

如何用JS实现你的个性化语言？(AST & PARSER)

1. 写一门语言需要考虑哪些模块？

- a. 流式读取字符
- b. 将连续的字符串标计化（tokenization），即将字符转化为Token。其主要的形式和分类如下：

JSON

```
1  { type: "punc", value: "(" }           // 符号: (), ,, ; {}等
2  { type: "num", value: 5 }             // 数字
3  { type: "str", value: "Hello World!" } // 字符串
4  { type: "kw", value: "function" }     // 关键字
5  { type: "var", value: "a" }           // 变量名
6  { type: "op", value: "!=" }           // 操作符
```

- c. 将Token数组转化为抽象语法树(AST)
- d. 将AST转化为可执行代码
 - i. 基于JS执行
 - ii. 转化为JS执行

其中 a 和 b 两个步骤合起来做的事情专业术语叫做词法分析，c叫做语法分析。

2. 实现极其简单的中文语言

下面我们将写一门属于自己简单的中文语言。随便起个名字比如：“中语言”吧。

2.1 定义语言的功能

首先我们需要定义“中语言”的功能，我们希望它能支持加减乘除的函数以及变量定义。

举几个“中语言”代码例子：（这里的阿拉伯数字就不转化为汉语了，不是重点）

JavaScript

```
1  const chineseAdd = `相加 为 函数 (甲, 乙) 甲 加 乙; 打印 (相加 (3, 4))`;
2  const chineseVarAdd = `定义 (甲 为 2, 乙 为 甲 加 7, 丙 为 甲 加 乙) 打印 (甲 加 乙 加 丙)`;
3  const chineseIf = `如果 (2 减 1 为 1) 「打印 (666)」`
```

根据第一节所总结的模块进行一一讲解，接下来就是流式读取字符。

2.2 流式读取字符

2.2.1 规则

- 从左往右，以字符为单位读取。
- 以空字符为结尾。
- 为了精确错误代码地址，需要以遇到\n为标记记录行数和列数。

2.2.2 代码

JavaScript

```
1 // 字符流分析器
2 function InputStream(input) {
3     let pos = 0, col = 0, line = 1;
4     return {
5         // 移动到下一位
6         next,
7         // 返回下一位
8         peek,
9         // 是否到末尾
10        eof,
11        // 解析报错
12        croak
13    };
14    function next() {
15        const char = input.charAt(pos++);
16        if (char === '\n') {
17            line++;
18            col = 0;
19        } else {
20            col++;
21        }
22        return char;
23    }
24
25    function peek() {
26        return input.charAt(pos);
27    }
28
29    function eof() {
30        return peek() === '';
31    }
32
33    function croak(msg) {
34        throw new Error(`${msg} (${line}:${col})`);
35    }
36 }
37 module.exports = InputStream;
```

2.2.3 总结

字符流分析器不是将字符串直接转化为另外的数据结构，而只是返回了操作字符串的几个方法，只有在真正需要操作字符串的时候才会移动当前“指针”。

2.3 将字符串转换为Token

2.3.1 规则

- a. Token的生成顺序依赖字符流的读取顺序
- b. 定义“中语言” token 分类
- c. 通过正则表达式根据读取到的连续子字符串创建相应的token

JSON

```
1 { type: "punc", value: "(" }           // 符号: 「」, , , ; () 等
2 { type: "num", value: 5 }              // 数字
3 { type: "str", value: "你好" }         // 字符串
4 { type: "kw", value: "函数" }          // 关键字
5 { type: "var", value: "甲" }           // 变量名
6 { type: "op", value: "加" }            // 操作符
```

2.3.2 代码

TokenStream返回的也是一批操作，其并没有改变字符串的数据结构，但是它在每次操作后会生成当前“指针”所代表的子字符串的token数据。从结果来看，在使用时，我们可以想象其会生成一个token 数组。

JavaScript

```
1 // 词法解析器
2 function TokenStream(input) {
3     let current = null;
4     // 关键词数组
5     const keywords = " 如果 则 否则 函数 真 假 定义 ";
6     return {
7         // 移动到下一个token 词组
8         next,
9         // 获取到下个词组
10        peek,
11        // 是否到了尾部
12        eof,
13        // 报错
14        croak: input.croak
15    };
16 }
```

```

16     function peek() {
17         return current || (current = read_next());
18     }
19     function next() {
20         let tok = current;
21         current = null;
22         return tok || read_next();
23     }
24     function eof() {
25         return peek() === null;
26     }
27     // 读取下一个词组
28     function read_next() {
29         read_while(is_whitespace);
30         if (input.eof()) return null;
31         const char = input.peek();
32         if (char === '#') {
33             skip_comment();
34             return read_next();
35         }
36         if (char === '"') return read_string();
37         if (is_digit(char)) return read_number();
38         // 操作符
39         if (is_op_char(char)) {
40             return {
41                 type: 'op',
42                 value: read_while(is_op_char)
43             }
44         }
45         if (is_id_start(char)) return read_ident();
46         // 符号
47         if (is_punc(char)) {
48             return {
49                 type: 'punc',
50                 value: input.next()
51             }
52         }
53
54         input.croak('无法解析的字符:' + char);
55     }
56     function is_keyword(x) {
57         return keywords.indexOf(" " + x + " ") >= 0;
58     }
59     function is_digit(ch) {
60         return /[0-9]/i.test(ch);

```

```

60     return /[\u4e00-\u9fa5a-z\u_]/i.test(ch);
61 }
62 function is_id_start(ch) {
63     return /[\u4e00-\u9fa5a-z\u_]/i.test(ch);
64 }
65 function is_id(ch) {
66     return is_id_start(ch) || '?!-<=>=123456789'.indexOf(ch) >= 0;
67 }
68 function is_op_char(ch) {
69     return '加减乘除为并或等于'.indexOf(ch) >= 0;
70 }
71 function is_punc(ch) {
72     return ", : () 「 」 ; ".indexOf(ch) >= 0;
73 }
74 function is_whitespace(ch) {
75     return " \t\n".indexOf(ch) >= 0;
76 }
77 // 连续读取属于predicate类型的字符
78 function read_while(predicate) {
79     let str = '';
80     while (!input.eof() && predicate(input.peek())) {
81         str += input.next();
82     }
83     return str;
84 }
85
86 function read_number() {
87     let has_dot = false;
88     let number = read_while(function (ch) {
89         if (ch === '.') {
90             if (has_dot) {
91                 return false;
92             }
93             has_dot = true;
94             return true;
95         }
96         return is_digit(ch);
97     });
98     return { type: 'num', value: parseFloat(number) };
99 }
100
101 function read_ident() {
102     const id = read_while(is_id);
103     return {
104         type: is_keyword(id) ? 'kw' : 'var',

```

```

105         value: id
106     }
107 }
108 // 读取带反编译\\的字符串
109 function read_escaped(end) {
110     var escaped = false, str = '';
111     input.next();
112     while (!input.eof()) {
113         let ch = input.next();
114         if (escaped) {
115             str += ch;
116             escaped = false;
117         } else if (ch === '\\') {
118             escaped = true;
119         } else if (ch === end) {
120             break;
121         } else {
122             str += ch;
123         }
124     }
125     return str;
126 }
127
128 function read_string() {
129     return { type: 'str', value: read_escaped('') };
130 }
131
132 function skip_comment() {
133     read_while(function (ch) { return ch !== '\n' });
134     input.next();
135 }
136 }
137 module.exports = TokenStream;

