



本科毕业设计（论文）

**外文参考文献译文及原文**

学 院 计算机学院

专 业 软件工程

年级班别 2016级（3）班

学 号 3116004824

学生姓名 林楷羽

指导教师 宋 玮

2020年 5 月

**目 录**

[1 外文文献译文 1](#_Toc40887210)

[1.1 简介 1](#_Toc40887211)

[1.2 JavaScript 1](#_Toc40887212)

[1.3 符号表 2](#_Toc40887213)

[1.4 词法单元 3](#_Toc40887214)

[1.5 作用域 5](#_Toc40887215)

[1.6 优先级 8](#_Toc40887216)

[1.7 表达式 9](#_Toc40887217)

[1.8 中缀运算符 9](#_Toc40887218)

[1.9 前缀运算符 12](#_Toc40887219)

[1.10 赋值运算符 13](#_Toc40887220)

[1.11 常量 14](#_Toc40887221)

[1.12 语句 14](#_Toc40887222)

[1.13 函数 18](#_Toc40887223)

[1.14 对象字面量 21](#_Toc40887224)

[1.15 未完成的功能以及思考 22](#_Toc40887225)

[2 外文文献原文 23](#_Toc40887226)

[2.1 Introduction 23](#_Toc40887227)

[2.2 JavaScript 23](#_Toc40887228)

[2.3 Symbol Table 24](#_Toc40887229)

[2.4 Tokens 26](#_Toc40887230)

[2.5 Scope 27](#_Toc40887231)

[2.6 Precedence 30](#_Toc40887232)

[2.7 Expressions 31](#_Toc40887233)

[2.8 Infix Operators 31](#_Toc40887234)

[2.9 Prefix Operators 34](#_Toc40887235)

[2.10 Assignment Operators 35](#_Toc40887236)

[2.11 Constants 36](#_Toc40887237)

[2.12 Statements 36](#_Toc40887238)

[2.13 Functions 41](#_Toc40887239)

[2.14 Object Literals 43](#_Toc40887240)

[2.15 Things to Do and Think About 44](#_Toc40887241)

# 外文文献译文

## 简介

1973年，Vaughan Pratt 在波士顿的 编程语言原则座谈会第一期刊发表 自顶向下算符优先（Top Down Operator Precedence）。该论文描述了一种解析技术，它融合了优良性能的递归向下（Recursive Descent）思想与Folyd的算符优先思想。它非常容易使用。它很像递归向下，不同的是它只需要更少的代码、同时有着更好的性能。Pratt 表示它非常容易理解，实现简单，性能突出，且非常灵活。它是动态的，真正支持语言级别的扩展。

奇怪的是，直到今天，这样一种对于编译构建来说非常理想的方法完全被忽略了。为什么会这样呢？Pratt在论文中解释：BNF语法以及它的各种各样的衍生产物、与之相关的自动机和定理的产生，限制了自动机不可见领域方向的发展。

另一种解释是，Pratt的这种技术只有在作用于动态的，函数式编程语言上才有很卓越的效率。应用于静态的、过程式语言则会非常困难。在论文中，Pratt 使用 LISP 语言，毫不费力地将Token流构建成语法树。但这种解析技术在 LISP 社区中没有体现出很大的价值。因为自从 LISP 诞生以来，已经出现了很多让它拥有类似于算法的语法，包括 Pratt的 CGOL, LISP 2, MLISP, Dylan, Interlisp 的 Clisp, 以及 McCarthy 的 original M-epressions。但全都没有被接受，因为 LISP 社区认为协调程序与数据相对于表达式语法来说更有价值。Pratt 的解析技术需要一门动态语言，但动态语言社区在历史上不曾使用过便于 Pratt 的解析技术的对应语法。

## JavaScript

随着 JavaScript 的出现，情况发送了变化。JavaScript 是一门动态的函数式语言，但语法上明显是 C 语言家族的一员。它是一门有着社区喜欢语法的动态语言。

