## **1.需求分析**

* 我们将会用它将 lisp 风格的函数调用转换为 C 风格

//假设我们有两个函数,`add` 和 `subtract`，那么它们的写法将会是下面这样

LISP C

2 + 2 (add 2 2) add(2, 2)

4 - 2 (subtract 4 2) subtract(4, 2)

2 + (4 - 2) (add 2 (subtract 4 2)) add(2, subtract(4, 2))

## **2.编译器分为三个阶段**

* 解析(Parsing) ****解析****是将最初原始的代码转换为一种更加抽象的表示(即AST)
* 转换(Transformation) ****转换****将对这个抽象的表示做一些处理,让它能做到编译器期望它做到的事情
* 代码生成(Code Generation) 接收处理之后的代码表示,然后把它转换成新的代码

### **2.1 解析(Parsing)**

* 解析一般来说会分成两个阶段：词法分析(Lexical Analysis)和语法分析(Syntactic Analysis)
  + \*词法分析\*\*接收原始代码,然后把它分割成一些被称为 token 的东西，这个过程是在词法分析器(Tokenizer或者Lexer)中完成的
  + Token 是一个数组，由一些代码语句的碎片组成。它们可以是数字、标签、标点符号、运算符或者其它任何东西
  + ****语法分析**** 接收之前生成的 token，把它们转换成一种抽象的表示，这种抽象的表示描述了代码语句中的每一个片段以及它们之间的关系。这被称为中间表示(intermediate representation)或抽象语法树(Abstract Syntax Tree, 缩写为AST)
  + 抽象语法树是一个嵌套程度很深的对象，用一种更容易处理的方式代表了代码本身，也能给我们更多信息

原始lisp代码

(add 2 (subtract 4 2))

tokens

[

{ type: 'paren', value: '(' },

{ type: 'name', value: 'add' },

{ type: 'number', value: '2' },

{ type: 'paren', value: '(' },

{ type: 'name', value: 'subtract' },

{ type: 'number', value: '4' },

{ type: 'number', value: '2' },

{ type: 'paren', value: ')' },

{ type: 'paren', value: ')' }

]

抽象语法树(AST)

{

type: 'Program',

body: [{

type: 'CallExpression',

name: 'add',

params: [{

type: 'NumberLiteral',

value: '2'

}, {

type: 'CallExpression',

name: 'subtract',

params: [{

type: 'NumberLiteral',

value: '4'

}, {

type: 'NumberLiteral',

value: '2'

}]

}]

}]

}

### **2.2 转换(Transformation)**

* 编译器的下一步就是转换,它只是把 AST 拿过来然后对它做一些修改.它可以在同种语言下操作 AST，也可以把 AST 翻译成全新的语言
* 你或许注意到了我们的 AST 中有很多相似的元素，这些元素都有type 属性，它们被称为 AST结点。这些结点含有若干属性，可以用于描述 AST 的部分信息
* 比如下面是一个NumberLiteral结点

{

type: 'NumberLiteral',

value: '2'

}

* 又比如下面是一个CallExpression结点

{

type: 'CallExpression',

name: 'subtract',

params: [...nested nodes go here...]

}

* 当转换 AST 的时候我们可以添加、移动、替代这些结点，也可以根据现有的 AST 生成一个全新的 AST
* 既然我们编译器的目标是把输入的代码转换为一种新的语言，所以我们将会着重于产生一个针对新语言的全新的 AST

### **2.3 遍历(Traversal)**

* 为了能处理所有的结点，我们需要遍历它们，使用的是深度优先遍历

{

type: 'Program',

body: [{

type: 'CallExpression',

name: 'add',

params: [{

type: 'NumberLiteral',

value: '2'

}, {

type: 'CallExpression',

name: 'subtract',

params: [{

type: 'NumberLiteral',

value: '4'

}, {

type: 'NumberLiteral',

value: '2'

}]

}]

}]

}

* 对于上面的 AST 的遍历流程是这样的

Program - 从 AST 的顶部结点开始

CallExpression (add) - Program 的第一个子元素

NumberLiteral (2) - CallExpression (add) 的第一个子元素

CallExpression (subtract) - CallExpression (add) 的第二个子元素

NumberLiteral (4) - CallExpression (subtract) 的第一个子元素

NumberLiteral (2) - CallExpression (subtract) 的第二个子元素

### **2.4 访问者(Visitors)**

* 我们最基础的想法是创建一个访问者(visitor)对象,这个对象中包含一些方法，可以接收不同的结点

**var** visitor = {

NumberLiteral() {},

CallExpression() {}

};

* 当我们遍历 AST 的时候，如果遇到了匹配 type 的结点，我们可以调用 visitor 中的方法
* 一般情况下为了让这些方法可用性更好，我们会把父结点也作为参数传入

