实验编号：3**四川师大 编译原理 实验报告 2019**年**4**月**12**日

**计算机科学学院** 2017级 4 班 实验名称： LR1分析法

姓名：\_\_韩勇\_\_ 学号：\_\_2017110409\_ 指导老师：\_\_吴贞东\_\_ 实验成绩:\_\_\_\_\_

**实验 三 LR1分析法\_**

1. 实验目的及要求

构造LR(1)分析程序，利用它进行语法分析，判断给出的符号串是否为该文 法识别的句子，了解LR（K）分析方法是严格的从左向右扫描，和自底向上的语 法分析方法。

1、编程时注意编程风格：空行的使用、注释的使用、缩进的使用等。

2、如果遇到错误的表达式，应输出错误提示信息。

3、程序输入/输出实例： 输入一以#结束的符号串(包括+\*（）i#)：在此位置输入符号串 输出过程如下： 步骤 状态栈 符号栈 剩余输入串 动作 1 0 # i+i\*i# 移进 i+i\*i的LR 分析过程 步骤 状态栈 符号栈 输入串 动作说明 1 0 # i+i\*i# ACTION[0,i]=S5,状态5 入栈 2 05 #i +i\*i# r6: F→i归约,GOTO(0,F)=3 入栈 3 03 #F +i\*i# r4: T→F 归约,GOTO(0,T)=3 入栈 4 02 #T +i\*i# r2: E→T 归约,GOTO(0,E)=1 入栈 19 5 01 #E +i\*i# ACTION[1,+]=S6,状态6 入栈 6 016 #E+ i\*i# ACTION[6,i]=S5,状态5 入栈 7 0165 #E+i \*i# r6: F→i归约,GOTO(6,F)=3 入栈 8 0163 #E+F \*i# r4: T→F 归约,GOTO(6,T)=9 入栈 9 0169 #E+T \*i# ACTION[9,\*]=S7,状态7 入栈 10 01697 #E+T\* i# ACTION[7,i]=S5,状态5 入栈 11 016975 #E+T\*i # r6:F→i归约,GOTO(7,F)=10 入栈 12 0169710 #E+T\*F # r3: T→T\*F 归约,GOTO(6,T)=9 入栈 13 0169 #E+T # r1:E→E+T,GOTO(0,E)=1 入栈 14 01 #E # Acc：分析成功 4、输入符号串为非法符号串(或者为合法符号串) 算术表达式文法的LR 分析表 状 态 ACTION GOTO i + \* ( ) # E T F 0 S5 S4 1 2 3 1 S6 acc 2 r2 S7 r2 r2 3 r4 r4 r4 r4 4 S5 S4 8 2 3 5 r6 r6 r6 r6 6 S5 S4 9 3 7 S5 S4 10 8 S6 S11 9 r1 S7 r1 r1 10 r3 r3 r3 r3 11 r5 r5 r5 r5

1. 实验内容

对下列文法，用LR（1）分析法对任意输入的符号串进行分析：

（1）E-> E+T

（2）E->T

（3）T-> T\*F

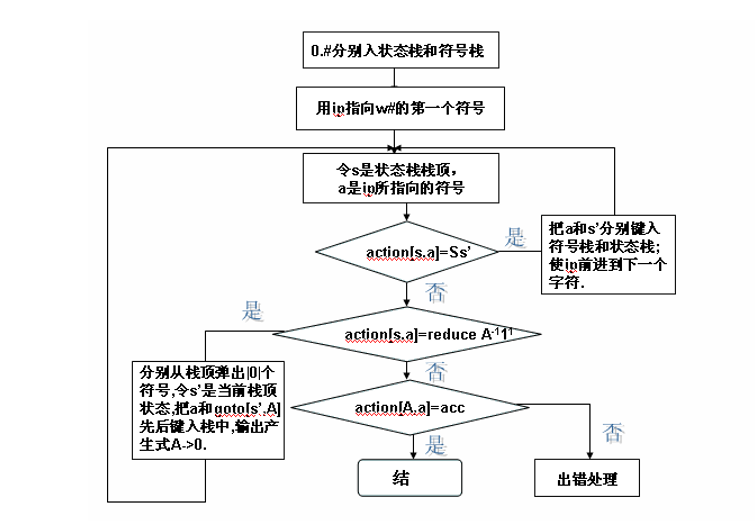
（4）T->F

（5）F-> (E)

