湖南大學
HUNAN UNIVERSITY

课程实验报告

课　程　名　称：编译原理

实验项目名称：上下文无关文法的 DFA 构建

专　业　班　级：软件 2105

姓　　名：马小梅

学　　号：202126010530

指　导　教　师：杨金民

完　成　时　间：2024 年 4 月 17 日

信息科学与工程学院

实验题目：语法分析表的构造

实验目的：

（1）学习和掌握消除左递归和提取左公因子的方法

（2）学习和掌握求解 FIRST 和 FOLLOW 函数的一般方法和步骤

（3）学习和掌握 DFA 的方法和步骤

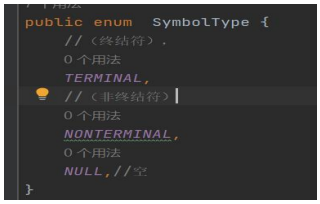
（4）掌握判断 LL(1)文法和 SLR(1)文法的一般算法

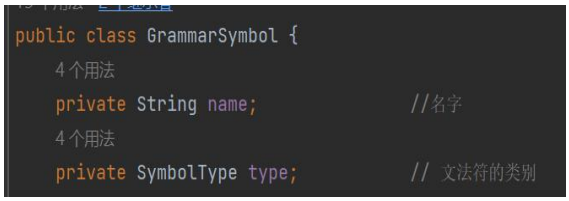
（5）掌握构造 LL(1)语法分析表和 LR(1)语法分析表的一般算法

实验内容及操作步骤：

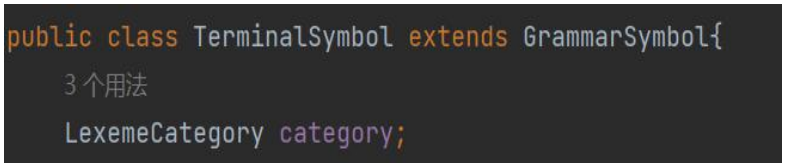
一、基本数据结构

1）文法符：作为终结符和非终结符的基类，其中 SymbolType 有三种类型：TERMINAL（终结符），NONTERMINAL（非终结符），NULL（ε）。





2）终结符定义：



3）词法类型定义。其中 LexemeCategory 定义

4）非终结符定义

```
public class NonTerminalSymbol extends GrammarSymbol{
    //有关非终结符构成的产生式
   6 个用法
   private ArrayList <Production>  pProductionTable;
    //产生式的个数
    6 个用法
    private int numOfProduction;
    //非终结符的FIRST函数值
    6 个用法
    private Set <TerminalSymbol>  pFirstSet;
    //非终结符的FOLLOW函数值
    7 个用法
    private Set <TerminalSymbol>  pFollowSet;
    //求非终结符的FOLLOW函数值时存放所依赖的非终结符。
    6 个用法
    private Set <NonTerminalSymbol> pDependentSetInFollow;
```

4）产生式定义

```
public class Production {
    //产生式序号，起标识作用
    4 个用法
    private int productionId;
    //产生式体中包含的文法符个数
    7 个用法
    private int bodySize;
    //产生式体中包含的文法符
    7 个用法
    private ArrayList <GrammarSymbol>  pBodySymbolTable;
    //产生式的FIRST函数值
    3 个用法
    private Set <TerminalSymbol>  pFirstSet;

    1 个用法
    public static int idNumber = 0;//产生式序号
```

5）LL(1)语法分析表中，每一格的定义

```
class Cell {
    2 个用法
    private NonTerminalSymbol  nonTerminalSymbol;
    2 个用法
    private TerminalSymbol  terminalSymbol;
    2 个用法
    private Production  production;
```

6）LR(0)项目定义

```
public class LR0Item {
       //LR(0)项目
   6 个用法
   private NonTerminalSymbol  nonTerminalSymbol;     //非终结符
   6 个用法
   private Production production;                  //产生式
   6 个用法
   private int dotPosition;          //圆点的位置
   6 个用法
   private ItemCategoy type;  //类型。两种：CORE(核心项)；NONCORE(非核心项)
```

7）LR(0)项集定义

```java
public class ItemSet {
    //LR(0)项集
    // 状态序号
    5 个用法
    private int stateId;
    // LR0 项目表
    9 个用法
    private ArrayList<LR0Item> pItemTable;
    1 个用法
    private static int idNumber = 0;
```