### **2.5 代码生成(Code Generation)**

* 编译器的最后一个阶段是代码生成，这个阶段做的事情有时候会和转换(transformation)重叠,但是代码生成最主要的部分还是根据 AST 来输出代码
* 代码生成有几种不同的工作方式，有些编译器将会重用之前生成的 token，有些会创建独立的代码表示，以便于线性地输出代码。但是接下来我们还是着重于使用之前生成好的 AST
* 我们的代码生成器需要知道如何打印AST 中所有类型的结点，然后它会递归地调用自身，直到所有代码都被打印到一个很长的字符串中

## **3.实现编译器**

### **3.1 词法分析器(Tokenizer)**

* 我们只是接收代码组成的字符串，然后把它们分割成 token 组成的数组

(add 2 (subtract 4 2)) => [{ type: 'paren', value: '(' }, ...]

main.js

**let** tokenizer = require('./tokenizer');**let** tokens = tokenizer("(add 11 22)");console.log(tokens);

tokenizer.js

**let** LETTERS = /[a-z]/i;**let** WHITESPACE = /\s/;**let** NUMBERS = /[0-9]/;**function** **tokenizer**(input){

//current类似指标,用于记录我们在代码字符串中的位置

**let** current=0;

//tokens是一个数组,用来放置我们的token

**let** tokens = [];

//可能会在单个循环中多次增加current

**while**(current < input.length){

//char 指向当前字符串

**let** char = input[current];

//先检查是不是一个左圆括号

**if**(char === '('){

//如果是的话,我们往tokens里push一个type为paren,value为左圆括号的对象

tokens.push({

type:'paren',

value:'('

});

//自增current

current++;

//结束本次循环,进入下一个循环

**continue**;

//如果token是函数名,函数名是由一系列字母组成 比如 (add 11 22)

}**else** **if**(LETTERS.test(char)){

**let** value = '';

//用内层循环遍历所有的字母,把它们存入value中

**while**(LETTERS.test(char)){

value+=char;

char = input[++current];

}

//然后添加一个类型为name的token,进入下一个循环

tokens.push({

type:'name',

value

});

**continue**;

}**else** **if**(WHITESPACE.test(char)){

//token并不是有效的token,所以直接进入下一个循环

current++;

**continue**;

}**else** **if**(NUMBERS.test(char)){

**let** value = '';

//用内层循环遍历所有的数字,把它们存入value中

**while**(NUMBERS.test(char)){

value+=char;

char = input[++current];

}

//然后添加一个类型为number的token,进入下一个循环

tokens.push({

type:'number',

value

});

**continue**;

}**else** **if**(char === ')'){

tokens.push({

type: 'paren',

value: ')'

});

current++;

**continue**;

}

//如果没有匹配上任何类型的token,则抛出一个错误

**throw** **new** TypeError('I dont know what this character is '+ char);

}

**return** tokens;

}module.exports = tokenizer;

### **3.2 语法分析器(Parser)**

* 语法分析器接受 token 数组，然后把它转化为 AST

#### **3.2.1 main.js**

let tokenizer = require('./tokenizer');+let parser = require('./parser');+let tokens = tokenizer("(add 11 (sub 3 1))");

console.log(tokens);+let ast = parser(tokens);+console.log(JSON.stringify(ast,null,2));

#### **3.2.2 parser.js**

/\*\*

\* 语法分析器接受 token 数组，然后把它转化为 AST

\* [{ type: 'paren', value: '(' }, ...] => { type: 'Program', body: [...] }

\*///现在我们定义parser函数,接受tokens数组**function** **parser**(tokens) { // (add 11 22) (add 2 (subtract 4 2))

//我们再次声明一个current变量指针

**let** current = 0;

//但是这次我们使用递归而不是 `while` 循环，所以我们定义一个 `walk` 函数

**function** **walk**() {

// walk函数里，我们从当前token开始

**let** token = tokens[current];

//我们检查是不是 CallExpressions 类型，我们从左圆括号开始

**if** (token.type === 'paren' && token.value == '(') {

// 我们会自增 `current` 来跳过这个括号，因为括号在 AST 中是不重要的

token = tokens[++current];

//我们创建一个类型为 `CallExpression` 的根节点，然后把它的 name 属性设置为当前token 的值

//因为紧跟在左圆括号后面的 token 一定是调用的函数的名字

**let** node = {

type: 'CallExpression',

name: token.value,

params: []

}

// 我们再次自增 `current` 变量，跳过当前的 token

token = tokens[++current];

//现在我们循环遍历接下来的每一个 token，直到我们遇到右圆括号，这些 token 将会是 `CallExpression` 的 `params`(参数)

