

Top Down Operator Precedence

Douglas Crockford

IN 1973, VAUGHAN PRATT PRESENTED “TOP DOWN OPERATOR PRECEDENCE”^{*} at the first annual Principles of Programming Languages Symposium in Boston. In the paper, Pratt described a parsing technique that combines the best properties of Recursive Descent and the Operator Precedence syntax technique of Robert W Floyd.[†] He claimed that the technique is simple to understand, trivial to implement, easy to use, extremely efficient, and very flexible. I will add that it is also beautiful.

It might seem odd that such an obviously utopian approach to compiler construction is completely neglected today. Why is this the case? Pratt suggested in the paper that preoccupation with BNF grammars and their various offspring, along with their related automata and theorems, have precluded development in directions that are not visibly in the domain of automata theory.

^{*} Pratt’s paper is available at <http://portal.acm.org/citation.cfm?id=512931>; more information about Pratt himself can be found at <http://boole.stanford.edu/pratt.html>.

[†] For a description of Floyd, see “Robert W Floyd, In Memoriam,” Donald E. Knuth, <http://sigact.acm.org/floyd>.

Another explanation is that his technique is most effective when used in a dynamic, functional programming language. Its use in a static, procedural language would be considerably more difficult. In his paper, Pratt used LISP and almost effortlessly built parse trees from streams of tokens.

But parsing techniques are not greatly valued in the LISP community, which celebrates the Spartan denial of syntax. There have been many attempts since LISP's creation to give the language a rich, ALGOL-like syntax, including:

Pratt's CGOL

<http://zane.brouhaha.com/~healyzh/doc/cgol.doc.txt>

LISP 2

http://community.computerhistory.org/scc/projects/LISP/index.html#LISP_2_

MLISP

<ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/68/92/CS-TR-68-92.pdf>

Dylan

<http://www.opendylan.org>

Interlisp's Clisp

<http://community.computerhistory.org/scc/projects/LISP/interlisp/Teitelman-3IJCAI.pdf>

McCarthy's original M-expressions

<http://www-formal.stanford.edu/jmc/history/lisp/lisp.html>

All failed to find acceptance. The functional programming community found the correspondence between programs and data to be much more valuable than expressive syntax. But the mainstream programming community likes its syntax, so LISP has never been accepted by the mainstream. Pratt's technique befits a dynamic language, but dynamic language communities historically have had no use for the syntax that Pratt's technique realizes.

JavaScript

The situation changed with the advent of JavaScript. JavaScript is a dynamic, functional language, but in a syntactic sense, it is obviously a member of the C family. JavaScript is a dynamic language with a community that likes syntax. In addition, JavaScript is object oriented. Pratt's 1973 paper anticipated object orientation but lacked an expressive notation for it. JavaScript has an expressive object notation. Thus, JavaScript is an ideal language for exploiting Pratt's technique. I will show that we can quickly and inexpensively produce parsers in JavaScript.

We don't have time in this short chapter to deal with the whole JavaScript language, and perhaps we wouldn't want to because the language is a mess. But it has some brilliant stuff in it, which is worthy of your consideration. We will build a parser that can process Simplified JavaScript, and we will write that parser in Simplified JavaScript. Simplified JavaScript is just the good stuff, including:

Functions as first-class objects

Functions are lambdas with lexical scoping.

Dynamic objects with prototypal inheritance

Objects are class-free. We can add a new member to any object by ordinary assignment.

An object can inherit members from another object.

Object literals and array literals.

This is a very convenient notation for creating new objects and arrays. JavaScript literals were the inspiration for the JSON data interchange (<http://www.JSON.org>) format.

We will take advantage of JavaScript's prototypal nature to make token objects that inherit from symbols, and symbols that inherit from an original symbol. We will depend on the object function, which makes a new object that inherits members from an existing object. Our implementation will also depend on a tokenizer that produces an array of simple token objects from a string. We will advance through the array of tokens as we weave our parse tree.