8）变迁边定义

```java
public class TransitionEdge {
    //变迁边
    /**
     * 驱动文法符有终结符和非终结符两种。
     * 在给driverSymbol赋值时，要先对驱动符进行强制类型转换，变成GrammarSymbol *类型。
     */
    4 个用法
    private GrammarSymbol driverSymbol;

    // 出发项集
    4 个用法
    private ItemSet fromItemSet; //起点
    // 到达项集
    4 个用法
    private ItemSet toItemSet; //目标
```

9）DFA 定义

```java
class Cell {
    2 个用法
    private NonTerminalSymbol  nonTerminalSymbol;
    2 个用法
    private TerminalSymbol  terminalSymbol;
    2 个用法
    private Production  production;
```

10）LR(1)语法分析表中 ACTION 部分的定义，其中 ActionCategory 有三种类型：r（reduce 规约，id 为产生式）、s（shift 移入，id 为状态）、a（accept，接受）

```java
public class ActionCell {
//    LR(1)语法分析表包含ACTION和GOTO两个部分
    //纵坐标：状态序号
    2 个用法
    private int stateId;
    // 横坐标：终结符
    2 个用法
    private String terminalSymbolName;
    // Action 类别
    /**
     * 取值有三种：'r'和's'，以及'a'。'r'是规约，'s'是移入，'a'是接受。
     * 当Action类别为规约时，id的取值为产生式id，当Action类别为移入时，id的取值为下一状态id。
     */
    2 个用法
    private ActionCategory type ;
    // Action 的 id
    2 个用法
    private int id;
```

11）LR(1)语法分析表中 GOTO 部分的定义

```java
public class GotoCell {
    // 纵坐标：状态序号
    2 个用法
    private int stateId;
    // 横坐标：非终结符
    2 个用法
    private String nonTerminalSymbolName;
    // 下一个状态
    2 个用法
    private int nextStateId;
```

12）产生式概述表定义

```java
public class ProductionInfo {
    // 产生式序号
    2 个用法
    private int indexId;
    // 头部非终结符
    2 个用法
    private String headName;
    // 产生式体中文法符的个数
    2 个用法
    private int bodySize;

    1 个用法
    private static int idNum = 0;
```

二、针对 LL 语法分析，实现如下函数

1）产生式的 FIRST 函数求解

实现方法：对于产生式 $X->Y_1Y_2.....Y_{n-1}Y_n$, $FIRST(Y1)\in FIRST(X)$ 显然成立。但如果从 Y_1 至 Y_j，$0<j<n$，全为非终结符，且都含虚产生式，那么 $FIRST(Y_{j+1})$ 属于 FIRST(X)。关联求文法符的 FERST 函数

实现函数：

```java
//产生式的FIRST集
9 个用法
public static Set<TerminalSymbol> getFirstSet(Production production) {
    //1.是否存在 连续的ε
    boolean flag = true;
    int i = 0;
    Set<TerminalSymbol> FirstList = new HashSet<>();
    ArrayList<GrammarSymbol> grammarSymbols = production.getpBodySymbolTable();
    //新建 ε
    TerminalSymbol epsilon = new TerminalSymbol( name: "ε", SymbolType.NULL);
    // 当前面文法符FIRST都包含ε时：从 Y1至 Yj，0<j<n，全为非终结符，且都含虚产生式，那么
    //FIRST=FIRST（Yj)- ε。( ε在Yj中则FIRST加入 ε)
    while(flag && i < production.getBodySize()) {
        // 获取产生式中当前文法符的FIRST
        Set<TerminalSymbol> first = firstOfSymbol(grammarSymbols.get(i));
        // 判断当前文法符FIRST是否包含ε
        if(first.contains(epsilon)) {
            // 跳转到下一个文法符，FIRST去掉ε
            i++;
            first.remove(epsilon);
        } else {
            flag = false;
        }
        // 把当前文法符的FIRST加入结果中
        FirstList.addAll(first);
    }
    // 如果最终能推导出ε，则FIRST集合中包含ε
    if(flag && i == production.getBodySize()) {
        FirstList.add(epsilon);
    }
    production.setpFirstSet(FirstList);
    return FirstList;
}
```