//这也是递归开始的地方，我们采用递归的方式来解决问题，而不是去尝试解析一个可能有无限层嵌套的结点

//(add 2 (subtract 4 2))

//所以我们创建一个 `while` 循环，直到遇到类型为 `'paren'`，值为右圆括号的 token

**while** (token.type != 'paren' || token.type == 'paren' && token.value != ')') {

//我们调用 `walk` 函数，它将会返回一个结点，然后我们把这个节点放入 `node.params` 中

node.params.push(walk());

token = tokens[current];

}

// 我们最后一次增加 `current`，跳过右圆括号

current++;

// 返回结点

**return** node;

//检查是不是 `number` 类型

}**else** **if**(token.type === 'number'){

// 如果是，`current` 自增。

current++;

// 然后我们会返回一个新的 AST 结点 `NumberLiteral`，并且把它的值设为 token 的值。

**return** {

type: 'NumberLiteral',

value: token.value

};

}

//同样，如果我们遇到了一个类型未知的结点，就抛出一个错误。

**throw** **new** TypeError(token.type);

}

//现在，我们创建 AST，根结点是一个类型为 `Program` 的结点

**var** ast = {

type: 'Program',

body: []

};

//现在我们开始 `walk` 函数，把结点放入 `ast.body` 中

//之所以在一个循环中处理，是因为我们的程序可能在 `CallExpressions` 后面包含连续的两个参数，而不是嵌套的

//(add 2 2) (subtract 4 2)

**while** (current < tokens.length) {

ast.body.push(walk());

}

// 最后我们的语法分析器返回 AST

**return** ast;

}module.exports = parser;

### **3.3 遍历器**

* 现在我们有了 AST，我们需要一个 visitor 去遍历所有的结点。当遇到某个类型的结点时，我们需要调用 visitor 中对应类型的处理函数

traverse(ast, {

Program(node, parent) {

console.log(node);

},

CallExpression(node, parent) {

console.log(node);

},

NumberLiteral(node, parent) {

console.log(node);

},

});

#### **3.3.1 main.js**

let tokenizer = require("./tokenizer");

let parser = require("./parser");+let traverser = require("./traverser");

let tokens = tokenizer("(add 11 (sub 3 1))");

console.log(tokens);

let ast = parser(tokens);

console.log(JSON.stringify(ast, null, 2));+let vistor = {+ Program(node, parent) {+ console.log(node);+ },+ CallExpression(node, parent) {+ console.log(node);+ },+ NumberLiteral(node, parent) {+ console.log(node);+ },+};+traverser(ast,vistor);

#### **3.3.2 traverser.js**

// 所以我们定义一个遍历器，它有两个参数，AST 和 vistor。在它的里面我们又定义了两个函数...**function** **traverser**(ast, visitor) {

// `traverseArray` 函数允许我们对数组中的每一个元素调用 `traverseNode` 函数。

**function** **traverseArray**(array, parent) {

array.forEach(**function**(child) {

traverseNode(child, parent);

});

}

// `traverseNode` 函数接受一个 `node` 和它的父结点 `parent` 作为参数，这个结点会被

// 传入到 visitor 中相应的处理函数那里。

**function** **traverseNode**(node,parent){

// 首先我们看看 visitor 中有没有对应 `type` 的处理函数。

**var** method = visitor[node.type];

// 如果有，那么我们把 `node` 和 `parent` 都传入其中。

**if** (method) {

method(node, parent);

}

// 下面我们对每一个不同类型的结点分开处理。

**switch** (node.type) {

//我们从顶层的 `Program` 开始，Program 结点中有一个 body 属性，它是一个由若干个结点组成的数组，所以我们对这个数组调用 `traverseArray`

//记住 `traverseArray` 会调用 `traverseNode`，所以我们会递归地遍历这棵树。

**case** 'Program':

traverseArray(node.body, node);

**break**;

//下面我们对 `CallExpressions` 做同样的事情，遍历它的 `params`。

**case** 'CallExpression':

traverseArray(node.params, node);

