend to be unary operators, such as the prefix increment operator `++$x` or the mathematical and boolean negation operators (`-$x` and `!$x`, respectively). *Postfix* operators appear after their operands (such as postfix increment `$x++`). *Circumfix* operators surround their operands, such as the anonymous hash and anonymous array creation operators or quoting operators (`{ ... }` and `[ ... ]` or `qq{ ... }`, for example). *Postcircumfix* operators surround some operands but follow others, as in the case of array or hash indices (`$hash{ ... }` and `$array[ ... ]`, for example).

Operator Types

Perl's pervasive contexts--especially value contexts (*value_contexts*)--extend to the behavior of its operators. Perl operators provide value contexts to their operands. Choosing the most appropriate operator for a given situation requires you to understand what type of value you expect to receive as well as the type of values on which you wish to operate.

Numeric Operators

The numeric operators enforce numeric contexts on their operands. They consist of the standard arithmetic operators such as addition (+), subtraction (`-`), multiplication (`*`), division (`/`), exponentiation (`**`), modulo (%), their in-place variants (+=, `-=`, `*=`, `/=`, `**=`, and %=), and auto-decrement (`--`), whether postfix or prefix.

While the auto-increment operator may seem like a numeric operator, it has special string behavior (*auto_increment_operator*).

Several comparison operators enforce 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

The string operators enforce string contexts on their operands. They consist of the positive and negative regular expression binding operators (=~ and `!~`, respectively), and the concatenation operator (`.`).

Several comparison operators enforce 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

The logical operators treat their operands in a boolean context. The `&&` and `and` operators test that both expressions are logically true, while the `||` and `or` operators test that either expression is true. All four are infix operators. All four perform *short-circuiting* behavior: if the evaluation of one expression will make the entire expression false, Perl will not evaluate the other expression. The word forms of these operators have lower precedence than the symbolic forms.

The defined-or operator, `//`, tests the *definedness* of its operand. Unlike `||` which tests the truthiness of its operand, `//` evaluates to a true value if its operand evalutes 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 `!` and `not` operators return the logical opposite of the boolean value of their operands. `not` has a lower precedence than `!`. These are prefix operators.

The `xor` operator is an infix operator which performs the exclusive or of its operands.

Bitwise Operators

The bitwise operators treat their operands numerically at the bit level. These are uncommon in most Perl 5 programs. They consist of left shift (<<), right shift (>>), bitwise and (&), bitwise or (|), and binary or (^), as well as their in-place variants (&=, |=, ^=, <<=, and >>=).

Special Operators

The auto-increment operator has a special case. If anything has ever used a variable in a numeric context (*cached_coercions*), it increments the numeric value of that variable. If the variable is obviously a string (and has never been evaluated in a numeric context), the string value increments with a carry, such that a increments to `b`, `zz` to `aaa`, and `a9` to `b0`.

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

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

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

    is( $num, 2, 'adding $str to $num should add numeric value of $str' );
    is( $str, 1, '... but $str should now autoincrement its numeric part'
);
```

Is there a better way to explain this? I may have confused myself writing it.

In list context, the repetition operator (`x`) changes its behavior based on its first operand. When given a list, it evaluates to that list repeated the number of times specified by its second operand. When given a scalar, it produces a string consisisting 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.

This is an infix operator:

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

    is( @scheherazade,                     1001, 'list repeated' );
    is( length $calendar, 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' );
```

Add readline, comma, range, flip-flop, and `...`.