```

2.3.3 总结

词法分析器会根据我们所定义的规则去生成token，如果我们写的代码不符合词法，它会提示出现无法解析的字符。

2.4 将Token转化为AST

2.4.1 规则

- 我们需要给AST定义相应的数据结构
 - 首先整个AST是树形结构
 - 每个节点可以是不同的类型，包含了不同的属性
 - 叶子节点一定是最终有具体值的节点类型，不会再嵌套AST

JSON

```
1 num { type: "num", value: NUMBER }
2 str { type: "str", value: STRING }
3 bool { type: "bool", value: 真 or 假 }
4 var { type: "var", value: NAME }
5 // 函数, 其body是另一颗AST树
6 func { type: "func", vars: [ NAME... ], body: AST }
7 call { type: "call", func: AST, args: [ AST... ] }
8 if { type: "if", cond: AST, then: AST, else: AST }
9 assign { type: "assign", operator: "为", left: AST, right: AST }
10 binary { type: "binary", operator: OPERATOR, left: AST, right: AST }
11 prog { type: "prog", prog: [ AST... ] }
12 let { type: "let", vars: [ VARS... ], body: AST }
```

- 不同的操作符有不同的权重

JavaScript

```
1 const PRECEDENCE = {
2   "为": 1,
3   "等于": 1.5,
4   "或": 2,
5   "并": 3,
6   "加": 10, "减": 10,
7   "乘": 20, "除": 20, "模": 20,
8 };
```

2.4.2 代码

JavaScript

```
1 const FALSE = { type: "bool", value: false };
2 // 操作符号权重
3
4 const PRECEDENCE = {
5   "为": 1,
6   "等于": 1.5,
```



```

7     "或": 2,
8     "并": 3,
9     "加": 10, "减": 10,
10    "乘": 20, "除": 20, "模": 20,
11 };
12
13 // 解析token
14 function parse(input) {
15     return parse_toplevel();
16     // 解析整个程序 根节点prog
17     function parse_toplevel() {
18         const prog = [];
19         while (!input.eof()) {
20             prog.push(parse_expression());
21             // 表达式以分号分割
22             if (!input.eof()) skip_punc("; ");
23         }
24         return { type: 'prog', prog };
25     }
26     // 解析表达式,
27     // 表达式可能是调用, 可能是定义。maybe_call
28     // 表达式可能是多个子表达式的集合。maybe_binary
29     function parse_expression() {
30         return maybe_call(function () {
31             return maybe_binary(parse_atom(), 0);
32         });
33     }
34     // 解析调用表达式
35     function maybe_call(exp) {
36         const expr = exp();
37         // 如果下一个token是 (
38         return is_punc('(') ? parse_call(expr) : expr;
39     }
40
41     // 构建调用表达式类型
42     function parse_call(func) {
43         return {
44             type: 'call',
45             func,
46             args: delimited('(', ')', ' ', ' ', parse_expression)
47         };
48     }
49     // 解析子表达式后带操作符的表达式
50     function maybe_binary(left, my_prec) {

```

```

51     const tok = is_op();
52
53     if (tok) {
54         const next_prec = PRECEDENCE[tok.value];
55         if (next_prec > my_prec) {
56             input.next();
57             return maybe_binary({
58                 type: tok.value === '为' ? 'assign' : 'binary',
59                 operator: tok.value,
60                 left,
61                 right: maybe_binary(parse_atom(), next_prec)
62             }, my_prec)
63         }
64     }
65     return left;
66 }
67
68
69 // 分解原子表达式
70 function parse_atom() {
71     return maybe_call(function () {
72         if (is_punc(' ( ')) {
73             input.next();
74             const exp = parse_expression();
75             skip_punc(' ' ');
76             return exp;
77         }
78         if (is_punc(' [ ')) {
79             return parse_prog();
80         }
81         if (is_kw('定义')) return parse_let();
82         if (is_kw('如果')) {
83             return parse_if();
84         }
85         if (is_kw('真') || is_kw('假')) return parse_bool();
86         if (is_kw('函数')) {
87             input.next();
88             return parse_func();
89         }
90         const tok = input.next();
91
92         if (tok.type === 'var' || tok.type === 'num' || tok.type === 'str')
93             return tok;
94     }

```

```

95         unexpected();
96     })
97 }
98
99 // 分解块
100 function parse_prog() {
101     const prog = delimited('「', '」', ';', parse_expression);
102     if (prog.length === 0) return FALSE;
103     if (prog.length === 1) return prog[0];
104     return { type: 'prog', prog };
105 }
106
107 // 分解if语句
108 function parse_if() {
109     skip_kw('如果');
110     const cond = parse_expression();
111     if (!is_punc('「')) skip_kw('则');
112     const then = parse_expression();
113     const ret = {
114         type: 'if',
115         cond,
116         then
117     }
118     if (is_kw('否则')) {
119         input.next();
120         ret.else = parse_expression();
121     }
122     return ret;
123 }
124
125 // 解析bool
126 function parse_bool() {
127     return {
128         type: 'bool',
129         value: input.next().value === '真'
130     }
131 }
132
133 // 解析函数
134 function parse_func() {
135     return {
136         type: 'func',
137         name: input.peek().type === 'var' ? input.next().value : null,
138         vars: delimited('(', ')', ',', parse_varname),
139         body: parse_expression()
140     }
141 }

```

```

139         body: parse_expression()
140     }
141 }
142
143 // 解析参数
144 function parse_varname() {
145     const name = input.next();
146     if (name.type !== 'var') input.croak('此处期望变量名');
147     return name.value;
148 }
149
150 // 解析定义
151 function parse_let() {
152     skip_kw('定义');
153     if (input.peek().type === 'var') {
154         const name = input.next().value;
155         const defs = delimited(" (", " ) ", " ", " ", parse_vardef);
156         return {
157             type: 'call',
158             func: {
159                 type: 'func',
160                 name,
161                 vars: defs.map((def) => def.name),
162                 body: parse_expression()
163             },
164             args: defs.map((def) => def.def || FALSE)
165         }
166     }
167     return {
168         type: 'let',
169         vars: delimited(' (', ' ) ', ' ', ' ', parse_vardef),
170         body: parse_expression()
171     }
172 }
173
174 // 解析 定义 的变量列表
175 function parse_vardef() {
176     let name = parse_varname(), def;
177     if (is_op('为')) {
178         input.next();
179         def = parse_expression();
180     }
181     return { name, def };
182 }
183