**break**;

// 如果是 `NumberLiterals`，那么就没有任何子结点了，所以我们直接 break

**case** 'NumberLiteral':

**break**;

// 同样，如果我们不能识别当前的结点，那么就抛出一个错误。

**default**:

**throw** **new** TypeError(node.type);

}

}

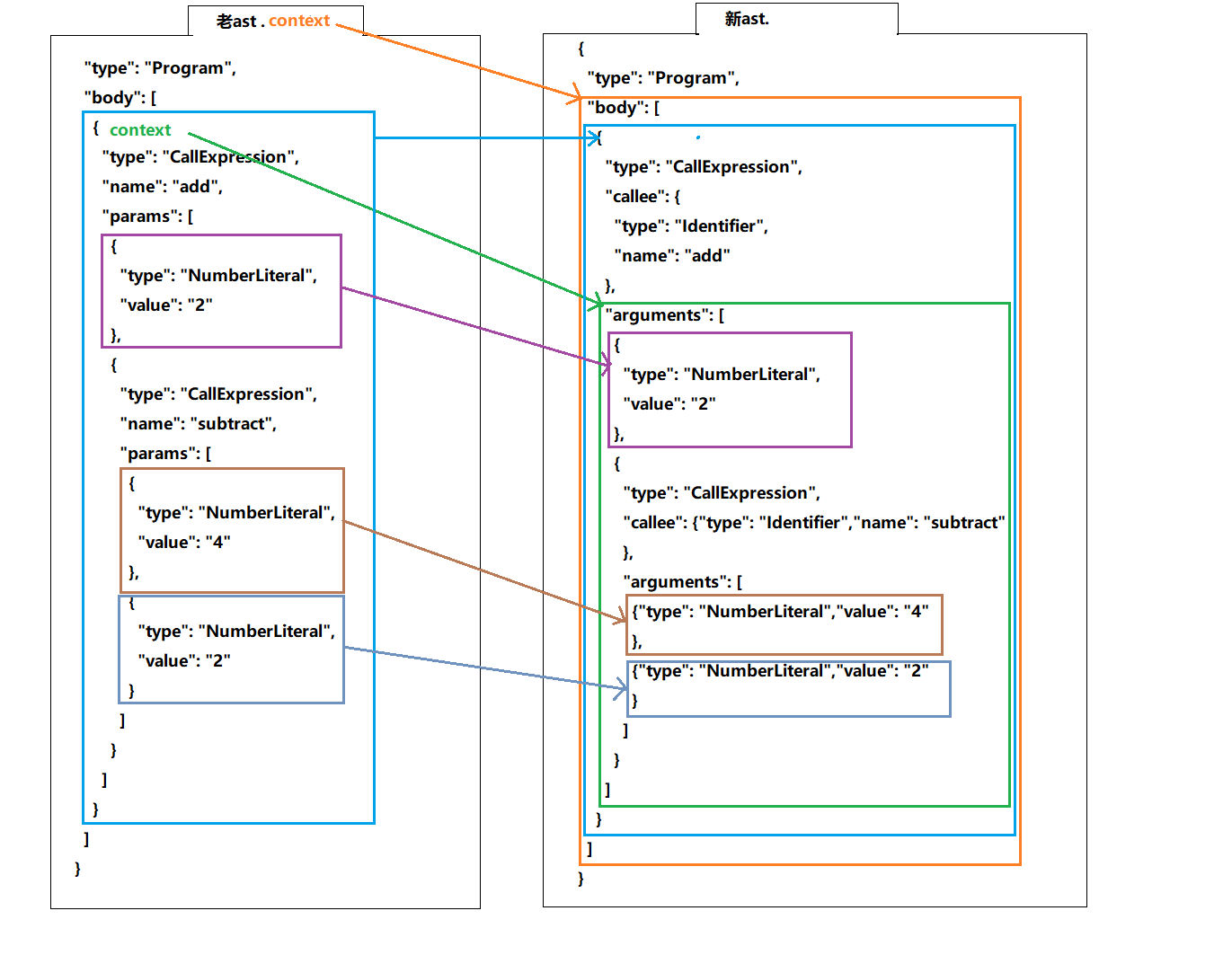
// 最后我们对 AST 调用 `traverseNode`，开始遍历。注意 AST 并没有父结点。

traverseNode(ast, null);

}module.exports = traverser;

### **3.4 转换AST**

* 下面是转换器。转换器接收我们在之前构建好的 AST，然后把它和 visitor 传递进入我们的遍历器中 ，最后得到一个新的 AST



#### **3.4.1 main.js**

let tokenizer = require("./tokenizer");

let parser = require("./parser");

let traverser = require("./traverser");+let transformer = require("./transformer");

let tokens = tokenizer("(add 2 (subtract 4 2))");

console.log(tokens);

let ast = parser(tokens);

console.log(JSON.stringify(ast, null, 2));

let vistor = {

Program(node, parent) {

console.log(node);

},

CallExpression(node, parent) {

console.log(node);

},

NumberLiteral(node, parent) {

console.log(node);

},

};

//traverser(ast,vistor);+var newAst = transformer(ast);+console.log(JSON.stringify(newAst, null, 2));

#### **3.4.2 transformer.js**

**let** traverser = require('./traverser');**function** **transformer**(ast) {

// 创建 `newAST`，它与我们之前的 AST 类似，有一个类型为 Program 的根节点。

**var** newAst = {

type: 'Program',

body: []