（6）F->i

1. 实验主要流程、基本操作或核心代码、算法片段（该部分如不够填写，请另加附页）

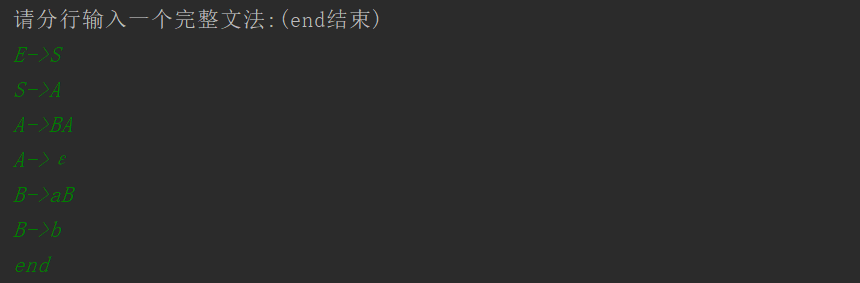
流程图：

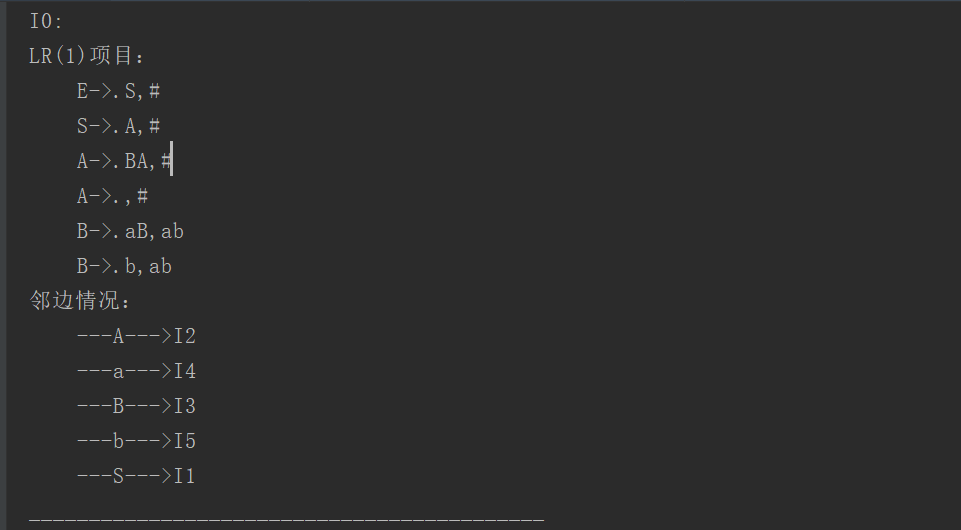


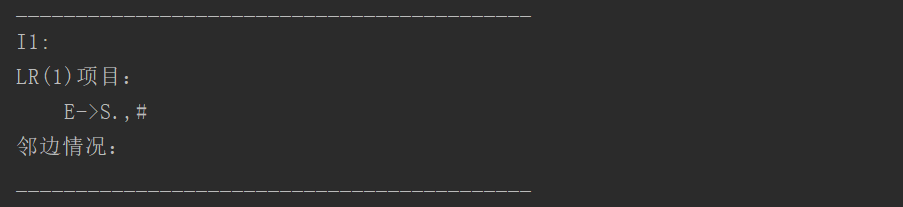
源程序：

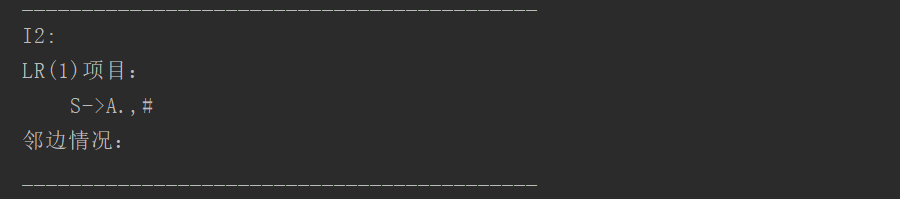
package compiler\_test;  
  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.Objects;  
import java.util.Set;  
  
public class Test\_03 {  
 //项目所属产生式,里面保存了项目的左部  
 private ProductionFormula productionFormula;  
 //项目右部 点之前的部分  
 private String rightPartBeforPoint;  
 //项目右部 点之后的部分  
 private String rightPartAfterPoint;  
 //展望信息 逗号之后的信息  
 private String rightPartAfterComma;  
  
 //移进项目  
 private Boolean isShift;  
 //规约项目  
 private Boolean isReduce;  
 //接受项目  
 private Boolean isAccept;  
 //待约项目  
 private Boolean isWaitReduce;  
  
