**广州大学学生实验报告**

**开课学院及实验室：**计算机科学与教育软件学院计算机软件实验室 **2020年5月30日**

1. 实验目的

掌握LR分析表的设计方法和语义加工程序的扩充

1. 实验原理
2. 自底向上的语法分析，从分析树的底部(叶节点)向顶部(根节点)方向构造分析树，可以看成是将输入串w归约为文法开始符号S的过程
3. 自底向上语法分析的通用框架，移入-归约分析(Shift-Reduce Parsing)过程：在对输入串的一次从左到右扫描过程中，语法分析 器将零个或多个输入符号移入到栈的顶端，直到它可以对栈顶的一个文法符号串β进行归约为止。 然后，它将β归约为某个产生式的左部。 语法分析器不断地重复这个循环，直到它检测到一。 个语法错误，或者栈中包含了开始符号且输入缓冲 区为空(当进入这样的格局时，语法分析器停止运行， 并宣称成功完成了语法分析)为止。
4. 增广文法(Augmented Grammar)，使得文法开始符号仅出现 在一个产生式的左边，从而使得分析器只有一个接受状态
5. FIRST ( X )：可以从X推导出的所有串首终结符构成的集合如果X \*ε，那么ε∈FIRST( X )。FOLLOW(A)：可能在某个句型中紧跟在A后边的终结符a的集合 FOLLOW(A)={a|S->\*αAaβ, a∈VT，α,β∈(VT∪VN)\*} 如果A是某个句型的的最右符号，则将结束符“$”添加到FOLLOW(A)中。
6. 构造LR(0)自动机的状态集，规范LR(0) 项集族(Canonical LR(0) Collection)
7. SLR分析表,动作表 ACTION,转移表 GOTO sn：将符号a、状态n压入栈 rn：用第n个产生式进行归约
8. SLR分析表构造算法
9. SLR文法解决归约移入冲突的方法

已知项目集I：

A1→α1.a1β1

A2→α2.a2β2

…

Am→αm.amβm

B1→γ1.

B2→γ2.

…

Bn→γn.

如果集合{a1, a2, …, am}和 FOLLOW(B1)，FOLLOW(B2)，…， FOLLOW(Bn)两两不相交，则项目 集I中的冲突可以按以下原则解决： 设a是下一个输入符号

若a∈{ a1, a2, …, am}，则移进a

若a∈FOLLOW(Bi)，则用产生式 Bi→γi归约

此外，报错

1. 实验环境

Java

Vscode

1. 实验内容

参照算术表达式LR分析表的设计方法，设计扩充后的算术表达式LR分析表，并对原语义加工程序修改，加入新添的内容。

算术表达式文法扩充如下：

E→E+E| E-E|E\*E |E/E| (E) | I

试根据该文法重新设计LR分析表，并修改语义加工程序，最后验证修改的结果。

增广文法G’为:

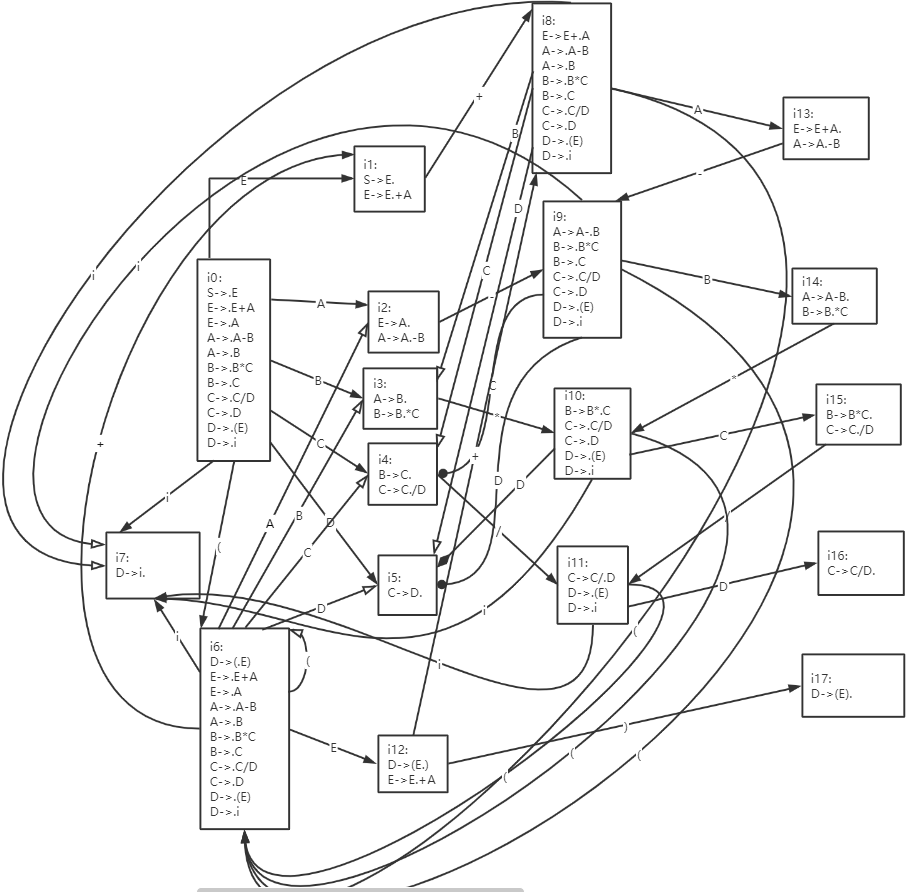
1. S->E
2. E->E+A
3. E->A
4. A->A-B
5. A->B
6. B->B\*C
7. B->C
8. C->C/D
9. C->D
10. D->(E)
11. D->i

