编译原理实验二报告

161250092 软件工程 苗卫星

一 目的

实验二实现了一个简易的语法分析器，加深对于编译原理第四章钟语法分析理论的理解，提高程序设计能力。实验可以采用LL(1)、LR(2)分析法，或者自己构建的YACC。本次采用LR(1).

二 内容描述

基于java实现一个c语言部分文法的语法分析器。输入为实验一词法分析器生成的token流。输出是cfg.txt中上下文无关文法生成的项集数、分析表、LR(1)分析中栈的动态变化、移入规约序列。如果在移入规约的过程中出现语法错误，会终止分析并给出提示。

三 方法

（1）人为定义终结符集与非终结符集合，根据上下文无关文法生成 产生式、终结符与非终结符的first集。
（2）构造I0项，递归生成所有的项集，由此构件DFA（核心：if A->α.Bβ,a　add B->γ,first(βa) into In）
（3）根据DFA构造分析表
（4）根据分析表，对输入进行分析
（5）打印结果

四 假设

（1）从词法分析器中获取的token序列全部符合文法的要求。
（2）若在构造分析表中出现了SR冲突，全部用采用移入。

五 相关的状态机描述

本次实验中状态机由读取文本里的产生式后自动生成。在运行的例子中，一共46个状态。
状态间的跳转记录在list中

```
//以下三个LIST长度相等   begin[i],end[i],shift[i] 代表从状态begin[i] 通过shift[i] 转移到状态end[i]
public ArrayList<Integer> begin = new ArrayList<~>();
public ArrayList<Integer> end = new ArrayList<~>();
public ArrayList<String>  shift = new ArrayList<~>();
```

本实验状态机具体内容保存在DFA.txt中，下面为部分截取。其余部分请参考DFA.txt。

```
共有46个项集
I0
S' -> S  (0 ,$ )
S -> if ( B ) S ;  (0 ,$ )
S -> if ( B ) S ; else S ;  (0 ,$ )
S -> id = E  (0 ,$ )
S -> S ; S  (0 ,$ )
S -> if ( B ) S ;  (0 ,; )
S -> if ( B ) S ; else S ;  (0 ,; )
S -> id = E  (0 ,; )
S -> S ; S  (0 ,; )

I1
S' -> S  (1 ,$ )
S -> S ; S  (1 ,$ )
S -> S ; S  (1 ,; )

I2
S -> S ; S  (2 ,$ )
S -> S ; S  (2 ,; )
S -> if ( B ) S ;  (0 ,$ )
S -> if ( B ) S ; else S ;  (0 ,$ )
S -> id = E  (0 ,$ )
S -> S ; S  (0 ,$ )
S -> if ( B ) S ;  (0 ,; )
S -> if ( B ) S ; else S ;  (0 ,; )
S -> id = E  (0 ,; )
S -> S ; S  (0 ,; )

I3
S -> S ; S  (3 ,$ )
S -> S ; S  (3 ,; )
S -> S ; S  (1 ,$ )
S -> S ; S  (1 ,; )

I4
S -> if ( B ) S ;  (1 ,$ )
```

```
I42
E -> E - E  (2 ,$ )
E -> E - E  (2 ,; )
E -> E - E  (2 ,+ )
E -> E - E  (2 ,- )
E -> E + E  (0 ,$ )
E -> E - E  (0 ,$ )
E -> num  (0 ,$ )
E -> id  (0 ,$ )
E -> E + E  (0 ,; )
E -> E - E  (0 ,; )
E -> num  (0 ,; )
E -> id  (0 ,; )
E -> E + E  (0 ,+ )
E -> E - E  (0 ,+ )
E -> num  (0 ,+ )
E -> id  (0 ,+ )
E -> E + E  (0 ,- )
E -> E - E  (0 ,- )
E -> num  (0 ,- )
E -> id  (0 ,- )

I43
E -> E - E  (3 ,$ )
E -> E - E  (3 ,; )
E -> E - E  (3 ,+ )
E -> E - E  (3 ,- )
E -> E + E  (1 ,$ )
E -> E - E  (1 ,$ )
E -> E + E  (1 ,; )
E -> E - E  (1 ,; )
E -> E + E  (1 ,+ )
E -> E - E  (1 ,+ )
E -> E + E  (1 ,- )
E -> E - E  (1 ,- )
```

.....