 //是否经过closure函数处理  
 private Boolean afterClosure;  
  
 public Boolean getAfterClosure() {  
 return afterClosure;  
 }  
  
 public void setAfterClosure(Boolean afterClosure) {  
 this.afterClosure = afterClosure;  
 }  
  
 public ProductionFormula getProductionFormula() {  
 return productionFormula;  
 }  
  
 public void setProductionFormula(ProductionFormula productionFormula) {  
 this.productionFormula = productionFormula;  
 }  
  
 public String getRightPartBeforPoint() {  
 return rightPartBeforPoint;  
 }  
  
 public void setRightPartBeforPoint(String rightPartBeforPoint) {  
 this.rightPartBeforPoint = rightPartBeforPoint;  
 }  
  
 public String getRightPartAfterPoint() {  
 return rightPartAfterPoint;  
 }  
  
 public void setRightPartAfterPoint(String rightPartAfterPoint) {  
 this.rightPartAfterPoint = rightPartAfterPoint;  
 }  
  
 public String getRightPartAfterComma() {  
 return rightPartAfterComma;  
 }  
  
 public void setRightPartAfterComma(String rightPartAfterComma) {  
 this.rightPartAfterComma = rightPartAfterComma;  
 }  
  
 public Boolean getShift() {  
 return isShift;  
 }  
  
 public void setShift(Boolean shift) {  
 isShift = shift;  
 }  
  
 public Boolean getReduce() {  
 return isReduce;  
 }  
  
 public void setReduce(Boolean reduce) {  
 isReduce = reduce;  
 }  
  
 public Boolean getAccept() {  
 return isAccept;  
 }  
  
 public void setAccept(Boolean accept) {  
 isAccept = accept;  
 }  
  
 public Boolean getWaitReduce() {  
 return isWaitReduce;  
 }  
  
 public void setWaitReduce(Boolean waitReduce) {  
 isWaitReduce = waitReduce;  
 }  
  
 public Test\_03() {  
 }  
  
 */\*\*  
 \** ***@param*** *productionFormula 产生式  
 \** ***@param*** *rightPartBeforPoint 点号之前  
 \** ***@param*** *rightPartAfterPoint 点号之后  
 \** ***@param*** *rightPartAfterComma 展望串  
 \** ***@param*** *allNonTerminals 所有的非终结符  
 \** ***@param*** *begin 文法开始符号  
 \*/* public Test\_03(ProductionFormula productionFormula, String rightPartBeforPoint, String rightPartAfterPoint, String rightPartAfterComma, ArrayList<String> allNonTerminals, String begin) {  
 this.productionFormula = productionFormula;  
 this.rightPartBeforPoint = rightPartBeforPoint;  
 this.rightPartAfterPoint = rightPartAfterPoint;  
 this.rightPartAfterComma = rightPartAfterComma;  
 this.isShift = false;  
 this.isReduce = false;  
 this.isAccept = false;  
 this.isWaitReduce = false;  
 this.afterClosure = false;  
 if (rightPartAfterPoint.length() == 0) {  
 if (productionFormula.getLeftPart().equals(begin)) {  
 isAccept = true;//接受项目，也是规约项目  
 }  
 this.isReduce = true;//规约项目  
 } else if (allNonTerminals.contains(rightPartAfterPoint.substring(0, 1))) {  
 this.isWaitReduce = true; //待约项目  
 } else {  
 this.isShift = true; //移进项目  
 }  
 }  
  
 @Override  
 public String toString() {  
 return productionFormula.getLeftPart() + "->" + rightPartBeforPoint + "." + rightPartAfterPoint + "," + rightPartAfterComma  
 /\*+ " "  
 + "[移进=" + isShift +  
 ", 规约=" + isReduce +  
 ", 接受=" + isAccept +  
 ", 待约=" + isWaitReduce + "]"\*/;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Test\_03 that = (Test\_03) o;  
 return Objects.equals(productionFormula, that.productionFormula) &&  
 Objects.equals(rightPartBeforPoint, that.rightPartBeforPoint) &&  
 Objects.equals(rightPartAfterPoint, that.rightPartAfterPoint) &&  
 Objects.equals(rightPartAfterComma, that.rightPartAfterComma) &&  
 Objects.equals(isShift, that.isShift) &&  
 Objects.equals(isReduce, that.isReduce) &&  
 Objects.equals(isAccept, that.isAccept) &&  
 Objects.equals(isWaitReduce, that.isWaitReduce);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.hash(productionFormula, rightPartBeforPoint, rightPartAfterPoint, rightPartAfterComma, isShift, isReduce, isAccept, isWaitReduce);  
 }  
}  
  