```

```

184 // 以start开始, end 结尾, separator分割, 被parser后的连续token 的AST
185 function delimited(start, end, separator, parser) {
186     const a = [];
187     let first = true;
188     // 删除开始的符号
189     skip_punc(start);
190     // token 数组里还有值
191     while (!input.eof()) {
192         // 碰到关闭符号
193         if (is_punc(end)) break;
194         // 开始符号后面的第一个token不会是分隔符
195         if (first) {
196             first = false;
197         } else {
198             skip_punc(separator);
199         }
200         if (is_punc(end)) {
201             break;
202         }
203         // 将不是符号的token解析后塞入结果
204         a.push(parser())
205     }
206     skip_punc(end);
207     return a;
208 }
209
210
211 function is_punc(ch) {
212     const tok = input.peek();
213     return tok && tok.type === 'punc' && (!ch || tok.value === ch) && tok;
214 }
215
216 function skip_punc(ch) {
217     if (is_punc(ch)) {
218         input.next();
219     } else {
220         input.croak(`期望的符号是 "${ch}"`)
221     }
222 }
223
224 function is_op(op) {
225     const tok = input.peek();
226     return tok && tok.type === 'op' && (!op || tok.value === op) && tok;
227 }
228

```

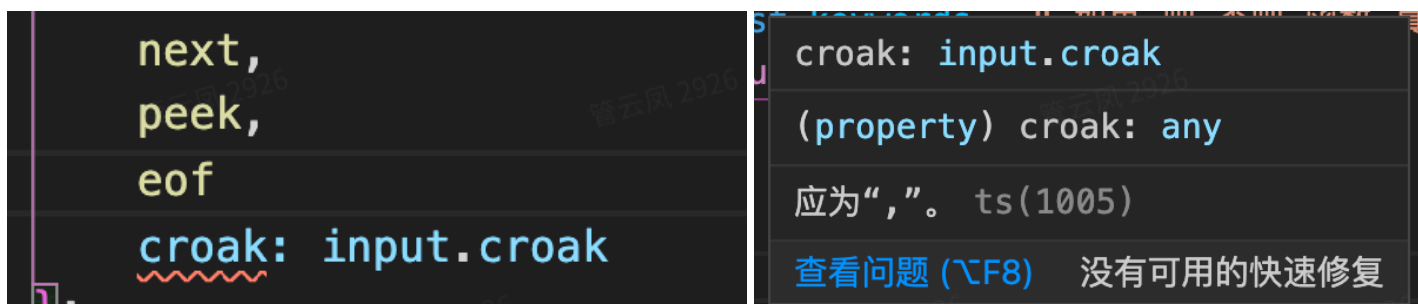
```

229     function is_kw(kw) {
230         const tok = input.peek();
231         return tok && tok.type === 'kw' && (!kw || tok.value === kw) && tok;
232     }
233
234     function skip_kw(kw) {
235         if (is_kw(kw)) { input.next(); }
236         else input.croak(`期望的关键字是:${kw}`)
237     }
238
239     function skip_op(op) {
240         if (is_op(op)) {
241             input.next();
242         } else {
243             input.croak(`期望的操作是${op}`)
244         }
245     }
246
247     function unexpected() {
248         input.croak(`不被期望出现的token` + JSON.stringify(input))
249     }
250
251     module.exports = parse

```

2.4.3 总结

- 我们先定义语法规则，再根据它，在转化Token为AST时忽略一些符号Token、解析有内容的Token生成AST的叶子节点，组合成一棵AST子树等等最终生成整个AST树。
- 每次让Token流“指针”向后移动的时机在不同类型的AST节点的解析器中是不一样的，依赖于语法规则以及当前指针的位置。
- 转化为AST是静态的语法分析。在日常开发中，我们在vscode中编写代码，在文件更新后，就会触发语法解析器parse，如果写错了语法立刻就会有错误提示。



- 对于上面2.1的“中语言”例子，我们会生成什么AST呢？

JavaScript

```
1  const chineseAdd = `相加 为 函数 (甲, 乙) 甲 加 乙; 打印 (相加 (3, 4))`;
2
3  const chineseVarAdd = `定义 (甲 为 2, 乙 为 甲 加 7, 丙 为 甲 加 乙) 打印 (甲 加 乙 加 丙)`;
4  const chineseIf = `如果 (2 减 1 等于 1) 「打印 (666)」`;
```

JSON

```
1  // chineseAdd
2  {
3      "type": "prog",
4      "prog": [
5          {
6              "type": "assign",
7              "operator": "为",
8              "left": {
9                  "type": "var",
10                 "value": "相加"
11             },
12             "right": {
13                 "type": "func",
14                 "name": null,
15                 "vars": [
16                     "甲",
17                     "乙"
18                 ],
19                 "body": {
20                     "type": "binary",
21                     "operator": "加",
22                     "left": {
23                         "type": "var",
24                         "value": "甲"
25                     },
26                     "right": {
27                         "type": "var",
28                         "value": "乙"
29                     }
30                 }
31             }
32         },
33         {
34             "type": "call",
```

```

35     "func": {
36         "type": "var",
37         "value": "打印"
38     },
39     "args": [
40         {
41             "type": "call",
42             "func": {
43                 "type": "var",
44                 "value": "相加"
45             },
46             "args": [
47                 {
48                     "type": "num",
49                     "value": 3
50                 },
51                 {
52                     "type": "num",
53                     "value": 4
54                 }
55             ]
56         }
57     ]
58 }
59 ]
60 }

```

```

62 // chineseVarAdd
63 {
64     "type": "prog",
65     "prog": [
66         {
67             "type": "let",
68             "vars": [
69                 {
70                     "name": "甲",
71                     "def": {
72                         "type": "num",
73                         "value": 2
74                     }
75                 },
76                 {
77                     "name": "乙",
78                     "def": {