函数测试：书上案例：

(1) $E \rightarrow TE'$
(2) $E' \rightarrow +TE'$
(3) $E' \rightarrow \varepsilon$
(4) $T \rightarrow FT'$
(5) $T' \rightarrow *FT'$
(6) $T' \rightarrow \varepsilon$
(7) $F \rightarrow (E)$
(8) $F \rightarrow id$

验证

产生式序号	产生式	产生式FIRST函数值
1	E→TE'	FIRST(F)={C,id}
2	E'→+TE'	+
3	E'→ε	ε
4	T→FT'	FIRST(F)={C,id}
5	T'→*FT'	
6	T'→ε	ε
7	F→(E)	(
8	F→id	id

结果：

```
产生式的FIRST集：
{ 产生式序号: 0, 文法符个数: 2, 产生式体文法符: TE'}  FIRST：{id, (, }

{ 产生式序号: 1, 文法符个数: 3, 产生式体文法符: +TE'}  FIRST：{+, }

{ 产生式序号: 2, 文法符个数: 1, 产生式体文法符: ε}  FIRST：{ε, }

{ 产生式序号: 3, 文法符个数: 2, 产生式体文法符: FT'}  FIRST：{id, (, }

{ 产生式序号: 4, 文法符个数: 3, 产生式体文法符: *FT'}  FIRST：{*, }

{ 产生式序号: 5, 文法符个数: 1, 产生式体文法符: ε}  FIRST：{ε, }

{ 产生式序号: 6, 文法符个数: 3, 产生式体文法符: (E)}  FIRST：{(, }

{ 产生式序号: 7, 文法符个数: 1, 产生式体文法符: id}  FIRST：{id, }
```

4）非终结符的 FIRST 函数求解

实现方法：对每个非终结符的产生式，求其 FIRST 函数，再将其合并即可。

实现函数：

```java
public static Set<TerminalSymbol> firstOfSymbol(GrammarSymbol grammarSymbol) {
    Set<TerminalSymbol> list = new HashSet<>();//first集合
    // 当前文法符为终结符或ε，则直接返回其本身
    if (grammarSymbol.getType() == SymbolType.TERMINAL || grammarSymbol.getType() == SymbolType.NULL) {
        list.add((TerminalSymbol) grammarSymbol);//转换类型！！
        return list;
    }
    // 当前文法符为非终结符，遍历每个产生式
    for (Production production: ((NonTerminalSymbol)grammarSymbol).getpProductionTable()) {
        if (production.getpBodySymbolTable().get(0) == grammarSymbol) {
            continue;
        }
        // 对每个产生式求其FIRST集
        for (TerminalSymbol terminalSymbol: getFirstSet(production)) {
            // 将未加入的终结符加入FIRST集合
            if (!list.contains(terminalSymbol)) {
                list.add(terminalSymbol);
            }
        }
    }
    // 设置FIRST非终结符的FIRST集合
    ((NonTerminalSymbol) grammarSymbol).setFirstSet(list);
    return list;
}
```

函数测试：文法同上，输出每个非终结符的 FIRST 函数

验证

```
非终结符的的FIRST集：
E FIRST: {id, (, }
E' FIRST: {+, ε, }
T FIRST: {id, (, }
T' FIRST: {*, ε, }
F FIRST: {id, (, }
```

```
E {
产生式个数: 1,
产生式集:[{ 产生式序号: 0, 文法符个数: 2, 产生式体文法符: TE'}],
FIRST集:{id, (, },
FOLLOW集：{},
FOLLOW集依赖的非终结符:{}
}
E' {
产生式个数: 2,
产生式集:[{ 产生式序号: 1, 文法符个数: 3, 产生式体文法符: +TE'}, { 产生式序号: 2, 文法符个数: 1, 产生式体文法符: ε}],
FIRST集:{+, ε, },
FOLLOW集：{},
FOLLOW集依赖的非终结符:{}
}
T {
产生式个数: 1,
产生式集:[{ 产生式序号: 3, 文法符个数: 2, 产生式体文法符: FT'}],
FIRST集:{id, (, },
FOLLOW集：{},
FOLLOW集依赖的非终结符:{}
}
T' {
产生式个数: 2,
产生式集:[{ 产生式序号: 4, 文法符个数: 3, 产生式体文法符: *FT'}, { 产生式序号: 5, 文法符个数: 1, 产生式体文法符: ε}],
FIRST集:{*, ε, },
FOLLOW集：{},
FOLLOW集依赖的非终结符:{}
}
F {
产生式个数: 2,
产生式集:[{ 产生式序号: 6, 文法符个数: 3, 产生式体文法符: (E)}, { 产生式序号: 7, 文法符个数: 1, 产生式体文法符: id}],
FIRST集:{id, (, },
FOLLOW集：{},
FOLLOW集依赖的非终结符:{}
}
```