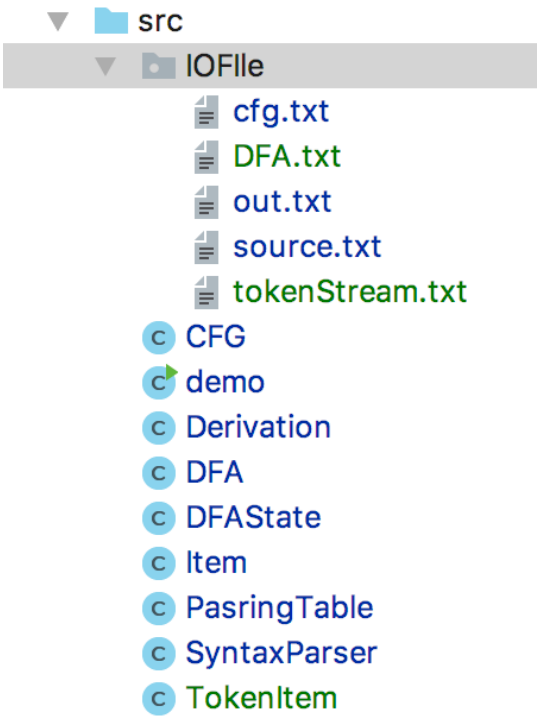
跳转内容包含在分析表中。
（左图仅为部分，详见out.txt）

分析表：

state	(	)	+	-	;	=	>=	else	id	if	num	$	B	E	S
0	x	x	x	x	x	x	x	x	S37	S4	x	x	-1	-1	1
1	x	x	x	x	S2	x	x	x	x	x	x	acc	-1	-1	-1
2	x	x	x	x	x	x	x	x	S37	S4	x	x	-1	-1	3
3	x	x	x	x	S2	x	x	x	x	x	R4	-1	-1	-1	
4	S5	x	x	x	x	x	x	x	x	x	x	x	-1	-1	-1
5	x	x	x	x	x	x	x	x	S36	x	S35	x	6	-1	-1
6	x	S7	x	x	x	S33	x	x	x	x	x	-1	-1	-1	
7	x	x	x	x	x	x	x	x	S24	S15	x	x	-1	-1	8
8	x	x	x	x	S9	x	x	x	x	x	x	-1	-1	-1	
9	x	x	x	x	R1	x	x	S10	S24	S15	x	R1	-1	-1	13
10	x	x	x	x	x	x	x	x	S24	S15	x	x	-1	-1	11
11	x	x	x	x	S12	x	x	x	x	x	x	x	-1	-1	-1
12	x	x	x	x	R2	x	x	x	S24	S15	x	R2	-1	-1	13
13	x	x	x	x	S14	x	x	x	x	x	x	x	-1	-1	-1
14	x	x	x	x	x	x	x	x	S24	S15	x	x	-1	-1	13
15	S16	x	x	x	x	x	x	x	x	x	x	x	-1	-1	-1
16	x	x	x	x	x	x	x	x	S36	x	S35	x	17	-1	-1
17	x	S18	x	x	x	S33	x	x	x	x	x	-1	-1	-1	
18	x	x	x	x	x	x	x	x	S24	S15	x	x	-1	-1	19
19	x	x	x	x	S20	x	x	x	x	x	x	x	-1	-1	-1
20	x	x	x	x	R1	x	x	S21	S24	S15	x	x	-1	-1	13
21	x	x	x	x	x	x	x	x	S24	S15	x	x	-1	-1	22
22	x	x	x	x	S23	x	x	x	x	x	x	x	-1	-1	-1
23	x	x	x	x	R2	x	x	x	S24	S15	x	x	-1	-1	13
24	x	x	x	x	x	S25	x	x	x	x	x	x	-1	-1	-1
25	x	x	x	x	x	x	x	x	S32	x	S31	x	-1	26	-1
26	x	x	S27	S29	R3	x	x	x	x	x	x	x	-1	-1	-1
27	x	x	x	x	x	x	x	x	S32	x	S31	x	-1	28	-1
28	x	x	S27	S29	R8	x	x	x	x	x	x	x	-1	-1	-1
29	x	x	x	x	x	x	x	x	S32	x	S31	x	-1	30	-1
30	x	x	S27	S29	R9	x	x	x	x	x	x	x	-1	-1	-1
31	x	x	R10	R10	R10	x	x	x	x	x	x	x	-1	-1	-1
32	x	x	R11	R11	R11	x	x	x	x	x	x	x	-1	-1	-1
33	x	x	x	x	x	x	x	x	S36	x	S35	x	34	-1	-1