```

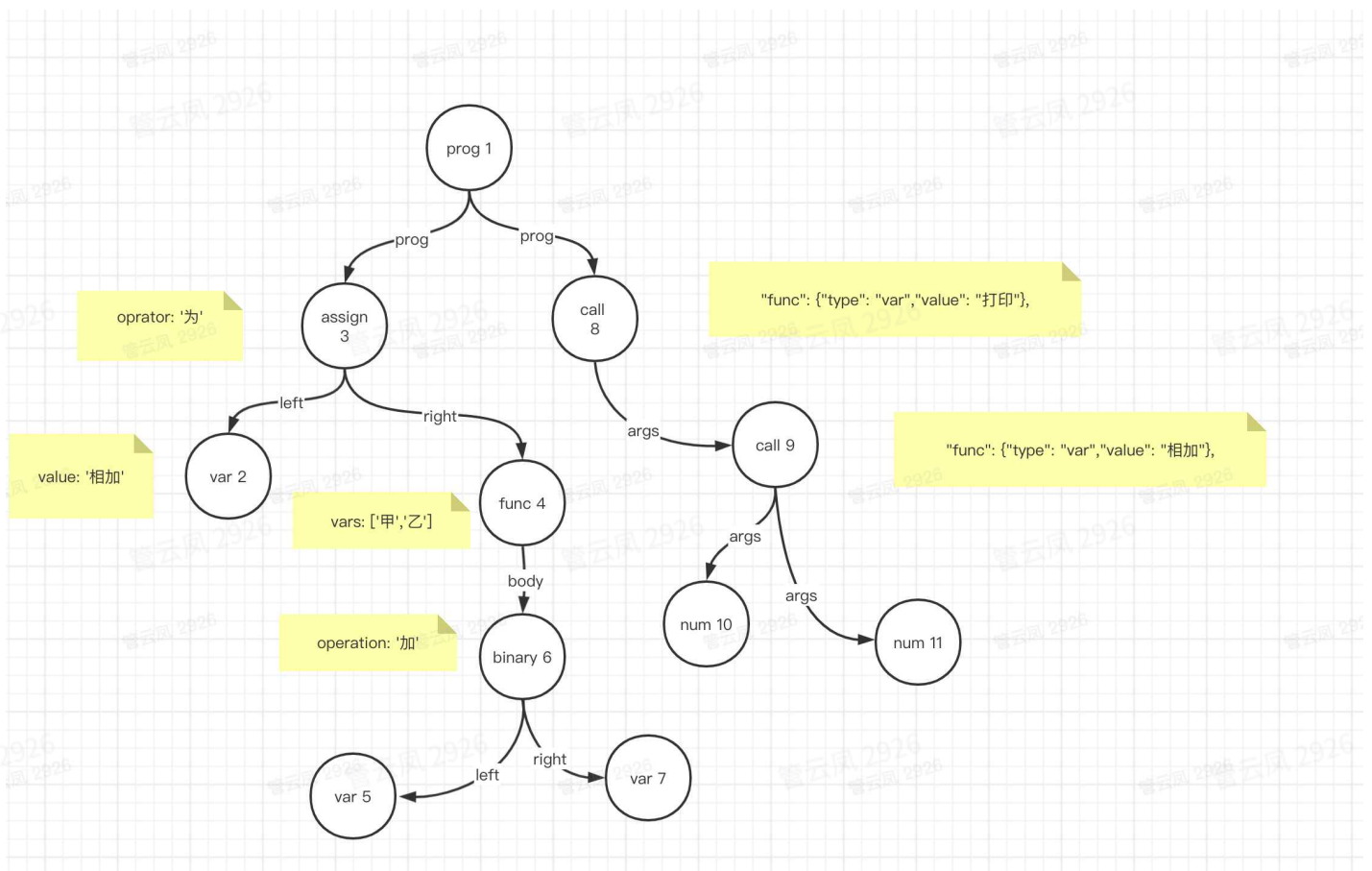


```

124         "right": {
125             "type": "var",
126             "value": "乙"
127         }
128     },
129     "right": {
130         "type": "var",
131         "value": "丙"
132     }
133 }
134 ]
135 }
136 }
137 ]
138 }
139

```

- 我没有写chineself的AST，既然我们知道了生成AST的逻辑，那我认为现在应该能看着“中语言”代码就可以在脑中生成对应的AST了。根据下面这张图的构造顺序(chineseAdd)，我们可以总结一下AST的生成规律：



- 1. 从左往右读取的流式token，token的顺序是节点解析的顺序
- 2. 利用了递归来构造ast的子树，每个节点根据其type具有特殊的属性

那么chineseIf 可以这么解析

JavaScript

```
1  const chineseIf = `如果 (2 减 1 为 1) 「打印 (666) 」`
2
3  // 生成的token, 简单描述如下
4  kw: '如果'
5  punc: '(',
6  num: 2,
7  op: 减,
8  num: 1,
9  op: '为',
10 num: 1 ,
11 punc: ') ',
12 punc: '「',
13 var: '打印',
14 punc: '(',
15 num: 666 ,
16 punc: ') '
17 punc: '」',
18
19 // 思考一下
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
```

```

37 // 生成的AST
38 {
39     "type": "prog",
40     "prog": [
41         {
42             "type": "if",
43             "cond": {
44                 "type": "binary",
45                 "operator": "等于",
46                 "left": {
47                     "type": "binary",
48                     "operator": "减",
49                     "left": {
50                         "type": "num",
51                         "value": 2
52                     },
53                     "right": {
54                         "type": "num",
55                         "value": 1
56                     }
57                 },
58                 "right": {
59                     "type": "num",
60                     "value": 1
61                 }
62             },
63             "then": {
64                 "type": "call",
65                 "func": {
66                     "type": "var",
67                     "value": "打印"
68                 },
69                 "args": [
70                     {
71                         "type": "num",
72                         "value": 666
73                     }
74                 ]
75             }
76         }
77     ]
78 }

```

2.5 将AST表示的数据结构用JS运行

对于简单的操作符号，我们可以直接根据节点的type, value, operator等属性进行JS代码转化操作。但是对于定义、函数、函数参数等复杂一点的节点需要有作用域的概念，否则如果所有的参数都挂在一个作用域下，就会导致覆盖、参数混乱的问题。

2.5.1 定义作用域类

JavaScript

```
1  function Environment(parent) {
2      this.vars = Object.create(parent ? parent.vars : null);
3      this.parent = parent;
4  }
5
6  Environment.prototype = {
7      extend: function () {
8          // 返回复制当前作用域的变量、和当前作用域为父作用域的作用域
9          return new Environment(this);
10     },
11     lookup: function (name) {
12         // 在作用域链中查找
13         let scope = this;
14         while (scope) {
15             if (Object.prototype.hasOwnProperty.call(scope.vars, name)) {
16                 return scope;
17             }
18             scope = scope.parent;
19         }
20     },
21     // 只获取当前作用域变量
22     get: function (name) {
23         if (name in this.vars) {
24             return this.vars[name];
25         }
26         throw new Error(`未定义的变量: ${name}`);
27     },
28     set: function (name, value) {
29         // 如果在作用域链中已经有这个变量，则覆盖
30         const scope = this.lookup(name);
31         // 不允许在嵌套环境内定义全局变量
32         if (!scope && this.parent) {
33             throw new Error(`未定义的变量: ${name}`);
34         }
35     }
36 }
```

```

35         return (scope || this).vars[name] = value;
36     },
37     // 只定义为当前作用域变量
38     def: function (name, value) {
39         return this.vars[name] = value
40     }
41
42 }
43
44
45 module.exports = Environment;

```

2.5.2 执行JS代码

- 在evaluate函数中需要传入当前需要执行的表达式和作用域
- 如果表达式的类型可以获取到具体的值就直接返回
- 如果表达式的值是比较复杂的类型就进行分解再递归执行

JavaScript

```

1  function evaluate(exp, env) {
2      switch (exp.type) {
3          case "num":
4          case "str":
5          case "bool":
6              return exp.value;
7              // 从作用域取得变量
8          case 'var':
9              return env.get(exp.value);
10             // 将作用域内左边的变量名设置为右边的表达式的值
11          case 'assign':
12              return env.set(exp.left.value, evaluate(exp.right, env));
13              // 将左边的表达式和右边的表达式用operator操作
14          case 'binary':
15              return apply_op(exp.operator, evaluate(exp.left, env),
16              evaluate(exp.right, env));
17              // 将func类型的节点转化成一个函数
18          case 'func':
19              return make_func(env, exp);
20              // 给每个变量定义创建一个新的作用域
21              // {定义 甲 为 1; {定义 乙 为 2}}
22          case "let":
23              exp.vars.forEach(function (v) {
24                  const scope = env.extend();