5）非终结符的 FOLLOW 函数求解

实现方法：穷举所有情形，找出跟在非终结符后面的终结符。产生式 $X->Y_1Y_2.....Y_{n-1}Y_n$ 蕴含有如下两个 FOLLOW 信息。

①对于末尾符 Yn，如果它为非终结符，那么 $FOLLOW(X)\in FOLLOW(Y_n)$。若 Y_i 为终结符（$0<i<n$），且从 Y_{i+1} 至 Y_n 全为非终结符，且都含虚产生式，那么 $FOLLOW(X)$ 属于 $FOLLOW(Y_i)$。

②除了末尾符 Y_n 之外，对于产生式右部中任一文法符 Y_i，其中 $0<i<n$，如果 Y_i 是一个非终结符，那么 $FIRST(Y_{i+1})-\varepsilon \in FOLLOW(Y_i)$。如果 Y_i 为非终结符（$0<i<n-1$），且从 Y_{i+1} 至 Y_j（$i+1<j<n$）全为非终结符，且都含虚产生式，那么 $FIRST(Y_{j+1})-\varepsilon \in FOLLOW(Y_i)$。

实现函数：

```java
public static void followOfSymbol(NonTerminalSymbol nonTerminalSymbol) {
    // 新建ε
    TerminalSymbol epsilon = new TerminalSymbol( name: "ε", SymbolType.NULL);
    // 遍历nonTerminalSymbol产生式
    for (Production production: nonTerminalSymbol.getpProductionTable()) {
        // 产生式文法符个数
        int size = production.getBodySize();
        // 获取最后一个文法符
        GrammarSymbol last = production.getpBodySymbolTable().get(size - 1);
        // nonTerminalSymbol->ε，表示为空，跳过该产生式
        if (last.getName().equals("ε")) {
            continue;
        }
        // 最后一个文法符为非终结符，其FOLLOW集合依赖于nonTerminalSymbol的FOLLOW集合
        if (last.getType() == SymbolType.NONTERMINAL) {
            ((NonTerminalSymbol) last).addDependentSetInFollow(nonTerminalSymbol);
        }
        // ε是否持续（从最后一个开始往前找）
        Boolean flag = true;
        // 从倒数第二个开始
        int i = size-2, j = size-1;
        while (i >= 0) {
            GrammarSymbol lasti = production.getpBodySymbolTable().get(i);
            // 当前文法符为非终结符
            if (lasti.getType() == SymbolType.NONTERMINAL) {
                // 遍历其后FIRST连续包含ε的非终结符
                for (int k = i+1; k <= j; k ++) {
                    // 若第k个文法符为终结符，将其自身加入FOLLOW集合即可
                    if (production.getpBodySymbolTable().get(k).getType() == SymbolType.TERMINAL) {
                        ((NonTerminalSymbol)lasti).addFollow((TerminalSymbol) production.getpBodySymbolTable().get(k));
                        flag = false;
                    }
                    else {
                        NonTerminalSymbol Yk = (NonTerminalSymbol)production.getpBodySymbolTable().get(k);
                        Set<TerminalSymbol> firstYk = Yk.removeREpsilon();//first(Yk)-ε
                        // 若当前终结符的后续非终结符的FIRST集合不包含ε，说明无法持续到最后，flag置0
                        if (!Yk.containsEpsilon()) {
                            flag = false;
                        }
                        // 将其FIRST集合-{ε}加入非终结符的FOLLOW集合
                        ((NonTerminalSymbol)lasti).addFollowSet(firstYk);
                    }
                }
                // 如果flag仍为true，表示当前文法符仍能到达最后，则其FOLLOW集合依赖于symbol
                if (flag) {
                    ((NonTerminalSymbol)lasti).addDependentSetInFollow(nonTerminalSymbol);
                }

                // 如果当前文法符的FIRST集合不包含ε，说明FOLLOW集合无法到达后续，将j修改为当前i
                if (!((NonTerminalSymbol)lasti).getpFirstSet().contains(epsilon)) {
                    j = i;
                }

            }
            else {
                j = i;
                if (flag) {
                    flag = false;
                }
            }
            i --;
        }
    }
}
```