*/\*\*  
 \** ***@author*** *magentaLi  
 \* 代表一个LR(1)项目族 即I0,I1,I2等  
 \*/*class LR\_1ProjectAggregate {  
  
 //状态号码 0 1 2 ...  
 private int index;  
 // 存储LR(1)项目的arrayList  
 private ArrayList<Test\_03> lr\_1Projects;  
 //存储连接边的hashMap  
 private HashMap<String, LR\_1ProjectAggregate> adjacentSide;  
 //是否经过Go函数处理  
 private Boolean afterGo;  
  
 public Boolean getAfterGo() {  
 return afterGo;  
 }  
  
 public void setAfterGo(Boolean afterGo) {  
 this.afterGo = afterGo;  
 }  
  
 public int getIndex() {  
 return index;  
 }  
  
 public void setIndex(int index) {  
 this.index = index;  
 }  
  
 public ArrayList<Test\_03> getLr\_1Projects() {  
 return lr\_1Projects;  
 }  
  
 public void setLr\_1Projects(ArrayList<Test\_03> lr\_1Projects) {  
 this.lr\_1Projects = lr\_1Projects;  
 }  
  
 public HashMap<String, LR\_1ProjectAggregate> getAdjacentSide() {  
 return adjacentSide;  
 }  
  
 public void setAdjacentSide(HashMap<String, LR\_1ProjectAggregate> adjacentSide) {  
 this.adjacentSide = adjacentSide;  
 }  
  
 public LR\_1ProjectAggregate(int index, ArrayList<Test\_03> lr\_1Projects, HashMap<String, LR\_1ProjectAggregate> adjacentSide) {  
 this.index = index;  
 this.lr\_1Projects = lr\_1Projects;  
 this.adjacentSide = adjacentSide;  
 this.afterGo = false;  
 }  
  
 @Override  
 public String toString() {  
 System.out.println("I" + index + ":");  
 System.out.println("LR(1)项目：");  
 for (Test\_03 lr\_1Project : lr\_1Projects) {  
 System.out.println("\t" + lr\_1Project.toString());  
 }  
 System.out.println("邻边情况：");  
 Set<String> keySet = adjacentSide.keySet();  
 for (String key : keySet) {  
 System.out.println("\t" + "---" + key + "--->" + "I" + adjacentSide.get(key).getIndex());  
 }  
 System.out.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");  
 return "";  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 LR\_1ProjectAggregate that = (LR\_1ProjectAggregate) o;  
 return index == that.index &&  
 Objects.equals(lr\_1Projects, that.lr\_1Projects) &&  
 Objects.equals(adjacentSide, that.adjacentSide);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.hash(index, lr\_1Projects, adjacentSide);  
 }  
}  
  
*/\*\*  
 \** ***@author*** *magentaLi  
 \*/*/\*  
 产生式类  
\*/  
 class ProductionFormula {  
  
 private String leftPart;  
 private String rightPart;  
  
 public ProductionFormula(String leftPart, String rightPart) {  
 this.leftPart = leftPart;  
 this.rightPart = rightPart;  
 }  
  
 public ProductionFormula() {  
 }  
  
 public String getLeftPart() {  
 return leftPart;  
 }  
  
 public void setLeftPart(String leftPart) {  
 this.leftPart = leftPart;  
 }  
  
 public String getRightPart() {  
 return rightPart;  
 }  
  
 public void setRightPart(String rightPart) {  
 this.rightPart = rightPart;  
 }  
  
 @Override  
 public String toString() {  
 return leftPart + "->" + rightPart;  
 }  
}  
  
*/\*\*  
 \** ***@author*** *magentaLi  
 \*/*/\*  
E->S  
S->A  
A->BA  
A->ε  
B->aB  
B->b  
end  
\*/  
class LRMain {  
 public static void main(String[] args) {  
 //输入一个文法  
 ArrayList<String[]> input = getInput();  
 //求出所有的非终结符  
 ArrayList<String> allNonTerminals = getAllNonTerminals(input);  
 //求出能直接推出ε的非终结符  
 ArrayList<String> nonTerminalsNullable = getAllNonTerminalsNullable(input);  
 //求解FIRST集  
 HashMap<String, ArrayList<String>> first = getFirst(allNonTerminals, nonTerminalsNullable, input);  
  