文法的first集和follow集

|  |  |
| --- | --- |
| X | First(X) |
| S | i ( |
| E | i ( |
| A | i ( |
| B | i ( |
| C | i ( |
| D | i ( |

|  |  |
| --- | --- |
| X | Follow(X) |
| S | $ |
| E | + $ ) |
| A | + - $ ) |
| B | + - \* $ ) |
| C | + - \* / $ ) |
| D | + - \* / $ ) |

自动机



SLR分析表

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 状态 | ACTION | | | | | | | | GOTO | | | | |
| i | + | - | \* | / | ( | ) | $ | A | B | C | D | E |
| 0 | S7 |  |  |  |  | S6 |  |  | 2 | 3 | 4 | 5 | 1 |
| 1 |  | S8 |  |  |  |  |  | accept |  |  |  |  |  |
| 2 |  | R2 | S9 |  |  |  | R2 | R2 |  |  |  |  |  |
| 3 |  | R4 | R4 | S10 |  |  | R4 | R4 |  |  |  |  |  |
| 4 |  | R6 | R6 | R6 | S11 |  | R6 | R6 |  |  |  |  |  |
| 5 |  | R8 | R8 | R8 | R8 |  | R8 | R8 |  |  |  |  |  |
| 6 | S7 |  |  |  |  | S6 |  |  | 2 | 3 | 4 | 5 | 12 |
| 7 |  | R10 | R10 | R10 | R10 |  | R10 | R10 |  |  |  |  |  |
| 8 | S7 |  |  |  |  | S6 |  |  | 13 | 3 | 4 | 5 |  |
| 9 | S7 |  |  |  |  | S6 |  |  |  | 14 | 4 | 5 |  |
| 10 | S7 |  |  |  |  | S6 |  |  |  |  | 15 | 5 |  |
| 11 | S7 |  |  |  |  | S6 |  |  |  |  |  | 16 |  |
| 12 |  | S8 |  |  |  |  | S17 |  |  |  |  |  |  |
| 13 |  | R1 | S9 |  |  |  | R1 | R1 |  |  |  |  |  |
| 14 |  | R3 | R3 | S10 |  |  | R3 | R3 |  |  |  |  |  |
| 15 |  | R5 | R5 | R5 | S11 |  | R5 | R5 |  |  |  |  |  |
| 16 |  | R7 | R7 | R7 | R7 |  | R7 | R7 |  |  |  |  |  |
| 17 |  | R9 | R9 | R9 | R9 |  | R9 | R9 |  |  |  |  |  |

