link: null
title: 珠峰架构师成长计划
description: 原始lisp代码
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=213 sentences=265, words=2499

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

```
                 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 &#x6253;&#x5370;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){

  let current=0;

  let tokens = [];

  while(current < input.length){

    let char = input[current];

    if(char === '('){

        tokens.push({
            type:'paren',
            value:'('
        });

        current++;

        continue;

    }else if(LETTERS.test(char)){
        let value = '';

        while(LETTERS.test(char)){
            value+=char;
            char = input[++current];
        }

        tokens.push({
            type:'name',
            value
        });
        continue;
    }else if(WHITESPACE.test(char)){

        current++;
        continue;
    }else if(NUMBERS.test(char)){
        let value = '';

        while(NUMBERS.test(char)){
            value+=char;
            char = input[++current];
        }

        tokens.push({
            type:'number',
            value
        });
        continue;
    }else if(char === ')'){
        tokens.push({
            type: 'paren',
            value: ')'
          });
        current++;
        continue;
    }

    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

```
function parser(tokens) {

    let current = 0;

    function walk() {

        let token = tokens[current];

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

            token = tokens[++current];

            let node = {
                type: 'CallExpression',
                name: token.value,
                params: []
            }

            token = tokens[++current];

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

                node.params.push(walk());
                token = tokens[current];
            }

            current++;

            return node;

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

            current++;

            return {
              type: 'NumberLiteral',
              value: token.value
            };
        }

        throw new TypeError(token.type);
    }

    var ast = {
        type: 'Program',
        body: []
    };

    while (current < tokens.length) {
        ast.body.push(walk());
    }

    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

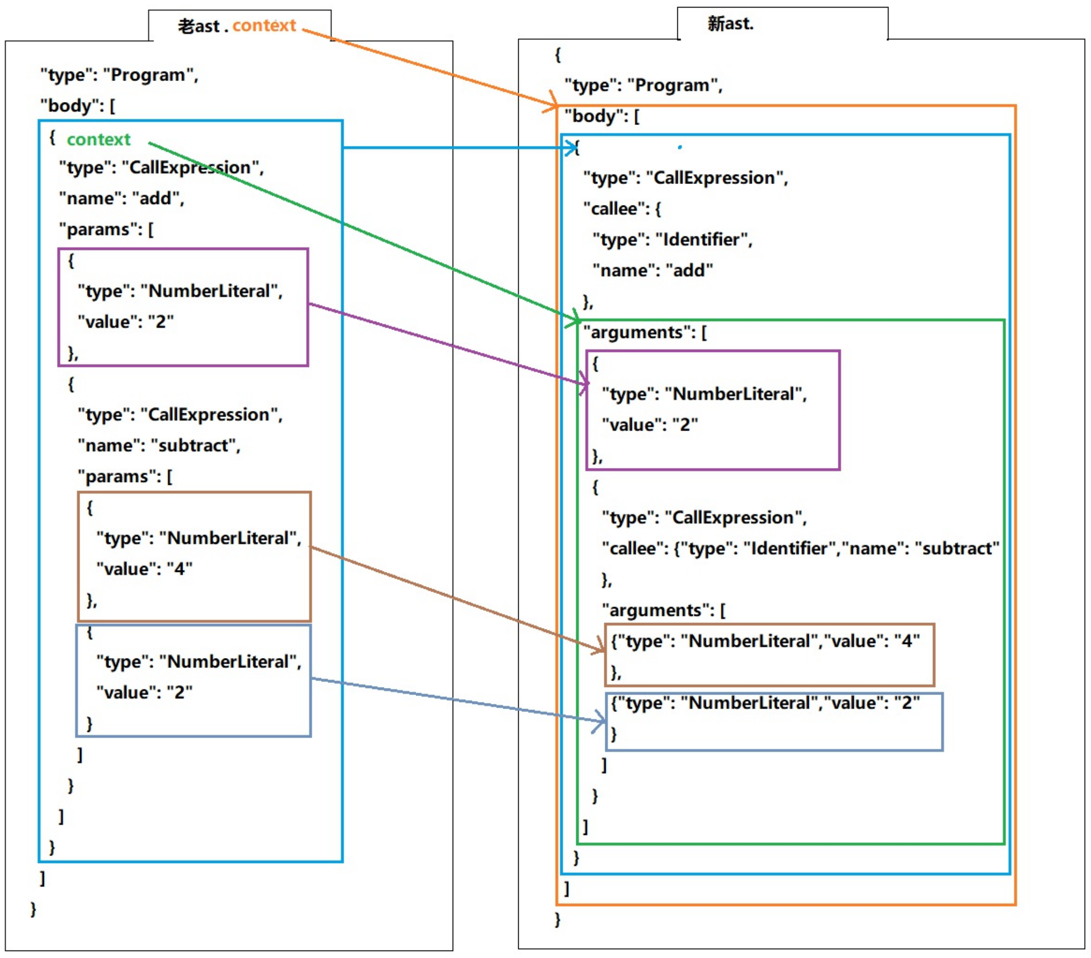
```
let tokenizer = require("./tokenizer");
let parser = require("./parser");
+let traverser = require("./traverser");
let tokens = tokenizer("(add 11 (sub 3 1))");
console.log(tokens);
let ast = parser(tokens);
console.log(JSON.stringify(ast, null, 2));
+let vistor = {
+  Program(node, parent) {
+      console.log(node);
+  },
+  CallExpression(node, parent) {
+    console.log(node);
+  },
+  NumberLiteral(node, parent) {
+    console.log(node);
+  },
+};
+traverser(ast,vistor);
```