根据每个非终结符求出 FOLLOW 相关信息后，将依赖的非终结符的 FOLLOW 信息加入其中：

```java
//FOLLOW集依赖的非终结符是否存在
2 个用法
public void addDependentSetInFollow(NonTerminalSymbol non) {
    if(non.getName() != getName()) {
        pDependentSetInFollow.add(non);
    }
}
```

函数测试：文法同上，先求其 FIRST 函数，再求 FOLLOW 函数

```
非终结符的的FOLLOW集：
E {
产生式个数: 1,
产生式集:[{ 产生式序号: 0, 文法符个数: 2, 产生式体文法符: TE'}],
FIRST集:{id, (, },
FOLLOW集：{), $, },
FOLLOW集依赖的非终结符:{}
}
E' {
产生式个数: 2,
产生式集:[{ 产生式序号: 1, 文法符个数: 3, 产生式体文法符: +TE'}, { 产生式序号: 2, 文法符个数: 1, 产生式体文法符: ε}],
FIRST集:{+, ε, },
FOLLOW集：{), $, },
FOLLOW集依赖的非终结符:{E }
}
T {
产生式个数: 1,
产生式集:[{ 产生式序号: 3, 文法符个数: 2, 产生式体文法符: FT'}],
FIRST集:{id, (, },
FOLLOW集：{), +, $, },
FOLLOW集依赖的非终结符:{E' E }
}
T' {
产生式个数: 2,
产生式集:[{ 产生式序号: 4, 文法符个数: 3, 产生式体文法符: *FT'}, { 产生式序号: 5, 文法符个数: 1, 产生式体文法符: ε}],
FIRST集:{*, ε, },
FOLLOW集：{), +, $, },
FOLLOW集依赖的非终结符:{T }
}
F {
产生式个数: 2,
产生式集:[{ 产生式序号: 6, 文法符个数: 3, 产生式体文法符: (E)}, { 产生式序号: 7, 文法符个数: 1, 产生式体文法符: id}],
FIRST集:{id, (, },
FOLLOW集：{), +, *, $, },
FOLLOW集依赖的非终结符:{T T' }
}
```

6）LL（1）文法的判断

实现方法：对于一个文法，其中的任一非终结符 X，设其实产生式有 $X->\alpha_1$，$X->\alpha_2$，……，$X->\alpha_n$，若满足 $FIRST(X->\alpha_i) \cap FIRST(X->\alpha_j) = \Phi$，其中 $i \neq j$ 且 $0<i,j\leqslant n$。如果 X 还有虚产生式 $X->\varepsilon$，若进一步满足 $FIRST(X->\alpha_i) \cap FOLLOW(X)=\Phi$，其中 $0<i\leqslant n$。具有这种特性的文法被称之为 LL(1)文法。

实现函数：

```java
public static Boolean isLL1 (NonTerminalSymbol nonTerminalSymbol) {
    Map<TerminalSymbol, Integer> map = new HashMap<>();
    // 判断是否有X->ε,若存在需将FOLLOW保存进map
    if (nonTerminalSymbol.containsEpsilon()) {
        for (TerminalSymbol s: nonTerminalSymbol.getpFollowSet()) {
            map.put(s, v: -1);
        }
    }
    // 遍历产生式
    for (Production production: nonTerminalSymbol.getpProductionTable()) {
        // 遍历产生式的FIRST集合，若其未出现过，说明无交集，否则有交集（不为LL（1）文法）
        for (TerminalSymbol s: production.getpFirstSet()) {
            if (map.get(s) == null) {
                map.put(s,production.getProductionId());
            }
            else {
                return false;
            }
        }
    }
    return true;
}
```

函数测试：文法同上，生成 FIRST 和 FOLLOW 函数后，对每个非终结符进行依次判断，若每个非终结符都满足条件，则其为 LL（1）文法。

(1) $E \rightarrow TE'$
(2) $E' \rightarrow +TE'$
(3) $E' \rightarrow \varepsilon$
(4) $T \rightarrow FT'$
(5) $T' \rightarrow *FT'$
(6) $T' \rightarrow \varepsilon$
(7) $F \rightarrow (E)$
(8) $F \rightarrow id$