六 数据结构的描述

文件结构如下。



（1）IOFile

1. cfg.txt 存放上下文无关无法
2. DFA.txt 存放DFA项集
3. out.txt为分析表、RS序列
4. source.txt 源程序
5. tokenStream.txt 经过词法分析器处理生成的token序列

（2） CFG

用来处理上下文无关文法，生成非终结符集、终结符集、产生式集、first集

```
/**
 * 上下文无关文法
 */
public class CFG {

    //非终结符
    public static TreeSet<String>  vn = new TreeSet<>();

    //终结符
    public static TreeSet<String>  vt = new TreeSet<>();

    //production 产生式
    public static ArrayList<Derivation> p = new ArrayList<>();


    //first集
    public static HashMap<String,TreeSet<String>>  firstSet = new HashMap<>();
```

（3）Derivation 产生式

成员变量包含左侧非终结符，右侧的推导式。 用一个List存放推导式每一个词法单元。

```
/**
 * 产生式
 */
public class Derivation {
    //NT
    private String left;

    //左侧文法文串
    private ArrayList<String> right = new ArrayList<>();
```

（4）Item 项

成员变量 PointIndex 项中 点的位置。predictiveSymbol 指预测符。 d是产生式

```java
//项
public class Item {

    private int PointIndex;
    private String predictiveSymbol;
    private Derivation d;


    public Item(Derivation d, int index, String symbol){
        PointIndex = index;
        predictiveSymbol = symbol;
        this.d = d;
    }
```

（5）DFAState 项集

成员变量 编号，以及包含的项

```java
//项集
public class DFAState {
    private int id;
    private ArrayList<Item>  ItemSet;

    public DFAState(int id){
        this.id = id;
        ItemSet = new ArrayList<Item>();
    }

    //该项集是否与另一个相同
```

（6）DFA 状态机

成员变量 所有的项集， 跳转信息

```java
/**
 * 状态机
 */
public class DFA {
    private ArrayList<DFAState> states;

    //以下三个LIST长度相等   begin[i],end[i],shift[i] 代表从状态begin[i] 通过shift[i] 转移到状态end[i]
    public ArrayList<Integer> begin = new ArrayList<~>();
    public ArrayList<Integer> end = new ArrayList<~>();
    public ArrayList<String>  shift = new ArrayList<~>();
```

（7）分析表
成员变量：如图

```
//预测分析表
public class PasringTable {

    DFA dfa;

    //状态数
    int stateNum;
    //非终结符数
    int gotoCol;
    //终结符数
    int actionCol;

    //用来存方 action goto 项，类似表头
    String[] actionItems;
    String[] gotoItems;

    // table body
    String[][] actionTable;
    int[][] gotoTable;
```

（8）语法分析

```
/**
 * 语法分析
 */
public class SyntaxParser {

    //分析表
    PasringTable parsingTable;

    //状态栈
    Stack<Integer> stack;

    //输入的token流
    Queue<TokenItem> tokens;

    //记录RS操作
    ArrayList<String> operations;
```

