自顶而下的运算符优先级

《代码之美》第九章

李长城

前言

- 为什么要讲这篇文章？我之所以要讲这篇文章：
- 第一是因为这种技术非常美妙：不光是从思想上，而且实现效率极高，逻辑清楚，层次分明，易于扩展和维护。
- 第二是因为它展示了JavaScript语言一些精彩的、值得程序语言设计者思考借鉴的特性。
- 第三是因为解析是解释与编译前非常重要的一步，而如果你不懂解释器，你就不会成为程序员中的master。

JavaScript简介

- JavaScript是一门动态函数语言，有丰富的支持面向对象编程的方法。但从句法看，却比较接近C。
- JavaScript是一种广泛用于客户端Web开发的脚本语言，常用来给HTML网页添加动态功能，比如响应用户的各种操作。它最初由网景公司的Brendan Eich设计，是一种动态、弱类型、基于原型的语言，内置支持类。以它为基础，制定了ECMAScript标准。（摘自维基百科）
- 虽然更常用于网页，但JavaScript也可以用于其他场合，比如服务器端编程（参见Rhino）。（摘自维基百科）
- 关于JavaScript更进一步的语言特性，会在讲文章的相应部分时讲解。

解析器简介

- 在此有必要先介绍一下什么叫解析器。右图是程序语言从原文件到解释、编译的一个流程。
- 其中FrontEnd有两个部分组成，第一部分把原文件字符串转换成一串语素，第二部分解析器则将语素串组织成表达式，语句，语块之类的语法结构，最后生成一个抽象语法树。

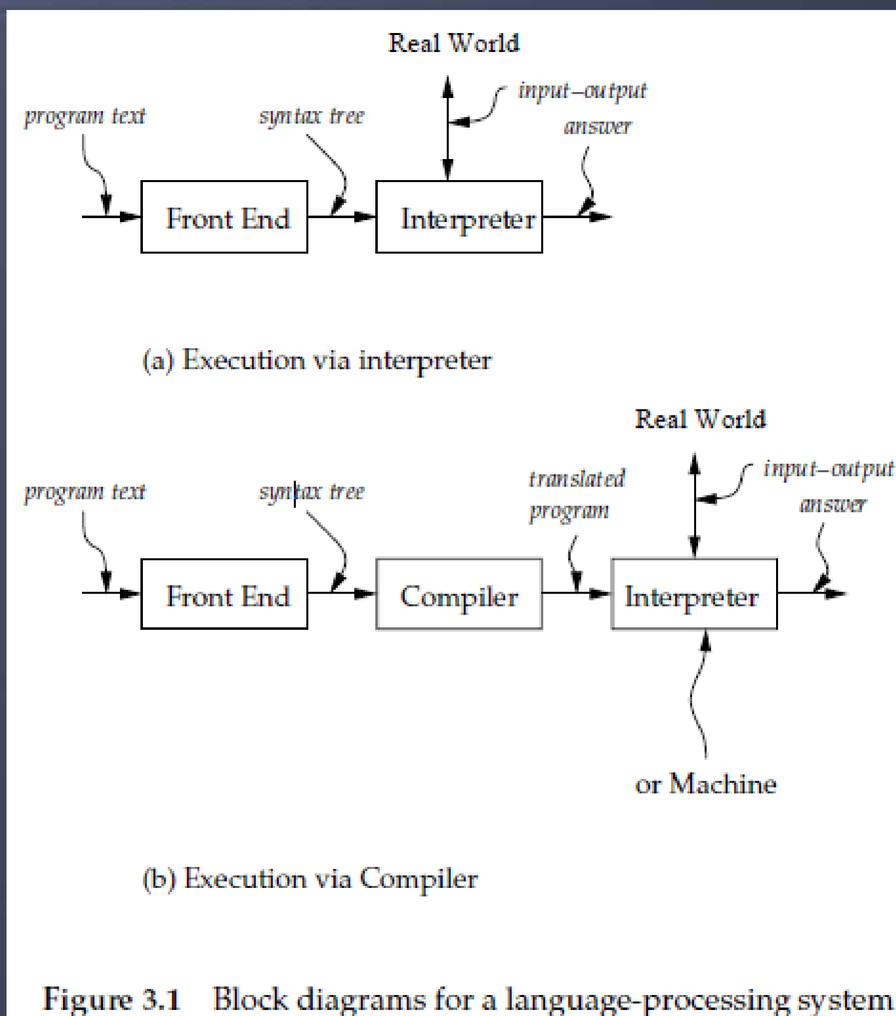

Figure 3.1 Block diagrams for a language-processing system

图片截自**Essentials of Programming Languages**

解析器实现需要？

- 1、正确根据运算符的优先级将语素串组织成表达式；
- 2、变量的处理——>作用域的处理：在作用域中定义新的元素，查找作用域内变量，创建新的作用域；
- 3、对函数声明及调用的处理。
- 4、对各种语句的处理。
- 其中的重点是优先级的处理。
- 为了更好的理解文章的内容，我们需要先做一下准备工作。