3.3.2 traverser.js

```
function traverser(ast, visitor) {

    function traverseArray(array, parent) {
        array.forEach(function(child) {
           traverseNode(child, parent);
        });
    }

    function traverseNode(node,parent){

       var method = visitor[node.type];

        if (method) {
            method(node, parent);
        }

        switch (node.type) {

            case 'Program':
              traverseArray(node.body, node);
            break;

            case 'CallExpression':
              traverseArray(node.params, node);
              break;

            case 'NumberLiteral':
                break;

            default:
                throw new TypeError(node.type);
            }
    }

    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



```
"type": "Program",
"body": [
  { context
    "type": "CallExpression",
    "name": "add",
    "params": [
      {
        "type": "NumberLiteral",
        "value": "2"
      },
      {
        "type": "CallExpression",
        "name": "subtract",
        "params": [
          {
            "type": "NumberLiteral",
            "value": "4"
          },
          {
            "type": "NumberLiteral",
            "value": "2"
          }
        ]
      }
    ]
  }
]
}
```

```
{
 "type": "Program",
 "body": [
  {
   "type": "CallExpression",
   "callee": {
    "type": "Identifier",
    "name": "add"
   },
   "arguments": [
    {
     "type": "NumberLiteral",
     "value": "2"
    },
    {
     "type": "CallExpression",
     "callee": {"type": "Identifier","name": "subtract"
     },
     "arguments": [
      {"type": "NumberLiteral","value": "4"
      },
      {"type": "NumberLiteral","value": "2"
      }
     ]
    }
   ]
  }
 ]
}
```

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) {

  var newAst = {
    type: 'Program',
    body: []
  };
  ast._context = newAst.body;

  traverser(ast,{
    NumberLiteral(node,parent){

        parent._context.push({
            type:'NumberLiteral',
            value:node.value
        });
    },
    CallExpression(node,parent){

      var expression = {
        type: 'CallExpression',
        callee: {
          type: 'Identifier',
          name: node.name
        },
        arguments: []
      };

      node._context = expression.arguments;

      parent._context.push(expression);
    }
  });

  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);
  },
};

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) {

    switch (node.type) {

      case 'Program':
        return node.body.map(codeGenerator)
          .join('\n');

      case 'CallExpression':
        return (
          codeGenerator(node.callee) +
          '(' +
          node.arguments.map(codeGenerator)
            .join(', ') +
          ')'
        );

      case 'Identifier':
        return node.name;

      case 'NumberLiteral':
        return node.value;

      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){

  let current=0;

  let tokens = [];

  while(current < input.length){

    let char = input[current];

    if(char === '('){

        tokens.push({
            type:'paren',
            value:'('
        });

        current++;

        continue;

    }else if(LETTERS.test(char)){
        let value = '';

        while(LETTERS.test(char)){
            value+=char;
            char = input[++current];
        }

        tokens.push({
            type:'name',
            value
        });
        continue;
    }else if(WHITESPACE.test(char)){

        current++;
        continue;
    }else if(NUMBERS.test(char)){
        let value = '';

        while(NUMBERS.test(char)){
            value+=char;
            char = input[++current];
        }

        tokens.push({
            type:'number',
            value
        });
        continue;
    }else if(char === ')'){
        tokens.push({
            type: 'paren',
            value: ')'
          });
        current++;
        continue;
    }

    throw new TypeError('I dont know what this character is '+ char);
  }
  return tokens;
}
module.exports = tokenizer;
```