JavaScript 同时也拥有面向对象特性。Pratt 1973 年发表的论文预测面向对象特性将成为一种趋势，但它却缺乏一种富于表达的表示法。对于 Pratt 的解析方法来说，JavaScript 是一门理想语言。接下来，我将会用 JavaScript 快速实现一个简单的编译器。

在这么短的章节中，我们没有足够的时间来实现完整的 JavaScript 语言体系，而且我们也没有必要这样去做。因为这门语言有一部分是糟粕，虽然如此，它还是有一些值得思考的精华。我们将会编写一个解析简洁版 JavaScript 的解析器，使用简介版的 JavaScript 编写。简洁版的 JavaScript 包含以下特性：

1. 函数作为一级对象。在简洁版 JavaScript 中，函数是拥有词法作用域的 lambdas 表达式。
2. 原型继承的动态对象。对象与 class 无关。我们可以使用赋值语法对任何一个对象添加一个新的成员属性。一个对象可以从另外一个对象继承成员属性。
3. 对象字面量与数组字面量。对于创建新的对象与数组，这是一种非常便利的表示法。JavaScript 字面量的灵感来自于 JSON 数据格式。

我们将利用 JavaScript 的原型继承的特性来编写 Token，让它继承自 symbols。 我们的实现依赖于Object.create方法（该方法使新对象继承现有对象的成员）和 Token 生成器（ Token 生成器从字符串生成简单 Token 数组）。 我们逐步遍历这个数组，以此来构建语法树。

## 符号表

每个Token，例如运算符或标识符都将从 symbol 继承。我们将所有 symbol（决定我们语言中标记的类型）保留在 symbol\_table 对象中。

var symbol\_table = {};

这个 original\_symbol 对象是所有其他 symbol 的原型。它的方法将被重写。（我们将会在关于优先级的章节来解释 nud 和 led 的作用以及优先级的含义）

var original\_symbol = {

nud: function () {

this.error("Undefined.");

},

led: function (left) {

this.error("Missing operator.");

}

}

让我们定义一个生成 symbol 的函数。它具有一个符号 id 和一个默认为 0 的可选优先级，并返回该 id 的 symbol 对象。如果 symbol\_table 中已经存在该符号，则该函数返回该符号对象。否则，它将创建一个继承自 original\_symbol 的新 symbol 对象，并将其存储在 symbol\_table 中，然后将其返回。符号对象最初包含一个 id ，一个值，一个左优先级以及它从 original\_symbol 继承的内容。

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;

}

常见的分隔符和结束符：

1. symbol(":");
2. symbol(";");
3. symbol(",");
4. symbol(")");
5. symbol("]");
6. symbol("}");
7. symbol("else");

(end) symbol 表示 token 流的结束。(name) symbol 则是新命名的原型，例如变量名。包含在 id 两边的括号主要用来防止与用户定义的 token 冲突.

symbol("(end"));

symbol("(name)");

## 词法单元

我们假设源代码以及被转换成一个由简单词法单元（token）对象构成的数组，每个对象包含一个类型（type）属性（包含名称（name），字符串（string），数字（number）或操作符（operator））以及一个值（value）属性，类型为数字或字符串。

变量 token 总是指向当前的词法单元。

var token;

函数 advance 从数组中的下一个简单词法单元创建一个新的词法单元对象，并且赋值给 token 变量。它拥有一个可选参数 id 用来检查是否和之前一个词法单元匹配。新的词法单元对象的原型是当前作用域中的 (name) 词法单元或者是符号表中的一个符号。新的词法单元的 arity(运算元) 是 名称(name)，直接量(literal)或者操作符(operator)。arity 随后也可能根据我们了解更多该词法单元在程序中的角色而变化成二元运算(binary)，一元运算(unary)或者语句(statement)。

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;

};

## 作用域

大部分语言都为定义新符号准备了一些表示方法（例如变量名）。在非常简单的语言中，当我们遇到一个新的单词时，我们给它一个声明然后放到符号表中。在一些更加复杂语言中，我们可以使用作用域，方便程序员控制这个变量的生命周期和可见性。