准备工作

- 为了后面方便，先介绍一些JavaScript语法。
- var语句：定义变量
- 函数定义：function 可选函数名{函数体}（和数、字符串一样，是值的一种）
- this:当前对象
- 对象：{key1:值,key2:值,......}（值的一种）
- 创建新对象（一种方法）：object（原型）
- ||：一种二元中置运算符，a||b当a为假值时返回b的值。

解析流程的大体解释

- 我们的解析器是由符号表驱动的，可理解为构造解析树的方法有以下几种：
- 1、符号表中符号的方法（构造以本符号为根的解析树的方法）；
- 2、解析表达式的方法expression；
- 3、解析语句的方法statement和语句组的方法statements。
- 这些过程之间相互调用。最终构造出了一颗解析树。

- 解析器工作的大体过程如下：
- 程序是有一条条的语句组成的，解析时首先调用statements，而statements则反复调用statement直至程序结束，将解析所得的结果并入一个列表作为最终解析结果。
- statement一次解析一条语句，如果语句开头的语素是语句类型的标识，则调用相应的方法，按相应的语法规则来解析语句；如果不是，则此语句应是表达式语句（赋值表达式或函数调用表达式）。
- 解析器的工作就主要转化为对表达式的解析，和应用语法规则解析语句的一个过程。

符号表

- var original_symbol_table = {};（对象，数组都是hash表）
- var original symbol = {
- nud: function ( ) {
- this.error("Undefined.");
- },
- led: function (left) {
- this.error("Missing operator.");
- }
- };
- 这是所有符号对象的原型。
- (nud与led是处理语素的方法,其具体含义将在后面解释）
- (bp为所绑定的权值，id为符号的标示）

- 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;（应为s）
- };

注意

- s = object(original_symbol);
- s.id = s.value = id;
- s.lbp = bp;
- symbol_table[id] = s;
- return z;（应为s）
- 这几行代码值得注意，这个函数不但创建了我们所需的一个符号，并将其加入符号表，而且将它作为返回值传回。
- 这点在仔细阅读后面的代码时值得注意。

token与advance（）

- 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;
- };

- token：当前语素，由advance过程生成，包含了此元素的类型，在语言中的作用，value等信息（advance也是表达式解析与语句解析的第一步，这也体现了符号表驱动的含义）。
- 这个函数的功能比较简单，是创建一个新的语素对象，并把它付给变量token。
- 这个函数有几点值得注意的地方：
- 一是有一些检查报错的步骤；
- 二是有o =scope.find(v);o = symbol_table[v];o = symbol_table["(literal)"];token = object(o);等一些步骤，这保证了token保持了原型（a+b*c中的“+”即以符号表中的“+”为原型）在符号表、作用域中所具有的一些方法和属性。也保证了token的生成不会破坏原型。

表达式的解析

```
{  "value": "+",
    "arity": "binary",
    "first": {
        "value": "a",
        "arity": "name"
    },
    "second": {
        "value": "*",
        "arity": "binary",
        "first": {
            "value": "b",
            "arity": "name"
        },
        "second": {
            "value": "c",
            "arity": "name"
        }
    }
}
```

- 为了更好的观察解析表达式的过程，我们不妨观察一个例子。
- 在这个例子中，我们解析了a+b*c这个表达式，我们可以看到位于树根处的是+这个优先级较低的运算符，表示最后一步运算，arity关键字表示此运算符是二元的，第一个参数是表达式"first": { "value": "a", "arity": "name"}，第二个参数是表达式"second": {"value": "*", "arity": "binary"......}

启示

- 从上面的观察我们可以得到以下启示，每个表达式树根都是最后进行的，优先级较低的运算符（不论是前置或是中置）（目前主要说的是算术运算符），而每个运算符的参数（枝叶）又是另一个表达式。