```

```

24         scope.def(v.name, v.def ? evaluate(v.def, env) : false);
25         env = scope;
26     });
27     return evaluate(exp.body, env);
28     // 如果条件成立就执行then, 不成立就执行else
29     case 'if':
30         const cond = evaluate(exp.cond, env);
31         if (cond !== false) return evaluate(exp.then, env);
32         return exp.else ? evaluate(exp.else, env) : false;
33         // 根节点的每个子节点都是一个表达式
34     case 'prog':
35         let val = false;
36         exp.prog.forEach(function (exp) {
37             val = evaluate(exp, env);
38         });
39         return val;
40         // 函数调用节点需要执行函数、其参数也可能是表达式, 所有也要先递归执行参数
41     case "call":
42         const func = evaluate(exp.func, env);
43         return func.apply(null, exp.args.map(function (arg) {
44             return evaluate(arg, env);
45         }));
46     default:
47         throw new Error("无法执行" + exp.type)
48 }
49 }
50
51 function apply_op(op, a, b) {
52     function num(x) {
53         if (typeof x !== 'number') {
54             throw new Error('希望获取到数字但是获取到' + x);
55         }
56         return x;
57     }
58
59     function div(x) {
60         if (num(x) === 0) {
61             throw new Error("除以0")
62         }
63         return x;
64     }
65
66     switch (op) {
67         case "加": return num(a) + num(b);
68         case "减": return num(a) - num(b);

```

```

68     case "减": return num(a) - num(b);
69     case "乘": return num(a) * num(b);
70     case "除": return num(a) / div(b);
71     case "模": return num(a) % div(b);
72     case "并": return a !== false && b;
73     case "或": return a !== false ? a : b;
74     case '等于': return a == b;
75   }
76   throw new Error("不能操作的符号" + op);
77 }
78
79 function make_func(env, exp) {
80   // 如果这个函数是有name的, 则在当前作用域下定义这个name为函数本身, 为了实现自己调用自己
81   if (exp.name) {
82     env = env.extend();
83     env.def(exp.name, func);
84   }
85   // 新建一个函数作用域, 将函数的参数名作为函数的变量
86   function func() {
87     const names = exp.vars;
88     const scope = env.extend();
89     for (let i = 0; i < names.length; i++) {
90       scope.def(names[i], i < arguments.length ? arguments[i] : false);
91     }
92     // 最后将新的作用域用于执行函数体
93     return evaluate(exp.body, scope);
94   }
95   return func;
96 }

```

2.6 将AST转换为JS再执行

- 对于简单的基础数据节点类型, 我们只需要将数据在JSON.stringify后直接返回
- 对于复杂数据节点类型, 需要转化为对应的JS代码
 - 带操作符(type: binary || assign)的表达式转化为(js(变量) 操作 js(变量))
 - 定义(表达式(type: let)转化为立即执行函数, 将当前定义变量名作为参数名, 后面的变量定义作为函数体; 这样就可以满足“let会定义新的作用域和变量”的功能。例如:

JavaScript

```

1
2 const chineseVarAdd = `定义 (甲 为 2, 乙 为 甲 加 7, 丙 为 甲 加 乙) 打印 (甲 加 乙 加

```



```

丙) `;
3 // 实际上
4 let a = 1, b = a + 1, c = a + b;
5 // 等于
6 (function (a) {
7     return (function(b) {
8         return (function(c) {
9             return c;
10         })(a+b))
11     })(a+1))
12 }(1))
13
14 // 解析整个表达式 将let ast将转化为call ast
15 {
16     "type": "call",
17     "func": {
18         "type": "func",
19         "vars": [
20             "甲"
21         ],
22         "body": {
23             "type": "let",
24             "vars": [
25                 {
26                     "name": "乙",
27                     "def": {
28                         "type": "binary",
29                         "operator": "加",
30                         "left": {
31                             "type": "var",
32                             "value": "甲"
33                         },
34                         "right": {
35                             "type": "num",
36                             "value": 7
37                         }
38                     }
39                 },
40                 {
41                     "name": "丙",
42                     "def": {
43                         "type": "binary",
44                         "operator": "加",
45                         "left": {
46                             "type": "var",

```

```

47         "value": "甲"
48     },
49     "right": {
50         "type": "var",
51         "value": "乙"
52     }
53 }
54 }
55 ],
56 "body": {
57     "type": "call",
58     "func": {
59         "type": "var",
60         "value": "打印"
61     },
62     "args": [
63         {
64             "type": "binary",
65             "operator": "加",
66             "left": {
67                 "type": "binary",
68                 "operator": "加",
69                 "left": {
70                     "type": "var",
71                     "value": "甲"
72                 },
73                 "right": {
74                     "type": "var",
75                     "value": "乙"
76                 }
77             },
78             "right": {
79                 "type": "var",
80                 "value": "丙"
81             }
82         }
83     ]
84 }
85 }
86 },
87 "args": [
88     {
89         "type": "num",
90         "value": 2
91     }

```

```

91     }
92 ]
93 }
94
95 // 解析乙
96 {
97     "type": "call",
98     "func": {
99         "type": "func",
100         "vars": [
101             "乙"
102         ],
103         "body": {
104             "type": "let",
105             "vars": [
106                 {
107                     "name": "丙",
108                     "def": {
109                         "type": "binary",
110                         "operator": "加",
111                         "left": {
112                             "type": "var",
113                             "value": "甲"
114                         },
115                         "right": {
116                             "type": "var",
117                             "value": "乙"
118                         }
119                     }
120                 }
121             ],
122             "body": {
123                 "type": "call",
124                 "func": {
125                     "type": "var",
126                     "value": "打印"
127                 },
128                 "args": [
129                     {
130                         "type": "binary",
131                         "operator": "加",
132                         "left": {
133                             "type": "binary",
134                             "operator": "加",
135                             "left": {

```

```

136         "type": "var",
137         "value": "甲"
138     },
139     "right": {
140         "type": "var",
141         "value": "乙"
142     }
143 },
144 "right": {
145     "type": "var",
146     "value": "丙"
147 }
148 }
149 ]
150 }
151 }
152 },
153 "args": [
154     {
155         "type": "binary",
156         "operator": "加",
157         "left": {
158             "type": "var",
159             "value": "甲"
160         },
161         "right": {
162             "type": "num",
163             "value": 7
164         }
165     }
166 ]
167 }

```

168 // 解析丙

```

170 {
171     "type": "call",
172     "func": {
173         "type": "func",
174         "vars": [
175             "丙"
176         ],
177         "body": {
178             "type": "let",
179             "vars": [],
180             "body": {

```

```

180     body: [
181         {
182             "type": "call",
183             "func": {
184                 "type": "var",
185                 "value": "打印"
186             },
187             "args": [
188                 {
189                     "type": "binary",
190                     "operator": "加",
191                     "left": {
192                         "type": "binary",
193                         "operator": "加",
194                         "left": {
195                             "type": "var",
196                             "value": "甲"
197                         },
198                         "right": {
199                             "type": "var",
200                             "value": "乙"
201                         }
202                     },
203                     "right": {
204                         "type": "var",
205                         "value": "丙"
206                     }
207                 }
208             ]
209         }
210     ],
211     "args": [
212         {
213             "type": "binary",
214             "operator": "加",
215             "left": {
216                 "type": "var",
217                 "value": "甲"
218             },
219             "right": {
220                 "type": "var",
221                 "value": "乙"
222             }
223         }
224     ]