相关数据结构：

private LinkedList<Character> list1 = new LinkedList<Character>();//保存输入字符串的列表

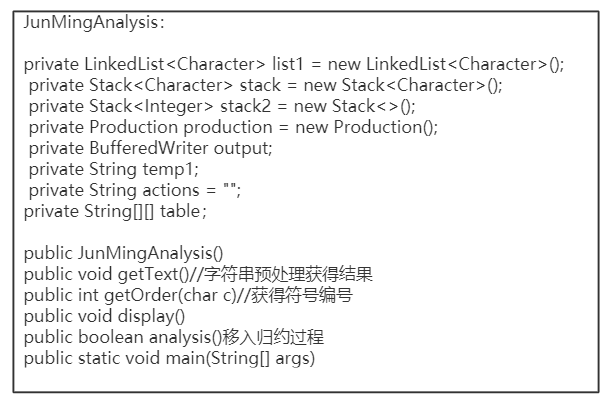
private Stack<Character> stack = new Stack<Character>();//保存符号的栈,移入符号到该栈，用顶部的符号进行归约

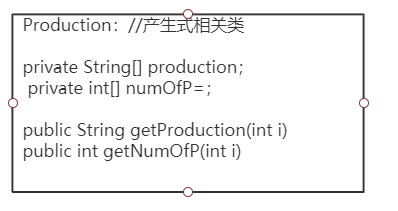
private Stack<Integer> stack2 = new Stack<>();//保存状态的栈，栈顶表示当前状态

private Production production = new Production();//描述产生式的类

private String[][] table//SLR分析表表的二维字符串数组

类设计:





1. 实验结果分析

输入：

i\*\*i

i+i-i\*(i/i)

输出：

state symbol input action

0 $ i\*\*i$ Shift to state 7

07 $i \*\*i$ Reduce by production 10

05 $D \*\*i$ Reduce by production 8

04 $C \*\*i$ Reduce by production 6

03 $B \*\*i$ Shift to state 10

语法错误不符合SLR(1)文法

-----------------------------------------

在状态10，下一个符号\*时，为空，所以出错

state symbol input action

0 $ i+i-i\*(i/i)$ Shift to state 7

07 $i +i-i\*(i/i)$ Reduce by production 10

05 $D +i-i\*(i/i)$ Reduce by production 8

04 $C +i-i\*(i/i)$ Reduce by production 6

03 $B +i-i\*(i/i)$ Reduce by production 4

02 $A +i-i\*(i/i)$ Reduce by production 2

01 $E +i-i\*(i/i)$ Shift to state 8

018 $E+ i-i\*(i/i)$ Shift to state 7

0187 $E+i -i\*(i/i)$ Reduce by production 10

0185 $E+D -i\*(i/i)$ Reduce by production 8

0184 $E+C -i\*(i/i)$ Reduce by production 6

0183 $E+B -i\*(i/i)$ Reduce by production 4

01813 $E+A -i\*(i/i)$ Shift to state 9

018139 $E+A- i\*(i/i)$ Shift to state 7

0181397 $E+A-i \*(i/i)$ Reduce by production 10

0181395 $E+A-D \*(i/i)$ Reduce by production 8

0181394 $E+A-C \*(i/i)$ Reduce by production 6

01813914 $E+A-B \*(i/i)$ Shift to state 10

0181391410 $E+A-B\* (i/i)$ Shift to state 6

01813914106 $E+A-B\*( i/i)$ Shift to state 7

018139141067 $E+A-B\*(i /i)$ Reduce by production 10

018139141065 $E+A-B\*(D /i)$ Reduce by production 8

018139141064 $E+A-B\*(C /i)$ Shift to state 11

01813914106411 $E+A-B\*(C/ i)$ Shift to state 7

018139141064117 $E+A-B\*(C/i )$ Reduce by production 10

0181391410641116 $E+A-B\*(C/D )$ Reduce by production 7

018139141064 $E+A-B\*(C )$ Reduce by production 6

018139141063 $E+A-B\*(B )$ Reduce by production 4

018139141062 $E+A-B\*(A )$ Reduce by production 2

0181391410612 $E+A-B\*(E )$ Shift to state 17

018139141061217 $E+A-B\*(E) $ Reduce by production 9

01813914105 $E+A-B\*D $ Reduce by production 8

018139141015 $E+A-B\*C $ Reduce by production 5

01813914 $E+A-B $ Reduce by production 3

01813 $E+A $ Reduce by production 1

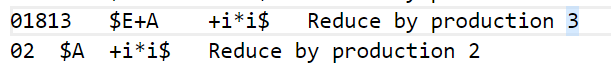
01 $E $ accept

语法正确，表达式符合SLR(1)文法

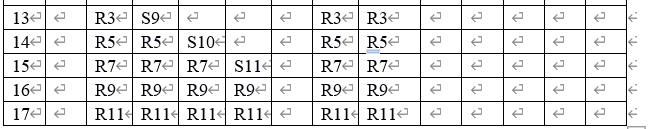
-----------------------------------------