作用域就是程序中变量被定义和可访问的一块区域。当前作用域可以被嵌套进其他作用域中。定义在内层作用域的变量对于外层作用域是不可见的。

我们将当前作用域对象保存在 scope 变量中。

var scope;

变量 original\_scope 是所有作用域对象的原型。它有一个 define 方法用来在该作用域中定义新的变量名。方法 define 将一个命名 token 转换成变量 token。果这个变量已经在作用域中定义过或者这个名字被用作保留字，那么将会报错。

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;

},

find 方法用于查找一个名称的定义。这个方法会从当前作用域开始查找，如果有必要的话，会顺着该作用域的父作用域链查找直到最后到达符号表。如果没有找到该名称的定义，则返回 symbol\_table['(name)']。

find 方法还会测试查找到的值，判断其不是 undefined （比如指向为未定义的变量名），也不是 function （可能与被继承的方法导致冲突）。

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)"];

}

}

},

方法 pop 可以清除一个作用域，返回到父作用域中。

pop: function () {

scope = this.parent;

},

方法 reserve 可以在当前作用域中声明一个需要被保留的关键字。

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;

}

};

我们需要为保留字准备一个策略。在一些语言中，被用于程序结构的词（例如 if）被当作保留字，不能用作变量的名称。由于我们解析器的灵活性，允许我们有一个更有效的策略。例如，我们可以说，在任何一个函数中，所有名称要么用作结构的单词，要么当作变量，但是不可能同时成为两者。那么我们只需当一个名称被用作保留词的时候，才将它保留在局部。这样的话，对于语言的设计者会更好，因为当要为语言添加新的结构单词时，不会破坏现有的程序，同样对于程序员也是好事，他们对于名称的使用不再受限于无关的约束。

每当我们想要为一个函数或者一个代码块建立一个新的作用域，我们调用 new\_scope 函数，这将新建一个对象实例，其原型指向 original\_scope 对象。

var new\_scope = function () {

var s = scope;

scope = Object.create(original\_scope);

scope.def = {};

scope.parent = s;

return scope;

};

## 优先级

词法单元对象上包含一些可以做优先级判断，匹配其他词法单元已经创建语法树（在未来更大的项目中，可以包含类型判断，代码优化及生成）的方法。优先级判断最基本的问题是：在两个操作符直接给定几个操作数，操作数是应该从左到右运算还是从右到左？

d A e B f

如果 A 和 B 是运算符的话，操作数 e 是应该与 A 还是 B 结合呢？换句话说：

(d A e) B f 还是 d A (e B f) 呢？

最终，这个复杂的解析过程给出了二义性的解决办法。我们实现用到的解决技术是这样的，每一个词法单元对象都有一个约束力(binding powers)（或者也可以说是优先级）作为成员，以及两个叫做 nud（空判定符）以及 lef（左判定符）的简单方法。nud 不管token 的左侧，而 led 却关注。nud 方法常用于值（例如变量和直接量）以及前缀操作符。led 方法常用于中缀和后缀运算符。一个 token 也可以同时拥有 nud 和 led 方法。例如，操作符 - 可以当作前缀运算符（负号），也可以当作中缀运算符（减号），所以它既拥有 nud 方法也拥有 led 方法。

在我们的论文中，我们使用这样的优先级：

|  |  |
| --- | --- |
| 优先级 | 操作符 |
| 0 | ; |
| 10 | 赋值号 = |
| 20 | ? |
| 30 | || && |
| 40 | 关系运算符 如： === |
| 50 | + - |
| 60 | \* / |
| 70 | 一元运算符 如：! |
| 80 | . [ ( |

## 表达式

Pratt 技术的核心就是 expression 函数。它需要一个右约束力（right binding power）参数，用来控制它对右侧词法单元有多大的约束力。

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 调用 token 的 nud 方法。nud 方法用于处理直接量，变量和前缀运算符。只要右约束力小于下一个 token 的左约束力，那么就对接下来的 token 调用 led 方法。led 方法用于处理中缀和后缀运算符。这一过程是递归的，因为 nud 和 led 方法都会调用 expression。