七 核心算法

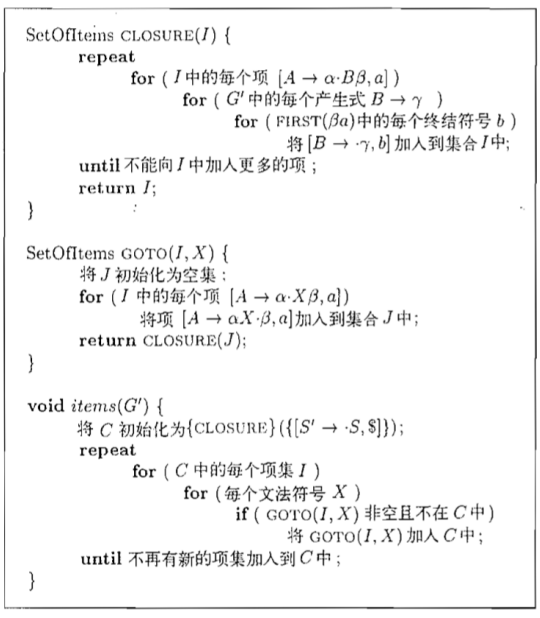
（1） 计算first集

```
private  static void getFirstSet(){
    /**
     * 终结符的first集就是自身
     * 非终结符的first集是其推导得到的串的首符号的集合;
     */

    for (String t : vt) {
        firstSet.put(t, new TreeSet<String>());
        firstSet.get(t).add(t);
    }


    for (String nt : vn) {
        firstSet.put(nt, new TreeSet<String>());

        //int productionKinds = p.size();
        for (Derivation d : p) {
            if (d.getLeft().equals(nt)) {
                //右式第一个是终结符
                if (vt.contains(d.getFirstItem()))
                    firstSet.get(nt).add(d.getFirstItem());
                else
                //右时第一个是非终结符
                    firstSet.get(nt).addAll(findFirst(d.getFirstItem()));
            }
        }//遍历所有产生式

    }//遍历所有非终结符

}

   private  static TreeSet<String>  findFirst(String vn){
       TreeSet<String> set = new TreeSet<>();

       for(Derivation d:p){
           if(d.getLeft().equals(vn)){
               //右式第一个是终结符
               if(vt.contains(d.getFirstItem()))
                   set.add(d.getFirstItem());
               else
               {
                   if(!vn.equals(d.getFirstItem()))//避免左递归
                       set.addAll(findFirst(d.getFirstItem()));
               }

           }

       }
       return set;
   }
```

解释：first(a)=a ,first(B)是从B可以推导得到的串的首符号的集合。先把终结符的first集求出来，再求非终结符，如果这个非终结符的产生式右侧第一个仍然是非终结符则调用 findFirst 方法。findFirst采用递归方式获取某个非终结符的first集。核心即　若A－>Bβ, 则First(A)包含First(B)

（2）构造DFA

DFA由许多DFA状态构成。算法主要采用龙书中文第二版4.53算法



SetOfItems CLOSURE(I) {
　　repeat
　　　　for (I 中的每个项 $[A \to \alpha \cdot B\beta, a]$)
　　　　　　for (G' 中的每个产生式 $B \to \gamma$)
　　　　　　　　for (FIRST(βa)中的每个终结符号 b)
　　　　　　　　　　将 $[B \to \cdot\gamma, b]$ 加入到集合 I 中;
　　until 不能向 I 中加入更多的项;
　　return I;
}

SetOfItems GOTO(I, X) {
　　将 J 初始化为空集;
　　for (I 中的每个项 $[A \to \alpha \cdot X\beta, a]$)
　　　　将项 $[A \to \alpha X \cdot \beta, a]$ 加入到集合 J 中;
　　return CLOSURE(J);
}

void *items*(G') {
　　将 C 初始化为{CLOSURE}({$[S' \to \cdot S, \$]$});
　　repeat
　　　　for (C 中的每个项集 I)
　　　　　　for (每个文法符号 X)
　　　　　　　　if (GOTO(I, X) 非空且不在 C 中)
　　　　　　　　　　将 GOTO(I, X) 加入 C 中;
　　until 不再有新的项集加入到 C 中;
}