Symbol Table

We will use a symbol table to drive our parser:

```
var symbol_table = {};
```

The `original_symbol` object will be the prototype for all other symbols. It contains methods that report errors. These will usually be overridden with more useful methods:

```
var original_symbol = {
  nud: function () {
    this.error("Undefined.");
  },
  led: function (left) {
    this.error("Missing operator.");
  }
};
```

Let's define a function that defines symbols. It takes a symbol id and an optional binding power that defaults to zero. It returns a symbol object for that id. If the symbol already exists in the `symbol_table`, it returns that symbol object. Otherwise, it makes a new symbol object that inherits from `original_symbol`, stores it in the symbol table, and returns it. A symbol object initially contains an id, a value, a left binding power, and the stuff it inherits from the `original_symbol`:

```
var symbol = function (id, bp) {
  var s = symbol_table[id];
  bp = bp || 0;
  if (s) {
    if (bp >= s.lbp) {
      s.lbp = bp;
    }
  }
}
```

```

    } else {
        s = object(original_symbol);
        s.id = s.value = id;
        s.lbp = bp;
        symbol_table[id] = s;
    }
    return z;
};

```

The following symbols are popular separators and closers:

```

symbol(":");
symbol(";");
symbol(",");
symbol(")");
symbol("]");
symbol("}");
symbol("else");

```

The (end) symbol indicates that there are no more tokens. The (name) symbol is the prototype for new names, such as variable names. They are spelled strangely to avoid collisions:

```

symbol("(end)");
symbol("(name)");

```

The (literal) symbol is the prototype for all string and number literals:

```

var itself = function () {
    return this;
};
symbol("(literal)").nud = itself;

```

The this symbol is a special variable. In a method invocation, it is the reference to the object:

```

symbol("this").nud = function () {
    scope.reserve(this);
    this.arity = "this";
    return this;
};

```

Tokens

We assume that the source text has been transformed into an array of simple token objects (tokens), each containing a type member that is a string ("name", "string", "number", "operator") and a value member that is a string or number.

The token variable always contains the current token:

```

var token;

```

The advance function makes a new token object and assigns it to the token variable. It takes an optional id parameter, which it can check against the id of the previous token. The new token object's prototype will be a name token in the current scope or a symbol from the symbol table. The new token's arity will be "name", "literal", or "operator". Its arity

may be changed later to "binary", "unary", or "statement" when we know more about the token's role in the program:

```
var advance = function (id) {
  var a, o, t, v;
  if (id && token.id !== id) {
    token.error("Expected '" + id + "'.");
  }
  if (token_nr >= tokens.length) {
    token = symbol_table["(end)"];
    return;
  }
  t = tokens[token_nr];
  token_nr += 1;
  v = t.value;
  a = t.type;
  if (a === "name") {
    o = scope.find(v);
  } else if (a === "operator") {
    o = symbol_table[v];
    if (!o) {
      t.error("Unknown operator.");
    }
  } else if (a === "string" || a === "number") {
    a = "literal";
    o = symbol_table["(literal)"];
  } else {
    t.error("Unexpected token.");
  }
  token = object(o);
  token.value = v;
  token.arity = a;
  return token;
};
```

Precedence



Tokens are objects that bear methods that allow them to make precedence decisions, match other tokens, and build trees (and in a more ambitious project also check types, optimize, and generate code). The basic precedence problem is this: given an operand between two operators, is the operand bound to the left operator or the right? Thus, if A and B are operators in:

d A e B f

does operand e bind to A or to B? In other words, are we talking about:

(d A e) B f

or:

d A (e B f)

Ultimately, the complexity in the process of parsing comes down to the resolution of this ambiguity. The technique we will develop here uses token objects whose members include binding powers (or precedence levels), and simple methods called *nud* (null denotation) and *led* (left denotation). A *nud* does not care about the tokens to the left. A *led* does. A *nud* method is used by values (such as variables and literals) and by prefix operators. A *led* method is used by infix operators and suffix operators. A token may have both a *nud* and a *led*. For example, `-` might be both a prefix operator (negation) and an infix operator (subtraction), so it would have both *nud* and *led* methods.