```

```

225 }
226
227 // 最终结果
228 (((function (甲) {return ((function (乙) {return ((function (丙) {return 打印
    (((甲+乙)+丙)) })((甲+乙))) })((甲+7))) })((2)))

```

- 函数表达式(type:func)将会拼接作用域符合"()"和参数符号，以及递归生成body的js代码
- 其他类型表达式都是根据js语法进行字符串拼接

完整代码：

JavaScript

```

1  const FALSE = { type: "bool", value: false };
2  function make_js(exp) {
3      return js(exp);
4
5      function js(exp) {
6          switch (exp.type) {
7              case "num":
8              case "str":
9              case "bool": return js_atom(exp);
10             case "var": return js_var(exp);
11             case "binary": return js_binary(exp);
12             case "assign": return js_assign(exp);
13             case "let": return js_let(exp);
14             case "func": return js_func(exp);
15             case "if": return js_if(exp);
16             case "prog": return js_prog(exp);
17             case "call": return js_call(exp);
18             default:
19                 throw new Error("Dunno how to make_js for " + JSON.stringify(exp));
20             }
21         }
22
23         function js_atom(exp) {
24             return JSON.stringify(exp.value);
25         }
26
27         function make_var(name) {
28             return name;
29         }
30
31         function js_var(exp) {

```

```

32     return make_var(exp.value);
33 }
34
35 function js_binary(exp) {
36     return "(" + js(exp.left) + transform_operator(exp.operator) + js(exp.ri
ght) + ")";
37 }
38
39 function transform_operator(operator) {
40     switch (operator) {
41         case '为': return '=';
42         case '加': return '+';
43         case "减": return '-';
44         case '乘': return '*';
45         case '除': return '/';
46         case '余': return '%';
47         case '等于': return '==';
48     }
49 }
50
51 function js_assign(exp) {
52     return js_binary(exp);
53 }
54
55 function js_func(exp) {
56     var code = "(function ";
57     if (exp.name)
58         code += make_var(exp.name);
59     code += "(" + exp.vars.map(make_var).join(", ") + ") {";
60     code += "return " + js(exp.body) + " }";
61     return code;
62 }
63
64 function js_let(exp) {
65     if (exp.vars.length == 0) {
66         return js(exp.body);
67     }
68     var iife = {
69         type: "call",
70         func: {
71             type: "func",
72             vars: [exp.vars[0].name],
73             body: {
74                 type: "let",

```

```

75         vars: exp.vars.slice(1),
76         body: exp.body
77     }
78 },
79     args: [exp.vars[0].def || FALSE]
80 };
81     return "(" + js(iife) + ")";
82 }
83
84     function js_if(exp) {
85         return "("
86             + js(exp.cond) + " !== false"
87             + " ? " + js(exp.then)
88             + " : " + js(exp.else || FALSE)
89             + ")";
90     }
91
92     function js_prog(exp) {
93         return "(" + exp.prog.map(js).join(", ") + ")";
94     }
95     function js_call(exp) {
96         return js(exp.func) + "(" + exp.args.map(js).join(", ") + ")";
97     }
98 }
99
100
101
102 module.exports = make_js

```

2.7 实践一下

打开vscode，进行演示

3. 基于JS代码执行 - 进阶

3.1 更高级的功能 - 回调

回调可以使得语言处理一些异步的操作，比如：

JavaScript

```
1 fs.readFile("file.txt", "utf8", function CC(error, data){});
2 // 第三个函数就是回调函数，其会让开发者在fs.readFile执行后得到错误或结果
```

如果需要我们的语言能够支持回调则需要更改evaluate函数里每个节点的返回逻辑。

- 给evaluate函数添加第三个参数，callback，callback的参数为执行结果。
- 对于简单的类型节点，直接将节点值作为callback函数的参数就可以。比如: num、str、bool、var。
- 对于部分复杂的类型节点，需要构造新的evaluate callback函数 来保证用户传入的callback得到的参数是最终答案。比如: assign、binary、if、func。
- 对于带参数的复杂类型节点，需要借用递归函数来构造新的上下文和callback。比如: let、prog、call。

完整代码：

JavaScript

```
1 function evaluate(exp, env, callback) {
2     switch (exp.type) {
3         case "num":
4         case "str":
5         case "bool":
6             return callback(exp.value);
7         case 'var':
8             return callback(env.get(exp.value));
9         case 'assign':
10            if (exp.left.type != "var")
11                throw new Error("Cannot assign to " + JSON.stringify(exp.left));
12            return evaluate(exp.right, env, function (right) { callback(env.set
13                (exp.left.value, right)) });
14        case 'binary':
15            evaluate(exp.left, env, function (left) {
16                evaluate(exp.right, env, function (right) {
17                    callback(apply_op(exp.operator, left, right));
18                });
19            });
20            return;
21        case 'func':
22            callback(make_func(env, exp));
23            return;
```

```

24     case "let":
25         // 当还有let变量未定义时用loop继续定义作用域变量
26         // 否则执行exp.body, exp.body执行时的scope已经记录好了所有的let变量
27         (function loop(env, i) {
28             if (i < exp.vars.length) {
29                 var v = exp.vars[i];
30                 if (v.def) evaluate(v.def, env, function (value) {
31                     var scope = env.extend();
32                     scope.def(v.name, value);
33                     loop(scope, i + 1);
34                 }); else {
35                     var scope = env.extend();
36                     scope.def(v.name, false);
37                     loop(scope, i + 1);
38                 }
39             } else {
40                 evaluate(exp.body, env, callback);
41             }
42         })(env, 0);
43         return;
44     case 'if':
45         evaluate(exp.cond, env, function (cond) {
46             if (cond !== false) evaluate(exp.then, env, callback);
47             else if (exp.else) evaluate(exp.else, env, callback);
48             else callback(false);
49         });
50         return;
51     case 'prog':
52
53         (function loop(last, i) {
54             if (i < exp.prog.length) evaluate(exp.prog[i], env, function (val) {
55                 loop(val, i + 1);
56             }); else {
57                 callback(last);
58             }
59         })(false, 0);
60         return;
61     case "call":
62
63         evaluate(exp.func, env, function (func) {
64             (function loop(args, i) {
65                 if (i < exp.args.length) evaluate(exp.args[i], env, function
66                 (arg) {
67                     args[i + 1] = arg;

```

```

67         loop(args, i + 1);
68     }); else {
69         func.apply(null, args);
70     }
71     })([callback], 0);
72 });
73 return;
74 default:
75     throw new Error("无法执行" + exp.type)
76 }
77 }
78
79 function apply_op(op, a, b) {
80     function num(x) {
81         if (typeof x !== 'number') {
82             throw new Error('希望获取到数字但是获取到' + x);
83         }
84         return x;
85     }
86
87     function div(x) {
88         if (num(x) === 0) {
89             throw new Error("除以0")
90         }
91         return x;
92     }
93
94     switch (op) {
95         case "加": return num(a) + num(b);
96         case "减": return num(a) - num(b);
97         case "乘": return num(a) * num(b);
98         case "除": return num(a) / div(b);
99         case "模": return num(a) % div(b);
100        case "并": return a !== false && b;
101        case "或": return a !== false ? a : b;
102        case '等于': return a == b;
103    }
104    throw new Error("不能操作的符号" + op);
105 }
106
107
108 function make_func(env, exp) {
109     function func(callback) {
110         const names = exp.vars;