实验结果与预期一致

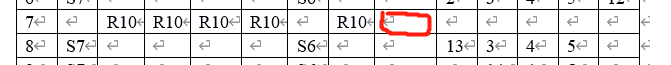
1. 实验总结
2. 由于产生式是从0开始的，而我在规约时想成从1开始，从而导致归约时发生错误，如下图所示，状态13遇到+号时用了产生式3归约导致错误，my god让我又回到自动机推断了好久，我以为13是状态1和3，把我绕晕了，后来才理清思路，



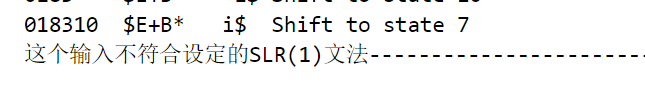
导致分析表里的归约全都写错了



1. 在状态7遇到$时忘记写上归约状态



导致归约时出错



后来我又按着自动机走了一遍并且查看follow集是否正确，才发现了错误

3.通过本次实验，使得我加深对自底向上分析方法的理解与操作，搞清楚了很多概念和算法，尤其是构造SLR1自动机和分析表，也使得我对栈链表等数据结构的使用更加熟练。

1. 源代码

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.util.LinkedList;

import java.util.Stack;

/\*\*

\*

\*/

/\*\*

\* @author zhongfang

\*

\*/

public class JunMingAnalysis {

private LinkedList<Character> list1 = new LinkedList<Character>();//保存输入字符串的列表

private Stack<Character> stack = new Stack<Character>();//保存符号的栈

private Stack<Integer> stack2 = new Stack<>();//保存状态的栈

private Production production = new Production();//描述产生式的类

private BufferedWriter output;

private String temp1;

private String actions = "";

//SLR分析表表的二维字符串数组

private String[][] table = { { "S7", "", "", "", "", "S6", "", "", "2", "3", "4", "5", "1" }, // 0

{ "", "S8", "", "", "", "", "", "accept", "", "", "", "", "" }, // 1

{ "", "R2", "R9", "", "", "", "R2", "R2", "", "", "", "", "" }, // 2

{ "", "R4", "R4", "S10", "", "", "R4", "R4", "", "", "", "", "" },// 3

{ "", "R6", "R6", "R6", "S11", "", "R6", "R6", "", "", "", "", "" },// 4

{ "", "R8", "R8", "R8", "R8", "", "R8", "R8", "", "", "", "", "" },// 5

{ "S7", "", "", "", "", "S6", "", "", "2", "3", "4", "5", "12" },// 6

{ "", "R10", "R10", "R10", "R10", "", "R10", "R10", "", "", "", "", "" },// 7

{ "S7", "", "", "", "", "S6", "", "", "13", "3", "4", "5", "" },// 8

{ "S7", "", "", "", "", "S6", "", "", "", "14", "4", "5", "" },// 9

{"S7", "", "", "", "", "S6", "", "", "", "", "15", "5", "" },// 10

{ "S7", "", "", "", "", "S6", "", "", "", "", "", "16", "" },// 11

{ "", "S8", "", "", "", "", "S17", "", "", "", "", "", "" },//12

{ "", "R1", "S9", "", "", "", "R1", "R1", "", "", "", "", "" },//13

{ "", "R3", "R3", "S10", "", "", "R3", "R3", "", "", "", "", "" },//14

{ "", "R5", "R5", "R5", "S11", "", "R5", "R5", "", "", "", "", "" },//15

{ "", "R7", "R7", "R7", "R7", "", "R7", "R7", "", "", "", "", "" },//16

{ "", "R9", "R9", "R9", "R9", "", "R9", "R9", "", "", "", "", "" },//17

};

public JunMingAnalysis() {

try {

output = new BufferedWriter(new FileWriter("output.txt"));

} catch (IOException e) {

e.printStackTrace();

}

}

public void getText() {

char a[];

BufferedReader bufferedReader = null;

try {

bufferedReader = new BufferedReader(new FileReader("input.txt"));

String lString;

while ((lString = bufferedReader.readLine()) != null) {

//数据预处理

stack.clear();

list1.clear();

stack2.clear();

temp1 = lString.trim();

temp1.replaceAll("\\s+", "");// 去掉一个以上的空白符，用一个空白代替

a = temp1.toCharArray();

for (char \_char : a) {

list1.offer(\_char);