图 4-40　为文法 G' 构造 LR(1)项集族的算法

首先 把S'-> .S ,$ 添加到I0，然后根据 if exist A->α.Bβ,a　add B->γ,first(βa) into I0 补完I0的项. 生成I0后，调用方法 addState(**int** previousState,String jump,ArrayList<Item> core) . 参数分别是前一个状态id，跳转符号，上一个状态跳转的核。 该函数依据核通过（if exist A->α.Bβ,a　add B->γ,first(βa) into I）补全项集，然后检测这个项集是否已经存在，若存在则结束递归。若不存在则继续递归。

如下

```
private void addState(int previousState,String jump,ArrayList<Item> core){
    //构建项集的核
    DFAState temp = new DFAState( id: -1);
    for(Item it:core){
        it.addIndex();//点全部向右移动1
        temp.addNewItem(it);
    }

    //core extend
    for(int i=0;i<temp.getItemSetSize();i++){
        Item item = temp.getItem(i);
    //    System.out.println(item.getPointIndex()+" "+item.getDSize());
        if(item.getPointIndex()<item.getDSize()){
            String strAfterPoint = item.getD().getNItem(item.getPointIndex());//B
            String beta = null;
            if(item.getPointIndex() == item.getDSize()-1)//when beta = null
                beta = item.getPredictiveSymbol(); //beta a = a ,first(beta a) = first(a)
            else
            /**
             * when bata != null;
             * beta = item.getD().getNItem(item.getPointIndex()+1);//beta = item after B,first(beta a)= first(beta)
             */
                beta = item.getStrAfterB();

            if(CFG.vn.contains(strAfterPoint)) {
                //if B is vn,temp add B -> gamma,first(beta a)
                ArrayList<String> PredictiveSymbolSet = CFG.getFirstSet(beta);
                ArrayList<Derivation> derivationFromB = CFG.getDerivation(strAfterPoint);
                for(int j=0;j<derivationFromB.size();j++) {
                    for (int m = 0; m < PredictiveSymbolSet.size(); m++) {
                        Item it = new Item(derivationFromB.get(j), index: 0,PredictiveSymbolSet.get(m));
                        temp.addNewItem(it);
                    }
                }
            }
        }
    }
```

```
    /**
     * 由核扩展的项集中的项推导结束，需要验证该项集是否已经存在
     * 如果该项集已经存在则终止递归
     */

    for(DFAState dfsState:states){
        if(temp.isSame(dfsState)) {
            this.recordShift(previousState,dfsState.getStateId(),jump);
            return;
        }
    }

    /**
     * 如果不存在该项集，则ID赋值为当前DFAstate.size()，并添加进DFA中
     */
    temp.changeStateId(states.size());
    states.add(temp);
    this.recordShift(previousState,temp.getStateId(),jump);

    for(String nextJump:temp.getJumpPath()){
        addState(temp.getStateId(),nextJump,temp.getJumpableItems(nextJump));
    }

}
```

（3）构造分析表

利用ParsingTable 模拟类似左部的分析表
其中的跳转信息存储在DFAState的三个list中。
只需要转换action、goto成员所在的下标和
action、goto成员本身即可完成表中S和num的
填充。然后扫描所有的项集，如果有.在推导式
最后情况，则进行规约，填充Rn。