};//老的ast有一个属性\_context指向新的ast的body

ast.\_context = newAst.body;

// 我们把 AST 和 visitor 函数传入遍历器

traverser(ast,{

NumberLiteral(node,parent){

// 我们创建一个新结点，名字叫 `NumberLiteral`，并把它放入父结点的 context 中。

parent.\_context.push({

type:'NumberLiteral',

value:node.value

});

},

CallExpression(node,parent){

//我们创建一个 `CallExpression` 结点，里面有一个嵌套的 `Identifier`

**var** expression = {

type: 'CallExpression',

callee: {

type: 'Identifier',

name: node.name

},

arguments: []

};

// 下面我们在原来的 `CallExpression` 结点上定义一个新的 context，它是 expression

// 中 arguments 这个数组的引用，我们可以向其中放入参数。

node.\_context = expression.arguments;

// 最后我们把 `CallExpression`放入父结点的 context 中。

parent.\_context.push(expression);

}

});

// 最后返回创建好的新 AST

**return** newAst;

}module.exports = transformer;

### **3.5 代码生成**

* 我们的代码生成器会递归地调用它自己，把 AST 中的每个结点打印到一个很大的字符串中

#### **3.5.1 main.js**

**let** tokenizer = require("./tokenizer");**let** parser = require("./parser");**let** traverser = require("./traverser");**let** transformer = require("./transformer");**let** codeGenerator = require("./codeGenerator");**let** tokens = tokenizer("(add 2 (subtract 4 2))");console.log(tokens);**let** ast = parser(tokens);console.log(JSON.stringify(ast, null, 2));**let** vistor = {

Program(node, parent) {

console.log(node);

},

CallExpression(node, parent) {

console.log(node);

},

NumberLiteral(node, parent) {

console.log(node);

},

};//traverser(ast,vistor);**var** newAst = transformer(ast);console.log(JSON.stringify(newAst, null, 2));

**let** newCode = codeGenerator(newAst);console.log(newCode);

#### **3.5.2 codeGenerator.js**

**function** **codeGenerator**(node) {

// 对于不同 `type` 的结点分开处理。

**switch** (node.type) {

// 如果是 `Program` 结点，那么我们会遍历它的 `body` 属性中的每一个结点，并且递归地

// 对这些结点再次调用 codeGenerator，再把结果打印进入新的一行中。

**case** 'Program':

**return** node.body.map(codeGenerator)

.join('\n');

// 对于 `CallExpressions`，我们会打印出 `callee`，接着是一个左圆括号，然后对

// arguments 递归调用 codeGenerator，并且在它们之间加一个逗号，最后加上右圆括号。

**case** 'CallExpression':

**return** (

codeGenerator(node.callee) +

'(' +

node.arguments.map(codeGenerator)

.join(', ') +

')'

);

// 对于 `Identifiers` 我们只是返回 `node` 的 name。

**case** 'Identifier':

**return** node.name;

// 对于 `NumberLiterals` 我们只是返回 `node` 的 value

**case** 'NumberLiteral':

**return** node.value;

// 如果我们不能识别这个结点，那么抛出一个错误。

**default**:

**throw** **new** TypeError(node.type);

}

}

module.exports = codeGenerator;

### **3.6 打包**

#### **3.6.1 main.js**

**const** compier = require("./compiler");

**let** compiler = require('./compiler');**let** output = compiler("(add 2 (subtract 4 2))");console.log(output);

#### **3.6.2 compiler\index.js**

compiler\index.js

**const** tokenizer = require("./tokenizer");**const** parser = require("./parser");**const** transformer = require("./transformer");**const** codeGenerator = require("./codeGenerator");

**function** **compier**(input){

**let** tokens = tokenizer(input);

**let** ast = parser(tokens);

**let** newAst = transformer(ast);

**let** output = codeGenerator(newAst);

**return** output;

}module.exports = compier;

#### **3.6.3 tokenizer.js**

compiler\tokenizer.js

**let** LETTERS = /[a-z]/i;**let** WHITESPACE = /\s/;**let** NUMBERS = /[0-9]/;**function** **tokenizer**(input){

//current类似指标,用于记录我们在代码字符串中的位置

**let** current=0;

//tokens是一个数组,用来放置我们的token

**let** tokens = [];

//可能会在单个循环中多次增加current

**while**(current < input.length){

//char 指向当前字符串

**let** char = input[current];

//先检查是不是一个左圆括号

**if**(char === '('){

//如果是的话,我们往tokens里push一个type为paren,value为左圆括号的对象

tokens.push({

type:'paren',

value:'('

