**广州大学学生实验报告**

**开课学院及实验室：计算机科学与工程实验室 2020年 11月 26日**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **学院** | 计算机科学与网络工程学院 | **年级/专业/班** | 计科182 | **姓名** | **莫广贤** | **学号** | **1806100104** |
| **实验课程名称** | 编译原理实验 | | | | | **成绩** |  |
| **实验项目名称** | 语法分析 | | | | | **指导老师** | 唐卷 |

**一、实验目的**

设计、编制并调试一个LL(1)语法分析程序，加深对语法分析原理的理解。

**二、实验内容**

1、上下文无关文法

2、无左递归、无回溯文法

3、LL(1)分析法

**三、实验设备/环境**

1、Windows操作系统

2、Javascript语言

**四、实验要求**

设有文法G[E]:  
E→E+T|T  
T→T\*F|F  
F→(E)|i

消除左递归后的文法为：

E→TE’

E’→+TE’|ε

T→FT’

T’→\*FT’|ε

F→(E)|i

实验要求，建立LL(1)分析表如下：

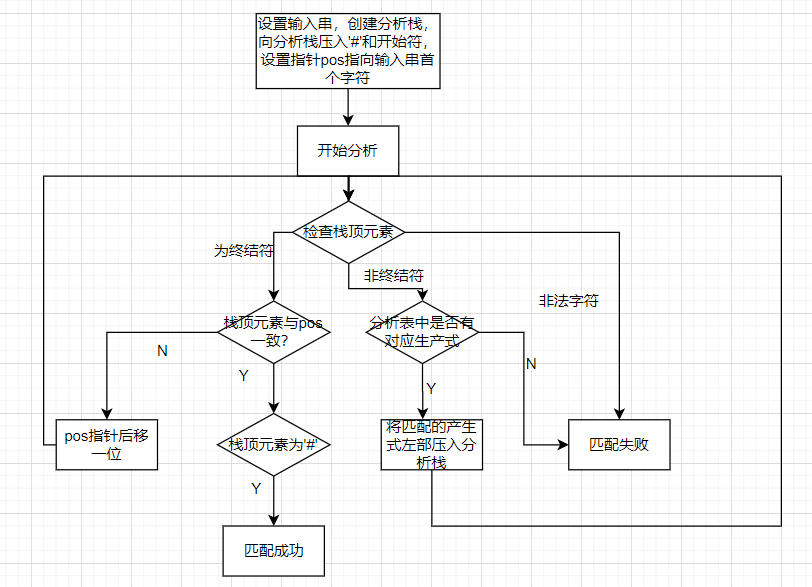
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **i** | **+** | **\*** | **(** | **)** | **#** |
| **E** | E→TE’ |  |  | E→TE’ |  |  |
| **E’** |  | E’→+TE’ |  |  | E’→ε | E’→ε |
| **T** | T→FT’ |  |  | T→FT’ |  |  |
| **T’** |  | T’→ε | T’→\*FT’ |  | T’→ε | T’→ε |
| **F** | F→i |  |  | F→(E) |  |  |