```

```

111     const scope = env.extend();
112     for (let i = 0; i < names.length; i++) {
113         scope.def(names[i], i + 1 < arguments.length ? arguments[i + 1] : false);
114     }
115     evaluate(exp.body, scope, callback);
116 };
117 return func;
118 }
119
120
121 module.exports = evaluate;

```

执行结果：

JavaScript

```

1  const InputStream = require('./InputStream');
2  const parse = require('./parse');
3  const TokenStream = require('./TokenStream');
4  const Environment = require('./environment');
5  const evaluate = require('./cps-evaluate');
6  const makeJs = require('./generator')
7
8
9  const code = `相加 为 函数 (甲, 乙) 甲 加 乙; 打印 (相加 (3, 4))`;
10 const ast = parse(TokenStream(InputStream(code)));
11 const globalEnv = new Environment();
12
13 globalEnv.def("打印", function (callback, txt) {
14     console.log(txt);
15     callback(txt);
16 });
17
18
19 evaluate(ast, globalEnv, function (result) {
20     console.log("result", result);
21 });
22
23 // 输出
24 // 7
25 // result 7

```

3.2 递归栈溢出 - 清除栈

现在我们可以给函数定义名字，那是否可以实现函数自己调用自己（递归）呢？

定义一个斐波那契数列函数：

TypeScript

```
1 const code = `斐波那契数列 为 函数 (n) 如果 n 小于 2 则 n 否则 斐波那契数列 (n 减 1)
  加 斐波那契数列 (n 减 2) ; 时间 (函数 () 打印 (斐波那契数列 (20) )) `;
```

执行函数命令行报错：maximum call stack即执行栈溢出。

```
guanyunfeng@C02XH711JG5L chinese-language % node cps-index
/Users/guanyunfeng/github/learning/projects/self/ast/chinese-language/environment.js:19
  get: function (name) {
    ^
RangeError: Maximum call stack size exceeded
    at Environment.get (/Users/guanyunfeng/github/learning/projects/self/ast/chinese-language/environment.js:19:19)
    at evaluate (/Users/guanyunfeng/github/learning/projects/self/ast/chinese-language/cps-evaluate.js:8:33)
    at evaluate (/Users/guanyunfeng/github/learning/projects/self/ast/chinese-language/cps-evaluate.js:15:13)
    at evaluate (/Users/guanyunfeng/github/learning/projects/self/a
```

如果一个函数调用另一个函数，JS引擎就会新建一个调用帧，并且把它放在调用栈的顶部，只有这个函数运行完并且其内部变量不被其他函数引用，才会从栈顶退出，再从内存里清除。

对于一次斐波那契数列函数 evaluate 调用，其调用栈的记录如下：

Apache

```
1  fib(20)
2  ## stack
3  evaluate(fib(20), k0)
4  evaluate(fib, K1)
5  evaluate(n 小于 2, K2)
6  evaluate(n, K3)
7  K3(n)
8  evaluate(2, K4)
9  K4(2)
10 evaluate(fib(19), k5)
11     ## 循环上面的步骤
12     ## 每个阶段都会引用到之前的fib结果
13
14
15 ## 更简单一点的, 直接到底
16 print(1 + 2 * 3);
17 ## stack:
18 ## Kn 代表一个栈帧
19 evaluate( print(1 + 2 * 3), K0 )
20 evaluate( print, K1 )
21 K1(print) # 可以从K1中获取到print, 则直接返回
22 evaluate( 1 + 2 * 3, K2 )
23 evaluate( 2 * 3, K3 )
24 evaluate( 2, K4 )
25 K4(2) # 2 是 常量, 直接返回
26 evaluate( 3, K5 ) # 3 是 常量, 直接返回
27 K5(3)
28 K3(6) # 返回 2 * 3
29 evaluate( 1, K6 ) # 1 是 常量, 直接返回
30 K6(1)
31 K2(7) # 返回 1+2*3
32 print(K0, 7) # 调用print
33 K0(false) # 结束顶级上下文
```

可以看出, evaluate 函数存在严重的性能问题。那么应该如何防止调用栈溢出呢?

我们可以记录当前是已经调用过多少次 内部函数 (evaluate), 如果超过了固定的最大栈值, 我们就抛出一个异常, 创建一个新的函数, 再重新调用, 这样就会生成新的调用栈, 之前的调用栈会被回收, 从而不会出现栈溢出。

关键代码:

JavaScript

```
1  var STACKLEN;
2  function GUARD(f, args) {
3      if (--STACKLEN < 0) throw new Continuation(f, args);
4  }
5
6  function Continuation(f, args) {
7      this.f = f;
8      this.args = args;
9  }
10
11 function Execute(f, args) {
12     while (true) try {
13         STACKLEN = 200;
14         return f.apply(null, args);
15     } catch (ex) {
16         if (ex instanceof Continuation) {
17             f = ex.f, args = ex.args;
18         }
19
20         else throw ex;
21     }
22 }
23
```

完整代码

JavaScript

```
1
2  const InputStream = require('./InputStream');
3  const parse = require('./parse');
4  const TokenStream = require('./TokenStream');
5  const Environment = require('./environment');
6
7  function evaluate(exp, env, callback) {
8      // 记录调用
9      GUARD(evaluate, arguments);
10     switch (exp.type) {
11         case "num":
12         case "str":
13         case "bool":
14             callback(exp.value);
15     }
16 }
```

```

15         return;
16
17     case "var":
18         callback(env.get(exp.value));
19         return;
20
21     case "assign":
22         if (exp.left.type != "var")
23             throw new Error("Cannot assign to " + JSON.stringify(exp.left));
24         evaluate(exp.right, env, function CC(right) {
25             GUARD(CC, arguments);
26             callback(env.set(exp.left.value, right));
27         });
28         return;
29
30     case "binary":
31         evaluate(exp.left, env, function CC(left) {
32             GUARD(CC, arguments);
33             evaluate(exp.right, env, function CC(right) {
34                 GUARD(CC, arguments);
35                 callback(apply_op(exp.operator, left, right));
36             });
37         });
38         return;
39
40     case "let":
41         (function loop(env, i) {
42             GUARD(loop, arguments);
43             if (i < exp.vars.length) {
44                 var v = exp.vars[i];
45                 if (v.def) evaluate(v.def, env, function CC(value) {
46                     GUARD(CC, arguments);
47                     var scope = env.extend();
48                     scope.def(v.name, value);
49                     loop(scope, i + 1);
50                 }); else {
51                     var scope = env.extend();
52                     scope.def(v.name, false);
53                     loop(scope, i + 1);
54                 }
55             } else {
56                 evaluate(exp.body, env, callback);
57             }
58         })(env, 0);