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*以下为新添加的代码  
 ArrayList<ProductionFormula> productionFormulas = new ArrayList<>();  
 //将文法转存到类中  
 for (String[] strings : input) {  
 ProductionFormula productionFormula = new ProductionFormula(strings[0], strings[1]);  
 productionFormulas.add(productionFormula);  
 }  
 //求解文法开始符号  
 String begin = productionFormulas.get(0).getLeftPart();  
 //求解DFA的过程  
 //第一个LR(1)项目  
 Test\_03 lr\_1Project = new Test\_03(productionFormulas.get(0), "", productionFormulas.get(0).getRightPart(), "#", allNonTerminals, begin);  
 ArrayList<Test\_03> lr\_1Projects = new ArrayList<>();  
 lr\_1Projects.add(lr\_1Project);  
 //声明 I0  
 LR\_1ProjectAggregate I0 = new LR\_1ProjectAggregate(0, lr\_1Projects, new HashMap<>());  
 //声明 DFA  
 ArrayList<LR\_1ProjectAggregate> DFA = new ArrayList<>();  
 //构造初始的 I0  
 DFA.add(I0);  
  
 //对 I0 调用closure函数  
 closure(I0, productionFormulas, allNonTerminals, first, begin);  
 //用于判断DFA是否还在增长  
 ArrayList<LR\_1ProjectAggregate> cloneDFA = new ArrayList<>(DFA);  
 while (!allAfterGo(DFA)) {  
 for (LR\_1ProjectAggregate I : DFA)  
 if (!I.getAfterGo()) {  
 go(I, cloneDFA, allNonTerminals, begin, productionFormulas, first);  
 }  
 DFA.clear();  
 DFA.addAll(cloneDFA);  
 }  
 System.out.println(DFA);  
 }  
  
 private static HashMap<String, ArrayList<String>> getFirst(ArrayList<String> allNonTerminals, ArrayList<String> nonTerminalsNullable, ArrayList<String[]> input) {  
 return null;  
 }  
  
 private static ArrayList<String> getAllNonTerminalsNullable(ArrayList<String[]> input) {  
 return null;  
 }  
  
 private static ArrayList<String> getAllNonTerminals(ArrayList<String[]> input) {  
 return null;  
 }  
  
 private static ArrayList<String[]> getInput() {  
 return null;  
 }  
  
 */\*\*  
 \* 判断所有的LR(1)项目集是否都经过Go函数处理  
 \*  
 \** ***@param*** *DFA DFA  
 \** ***@return*** *true OR false  
 \*/* private static Boolean allAfterGo(ArrayList<LR\_1ProjectAggregate> DFA) {  
 for (LR\_1ProjectAggregate I : DFA)  
 if (!I.getAfterGo()) {  
 return false;  
 }  
 return true;  
 }  
  
 */\*\*  
 \* go函数  
 \*  
 \** ***@param*** *I 项目族  
 \** ***@param*** *DFA DFA  
 \** ***@param*** *allNonTerminals 所有的非终结符  
 \** ***@param*** *begin 文法开始符号  
 \** ***@param*** *productionFormulas 所有的产生式  
 \** ***@param*** *first first集合  
 \*/* private static void go(LR\_1ProjectAggregate I, ArrayList<LR\_1ProjectAggregate> DFA, ArrayList<String> allNonTerminals, String begin, ArrayList<ProductionFormula> productionFormulas, HashMap<String, ArrayList<String>> first) {  
 //遍历所有的LR(1)项目  
 for (Test\_03 l : I.getLr\_1Projects()) {  
 //如果是规约项目  
 if (l.getReduce()) {  
 continue;  
 }  
 //如果不是规约项目  
 String afterPoint = l.getRightPartAfterPoint();  
 if (afterPoint.length() != 0) {  
 String firstChar = afterPoint.substring(0, 1);  
 //构造有个新的LR(1)项目  
 Test\_03 newLR\_1 = makeLR1ForGo(l, allNonTerminals, begin);  
 int res = notNewLR\_1(DFA, newLR\_1);  
 if (res == 10000) {//是全新的LR(1)项目  
 //构造一个全新的I  
 int index = DFA.size();  
 ArrayList<Test\_03> newList = new ArrayList<>();  
 newList.add(newLR\_1);  
 LR\_1ProjectAggregate newI = new LR\_1ProjectAggregate(index, newList, new HashMap<>());  
 //对新的I调用closure函数进行扩充  
 closure(newI, productionFormulas, allNonTerminals, first, begin);  
 DFA.add(newI);  
 //将邻边加入原来的I中  
 I.getAdjacentSide().put(firstChar, getAggregateByIndex(index, DFA));  
 } else {//不是全新的LR(1)项目  
 //则将邻边加入I中  
 I.getAdjacentSide().put(firstChar, getAggregateByIndex(res, DFA));  
 }  
 } else break;  
 }  
 //将 I 置为已经过Go函数处理的项目族  
 I.setAfterGo(true);  
 }  
  