- 则我们可为我们对表达式的解析大致构想为以下的形式：

- expression作为解析表达式的函数，带有一个权值参数。（既可解析独立的表达式也可为某一运算符生成右枝解析树）。
- 构造解析树从当前运算符右侧的值开始。原则是跟谁的结合更紧密，解析树就传给谁。
- 解析遇到低于此权值的运算符则停止构造，构造出来的即是左侧运算符的右子树。以保证不会将低等级的运算符包含进高等级的运算符的子树中。
- 而遇到高等级的运算符时，则说明目前构造的解析树与此高级运算符结合更紧密，将已有的解析树作为参数传给此高等级的运算符的解析树构造方法，构造以此高级运算符为根的解析树……
- 则我们可将expression写成以下形式：

- var expression = function (rbp) {var left;
- var t = token;
- advance( );
- left = t.nud( );
- //由值对象或前置运算符生成解 //析树
- while (rbp < token.lbp) {
- //右侧的中置运算符权值较大
- t = token;
- advance( );
- left = t.led(left);
- /*由右侧的中置运算符led方法生成解析树，此时原来的left解析树成为右侧的中置运算符的子树，生成的left解析树作为新的left，而后进行新的循环*/
- }
- return left;}

- （方法nud和方法led是以所处理语素为根生成解析树的方法。nud不在乎左对象，而led在乎（所以led有参数代表左面的解析树）。所以值对象（变量和字面量）和前置运算符用nud方法，中置运算符用led方法。）
- 从这里可以看出，expression有一个参数（为左侧运算符的优先级），如越小，则expression读取的范围也越大，左侧运算符也就越在靠近树根的一侧。这符合我们的要求。

下面具体观察几个中置运算符的例子。

- symbol("+", 60).led = function (left) {this.first = left;
- //接受传来的参数为解析树左支
- this.second = expression(60);
- //构造解析树右支，60控制构造范围，确保不会将优先级较低的运算符包含。
- this.arity = "binary";
- return this;};
- //最后传回以本身为根的解析树。
- symbol("*", 70).led = function (left) {this.first = left;
- this.second = expression(70);
- this.arity = "binary";
- return this;};
- (思考？这种写法是什么意思？)

- 观察以上几个例子，我们可以抽象出一种定义中置运算符的模式，它可让我们写出的解析器更清晰：
- 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; };
- 其中bp为要绑定的权值，这一模式适用于大多数中置的算数运算符。

- 由expression中的while (rbp < token.lbp)，而在上面的运算符抽象中，rbp极为运算符的优先级本身，则可以知道，此模式默认创造一个左结合的运算符，即a+b+c会被解析成右边的形式，第二个加号处于树的根部。
- 也可通过减小右绑定权值来创建右结合运算符模式。在此不予赘述。

```
{ "value": "+",
        "arity": "binary",
        "first": {
            "value": "+",
            "arity": "binary",
            "first": {
                "value": "a",
                "arity": "name"
            },
            "second": {
                "value": "b",
                "arity": "name"
            }
        },
        "second": {
            "value": "c",
            "arity": "name"
        }
    }
```

- 除了以上模式定义中置运算符。有一些比较特殊的中置运算符的了的方法还需单独定义。如表示选择的？：三元运算符，用于选择对象成员的点“.”运算符以及用于从对象和数组中选出成员的“[”运算符。
- 考虑其使用的方法（与左侧的运算符的竞争），则除了？：运算符优先级应低于除了赋值运算符的其他运算符外。“.”和“[”运算优先级应高于其他所有运算符。
- 应该说这些运算符的使用都有比较明确的语法规则，不会与右侧的其他运算符竞争，只需考虑与左侧的其他运算符的竞争。因此其led方法写起来只须遵照语法规范即可。在此略去不讲。
- 定义led方法时唯一需要特别注意的是与expression的结合，特别是advance的使用。

前置运算符

- 前置运算符的处理和前面类似。前置运算符是右结合的，它没有左绑定权值，因为它不向左绑定。前置运算符有时也可以是保留字。下面是定义前置运算符的一个一般方法。
- var prefix = function (id, nud) {
- var s = symbol(id);
- s.nud = nud || function ( ) {
- scope.reserve(this);
- this.first = expression(80);//其优先级应高于一般的中置运算符。
- this.arity = "unary";
- return this;
- };
- return s;
- }
- 因“前置运算符”“（”（不会成为解析树的一部分）的优先级要比任何运算符都要高，为避免优先级的混乱等因素，我们可单独写出“（”的nud方法，比较简单，在此略去。