});

//自增current

current++;

//结束本次循环,进入下一个循环

**continue**;

//如果token是函数名,函数名是由一系列字母组成 比如 (add 11 22)

}**else** **if**(LETTERS.test(char)){

**let** value = '';

//用内层循环遍历所有的字母,把它们存入value中

**while**(LETTERS.test(char)){

value+=char;

char = input[++current];

}

//然后添加一个类型为name的token,进入下一个循环

tokens.push({

type:'name',

value

});

**continue**;

}**else** **if**(WHITESPACE.test(char)){

//token并不是有效的token,所以直接进入下一个循环

current++;

**continue**;

}**else** **if**(NUMBERS.test(char)){

**let** value = '';

//用内层循环遍历所有的数字,把它们存入value中

**while**(NUMBERS.test(char)){

value+=char;

char = input[++current];

}

//然后添加一个类型为number的token,进入下一个循环

tokens.push({

type:'number',

value

});

**continue**;

}**else** **if**(char === ')'){

tokens.push({

type: 'paren',

value: ')'

});

current++;

**continue**;

}

//如果没有匹配上任何类型的token,则抛出一个错误

**throw** **new** TypeError('I dont know what this character is '+ char);

}

**return** tokens;

}module.exports = tokenizer;

#### **3.6.4 parser.js**

compiler\parser.js

/\*\*

\* 语法分析器接受 token 数组，然后把它转化为 AST

\* [{ type: 'paren', value: '(' }, ...] => { type: 'Program', body: [...] }

\*///现在我们定义parser函数,接受tokens数组**function** **parser**(tokens) { // (add 11 22) (add 2 (subtract 4 2))

//我们再次声明一个current变量指针

**let** current = 0;

//但是这次我们使用递归而不是 `while` 循环，所以我们定义一个 `walk` 函数

**function** **walk**() {

// walk函数里，我们从当前token开始

**let** token = tokens[current];

//我们检查是不是 CallExpressions 类型，我们从左圆括号开始

**if** (token.type === 'paren' && token.value == '(') {

// 我们会自增 `current` 来跳过这个括号，因为括号在 AST 中是不重要的

token = tokens[++current];

//我们创建一个类型为 `CallExpression` 的根节点，然后把它的 name 属性设置为当前token 的值

//因为紧跟在左圆括号后面的 token 一定是调用的函数的名字

**let** node = {

type: 'CallExpression',

name: token.value,

params: []

}

// 我们再次自增 `current` 变量，跳过当前的 token

token = tokens[++current];

//现在我们循环遍历接下来的每一个 token，直到我们遇到右圆括号，这些 token 将会是 `CallExpression` 的 `params`(参数)

//这也是递归开始的地方，我们采用递归的方式来解决问题，而不是去尝试解析一个可能有无限层嵌套的结点

//(add 2 (subtract 4 2))

//所以我们创建一个 `while` 循环，直到遇到类型为 `'paren'`，值为右圆括号的 token

**while** (token.type != 'paren' || token.type == 'paren' && token.value != ')') {

//我们调用 `walk` 函数，它将会返回一个结点，然后我们把这个节点放入 `node.params` 中

node.params.push(walk());

token = tokens[current];

}

// 我们最后一次增加 `current`，跳过右圆括号

current++;

// 返回结点

**return** node;

//检查是不是 `number` 类型

}**else** **if**(token.type === 'number'){

// 如果是，`current` 自增。

current++;

// 然后我们会返回一个新的 AST 结点 `NumberLiteral`，并且把它的值设为 token 的值。

**return** {

type: 'NumberLiteral',

value: token.value

};

}

//同样，如果我们遇到了一个类型未知的结点，就抛出一个错误。

**throw** **new** TypeError(token.type);

}

//现在，我们创建 AST，根结点是一个类型为 `Program` 的结点

**var** ast = {

type: 'Program',

body: []

};

//现在我们开始 `walk` 函数，把结点放入 `ast.body` 中

//之所以在一个循环中处理，是因为我们的程序可能在 `CallExpressions` 后面包含连续的两个参数，而不是嵌套的

//(add 2 2) (subtract 4 2)

**while** (current < tokens.length) {

ast.body.push(walk());

}

// 最后我们的语法分析器返回 AST

**return** ast;

}module.exports = parser;

#### **3.6.5 transformer.js**

**let** traverser = require('./traverser');**function** **transformer**(ast) {

// 创建 `newAST`，它与我们之前的 AST 类似，有一个类型为 Program 的根节点。

**var** newAst = {

type: 'Program',

body: []

};//老的ast有一个属性\_context指向新的ast的body

ast.\_context = newAst.body;