  
 */\*\*  
 \* 通过index查找I  
 \*  
 \** ***@param*** *index 下标  
 \** ***@param*** *DFA DFA  
 \** ***@return*** *I  
 \*/* private static LR\_1ProjectAggregate getAggregateByIndex(int index, ArrayList<LR\_1ProjectAggregate> DFA) {  
 for (LR\_1ProjectAggregate I : DFA) {  
 if (I.getIndex() == index) {  
 return I;  
 }  
 }  
 return null;  
 }  
  
 */\*\*  
 \** ***@param*** *DFA DFA  
 \** ***@param*** *lr\_1Project 待判断是否被包含的LR(1)项目  
 \** ***@return*** *10000不包含 或者包含这个LR(1)项目的项目族的编号  
 \*/* private static int notNewLR\_1(ArrayList<LR\_1ProjectAggregate> DFA, Test\_03 lr\_1Project) {  
 for (LR\_1ProjectAggregate aggregate : DFA) {  
 for (Test\_03 l : aggregate.getLr\_1Projects()) {  
 if (l.equals(lr\_1Project)) {  
 return aggregate.getIndex();  
 }  
 }  
 }  
 return 10000;  
 }  
  
 */\*\*  
 \* 为go函数构建新的LR(1)项目  
 \*  
 \** ***@param*** *project LR(1)项目  
 \** ***@param*** *allNonTerminals 所有的非终结符  
 \** ***@param*** *begin 文法开始符号  
 \** ***@return*** *LRI(1)项目  
 \*/* private static Test\_03 makeLR1ForGo(Test\_03 project, ArrayList<String> allNonTerminals, String begin) {  
 String rightPartBeforPoint = project.getRightPartBeforPoint();  
 String rightPartAfterPoint = project.getRightPartAfterPoint();  
 String newRightPartBeforPoint = rightPartBeforPoint + rightPartAfterPoint.substring(0, 1);  
 String newRightPartAfterPoint = rightPartAfterPoint.substring(1);  
 return new Test\_03(project.getProductionFormula(), newRightPartBeforPoint, newRightPartAfterPoint, project.getRightPartAfterComma(), allNonTerminals, begin);  
 }  
  
 */\*\*  
 \* closure 函数  
 \*  
 \** ***@param*** *I 项目族  
 \** ***@param*** *productionFormulas 产生式  
 \** ***@param*** *allNonTerminals 所有的非终结符  
 \** ***@param*** *first first集合  
 \** ***@param*** *begin 文法开始符号  
 \*/* private static void closure(LR\_1ProjectAggregate I, ArrayList<ProductionFormula> productionFormulas, ArrayList<String> allNonTerminals, HashMap<String, ArrayList<String>> first, String begin) {  
 ArrayList<Test\_03> lr\_1Projects = I.getLr\_1Projects();  
 ArrayList<Test\_03> clone = new ArrayList<>(lr\_1Projects);  
 while (!allAfterClosure(I)) {  
 //遍历I里面的LR(1)项目  
 for (Test\_03 lr\_1Project : lr\_1Projects) {  
 //如果该LR(1)项目是待约项目或者移进项目  
 if ((lr\_1Project.getWaitReduce() || lr\_1Project.getReduce()) && !lr\_1Project.getAfterClosure()) {  
 //则用该LR(1)项目构造LR(1)新得项目并加入I中  
 ArrayList<Test\_03> projects = makeLR1ForClosure(lr\_1Project, productionFormulas, allNonTerminals, first, begin);  
 lr\_1Project.setAfterClosure(true);  
 clone.addAll(projects);  
 }  
 }  
 lr\_1Projects.clear();  
 lr\_1Projects.addAll(clone);  
 }  
 }  
  
 */\*\*  
 \* 判断项目族中的所有项目是否都经过处理了  
 \*  
 \** ***@param*** *I 项目族  
 \** ***@return*** *true Or false  
 \*/* private static Boolean allAfterClosure(LR\_1ProjectAggregate I) {  
 ArrayList<Test\_03> projects = I.getLr\_1Projects();  
 for (Test\_03 p : projects) {  
 if (p.getWaitReduce()) {  
 if (!p.getAfterClosure()) {  
 return false;  
 }  
 }  
 }  
 return true;  
 }  
  