- 为完善处理表达式的方法，我们还应增加处理赋值运算符，在被解析的语言中植入常数，及处理函数声明，函数调用，数组和对象字面量的方法。
- 下面是对赋值运算符的处理。
- var assignment = function (id) {
- return infixr(id, 10, function (left) {//infixr是创建右结合的运算符的过程
- if (left.id !== "." && left.id !== "[" &&
- left.arity !== "name") {
- left.error("Bad lvalue.");//检查左侧的解析树是否符合左值的要求
- }
- this.first = left;
- this.second = expression(9);//右结合的运算符
- this.assignment = true;//设置assignment标记
- this.arity = "binary";
- return this;
- });
- };
- 其中常数的处理方法略去。
- 函数声明、调用，数组和对象字面量将单独讲解。

做一做

- 一个算术表达式(a+b*c)*(d*e+f+g)的解析过程是怎样的，其生成的树又是怎样的呢？

对作用域的处理

- 我们在前面的advance函数中已经看到对scope的利用。在这里我们具体来看一下解释器对作用域的处理。
- 根据JavaScript语言的特点，我们的解释器对作用域的处理必须符合以下几个要求：
- 1、新函数的建立与语句块（？）的建立会建立新的作用域；
- 2、内部的函数与语句块可以看到外部的作用域。
- 3、变量必须先声明然后使用（？）。

- 我们可以相应的提出一种scope的设计方案：
- 每一个scope都应具有：
- 1、定义变量、查找变量、定义保留字、退出当前作用域的方法；
- 2、都应具有指向父作用域的指针（方便逐级查找，这样就保证了内部的函数可看到外部的作用域）（应在新建作用域时生成）。
- 并且还应有一个新建新的作用域的全局函数。
- 后面是相应的具体构造：

Scope

- 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;
- //特别要注意以上这句话，名字的 //value是字符串，这使得查找成为 //可能
- n.reserved = false;
- n.nud = itself;
- n.led = null;
- n.std = null;
- n.lbp = 0;
- n.scope = scope;
- return n;},

- 这是所有作用域对象的原型。
- 这是定义名字的方法，如果名字已被定义或用于保留字，则报错。
- 注意检查是否定义只在当前作用域中检查，则在内层作用域内的定义或保留可遮蔽外层作用域。
- def对象内保存了在当前作用域下定义的变量。

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

- find是作用域内的查找功能，从当前作用域开始逐层向上查找，直至找到合适的结果。
- 注意e = e.parent;保证了当前作用域不被破坏。
- pop则会退出当前作用域，返回上级作用域。

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

- reserve用于将一个名字定义为保留字。
- new_scope则用于创建一个新的作用域，它以original_scope为原型，拥有指向外层作用域的指针。

语句的处理

- 为了使我们的解析器支持语句的处理，我们在语素里加入另一个方法std，和函数nud一样，它生成一棵当前语句的解析树。
- 函数statement一次解析一条语句。如果当前语素有std方法，则保留该语素，并调用它的std函数。不然，我们假设一条语句以“；”结尾。为了可靠性，我们不接手既非赋值也非函数调用的表达式语句：

- 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 !== "(")
- //这也反映了我们为什么不把前置的“（”放入解析树中
- {v.error("Bad expression statement.");
- }
- advance(";");
- return v;
- };