3.6.4 parser.js

compiler\parser.js

```
function parser(tokens) {

    let current = 0;

    function walk() {

        let token = tokens[current];

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

            token = tokens[++current];

            let node = {
                type: 'CallExpression',
                name: token.value,
                params: []
            }

            token = tokens[++current];

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

                node.params.push(walk());
                token = tokens[current];
            }

            current++;

            return node;

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

            current++;

            return {
              type: 'NumberLiteral',
              value: token.value
            };
        }

        throw new TypeError(token.type);
    }

    var ast = {
        type: 'Program',
        body: []
    };

    while (current < tokens.length) {
        ast.body.push(walk());
    }

    return ast;
}
module.exports  = parser;
```

3.6.5 transformer.js #

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

  var newAst = {
    type: 'Program',
    body: []
  };
  ast._context = newAst.body;

  traverser(ast,{
    NumberLiteral(node,parent){

        parent._context.push({
            type:'NumberLiteral',
            value:node.value
        });
    },
    CallExpression(node,parent){

      var expression = {
        type: 'CallExpression',
        callee: {
          type: 'Identifier',
          name: node.name
        },
        arguments: []
      };

      node._context = expression.arguments;

      parent._context.push(expression);
    }
  });

  return newAst;
}
module.exports = transformer;
```

3.6.6 codeGenerator.js #

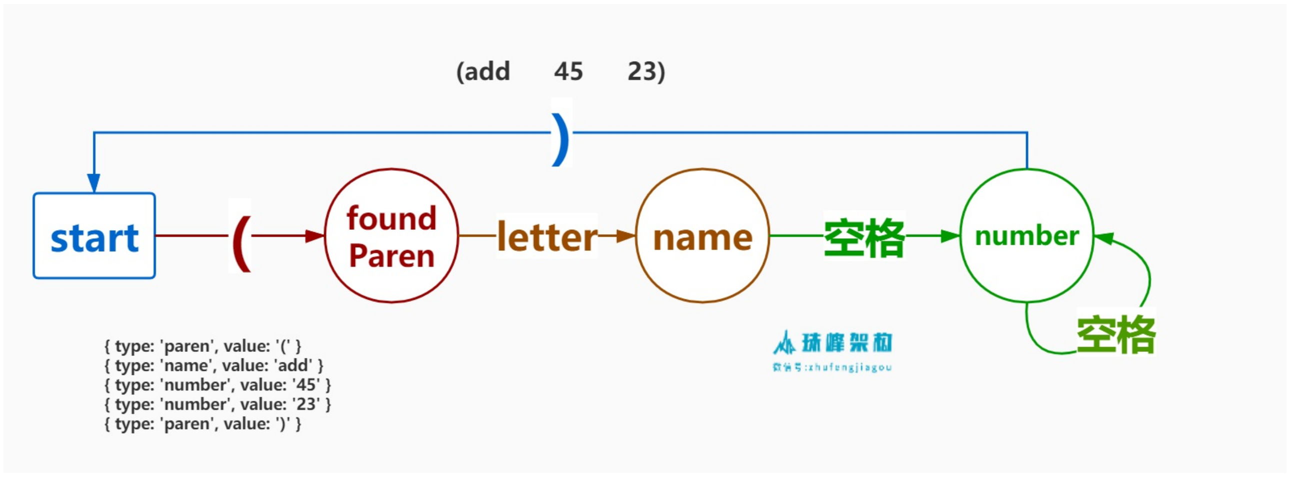
```
function codeGenerator(node) {

    switch (node.type) {

      case 'Program':
        return node.body.map(codeGenerator)
          .join('\n');

      case 'CallExpression':
        return (
          codeGenerator(node.callee) +
          '(' +
          node.arguments.map(codeGenerator)
            .join(', ') +
          ')'
        );

      case 'Identifier':
        return node.name;

      case 'NumberLiteral':
        return node.value;

      default:
        throw new TypeError(node.type);
    }
  }
  module.exports = codeGenerator;
```

3.6.7 compiler\traverser.js

```

function traverser(ast, visitor) {

    function traverseArray(array, parent) {
        array.forEach(function(child) {
           traverseNode(child, parent);
        });
    }

    function traverseNode(node,parent){

       var method = visitor[node.type];

        if (method) {
            method(node, parent);
        }

        switch (node.type) {

            case 'Program':
              traverseArray(node.body, node);
            break;

            case 'CallExpression':
              traverseArray(node.params, node);
              break;

            case 'NumberLiteral':
                break;

            default:
                throw new TypeError(node.type);
            }
    }

    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



```
{ type: 'paren', value: '(' }
{ type: 'name', value: 'add' }
{ type: 'number', value: '45' }
{ type: 'number', value: '23' }
{ type: 'paren', value: ')' }
```

```
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);
```