 */\*\*  
 \* 新增LR(1)项目  
 \*  
 \** ***@param*** *lr\_1Project LR(1)项目  
 \** ***@param*** *productionFormulas 产生式  
 \** ***@param*** *allNonTerminals 所有的非终结符  
 \** ***@param*** *first first集合  
 \** ***@param*** *begin 文法开始符号  
 \** ***@return*** *新增的LR(1)项目集合  
 \*/* private static ArrayList<Test\_03> makeLR1ForClosure(Test\_03 lr\_1Project, ArrayList<ProductionFormula> productionFormulas, ArrayList<String> allNonTerminals, HashMap<String, ArrayList<String>> first, String begin) {  
 ArrayList<Test\_03> myRes = new ArrayList<>();  
 //1.获取点后的第一个非终结符  
 if (lr\_1Project.getRightPartAfterPoint().length() >= 1) {  
 String leftChar = lr\_1Project.getRightPartAfterPoint().substring(0, 1);  
 //当前LR(1）项目的点之后的部分  
 String currentPF = lr\_1Project.getRightPartAfterPoint();  
 //当前LR(1)项目逗号后的部分  
 String currentAFC = lr\_1Project.getRightPartAfterComma();  
 //2.获取相关的产生式  
 ArrayList<ProductionFormula> productionFormulasUseful = getProductionFormulaByLeft(productionFormulas, leftChar);  
 for (ProductionFormula p : productionFormulasUseful) {  
 //3.计算逗号前的部分  
 //3.1 点前的部分  
 String resOfRightPartBeforPoint = "";  
 //3.2 点后的部分  
 String resOfRightPartAfterPoint = p.getRightPart();  
 if (p.getRightPart().equals("ε")) {  
 resOfRightPartAfterPoint = "";  
 }  
 //4.计算逗号后的部分  
 String resOfRightPartAfterComma = "";  
 //4.1获取点后2位 的串数组  
 if (currentPF.length() <= 1) {  
 resOfRightPartAfterComma = "#";  
 char[] chars = currentAFC.toCharArray();  
 for (char aChar : chars)  
 if (!resOfRightPartAfterComma.contains(String.valueOf(aChar))) {  
 resOfRightPartAfterComma += String.valueOf(aChar);  
 }  
 if (resOfRightPartAfterComma.length() > 1)  
 resOfRightPartAfterComma = resOfRightPartAfterComma.replace("#", "");  
 } else {  
 String s = currentPF.substring(currentPF.indexOf(".") + 2);  
 if (s.equals("")) {  
 resOfRightPartAfterComma = "#";  
 char[] chars = currentAFC.toCharArray();  
 for (char aChar : chars)  
 if (!resOfRightPartAfterComma.contains(String.valueOf(aChar))) {  
 resOfRightPartAfterComma += String.valueOf(aChar);  
 }  
 if (resOfRightPartAfterComma.length() > 1)  
 resOfRightPartAfterComma = resOfRightPartAfterComma.replace("#", "");  
 } else {  
 ArrayList<String> strings = getStringFirst(allNonTerminals, first, currentPF.toCharArray());  
 //4.2拼接逗号后的部分  
 for (String res : strings) {  
 resOfRightPartAfterComma += res;  
 }  
 char[] chars = currentAFC.toCharArray();  
 for (char aChar : chars)  
 if (!resOfRightPartAfterComma.contains(String.valueOf(aChar))) {  
 resOfRightPartAfterComma += String.valueOf(aChar);  
 }  
 if (resOfRightPartAfterComma.length() > 1) {  
 resOfRightPartAfterComma = resOfRightPartAfterComma.replace("#", "");  
 }  
 }  
 }  
 Test\_03 lr\_1Project1 = new Test\_03(p, resOfRightPartBeforPoint, resOfRightPartAfterPoint, resOfRightPartAfterComma, allNonTerminals, begin);  
 myRes.add(lr\_1Project1);  
 }  
 } else {  
 return myRes;  
 }  
 return myRes;  
 }  
  
 private static ArrayList<String> getStringFirst(ArrayList<String> allNonTerminals, HashMap<String, ArrayList<String>> first, char[] toCharArray) {  
 return null;  
 }  
  
  
 */\*\*  
 \* 要增加的产生式集合  
 \*  
 \** ***@param*** *productionFormulas 所有的产生式  
 \** ***@param*** *left 左部  
 \** ***@return*** *产生式集合  
 \*/* private static ArrayList<ProductionFormula> getProductionFormulaByLeft(ArrayList<ProductionFormula> productionFormulas, String left) {  
 ArrayList<ProductionFormula> res = new ArrayList<>();  
 for (ProductionFormula p : productionFormulas) {  
 if (p.getLeftPart().equals(left)) {  
 res.add(p);  
 }  
 }  
 return res;  
 }  
  