Expressions

The heart of Pratt's technique is the expression function. It takes a right binding power that controls the aggressiveness of its consumption of tokens that it sees to the right. It returns the result of calling methods on the tokens it acts upon:

```
var expression = function (rbp) {
  var left;
  var t = token;
  advance();
  left = t.nud();
  while (rbp < token.lbp) {
    t = token;
    advance();
    left = t.led(left);
  }
  return left;
}
```

`expression` calls the *nud* method of the token. The *nud* is used to process literals, variables, and prefix operators. After that, as long as the right binding power is less than the left binding power of the next token, the *led* methods are invoked. The *led* is used to process infix and suffix operators. This process can be recursive because the *nud* and *led* methods can call `expression`.

Infix Operators

The `+` operator is an infix operator, so it will have a *led* method that makes the token object into a tree, where the two branches are the operand to the left of the `+` and the operand to the right. The left operand is passed into the *led*, and the right is obtained by calling the `expression` method.

The number 60 is the binding power of `+`. Operators that bind tighter or have higher precedence have greater binding powers. In the course of mutating the stream of tokens into a parse tree, we will use the operator tokens as containers of operand nodes:

```
symbol("+", 60).led = function (left) {
  this.first = left;
  this.second = expression(60);
  this.arity = "binary";
  return this;
};
```

When we define the symbol for *, we see that only the id and binding powers are different. It has a higher binding power because it binds more tightly:

```
symbol("*", 70).led = function (left) {
  this.first = left;
  this.second = expression(70);
  this.arity = "binary";
  return this;
};
```

Not all infix operators will be this similar, but many will, so we can make our work easier by defining an infix function that will help us specify infix operators. The infix function will take an id and a binding power, and optionally a led function. If a led function is not provided, it will supply a default led that is useful in most cases:

```
var infix = function (id, bp, led) {
  var s = symbol(id, bp);
  s.led = led || function (left) {
    this.first = left;
    this.second = expression(bp);
    this.arity = "binary";
    return this;
  };
  return s;
}
```

This allows a more declarative style for specifying operators:

```
infix("+", 60);
infix("-", 60);
infix("*", 70);
infix("/", 70);
```

The string === is JavaScript's exact-equality comparison operator:

```
infix("===", 50);
infix("!==", 50);
infix("<", 50);
infix("<=", 50);
infix(">", 50);
infix(">=", 50);
```

The ternary operator takes three expressions, separated by ? and :. It is not an ordinary infix operator, so we need to supply its led function:

```
infix(">", 20, function (left) {
  this.first = left;
  this.second = expression(0);
  advance(":");
  this.third = expression(0);
  this.arity = "ternary";
  return this;
});
```

The `.` operator is used to select a member of an object. The token on the right must be a name, but it will be used as a literal:

```
infix(".", 90, function (left) {
  this.first = left;
  if (token.arity !== "name") {
    token.error("Expected a property name.");
  }
  token.arity = "literal";
  this.second = token;
  this.arity = "binary";
  advance();
  return this;
});
```

a.X
↑

The `[` operator is used to dynamically select a member from an object or array. The expression on the right must be followed by a closing `]`:

```
infix("[", 90, function (left) {
  this.first = left;
  this.second = expression(0);
  this.arity = "binary";
  advance("]");
  return this;
});
```

Those infix operators are left associative. We can also make right associative operators, such as short-circuiting logical operators, by reducing the right binding power.

```
var infixr = function (id, bp, led) {
  var s = symbol(id, bp);
  s.led = led || function (left) {
    this.first = left;
    this.second = expression(bp - 1);
    this.arity = "binary";
    return this;
  };
  return s;
};
```

bp
0 bp

The `&&` operator returns the first operand if the first operand is falsy. Otherwise, it returns the second operand. The `||` operator returns the first operand if the first operand is truthy; otherwise, it returns the second operand.

```
infixr("&&", 40);
infixr("||", 40);
```