## 中缀运算符

+ 是一个前缀运算符，所以他有一个 led 方法用于将 + 左侧和右侧的 token 对象转换为树的两支（first 和 second）。左侧的操作数传入 led 中运算，在运算中通过调用 expression 函数来获得右侧操作数。

symbol("+", 50).led = function (left) {

this.first = left;

this.second = expression(50);

this.arity = "binary";

return this;

};

符号 \* 与 + 很类似，除了 id 和约束力不同之外。由于它的优先级更高,所以拥有更大的约束力。

symbol("\*", 60).led = function (left) {

this.first = left;

this.second = expression(60);

this.arity = "binary";

return this;

};

并不是所有的中缀运算符都与这相似，但是大部分都会是这样的，所以为了让我们的工作更简单，我们定义了一个 infix 函数，用来帮助我们创建中缀运算符的符号对象。infix 函数需要 id，约束力，以及一个可选的 led 函数作为参数。如果没有提供 led 函数，那么 infix 函数提供一个大部分情况下都有用的 led 默认函数。

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;

}

这就允许我们使用一种更具有表达力的写法来表示中缀运算法：

infix("+", 50);

infix("-", 50);

infix("\*", 60);

infix("/", 60);

JavaScript 中表示精确等于的符号是 ===

infix("===", 40);

infix("!==", 40);

infix("<", 40);

infix("<=", 40);

infix(">", 40);

infix(">=", 40);

三元运算符需要三个部分表达式，用 ? 和 : 分隔。这不是一个常规的中缀运算符，所以我们需要提供 led 函数。

infix("?", 20, function (left) {

this.first = left;

this.second = expression(0);

advance(":");

this.third = expression(0);

this.arity = "ternary";

return this;

});

. 运算符用于获取一个对象的成员属性。右侧的 token 必须是一个变量名，且它会被当作字面量使用。

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;

});

[ 运算符用于动态地从一个对象或者数组中获取成员。右侧的表达式必须以 ] 结尾。

infix("[", 80, function (left) {

this.first = left;

this.second = expression(0);

this.arity = "binary";

advance("]");

return this;

});

上述这些中缀运算符都是左结合的。我们还需要创造右结合的运算符，例如：短路逻辑运算符，这是通过降低右约束力来实现的。

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;

}

如果 && 运算符的第一个操作数为假，那么就返回第一个操作数。否则返回第二个。如果 || 运算符的第一个操作数为真，那么就返回第一个操作数，否则返回第二个。（这里“假”的含义包括数字 0，空字符串 ''以及 false 值和 null 值。所有其他值（包括所有的对象）都是“真”）。

infixr("&&", 30);

infixr("||", 30);

## 前缀运算符

我们用于右结合中缀运算符的代码，可以适配到前缀运算符。前缀运算符是右结合的。由于前缀运算符不需要项做绑定，所以没有左约束力。前缀运算符同时也可用作保留关键字。

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");

（ 符号的 nud 函数需要调用 advance（“)”）去匹配 ） token。由于 nud 函数返回内部表达式，所以 （ token 不会成为语法树的一部分。

prefix("(", function () {

var e = expression(0);

advance(")");

return e;

});

## 赋值运算符

我们可以用 infixr 来定义赋值运算符，但是我们还需要对 assignment 函数做一些特殊定制，来让它多做两件事：一是验证左侧操作数，确保是正确的左值，二是添加一个 assignment 成员，之后便可以快速地判断出赋值语句。

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("-=");

需要注意的是，我们这里的实现利用了一些列的继承模式，assignment 返回调用 infixr 的结果，而 infixr 则返回调用 symbol 的结果。

## 常量

constant 函数用于构建语言内部的常量。nud 方法将一个名称词法单元转换成直接量词法单元。

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);

(literal) 是所有字符串直接量和数字直接量的原型。直接量词法单元的 nud 方法返回词法单元自身。

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

## 语句

Pratt 原始的陈述体系只针对那些所有东西都是表达式的函数式语言有效。大部分主流语言不像那些层层嵌套的表达式一样，而是拥有语句的概念。只要为词法单元再添加一个 std（语句描述符）方法，就可以很轻松地处理语句。std 很像 nud，除了它只在语句开头调用外。

statement 方法解析一个语句。如果当前词法单元有 std 方法，那么这个词法单元就会被保留做关键词，然后调用 std 方法。否则，我们则假设一个以分号结尾的表达式语句。为了代码更好的可读性，我们只允许赋值或者调用的表达式语句。

