Top Down Operator Precedence

Douglas Crockford

2007-02-21

This is chapter 9 of Beautiful Code.

Introduction

Vaughan Pratt presented "Top Down Operator Precedence" at the first annual Principles of Programming Languages Symposium in Boston in 1973. In the paper Pratt described a parsing technique that combines the best properties of Recursive Descent and Floyd's Operator Precedence. It is easy to use. It feels a lot like Recursive Descent, but with the need for less code and with significantly better performance. He claimed the technique is simple to understand, trivial to implement, easy to use, extremely efficient, and very flexible. It is dynamic, providing support for truly extensible languages.

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

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 the 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, LISP 2, MLISP, Dylan, Interlisp's Clisp, and McCarthy's original M-expressions. All failed to find acceptance. That 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 wants a dynamic language, but dynamic language communities historically have had no use for the syntax that Pratt's technique conveniently realizes.

JavaScript

The situation changes with the advent of JavaScript. JavaScript is a dynamic, functional language, but syntactically it is obviously a member of the C family. It is a dynamic language with a community that likes syntax.

JavaScript is also object-oriented. Pratt's 1973 paper anticipated object orientation but lacked an expressive notation for it. 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 that is well worth consideration. We will build a parser that can process Simplified JavaScript. We will write the parser in Simplified JavaScript. Simplified JavaScript is just the good stuff, including:

- Functions as first class objects. Functions in Simplified JavaScript 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 format.

We will take advantage of JavaScript's prototypal nature to make token objects that inherit from symbols. Our implementation depends on an `Object.create` method (which makes a new object that inherits members from an existing object) and a tokenizer (which produces an array of simple token objects from a string). We will advance through this array of tokens as we grow our parse tree.

Symbol Table

Every token, such as an operator or identifier, will inherit from a symbol. We will keep all of our symbols (which determine the types of tokens in our language) in a `symbol_table` object.

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

The `original_symbol` object is the prototype for all other symbols. Its methods will usually be overridden. (We will describe the role of `nud` and `led` and binding powers in the section on Precedence below).

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

Let's define a function that makes symbols. It takes a symbol `id` and an optional binding power that defaults to 0 and returns a symbol object for that `id`. If the symbol already exists in the `symbol_table`, the function returns that symbol object. Otherwise, it makes a new symbol object that inherits from the `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.create(original_symbol);
        s.id = s.value = id;
        s.lbp = bp;
        symbol_table[id] = s;
    }
```

```
        return s;
    };
```

The following symbols are popular separators and closers.

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

The `(end)` symbol indicates the end of the token stream. The `(name)` symbol is the prototype for new names, such as variable names. The parentheses that I've included in the ids of these symbols avoid collisions with user-defined tokens.

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

Tokens

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

The `token` variable always contains the current token.

```
    var token;
```

The `advance` function makes a new token object from the next simple token in the array and assigns it to the `token` variable. It can take an optional `id` parameter which it can check against the `id` of the previous token. The new token object's prototype is a `(name)` token in the current scope or a symbol from the symbol table. The new token's `arity` is `"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.create(o);
        token.value = v;
        token.arity = a;
        return token;
    };
```

Scope

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 to have 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;
```

The `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 itself = function () {
    return this;
};

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 seeks, 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.

The `find` method tests the values it finds to determine that they are not `undefined` (which would indicate an undeclared name) and not a function (which would indicate a collision with an inherited method).

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

The `pop` method closes a scope, giving focus back to the parent.

```
    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. 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.create(original_scope);
    scope.def = {};
    scope.parent = s;
    return scope;
};
```

Precedence

Tokens are objects that bear methods allowing them to make precedence decisions, match other tokens, and build trees (and in a more ambitious project, also check types and 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?

d Ⓐ e Ⓑ f

If Ⓐ and Ⓑ are operators, does operand `e` bind to Ⓐ or to Ⓑ? In other words, are we talking about

(d Ⓐ e) Ⓑ f   or   d Ⓐ (e Ⓑ 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` method and a `led` method. For example, - might be both a prefix operator (negation) and an infix operator (subtraction), so it would have both `nud` and `led` methods.

In our parser, we will use these binding powers:

0	non-binding operators like ;
10	assignment operators like =
20	?
30	|| &&
40	relational operators like ===
50	+ -
60	* /
70	unary operators like !
80	. [(

Expressions

The heart of Pratt's technique is the `expression` function. It takes a right binding power that controls how aggressively it binds to tokens on its right.