0 ||

Prefix Operators

We can do a similar thing for prefix operators. Prefix operators are right associative. A prefix does not have a left binding power because it does not bind to the left. Prefix operators can sometimes be reserved words (reserved words are discussed in the section “Scope,” later in this chapter):


```

var prefix = function (id, nud) {
  var s = symbol(id);
  s.nud = nud || function () {
    scope.reserve(this);
    this.first = expression(80);
    this.arity = "unary";
    return this;
  };
  return s;
}
prefix("-");
prefix("!");
prefix("typeof");

```

= prefix precedence

The nud of (will call advance("(") to match a balancing) token. The (token does not become part of the parse tree because the nud returns the expression:

```

prefix("(", function () {
  var e = expression(0);
  advance("(");
  return e;
});

```

Assignment Operators

We could use infixr to define our assignment operators, but we want to do two extra bits of business, so we will make a specialized assignment function. It will examine the left operand to make sure that it is a proper lvalue. We will also set an assignment flag so that we can later quickly identify assignment statements:

```

var assignment = function (id) {
  return infixr(id, 10, function (left) {
    if (left.id !== "." && left.id !== "[" &&
        left.arity !== "name") {
      left.error("Bad lvalue.");
    }
    this.first = left;
    this.second = expression(9);
    this.assignment = true;
    this.arity = "binary";
    return this;
  });
};
assignment("=");
assignment("+=");
assignment("-=");

```

right associative

= 10-1

Notice that we have a sort of inheritance pattern, where assignment returns the result of calling infixr, and infixr returns the result of calling symbol.

Constants

The constant function builds constants into the language. The nud mutates a name token into a literal token:

```
var constant = function (s, v) {
  var x = symbol(s);
  x.nud = function () {
    scope.reserve(this);
    this.value = symbol_table[this.id].value;
    this.arity = "literal";
    return this;
  };
  x.value = v;
  return x;
};
constant("true", true);
constant("false", false);
constant("null", null);
constant("pi", 3.141592653589793);
```

Scope

We use functions such as infix and prefix to define the symbols used in the language. Most languages have some notation for defining new symbols, such as variable names. In a very simple language, when we encounter a new word, we might give it a definition and put it in the symbol table. In a more sophisticated language, we would want a notion of scope, giving the programmer convenient control over the lifespan and visibility of a variable.

A *scope* is a region of a program in which a variable is defined and accessible. Scopes can be nested inside of other scopes. Variables defined in a scope are not visible outside of the scope.

We will keep the current scope object in the scope variable:

```
var scope;
```

original_scope is the prototype for all scope objects. It contains a define method that is used to define new variables in the scope. The define method transforms a name token into a variable token. It produces an error if the variable has already been defined in the scope or if the name has already been used as a reserved word:

```
var original_scope = {
  define: function (n) {
    var t = this.def[n.value];
    if (typeof t === "object") {
      n.error(t.reserved ?
        "Already reserved." :
        "Already defined.");
    }
  }
}
```

```

        this.def[n.value] = n;
        n.reserved = false;
        n.nud      = itself;
        n.led      = null;
        n.std      = null;
        n.lbp      = 0;
        n.scope    = scope;
        return n;
    },

```

The find method is used to find the definition of a name. It starts with the current scope and will go, if necessary, back through the chain of parent scopes and ultimately to the symbol table. It returns `symbol_table["(name)"]` if it cannot find a definition:

```

    find: function (n) {
        var e = this;
        while (true) {
            var o = e.def[n];
            if (o) {
                return o;
            }
            e = e.parent;
            if (!e) {
                return symbol_table[
                    symbol_table.hasOwnProperty(n) ?
                    n : "(name)"];
            }
        }
    },

```

The pop method closes a scope:

```

    pop: function () {
        scope = this.parent;
    },

```

The reserve method is used to indicate that a name has been used as a reserved word in the current scope:

```

    reserve: function (n) {
        if (n.arity !== "name" || n.reserved) {
            return;
        }
        var t = this.def[n.value];
        if (t) {
            if (t.reserved) {
                return;
            }
            if (t.arity === "name") {
                n.error("Already defined.");
            }
        }
        this.def[n.value] = n;
        n.reserved = true;
    }
};

```

We need a policy for reserved words. In some languages, words that are used structurally (such as `if`) are reserved and cannot be used as variable names. The flexibility of our parser allows us to have a more useful policy. For example, we can say that in any function, any name may be used as a structure word or as a variable, but not as both. Also, we will reserve words locally only after they are used as reserved words. This makes things better for the language designer because adding new structure words to the language will not break existing programs, and it makes things better for programmers because they are not hampered by irrelevant restrictions on the use of names.