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;

};

statements 函数解析所有语句直到遇到 (end) 或者 } 这些表示块结束的标识符。这个函数的返回值是一个语句，或者包含很多语句的一个数组，或者是 null 用来表示没有语句。

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;

};

stmt 函数用于将语句符号添加到符号表中。需要两个参数，语句的 id 和 std 函数。

var stmt = function (s, f) {

var x = symbol(s);

x.std = f;

return x;

};

语句块是用花括号括起来的一些列语句，同时也要给它们一个新的作用域。（JavaScript 没有块级作用域，简化版的 JavaScript 修正了这一点。）

stmt("{", function () {

new\_scope();

var a = statements();

advance("}");

scope.pop();

return a;

});

block 函数用于解析语法块。

var block = function () {

var t = token;

advance("{");

return t.std();

};

var 语句用于在当前语句块里定义一个或多个变量。每一个名称后面跟一个可选的 = 用于初始化表达式。

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;

});

while 语句用于定义循环。它的圆括号里包含一个表达式，接下来是语句块。

stmt("while", function () {

advance("(");

this.first = expression(0);

advance(")");

this.second = block();

this.arity = "statement";

return this;

});

if 语句允许有条件地执行。如果我们看见 else 符号后面还有一个语句块，那么我们就接着解析下面的语句块或者是 if 语句。

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;

});

break 可以跳出循环。

stmt("break", function () {

advance(";");

if (token.id !== "}") {

token.error("Unreachable statement.");

}

this.arity = "statement";

return this;

});

return 语句用于从函数中返回，后面可以带一个可选的表达式。

stmt("return", function () {

if (token.id !== ";") {

this.first = expression(0);

}

advance(";");

if (token.id !== "}") {

token.error("Unreachable statement.");

}

this.arity = "statement";

return this;

});

## 函数

函数是可执行的对象。函数有一个可选的函数名（因此它可以递归地调用自己），用圆括号括起来的用逗号分隔的参数以及用花括号括起来的有一系列语句组成的函数体。函数有自己的作用域。

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;

});

函数使用 ( 运算符调用的。其中包含零个或多个用逗号分隔的参数。我们需要判断左侧操作数是否是一个不可能为函数的表达式。

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;

});

this 符号是一个特殊的变量。当函数是以方法的形式被调用，那它就指向那个对象。

symbol("this").nud = function () {

scope.reserve(this);

this.arity = "this";

return this;

};

## 对象字面量

数组直接量是由一对方括号括起来，其中包含零个或多个由逗号分隔的表达式。每一个表达式都是可计算的，且他们的计算结果就是数组的值。

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;

});

对象直接量是由一对花括号括起来，其中包含一个或多个由逗号分隔的键值对组成。每一个键值对是由一个分号分隔键与值。且键必须是一个直接量或者是可以被当作直接量的名称组成。

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;

});

## 未完成的功能以及思考

语法树可以被代码生成器解析，或者直接交由翻译器执行。产生语法树的计算是很简单的，正如我们所见，通过程序来创建一棵语法树也很简单。

我们可以让 infix 函数接受一个 opcode 参数，这样有助于代码生成。我们同样也可以添加一些其他方法来支持常量折叠或者代码生成。

我们还可以继续添加更多的语句支持（例如，for, switch, try），语句标签，更多的错误检查，错误恢复以及更多的运算符。我们还可以添加类型规范已经类型推断。

我们可以让我们的语言更具扩展性。可以让定义变量更加方便，同样也可以让程序员自己添加新的运算符和语句。

快来试试我们在这篇论文中描述的解析器。

关于解析器的其他技术例子可以在 JSLint 中找到。

# 外文文献原文

## 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 A e B f

If A and B are operators, 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) 呢？

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.