}

list1.offerLast('$');

stack.push('$');

stack2.push(0);

output.write("state\t symbol\t input\t action");

output.newLine();

boolean b = analysis();

if (b){

output.write("语法正确，表达式符合SLR(1)文法");

output.newLine();

}

else{

output.write("语法错误不符合SLR(1)文法");

output.newLine();

}

output.write("-----------------------------------------");

output.newLine();

}

} catch (Exception e) {

} finally { // 关闭资源

if (bufferedReader != null)

try {

bufferedReader.close();

} catch (IOException e) {

e.printStackTrace();

}

if (output != null)

try {

output.close();

} catch (Exception e2) {

}

}

}

public int getOrder(char c) { //获取符号（终结符或非终结符）的编号（符号表中的横向顺序）

if (c == 'i')

return 0;

else if (c == '+')

return 1;

else if (c == '-')

return 2;

else if (c == '\*')

return 3;

else if (c == '/')

return 4;

else if (c == '(')

return 5;

else if (c == ')')

return 6;

else if (c == '$')

return 7;

else if(c=='A')

return 8;

else if(c=='B')

return 9;

else if(c=='C')

return 10;

else if(c=='D')

return 11;

else if(c=='E')

return 12;

else

return -1;

}

public void display() { //读SymbolStack、StateStack和input里的所有字符，显示到输出文件

String symbols = "";

String states = "";

String input = "";

Object[] symbolObjects = stack.toArray();

for (int i = 0; i < symbolObjects.length; i++) {

symbols += symbolObjects[i].toString();

}

Object[] stateObjects = stack2.toArray();

for (int i = 0; i < stack2.size(); i++) {

states += stateObjects[i].toString();

}

Object[] inputObjects = list1.toArray();

for (int i = 0; i < list1.size(); i++) {

input += inputObjects[i].toString();

}

try {

output.write(states + "\t" + symbols + "\t" + input + "\t"

+ actions);

output.newLine();

} catch (IOException e) {

e.printStackTrace();

}

}

public boolean analysis() {

while (true) {

actions = "";

char c = list1.peekFirst();

int i = getOrder(c);

if(i==-1) //如果输入是除了规定的终结符和非终结符以外的符号，返回false

return false;

String string2 = table[stack2.peek()][i];

if (string2.trim().equals("".trim()))

return false;

else if (string2.equals("accept")) {//接收

actions+="accept";

display();

return true;

} else if (string2.charAt(0) == 'S') { // 移进

String s = string2.substring(1); // 取S后面的状态数

int n = Integer.parseInt(s);

System.out.println("Shift " + s);

actions += "Shift to state " + s;

display();

list1.pollFirst(); // 从输入带里弹出第一个字符，并把该字符送symbolStack,同时向StateStack压入取得的状态数

stack2.push(n);

stack.push(c);

} else if (string2.charAt(0) == 'R') { // 规约

String s = string2.substring(1); // 取r后面的产生式编号

int n = Integer.parseInt(s);

System.out.println("Reduce " + s);

actions += "Reduce by production " + s;

display();

int n2 = production.getNumOfP(n);// 取产生式右部的字符个数（应该在SymbolStack和StateStack中弹出来的个数）

for (int i1 = 0; i1 < n2; i1++) {

stack.pop();

stack2.pop();

}

char \_char1 = production.getProduction(n).charAt(0);// 获取产生式左边的非终结符，压入SymbolStack

stack.push(\_char1);

String s1 = table[stack2.peek()][getOrder(\_char1)];// 查找goto字表，找到该终结符和当前状态对应的编号，压入StateStack

if (s1.trim().equals(""))

return false;

else

stack2.push(Integer.parseInt(s1));

}

}

}

public static void main(String[] args) {

JunMingAnalysis ananlyzer = new JunMingAnalysis();

ananlyzer.getText();

}

}

public class Production {

private String[] productions={"S->E","E->E+A","E->A","A->A-B","A->B","B->B\*C","B->C","C->C/D","C->D","D->(E)","D->i"};

private int[] numOfP={1,3,1,3,1,3,1,3,1,3,1};

public String getProduction(int i){

return productions[i];

}

public int getNumOfP(int i){

return numOfP[i];

}

}