```
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*.

28.1.2 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 §28.1.30.