// 我们把 AST 和 visitor 函数传入遍历器

traverser(ast,{

NumberLiteral(node,parent){

// 我们创建一个新结点，名字叫 `NumberLiteral`，并把它放入父结点的 context 中。

parent.\_context.push({

type:'NumberLiteral',

value:node.value

});

},

CallExpression(node,parent){

//我们创建一个 `CallExpression` 结点，里面有一个嵌套的 `Identifier`

**var** expression = {

type: 'CallExpression',

callee: {

type: 'Identifier',

name: node.name

},

arguments: []

};

// 下面我们在原来的 `CallExpression` 结点上定义一个新的 context，它是 expression

// 中 arguments 这个数组的引用，我们可以向其中放入参数。

node.\_context = expression.arguments;

// 最后我们把 `CallExpression`放入父结点的 context 中。

parent.\_context.push(expression);

}

});

// 最后返回创建好的新 AST

**return** newAst;

}module.exports = transformer;

#### **3.6.6 codeGenerator.js**

**function** **codeGenerator**(node) {

// 对于不同 `type` 的结点分开处理。

**switch** (node.type) {

// 如果是 `Program` 结点，那么我们会遍历它的 `body` 属性中的每一个结点，并且递归地

// 对这些结点再次调用 codeGenerator，再把结果打印进入新的一行中。

**case** 'Program':

**return** node.body.map(codeGenerator)

.join('\n');

// 对于 `CallExpressions`，我们会打印出 `callee`，接着是一个左圆括号，然后对

// arguments 递归调用 codeGenerator，并且在它们之间加一个逗号，最后加上右圆括号。

**case** 'CallExpression':

**return** (

codeGenerator(node.callee) +

'(' +

node.arguments.map(codeGenerator)

.join(', ') +

')'

);

// 对于 `Identifiers` 我们只是返回 `node` 的 name。

**case** 'Identifier':

**return** node.name;

// 对于 `NumberLiterals` 我们只是返回 `node` 的 value

**case** 'NumberLiteral':

**return** node.value;

// 如果我们不能识别这个结点，那么抛出一个错误。

**default**:

**throw** **new** TypeError(node.type);

}

}

module.exports = codeGenerator;

#### **3.6.7 compiler\traverser.js**

// 所以我们定义一个遍历器，它有两个参数，AST 和 vistor。在它的里面我们又定义了两个函数...**function** **traverser**(ast, visitor) {

// `traverseArray` 函数允许我们对数组中的每一个元素调用 `traverseNode` 函数。

**function** **traverseArray**(array, parent) {

array.forEach(**function**(child) {

traverseNode(child, parent);

});

}

// `traverseNode` 函数接受一个 `node` 和它的父结点 `parent` 作为参数，这个结点会被

// 传入到 visitor 中相应的处理函数那里。

**function** **traverseNode**(node,parent){

// 首先我们看看 visitor 中有没有对应 `type` 的处理函数。

**var** method = visitor[node.type];

// 如果有，那么我们把 `node` 和 `parent` 都传入其中。

**if** (method) {

method(node, parent);

}

// 下面我们对每一个不同类型的结点分开处理。

**switch** (node.type) {

//我们从顶层的 `Program` 开始，Program 结点中有一个 body 属性，它是一个由若干个结点组成的数组，所以我们对这个数组调用 `traverseArray`

//记住 `traverseArray` 会调用 `traverseNode`，所以我们会递归地遍历这棵树。

**case** 'Program':

traverseArray(node.body, node);

**break**;

//下面我们对 `CallExpressions` 做同样的事情，遍历它的 `params`。

**case** 'CallExpression':

traverseArray(node.params, node);