Whenever we want to establish a new scope for a function or a block, we call the `new_scope` function, which makes a new instance of the original scope prototype:

```
var new_scope = function () {
  var s = scope;
  scope = object(original_scope);
  scope.def = {};
  scope.parent = s;
  return scope;
};
```

Statements

Pratt's original formulation worked with functional languages in which everything is an expression. Most mainstream languages have statements. We can easily handle statements by adding another method to tokens, `std` (statement denotation). An `std` is like a `nud` except that it is only used at the beginning of a statement.

The `statement` function parses one statement. If the current token has an `std` method, the token is reserved and the `std` is invoked. Otherwise, we assume an expression statement terminated with a `;`. For reliability, we reject an expression statement that is not an assignment or invocation:

```
var statement = function () {
  var n = token, v;
  if (n.std) {
    advance();
    scope.reserve(n);
    return n.std();
  }
  v = expression(0);
  if (!v.assignment && v.id !== "(") {
    v.error("Bad expression statement.");
  }
  advance(";");
  return v;
};
```

特殊语句

表达式语句

The `statements` function parses statements until it sees `(end)` or `}` signaling the end of a block. It returns a statement, an array of statements, or (if there were no statements present) simply `null`:

```

var statements = function () {
  var a = [], s;
  while (true) {
    if (token.id === "}" || token.id === "(end)") {
      break;
    }
    s = statement();
    if (s) {
      a.push(s);
    }
  }
  return a.length === 0 ? null : a.length === 1 ? a[0] : a;
};

```

The stmt function is used to add statements to the symbol table. It takes a statement id and an std function:

```

var stmt = function (s, f) {
  var x = symbol(s);
  x.std = f;
  return x;
};

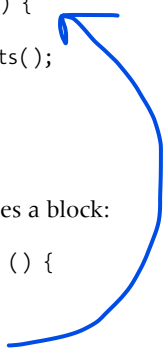
```

The block statement wraps a pair of curly braces around a list of statements, giving them a new scope:

```

stmt("{", function () {
  new_scope();
  var a = statements();
  advance("}");
  scope.pop();
  return a;
});

```



The block function parses a block:

```

var block = function () {
  var t = token;
  advance("{");
  return t.std();
};

```

The var statement defines one or more variables in the current block. Each name can optionally be followed by = and an expression:

```

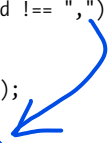
stmt("var", function () {
  var a = [], n, t;
  while (true) {
    n = token;
    if (n.arity !== "name") {
      n.error("Expected a new variable name.");
    }
    scope.define(n);
    advance();
    if (token.id === "=") {

```

```

        t = token;
        advance("=");
        t.first = n;
        t.second = expression(0);
        t.arity = "binary";
        a.push(t);
    }
    if (token.id !== ",") {
        break;
    }
    advance(",");
}
advance(";");
return a.length === 0 ? null : a.length === 1 ? a[0] : a;
});

```



The while statement defines a loop. It contains an expression in parentheses and a block:

```

stmt("while", function () {
    advance("(");
    this.first = expression(0);
    advance(")");
    this.second = block();
    this.arity = "statement";
    return this;
});

```

The if statement allows for conditional execution. If we see the else symbol after the block, we parse the next block or if statement:

```

stmt("if", function () {
    advance("(");
    this.first = expression(0);
    advance(")");
    this.second = block();
    if (token.id === "else") {
        scope.reserve(token);
        advance("else");
        this.third = token.id === "if" ? statement() : block();
    }
    this.arity = "statement";
    return this;
});