}

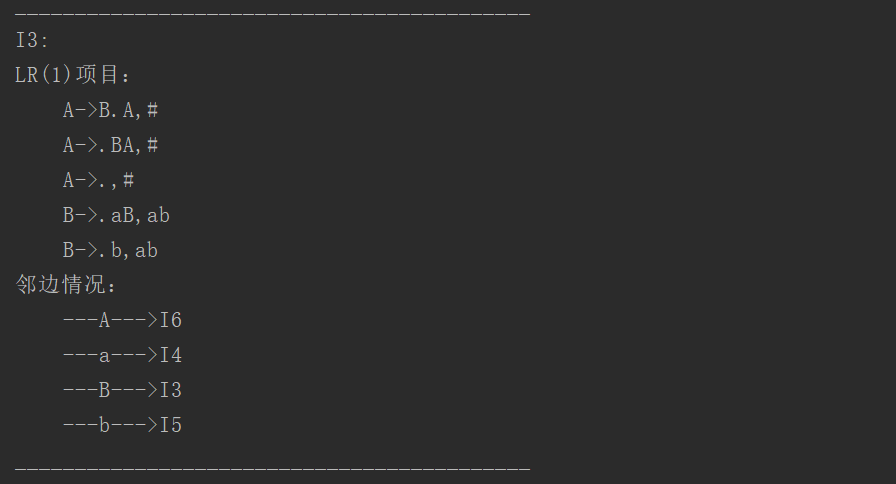
运行结果截图：

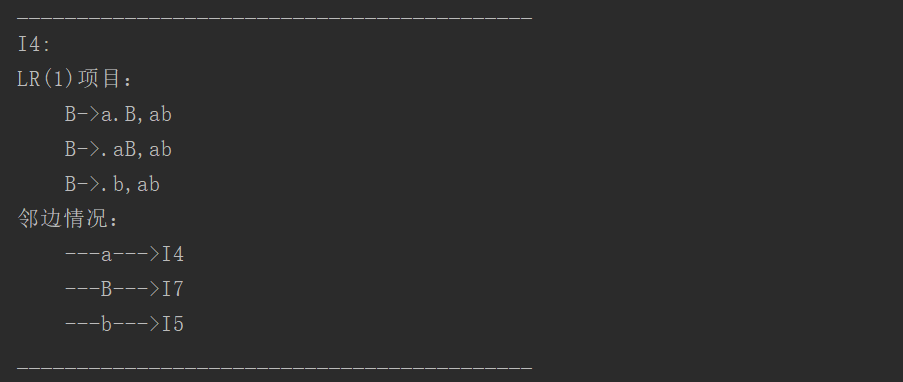


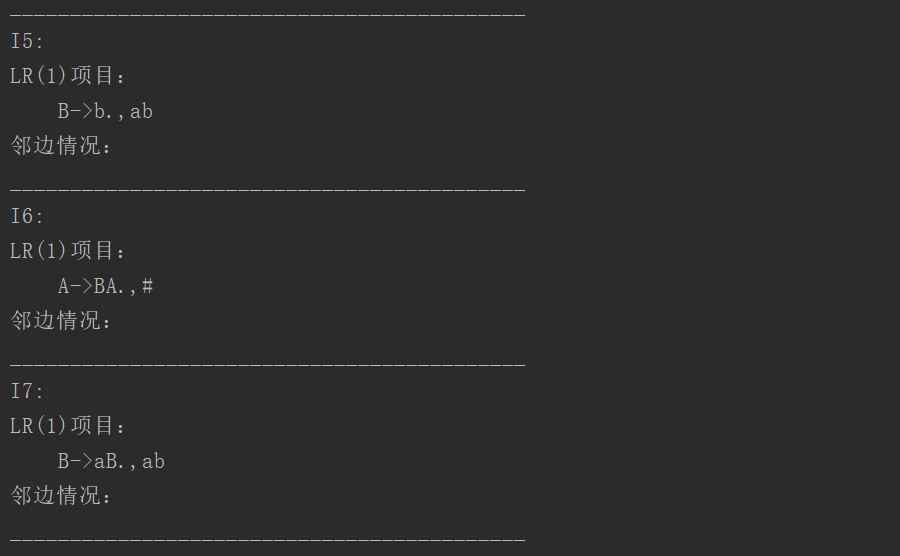












1. 实验结果的分析与评价（该部分如不够填写，请另加附页）

本实验的难度相比于上一个的难度要小一些，关键在于弄清楚识别活前缀的DFA的构造过程，以及存储各类结果的数据结构即可。

注：实验成绩等级分为（90－100分）优，（80－89分）良，(70-79分)中，（60－69分）及格，（59分）不及格。