请分析输入串i+i\*i是否为该文法的句子，输出其分析过程。

**五、实验原理**

1、数据结构：使用二维数组存储LL(1)分析表，产生式由一个对象含有产生式左部和右部的对象构成。

2、流程图

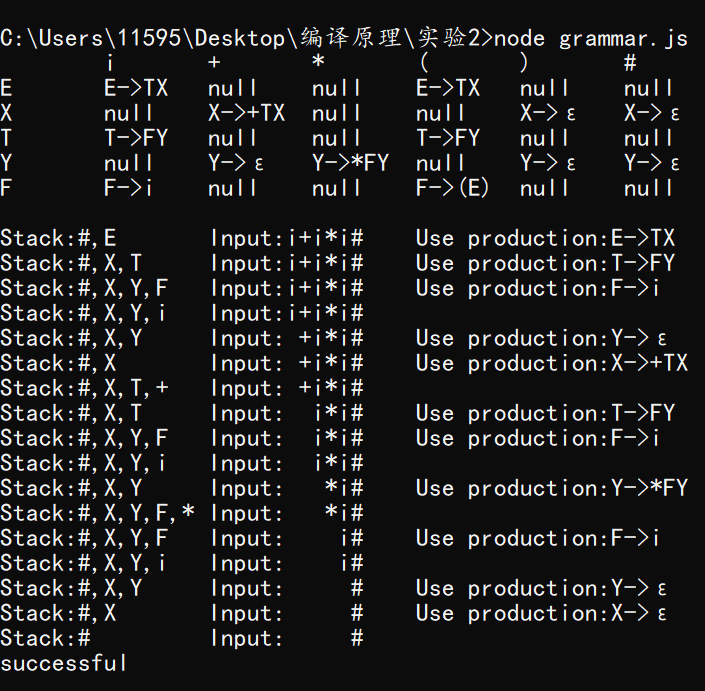


**六、实验步骤与结果**

1.实验步骤

依照实验原理中的流程图编写程序，并对程序进行测试。

2.实验结果



**七、总结心得**

本次实验自己编写了一个简单的LL(1)语法分析程序。加深了对语法分析原理的理解。此实验还可以进阶，即自动生成分析表，此处并未编写对应程序，略有不足。但是理解了语法分析的过程，还是提高了对代码编译的理解。

**八、程序源代码**

const { stdout, exit } = require('process');

const terminators = ['i', '+', '\*', '(', ')', '#'];

const nonTerminators = ['E', 'X', 'T', 'Y', 'F'];

class Pro { //产生式

    constructor(condition, conclusion) {

        this.condition = condition; //产生式左部

        this.conclusion = conclusion;   //产生式右部

    }

}

const analyzeTable = [  //初始化分析表

    [new Pro('E', 'TX'), null, null, new Pro('E', 'TX'), null, null],

    [null, new Pro('X', '+TX'), null, null, new Pro('X', 'ε'), new Pro('X', 'ε')],

    [new Pro('T', 'FY'), null, null, new Pro('T', 'FY'), null, null],

    [null, new Pro('Y', 'ε'), new Pro('Y', '\*FY'), null, new Pro('Y', 'ε'), new Pro('Y', 'ε')],

    [new Pro('F', 'i'), null, null, new Pro('F', '(E)'), null, null]

]

//打印LL(1)分析表

console.log('\ti\t+\t\*\t(\t)\t#');

for (let i = 0; i < analyzeTable.length; i++) {

    stdout.write(`${nonTerminators[i]}\t`);

    for (let j = 0; j < analyzeTable[i].length; j++) {

        if (analyzeTable[i][j]) stdout.write(analyzeTable[i][j].condition + '->' + analyzeTable[i][j].conclusion + '\t');

        else stdout.write('null\t');

    }

    stdout.write('\n');

}

console.log('');

const input = 'i+i\*i#'; //设置输入串

const stack = new Array();  //创建分析栈，并压入'#'

stack.push('#');

stack.push('E');//压入输入开始符

let pos = 0;    //指向当前输入符号

while (pos < input.length && stack.length) {    //开始分析

    stdout.write(`Stack:${stack.toString().padEnd(3, ' ')}\tInput:${input.slice(pos).padStart(input.length, ' ')}\t`);

    const c = stack.pop();

    if (c === input[pos]) { //若栈顶元素和当前输入符号一致

        if (c === '#') {    //若栈顶元素为'#'，则表示输入串为当前文法的句子

            console.log('\nsuccessful');

            break;

        } else { pos++; console.log(); continue; }  //当前输入符号后移一位

    } else if (nonTerminators.indexOf(c) !== -1) {

        const x = nonTerminators.indexOf(c), y = terminators.indexOf(input[pos]);

        const production = analyzeTable[x][y];  //在分析表中找到对应的产生式

        if (production === null) {    //分析表中无对应的产生式，则输入串不是当前文法的句子

            console.log(`\nError, there is no prodution. The top of stack is ${c}, current char is ${input[pos]}.`);

            break;

        }

        stdout.write(`Use production:${production.condition}->${production.conclusion}\n`)

        if (production.conclusion !== 'ε') {    //若产生式右部不为空则把左部的字符逆序压入分析栈

            for (let i = production.conclusion.length - 1; i >= 0; i--)

                stack.push(production.conclusion[i]);

        }

    } else {    //当前字符为非法字符

        console.log(`\nError, can not match this char. The top of stack is ${c}, current char is ${input[pos]}.`);

        break;

    }

}

exit();