```
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. Then as long as the right binding power is less than the left binding power of the next token, the `led` method is invoked on the following token. 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 has a `led` method that weaves the token object into a tree whose two branches (`first` and `second`) are the operand to the left of the `+` and the operand to the right. The left operand is passed into the `led`, which then obtains the right operand by calling the `expression` function.

```
symbol("+", 50).led = function (left) {
    this.first = left;
    this.second = expression(50);
    this.arity = "binary";
    return this;
};
```

The symbol for `*` is the same as `+` except for the `id` and binding powers. It has a higher binding power because it binds more tightly.

```
symbol("*", 60).led = function (left) {
    this.first = left;
    this.second = expression(60);
    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 make symbols for infix operators. The `infix` function takes an `id`, a binding power, and an optional `led` function. If a `led` function is not provided, the `infix` function supplies 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 infix operators:

```
    infix("+", 50);
    infix("-", 50);
    infix("*", 60);
    infix("/", 60);
```

`===` is JavaScript's exact equality comparison operator.

```
    infix("===", 40);
    infix("!==", 40);
    infix("<", 40);
    infix("<=", 40);
    infix(">", 40);
    infix(">=", 40);
```

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(".", 80, 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;
});
```

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("[", 80, 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;
}
```

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. (The falsy values are the number `0`, the empty string `""`, and the values `false` and `null`. All other values (including all objects) are truthy.)

```
        infixr("&&", 30);
        infixr("||", 30);
```

Prefix Operators

The code we used for right associative infix operators can be adapted 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 also sometimes be reserved words.

```
        var prefix = function (id, nud) {
            var s = symbol(id);
            s.nud = nud || function () {
                scope.reserve(this);
                this.first = expression(70);
                this.arity = "unary";
                return this;
            };
            return s;
        }

        prefix("-");
        prefix("!");
        prefix("typeof");
```

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 inner expression.

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

Assignment Operators

We could use `infixr` to define our assignment operators, but we will make a specialized `assignment` function because we want it to do two extra bits of business: examine the left operand to make sure that it is a proper lvalue, and set an `assignment` member 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("-=");
```

Notice that we have implemented 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);
```

The `(literal)` symbol is the prototype for all string and number literals. The `nud` method of a literal token returns the token itself.

```
symbol("(literal)").nud = itself;
```

Statements

Pratt's original formulation worked with functional languages in which everything is an expression. Most mainstream languages have statements that are not as nestable as expressions. We can easily handle statements by adding another method to tokens, the `std` (statement denotation). A `std` is like a `nud` except that it is used only 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 semi-colon. For reliability, we will 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;
};
```

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

```
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 statement symbols 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. (JavaScript does not have block scope. Simplified JavaScript corrects that.)

```
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 initializing 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 parens 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, then 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();
        } else {
            this.third = null;
        }
        this.arity = "statement";
        return this;
    });
```

The `break` statement is used to break out of loops.

```
    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 take 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 parens, and a body that is a list of statements wrapped in curly braces. A function has its own scope.

```
prefix("function", function () {
    var a = [];
    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("(", 80, function (left) {
        var a = [];
        if (left.id === "." || left.id === "[") {
            this.arity = "ternary";
            this.first = left.first;
            this.second = left.second;
            this.third = a;
        } else {
            this.arity = "binary";
            this.first = left;
            this.second = a;
            if ((left.arity !== "unary" || left.id !== "function") &&
                    left.arity !== "name" && left.id !== "(" &&
                    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;
    });
```

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;
    };
```

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;
});
```

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 colon (:). The key is a literal or a name which is 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;
});
```

Things to Do and Think About

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`), statement labels, more error checking, error recovery, and 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.

Try the demonstration of the parser that was described in this paper.

Another example of this parsing technique can be found in JSLint.