```

The break statement is used to break out of loops. We make sure that the next symbol is }:

```

stmt("break", function () {
    advance(";");
    if (token.id !== "}") {
        token.error("Unreachable statement.");
    }
    this.arity = "statement";
    return this;
});

```

The return statement is used to return from functions. It can return an optional expression:

```
stmt("return", function () {
  if (token.id !== ";") {
    this.first = expression(0);
  }
  advance(";");
  if (token.id !== "}") {
    token.error("Unreachable statement.");
  }
  this.arity = "statement";
  return this;
});
```

Functions

Functions are executable object values. A function has an optional name (so that it can call itself recursively), a list of parameter names wrapped in parentheses, and a body that is a list of statements wrapped in curly braces. A function has its own scope:

```
prefix("function", function () {
  var a = [];
  scope = new_scope();
  if (token.arity === "name") {
    scope.define(token);
    this.name = token.value;
    advance();
  }
  advance("(");
  if (token.id !== ")") {
    while (true) {
      if (token.arity !== "name") {
        token.error("Expected a parameter name.");
      }
      scope.define(token);
      a.push(token);
      advance();
      if (token.id !== ",") {
        break;
      }
      advance(",");
    }
  }
  this.first = a;
  advance(")");
  advance("{");
  this.second = statements();
  advance("}");
  this.arity = "function";
  scope.pop();
  return this;
});
```

Functions are invoked with the (operator. It can take zero or more comma-separated arguments. We look at the left operand to detect expressions that cannot possibly be function values:

```
infix("(", 90, function (left) {
  var a = [];
  this.first = left;
  this.second = a;
  this.arity = "binary";
  if ((left.arity !== "unary" ||
      left.id !== "function") &&
      left.arity !== "name" &&
      (left.arity !== "binary" ||
       (left.id !== "." &&
        left.id !== "(" &&
        left.id !== "["))) {
    left.error("Expected a variable name.");
  }
  if (token.id !== ")") {
    while (true) {
      a.push(expression(0));
      if (token.id !== ",") {
        break;
      }
      advance(",");
    }
  }
  advance(")");
  return this;
});
```

Array and Object Literals

An array literal is a set of square brackets around zero or more comma-separated expressions. Each of the expressions is evaluated, and the results are collected into a new array:

```
prefix("[", function () {
  var a = [];
  if (token.id !== "]") {
    while (true) {
      a.push(expression(0));
      if (token.id !== ",") {
        break;
      }
      advance(",");
    }
  }
  advance("]");
  this.first = a;
  this.arity = "unary";
  return this;
});
```

a = [1, 2, 3, 4, 5...]
↑
prefix

An object literal is a set of curly braces around zero or more comma-separated pairs. A *pair* is a key/expression pair separated by a `:`. The key is a literal or a name treated as a literal:

```
prefix("{", function () {
  var a = [];
  if (token.id !== "}") {
    while (true) {
      var n = token;
      if (n.arity !== "name" && n.arity !== "literal") {
        token.error("Bad key.");
      }
      advance();
      advance(":");
      var v = expression(0);
      v.key = n.value;
      a.push(v);
      if (token.id !== ",") {
        break;
      }
      advance(",");
    }
  }
  advance("}");
  this.first = a;
  this.arity = "unary";
  return this;
});
```

$= \{ k_1 : v_1, k_2 : v_2, \dots \}$

Things to Do and Think About

The simple parser shown in this chapter is easily extensible. The tree could be passed to a code generator, or it could be passed to an interpreter. Very little computation is required to produce the tree. And as we saw, very little effort was required to write the programming that built the tree.

We could make the infix function take an opcode that would aid in code generation. We could also have it take additional methods that would be used to do constant folding and code generation.

We could add additional statements, such as `for`, `switch`, and `try`. We could add statement labels. We could add more error checking and error recovery. We could add lots more operators. We could add type specification and inference.

We could make our language extensible. With the same ease that we can define new variables, we can let the programmer add new operators and new statements.

You can try the demonstration of the parser that was described in this chapter at <http://javascript.crockford.com/tdop/index.html>.

Another example of this parsing technique can be found in JSLint at <http://JSLint.com>.