```



```

59         return;
60
61     case "func":
62         callback(make_func(env, exp));
63         return;
64
65     case "if":
66         evaluate(exp.cond, env, function CC(cond) {
67             GUARD(CC, arguments);
68             if (cond !== false) evaluate(exp.then, env, callback);
69             else if (exp.else) evaluate(exp.else, env, callback);
70             else callback(false);
71         });
72         return;
73
74     case "prog":
75         (function loop(last, i) {
76             GUARD(loop, arguments);
77             if (i < exp.prog.length) evaluate(exp.prog[i], env, function CC(
val) {
78                 GUARD(CC, arguments);
79                 loop(val, i + 1);
80             }); else {
81                 callback(last);
82             }
83         })(false, 0);
84         return;
85
86     case "call":
87         evaluate(exp.func, env, function CC(func) {
88             GUARD(CC, arguments);
89             (function loop(args, i) {
90                 GUARD(loop, arguments);
91                 if (i < exp.args.length) evaluate(exp.args[i], env, function
CC(arg) {
92                     GUARD(CC, arguments);
93                     args[i + 1] = arg;
94                     loop(args, i + 1);
95                 }); else {
96                     func.apply(null, args);
97                 }
98             })([callback], 0);
99         });
100         return;
101

```

```

102         default:
103             throw new Error("I don't know how to evaluate " + exp.type);
104     }
105 }
106 function apply_op(op, a, b) {
107     function num(x) {
108         if (typeof x !== 'number') {
109             throw new Error('希望获取到数字但是获取到' + x);
110         }
111         return x;
112     }
113
114     function div(x) {
115         if (num(x) === 0) {
116             throw new Error("除以0")
117         }
118         return x;
119     }
120
121     switch (op) {
122         case "加": return num(a) + num(b);
123         case "减": return num(a) - num(b);
124         case "乘": return num(a) * num(b);
125         case "除": return num(a) / div(b);
126         case "模": return num(a) % div(b);
127         case "并": return a !== false && b;
128         case "或": return a !== false ? a : b;
129         case '等于': return a == b;
130         case "小于": return a < b;
131     }
132     throw new Error("不能操作的符号" + op);
133 }
134 function make_func(env, exp) {
135     if (exp.name) {
136         env = env.extend();
137         env.def(exp.name, func);
138     }
139     function func(callback) {
140         GUARD(func, arguments);
141         var names = exp.vars;
142         var scope = env.extend();
143         for (var i = 0; i < names.length; ++i)
144             scope.def(names[i], i + 1 < arguments.length ? arguments[i + 1] : false);
145         evaluate(exp.body, scope, callback);

```

```

145     evaluate(exp.body, scope, callback);
146   }
147   return func;
148 }
149
150 var STACKLEN;
151 function GUARD(f, args) {
152   if (--STACKLEN < 0) throw new Continuation(f, args);
153 }
154
155 function Continuation(f, args) {
156   this.f = f;
157   this.args = args;
158 }
159
160 function Execute(f, args) {
161   while (true) try {
162     STACKLEN = 200;
163     return f.apply(null, args);
164   } catch (ex) {
165     if (ex instanceof Continuation) {
166       f = ex.f, args = ex.args;
167     }
168
169     else throw ex;
170   }
171 }
172 const globalEnv = new Environment();
173
174
175
176 const code = `斐波那契数列 为 函数 (n) 如果 n 小于 2 则 n 否则 斐波那契数列 (n 减 1)
  加 斐波那契数列 (n 减 2) ; 时间 (函数 () 打印 (斐波那契数列 (20) )) `
177 const token = TokenStream(InputStream(code));
178 const ast = parse(token);
179
180 globalEnv.def("打印", function (k, val) {
181   console.log(val)
182   k(false);
183 });
184
185 globalEnv.def("时间", function (k, func) {
186   console.time("time");
187   func(function (ret) {
188     console.timeEnd("time");

```

```
189         k(ret);
190     });
191 });
192
193
194 console.log('ast', JSON.stringify(ast));
195 Execute(evaluate, [ast, globalEnv, function (result) {
196     console.log("*** Result:", result);
197 }]);
198
```

运行结果：

```
guanyunfeng@C02XH711JG5L chinese-language % node cc
6765
time: 148.924ms
*** Result: false
```

4. 完整的AST

到目前为止比较完整的AST节点类型可以参考我们都在用的babel-parser，其包括了：

- Identifier
- PrivateName
- Literals
 - RegExpLiteral
 - NullLiteral
 - StringLiteral
 - BooleanLiteral
 - NumericLiteral
- Programs
- Functions
- Statements
 - ExpressionStatement
 - BlockStatement
 - EmptyStatement

- DebuggerStatement
- WithStatement
- Control flow
 - ReturnStatement
 - LabeledStatement
 - BreakStatement
 - ContinueStatement
- Choice
 - IfStatement
 - SwitchStatement
 - SwitchCase
- Exceptions
 - ThrowStatement
 - TryStatement
 - CatchClause
- Loops
 - WhileStatement
 - DoWhileStatement
 - ForStatement
 - ForInStatement
 - ForOfStatement
- Declarations
 - FunctionDeclaration
 - VariableDeclaration
 - VariableDeclarator
- Misc
 - Decorator
 - Directive
 - DirectiveLiteral
- Expressions
 - Super
 - Import

- ThisExpression
- ArrowFunctionExpression
- YieldExpression
- AwaitExpression
- ArrayExpression
- ObjectExpression
 - ObjectMember
 - ObjectProperty
 - ObjectMethod
- FunctionExpression
- Unary operations
 - UnaryExpression
 - UnaryOperator
 - UpdateExpression
 - UpdateOperator
- Binary operations
 - BinaryExpression
 - BinaryOperator
 - AssignmentExpression
 - AssignmentOperator
 - LogicalExpression
 - LogicalOperator
 - SpreadElement
 - MemberExpression
 - BindExpression
- ConditionalExpression
- CallExpression
- NewExpression
- SequenceExpression
- DoExpression
- Template Literals
 - TemplateLiteral

- TaggedTemplateExpression
- TemplateElement
- Patterns
 - ObjectPattern
 - ArrayPattern
 - RestElement
 - AssignmentPattern
- Classes
 - ClassBody
 - ClassMethod
 - ClassPrivateMethod
 - ClassProperty
 - ClassPrivateProperty
 - ClassDeclaration
 - ClassExpression
 - MetaProperty
- Modules
 - ModuleDeclaration
 - ModuleSpecifier
 - Imports
 - ImportDeclaration
 - ImportSpecifier
 - ImportDefaultSpecifier
 - ImportNamespaceSpecifier
 - Exports
 - ExportNamedDeclaration
 - ExportSpecifier
 - ExportDefaultDeclaration
 - ExportAllDeclaration

更复杂的节点类型意味着在parse中需要对不同type的token进行更复杂处理。

具体的处理方式感兴趣的可以参考babel-parser的源码。

5. 参考链接

- [How to implement a programming language \(tutorial for beginners\)](#)
- [ES6 入门教程](#)
- [babel/babylon](#)

6. 分享结束，谢谢大家

- 所有讲解，[完整代码参考](#)。