statements

- 函数statements逐条解析语句，直到它看到表示程序快结束的“（end）”或者“}”。它返回一条语句、一条语句，或者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;
};
```

stmt

- 函数stmt用于将语句加入符号表。
- var stmt = function (s, f) {
- var x = symbol(s);
- x.std = f;
- return x;
- };

- 我们可以定义很多语句，书上定义了程序块语句，解析程序快的block函数，声明变量的var语句，while、if、break、return等流程控制语句。
- 它们的std方法都不是很难理解，同学们可自己研究。

函数的声明

- 函数是可以执行的对象值。函数可以有名字（用来递归调用自己），一组放在括号内的参数名和函数体。函数体是包含在花括号内的一组语句。函数有自己的作用域。
- prefix("function", function ( ) {
- var a = [];//参数表
- scope = new_scope( );//新建作用域
- if (token.arity === "name") {
- //如果函数有名称，则在作用域中定义
- scope.define(token);
- this.name = token.value;
- advance( );
- }
- advance("(");

- if (token.id !== ")") {
- while (true) {
- if (token.arity !== "name") {
- token.error("Expected a parameter name.");
- }
- //在作用域中定义
- scope.define(token);
- a.push(token);//构造参数表
- advance( );
- if (token.id !== ",") {
- break;
- }
- advance(",");
- }
- }
- this.first = a;
- advance(")");
- advance("{");
- //读函数体
- this.second = statements( );
- advance("}");
- this.arity = "function";
- scope.pop( );
- return this;
- });

思考？

- 在刚才的函数中，有这样一段代码：
- scope = new_scope( );//新建作用域
- if (token.arity === "name") {
- //如果函数有名称，则在作用域中定义
- scope.define(token);
- this.name = token.value;
- ....
- 这说明了什么呢？注意新建作用域和将函数写入作用域的先后顺序。
- 注意我们的作用域查找规则，也就是变量作用范围的规则。
- 函数究竟应该怎样调用呢？
- 这又会造成什么样有趣的特性呢？

函数的调用

- 函数通过“（”运算符调用。它可以接受零个或多个用逗号分隔的参数。我们通过考察左运算对象来判别不能当做函数的表达式：

- infix("(", 90, function (left) {
- var a = [];//实参表
- this.first = left;
- //函数体或函数名或方法名
- 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.");
- }//左值是否合理的判断
- if (token.id !== ")") {
- while (true) {
- a.push(expression(0));
- if (token.id !== ",") {
- break;
- }
- advance(",");
- }
- }
- advance(")");
- return this;
- });

数组字面量

- 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;});

- 数组字面量是包含零个或多个表达式的一组方括号。表达式用逗号分隔。每个表达式的值收集起来，得出一个新的数组。

对象字面量

- 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;});

- 对象字面量是包含零个或多个键值对的一组花括号。键值对使用冒号分隔的键/表达式对：键要么是字面量，要么是被当做字面量的名字。
- 注意这里实现的对象较弱，甚至不可能定义私有元素，在这个实现中，只有语句块和函数能实现私有元素。
- 思考：怎样增强对象的能力呢？
- 更进一步，JavaScript完整的支持对象的功能如何实现呢？

要做和要思考的事

- 本文展示的简单解析器容易扩展。生成的解析树可以传给代码生成器，也可以传给解释器。生成树的计算量极小。而且我们也看到了，编程生成解析树也不费什么功夫。
- 我们可以让infix函数接受一个操作码参数，用来辅助代码生成。我们也可以让他接受额外的方法参数。这些方法可以用于常数展开和代码生成。
- 我们可以加上其他语句，比如说for、switch和try。我们还可以加上语句标签。我们可以加上更多的错误检查和错误恢复。我们可以加入很多的运算符。我们可以加入类型规则和推断。
- 可以让我们的语言具备扩展能力，使得程序员加入新运算符和语句就跟加入新的变量一样容易。
- 这章描述的解析器有演示程序。你可以到下面的网址体验：http://javascript.crockford.com/tdop/index.html。
- JSLint也用到了这里的解析技术：http://JSLint.com。

- 注意：此网站的解析器和文中所描述的在算法上有一定差别，不过很细微，有兴趣的同学可以体验研究一下，看看差别是什么，有什么不同之处。还可以实验一下，看自己写的代码能否通过解析器，生成的树又是什么样的。
- 有兴趣的同学还可以观察本文所实现的解析器所支持的简易JavaScript语言除了功能少之外，还在哪些方面与目前的浏览器支持的JavaScript有何不同。
- 有兴趣的同学甚至可以自行编一个相应的解释器，它可解释运行本文展示的解析器所生成的抽象语法树。在本文所实现的解析器的基础上，这不难办到。

相关资料

- 对于程序语言的解析与解释、编译等技术有兴趣的同学还可以参考以下一些书籍，可以更好的理解这项技术
- The.MIT.Press.Essentials.of.Programming.Languages.3rd.Edition
- 这篇文章比较全面的介绍了程序语言的解析与解释技术
- Smalltalk-80：The Language and its Implementation
- 这本书的第四部分介绍了一个smalltalk的实现，smalltalk是一种早期的面向对象的语言。对于这种语言有兴趣的同学可以参考一下这本书。
- ALittleSmalltalk
- 这本书也介绍了一下smalltalk的实现。
- Queinnec.C. LISP in small pieces 1996
- 这本书全面介绍了lisp语言的解释实现与编译实现，想更深层次的了解lisp语言的同学以及对程序语言实现有兴趣的同学可参考此书。
- Write Yourself a Scheme in 48 Hours
- 一本教你写出自己的scheme解释器的Haskell教程。
- The Little JavaScripter Douglas Crockford
- 这篇文章的作者所写的另一篇文章，教你怎样用JavaScript写出scheme解释器。

- 而对于javascript比较感兴趣的同学除了可以找些javascript教程来看，也可到以下的网址，上面有一些关于javascript的文章，绝对可以令你加深对这门语言的理解。
- http://javascript.crockford.com/

谢谢参与！！！