状态	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

（4）语法分析

源程序通过词法分析器处理后，作为输入

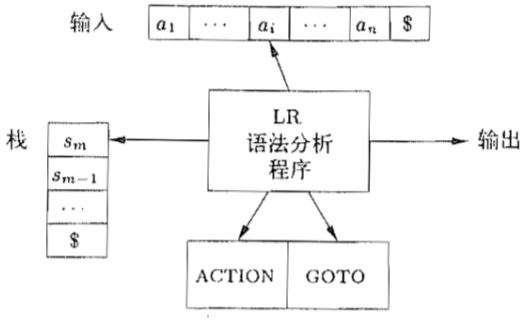


图 4-35　一个 LR 语法分析器的模型

接着模拟左图的形式进行分析

	栈	符号	输入	动作
(1)	0		id * id + id $	移入
(2)	0 5	id	* id + id $	根据 $F \to$ id 归约
(3)	0 3	F	* id + id $	根据 $T \to F$ 归约
(4)	0 2	T	* id + id $	移入
(5)	0 2 7	T *	id + id $	移入
(6)	0 2 7 5	T * id	+ id $	根据$F \to$ id 归约
(7)	0 2 7 10	T * F	+ id $	根据$T \to T * F$归约
(8)	0 2	T	+ id $	根据$E \to T$ 归约
(9)	0 1	E	+ id $	移入
(10)	0 1 6	E +	id $	移入
(11)	0 1 6 5	E + id	$	根据 $F \to$ id 归约
(12)	0 1 6 3	E + F	$	根据 $T \to F$ 归约
(13)	0 1 6 9	E + T	$	根据 $E \to E + T$归约
(14)	0 1	E	$	接受

具体算法采用龙书第二版160页例4.45

```
令 a 为w$的第一个符号；
while(1) { /* 永远重复 */
    令 s 是栈顶的状态；
    if ( ACTION[s, a] = 移入 t ) {
        将 t 压入栈中；
        令 a 为下一个输入符号；
    } else if ( ACTION[s, a] = 归约 A → β ) {
        从栈中弹出|β|个符号；
        令 t 为当前的栈顶状态；
        将 GOTO[t, A]压入栈中；
        输出产生式 A → β;
    } else if ( ACTION[s, a] = 接受 ) break; /* 语法分析完成 */
    else 调用错误恢复例程；
}
```

部分具体实现如下

```
        while( !(stack.peek()==1&&tokens.element().content.equals("$") ) ){
                String nextState = getJumpState(stack.peek(),tokens.element().type);
System.out.println("使用"+nextState);
                if(nextState.equals("x")){
                    System.out.println("语法错误");
                    break;
                }
                else{
                    String[] SnOrRn =new String[2];
                    SnOrRn[0] = nextState.substring( beginIndex: 0, endIndex: 1);
                    SnOrRn[1] = nextState.substring(1);
                    if(SnOrRn[0].equals("S")) {//移入
                          int next = Integer.parseInt(SnOrRn[1]);//将入栈的状态
                        System.out.println(next + " pushed into stack");
                          stack.push(next);
                          operations.add("移入"+tokens.element().type);
                          tokens.remove();
                    }
                    else if(SnOrRn[0].equals("R")){//规约
                          int reduceP = Integer.parseInt(SnOrRn[1]);//将规约的式子编号
                          Derivation d = CFG.p.get(reduceP);

                          String A = d.getLeft();
                          int r = d.getRightSize();

                          for(int i=0;i<r;i++)//Sm-r
                              stack.pop();

                          int gotoCol = parsingTable.turnGotoJumpToIndex(A);
                          int next = parsingTable.gotoTable[stack.peek()][gotoCol];
                          stack.push(next);

                          operations.add("规约，使用 "+reduceP+" 产生式");
                    }
                }
                showStack();
        }
```

八 运行例子

1.CFG:

```
S'->S
S->if ( B ) S ;|if ( B ) S ; else S ;|id = E|S ; S
B->B >= B|num|id
E->E + E|E - E|num|id
```

2.source code:

CFG.java | cfg.txt | DFA.java

```
1  x=y+2;
2  if (x>=2) z=1;
3  else z=z-1;
```

3.tokenStream

```
id,71,x
=,-1,=
id,71,y
+,-1,+
num,-1,2
;,-1,;
if,19,if
(,-1,(
id,71,x
>=,-1,>=
num,-1,2
),-1,)
id,71,z
=,-1,=
num,-1,1
;,-1,;
else,13,else
id,71,z
=,-1,=
id,71,z
-,-1,-
num,-1,1
;,-1,;
```

结果:

共有46个项集

分析表:

```
state  ( ) + − ; = >= else id if num $ B E S
0 x x x x x x x S37 S4 x x −1 −1 1
1 x x x x S2 x x x x x x acc −1 −1 −1
2 x x x x x x x S37 S4 x x −1 −1 3
3 x x x x S2 x x x x x x R4 −1 −1 −1
4 S5 x x x x x x x x x x −1 −1 −1
5 x x x x x x x S36 x S35 x 6 −1 −1
6 x S7 x x x x S33 x x x x x −1 −1 −1
7 x x x x x x x x S24 S15 x x −1 −1 8
8 x x x x x S9 x x x x x x x −1 −1 −1
9 x x x x R1 x x S10 S24 S15 x R1 −1 −1 13
10 x x x x x x x x S24 S15 x x −1 −1 11
11 x x x x S12 x x x x x x x −1 −1 −1
12 x x x x R2 x x x S24 S15 x R2 −1 −1 13
13 x x x x S14 x x x x x x x −1 −1 −1
14 x x x x x x x x S24 S15 x x −1 −1 13
15 S16 x x x x x x x x x x x −1 −1 −1
16 x x x x x x x x S36 x S35 x 17 −1 −1
17 x S18 x x x x S33 x x x x x −1 −1 −1
18 x x x x x x x x S24 S15 x x −1 −1 19
19 x x x x S20 x x x x x x x −1 −1 −1
20 x x x x R1 x x S21 S24 S15 x x −1 −1 13
21 x x x x x x x x S24 S15 x x −1 −1 22
22 x x x x S23 x x x x x x x −1 −1 −1
23 x x x x R2 x x x S24 S15 x x −1 −1 13
24 x x x x x S25 x x x x x x −1 −1 −1
25 x x x x x x x x S32 x S31 x −1 26 −1
26 x x S27 S29 R3 x x x x x x x −1 −1 −1
27 x x x x x x x x S32 x S31 x −1 28 −1
28 x x S27 S29 R8 x x x x x x x −1 −1 −1
29 x x x x x x x x S32 x S31 x −1 30 −1
30 x x S27 S29 R9 x x x x x x x −1 −1 −1
31 x x R10 R10 R10 x x x x x x x −1 −1 −1
32 x x R11 R11 R11 x x x x x x x −1 −1 −1
33 x x x x x x x x S36 x S35 x 34 −1 −1
34 x R5 x x x x S33 x x x x x −1 −1 −1
35 x R6 x x x x R6 x x x x x −1 −1 −1
36 x R7 x x x x R7 x x x x x −1 −1 −1
37 x x x x x S38 x x x x x x −1 −1 −1
38 x x x x x x x x S45 x S44 x −1 39 −1
39 x x S40 S42 R3 x x x x x x R3 −1 −1 −1
40 x x x x x x x x S45 x S44 x −1 41 −1
41 x x S40 S42 R8 x x x x x x R8 −1 −1 −1
42 x x x x x x x x S45 x S44 x −1 43 −1
43 x x S40 S42 R9 x x x x x x R9 −1 −1 −1
44 x x R10 R10 R10 x x x x x x R10 −1 −1 −1
45 x x R11 R11 R11 x x x x x x R11 −1 −1 −1
```

栈的变化过程:

使用S37

```
37 pushed into stack
now stack is: [0, 37]
```

使用S38

```
38 pushed into stack
now stack is: [0, 37, 38]
```

使用S45

```
45 pushed into stack
now stack is: [0, 37, 38, 45]
```

使用R11

```
now stack is: [0, 37, 38, 39]
```

使用S40

```
40 pushed into stack
now stack is: [0, 37, 38, 39, 40]
```

使用S44

```
44 pushed into stack
now stack is: [0, 37, 38, 39, 40, 44]
```

```
使用R10
now stack is: [0, 37, 38, 39, 40, 41]
使用R8
now stack is: [0, 37, 38, 39]
使用R3
now stack is: [0, 1]
使用S2
2 pushed into stack
now stack is: [0, 1, 2]
使用S4
4 pushed into stack
now stack is: [0, 1, 2, 4]
使用S5
5 pushed into stack
now stack is: [0, 1, 2, 4, 5]
使用S36
36 pushed into stack
now stack is: [0, 1, 2, 4, 5, 36]
使用R7
now stack is: [0, 1, 2, 4, 5, 6]
使用S33
33 pushed into stack
now stack is: [0, 1, 2, 4, 5, 6, 33]
使用S35
35 pushed into stack
now stack is: [0, 1, 2, 4, 5, 6, 33, 35]
使用R6
now stack is: [0, 1, 2, 4, 5, 6, 33, 34]
使用R5
now stack is: [0, 1, 2, 4, 5, 6]
使用S7
7 pushed into stack
now stack is: [0, 1, 2, 4, 5, 6, 7]
使用S24
24 pushed into stack
now stack is: [0, 1, 2, 4, 5, 6, 7, 24]
使用S25
25 pushed into stack
now stack is: [0, 1, 2, 4, 5, 6, 7, 24, 25]
使用S31
31 pushed into stack
now stack is: [0, 1, 2, 4, 5, 6, 7, 24, 25, 31]
使用R10
now stack is: [0, 1, 2, 4, 5, 6, 7, 24, 25, 26]
使用R3
now stack is: [0, 1, 2, 4, 5, 6, 7, 8]
使用S9
9 pushed into stack
now stack is: [0, 1, 2, 4, 5, 6, 7, 8, 9]
使用S10
10 pushed into stack
now stack is: [0, 1, 2, 4, 5, 6, 7, 8, 9, 10]
使用S24
24 pushed into stack
now stack is: [0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 24]
使用S25
25 pushed into stack
now stack is: [0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 24, 25]
使用S32
32 pushed into stack
now stack is: [0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 24, 25, 32]
使用R11
now stack is: [0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 24, 25, 26]
使用S29
29 pushed into stack
```

```
now stack is: [0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 24, 25, 26, 29]
使用S31
31 pushed into stack
now stack is: [0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 24, 25, 26, 29, 31]
使用R10
now stack is: [0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 24, 25, 26, 29, 30]
使用R9
now stack is: [0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 24, 25, 26]
使用R3
now stack is: [0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 11]
使用S12
12 pushed into stack
now stack is: [0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12]
使用R2
now stack is: [0, 1, 2, 3]
使用R4
now stack is: [0, 1]
```

RS动作描述:

```
移入id
移入=
移入id
规约，使用 11 产生式
移入+
移入num
规约，使用 10 产生式
规约，使用 8 产生式
规约，使用 3 产生式
移入;
移入if
移入(
移入id
规约，使用 7 产生式
移入>=
移入num
规约，使用 6 产生式
规约，使用 5 产生式
移入)
移入id
移入=
移入num
规约，使用 10 产生式
规约，使用 3 产生式
移入;
移入else
移入id
移入=
移入id
规约，使用 11 产生式
移入-
移入num
规约，使用 10 产生式
规约，使用 9 产生式
规约，使用 3 产生式
移入;
规约，使用 2 产生式
规约，使用 4 产生式
接受
```

如果将tokenStream里删除一项

```
id,71,x
=,-1,=
id,71,y
+,-1,+
num,-1,2
;,-1,;
if,19,if
(,-1,(
id,71,x
num,-1,2
),-1,)
id,71,z
=,-1,=
num,-1,1
;,-1,;
else,13,else
id,71,z
=,-1,=
id,71,z
-,-1,-
num,-1,1
```

则：

```
RS动作描述:
移入id
移入=
移入id
规约，使用 11 产生式
移入+
移入num
规约，使用 10 产生式
规约，使用 8 产生式
规约，使用 3 产生式
移入;
移入if
移入(
移入id
语法错误
```

九 遇到的问题以及相关的解决方式

（1）刚开始求first集时思路不明确，甚至打算人工计算，硬编码写入。后来考虑到first集的特性想到可以采用递归。

（2）在构造DFA时，由于没注意到java函数 的引用传递，导致临时的DFAState中的项其实是其他项集中的项。而临时的DFAState会对这些项进行操作，改变它的属性，导致之前的项集中项被改变，导致程序错误。后来增加一个函数，返回一个属性相同的新对象。

（3）该文法存在SR冲突。 后来选择在冲突处全部选择移入。

十 你的感受以及评论

通过这次实验，我对于LR(1)分析法的理解更加透彻，对其中涉及的算法进行实现后，不仅提高了自身的程序设计能力，而且纠正了理解误差。除此之外，还发现了自己对于引用类型处理的误区。构造状态机的过程使我更加感受到状态机在编译中重要性。