**break**;

// 如果是 `NumberLiterals`，那么就没有任何子结点了，所以我们直接 break

**case** 'NumberLiteral':

**break**;

// 同样，如果我们不能识别当前的结点，那么就抛出一个错误。

**default**:

**throw** **new** TypeError(node.type);

}

}

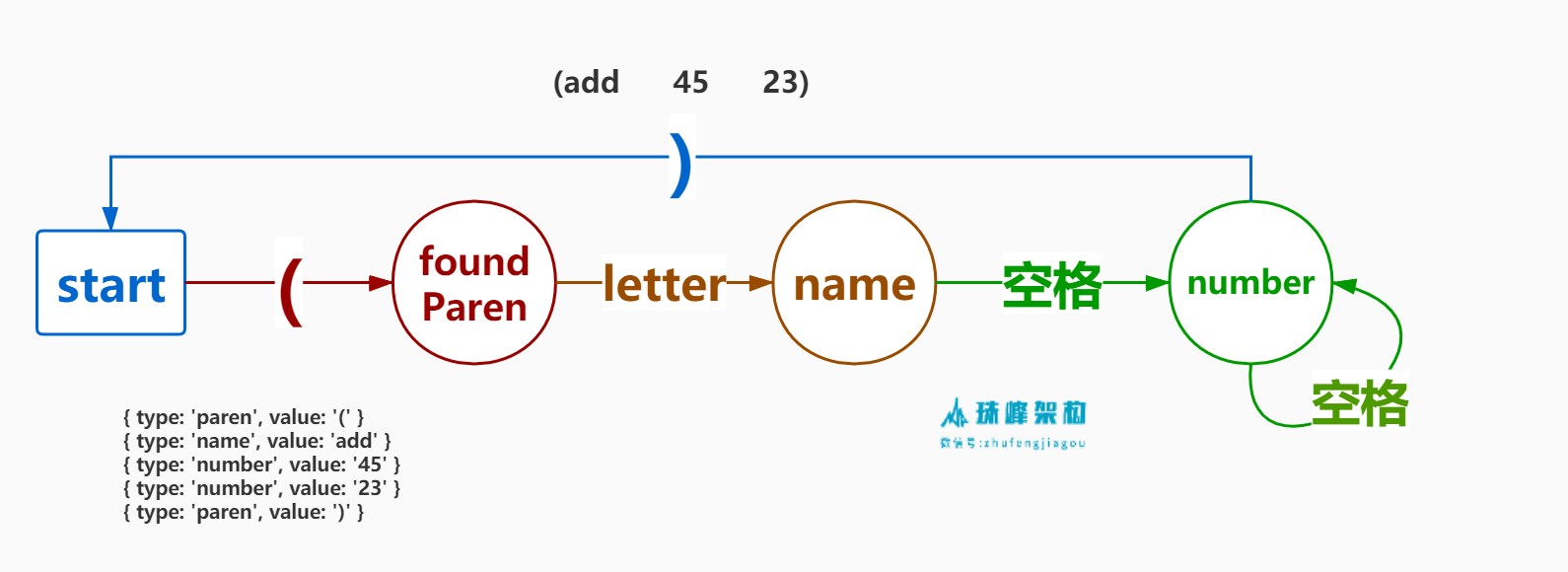
// 最后我们对 AST 调用 `traverseNode`，开始遍历。注意 AST 并没有父结点。

traverseNode(ast, null);

}module.exports = traverser;

## **4.有限状态机**

* 每一个状态都是一个机器,每个机器都可以接收输入和计算输出
* 机器本身没有状态,每一个机器会根据输入决定下一个状态



**let** LETTERS = /[a-z]/i;**let** WHITESPACE = /\s/;**let** NUMBERS = /[0-9]/;**let** currentToken;

**function** **start**(char){

**if**(char === '('){

emit({ type: 'paren', value: '('});

**return** foundParen;

}**else**{

**return** start;

}

}**function** **foundParen**(char){

**if**(LETTERS.test(char)){

currentToken = {

type:'name',

value:''

}

**return** name(char);

}

**throw** **new** TypeError('函数名必须是字符 '+ char);

}**function** **name**(char){

**if**(char.match(/^[a-zA-Z]$/)){

currentToken.value += char;

**return** name;

}**else** **if**(char == " "){

emit(currentToken);

currentToken = {

type:'number',

value:''

}

**return** number;

}

**throw** **new** TypeError('函数名必须以空格结束 '+ char);

}**function** **number**(char){

**if**(NUMBERS.test(char)){

currentToken.value += char;

**return** number;

}**else** **if**(char == " "){

emit(currentToken);

currentToken = {

type:'number',

value:''

}

**return** number;

}**else** **if**(char == ")"){

emit(currentToken);

emit({ type: 'paren', value: ')'});

**return** start;

}

**throw** **new** TypeError('参数必须是数字 '+ char);

}

**function** **tokenizer**(input){

**let** state = start;

**for**(**let** char **of** input){

state = state(char);

}

}

**function** **emit**(token){

console.log(token);

}

tokenizer('(add 45 23)');

## **5.正则分词**

**let** RegExpObject = /([0-9]+)|([ ])|(\+)|(\-)|(\\*)|(\/)|([\(])|([\)])/g;**let** names = ["Number","Space","+","-","\*","/","(",")"];

**function**\* **tokenize**(source){

**let** result = null;

**while**(true){

result = RegExpObject.exec(source);

**if**(!result) **break**;

**let** token = {type:null,value:null};

**let** index = result.find((item,index)=>index>0&&!!item);

token.type = names[index];

token.value = (result[0]);

**yield** token;

}

}**let** tokens = [];**for**(**let** token **of** tokenize("33+44-55\*66\*(77+55)")){

tokens.push(token);

}console.log(tokens);