验证

```
LL1分析表信息
E:true
nonEE:true
T:true
nonTT:true
F:true
```

7）LL（1）语法分析表的填写

实现方法：对于 LL(1)文法，在自顶向下最左推导当中，设当前要推导的非终结符为 X，当前词为 w，如果 w∈FIRST(X->α_i)，就选择 X->α_i 进行推导。如果存在有 X-> ε 且 w∈FOLLOW(X)，就选择 X-> ε 进行推导。

实现函数：

```
public static ArrayList<Cell> parseTable(ArrayList<NonTerminalSymbol> pNonTerminalSymbolTable) {
    ArrayList<Cell> pParseTableOfLL = new ArrayList<>();
    // 遍历每个非终结符
    for (NonTerminalSymbol nonTerminalSymbol: pNonTerminalSymbolTable) {
        // 遍历每个产生式
        for (Production production: nonTerminalSymbol.getpProductionTable()) {
            // 若该产生式为X->ε，则选择FOLLOW集合中的终结符填入该产生式
            if (production.isEpsilon()) {
                for (TerminalSymbol t: nonTerminalSymbol.getpFollowSet()) {
                    Cell cell = new Cell(nonTerminalSymbol, t, production);
                    pParseTableOfLL.add(cell);
                }
            }
            // 否则选择FIRST集合中的终结符填入该产生式
            else {
                for (TerminalSymbol t: production.getpFirstSet()) {
                    Cell cell = new Cell(nonTerminalSymbol, t, production);
                    pParseTableOfLL.add(cell);
                }
            }
        }
    }
    return pParseTableOfLL;
}
```

函数测试：文法同上，求出 FIRST 和 FOLLOW 函数判断其是否为 LL（1）文法，再构造语法分析表

(1) $E \rightarrow TE'$
(2) $E' \rightarrow +TE'$
(3) $E' \rightarrow \varepsilon$
(4) $T \rightarrow FT'$
(5) $T' \rightarrow *FT'$
(6) $T' \rightarrow \varepsilon$
(7) $F \rightarrow (E)$
(8) $F \rightarrow id$

表 3.14 文法 5 的 LL(1)语法分析表

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$F \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

验证结果如下：

```
[Cell{非终结符=E, 终结符=id产生式{ 产生式序号: 0, 文法符个数: 2, 产生式体文法符: TE'}}
, Cell{非终结符=E, 终结符=(产生式{ 产生式序号: 0, 文法符个数: 2, 产生式体文法符: TE'}}
, Cell{非终结符=E', 终结符=+产生式{ 产生式序号: 1, 文法符个数: 3, 产生式体文法符: +TE'}}
, Cell{非终结符=E', 终结符=)产生式{ 产生式序号: 2, 文法符个数: 1, 产生式体文法符: ε}}
, Cell{非终结符=E', 终结符=$产生式{ 产生式序号: 2, 文法符个数: 1, 产生式体文法符: ε}}
, Cell{非终结符=T, 终结符=id产生式{ 产生式序号: 3, 文法符个数: 2, 产生式体文法符: FT'}}
, Cell{非终结符=T, 终结符=(产生式{ 产生式序号: 3, 文法符个数: 2, 产生式体文法符: FT'}}
, Cell{非终结符=T', 终结符=*产生式{ 产生式序号: 4, 文法符个数: 3, 产生式体文法符: *FT'}}
, Cell{非终结符=T', 终结符=)产生式{ 产生式序号: 5, 文法符个数: 1, 产生式体文法符: ε}}
, Cell{非终结符=T', 终结符=+产生式{ 产生式序号: 5, 文法符个数: 1, 产生式体文法符: ε}}
, Cell{非终结符=T', 终结符=$产生式{ 产生式序号: 5, 文法符个数: 1, 产生式体文法符: ε}}
, Cell{非终结符=F, 终结符=(产生式{ 产生式序号: 6, 文法符个数: 3, 产生式体文法符: (E)}}
, Cell{非终结符=F, 终结符=id产生式{ 产生式序号: 7, 文法符个数: 1, 产生式体文法符: id}}
]
```

三、针对 LR 语法分析，实现如下函数。

1）void getClosure(ItemSet itemSet);

函数作用：基于 LR（0）核心项的闭包求解。

实现方法：找到所有待约项目，根据待约项目推导出非核心项。

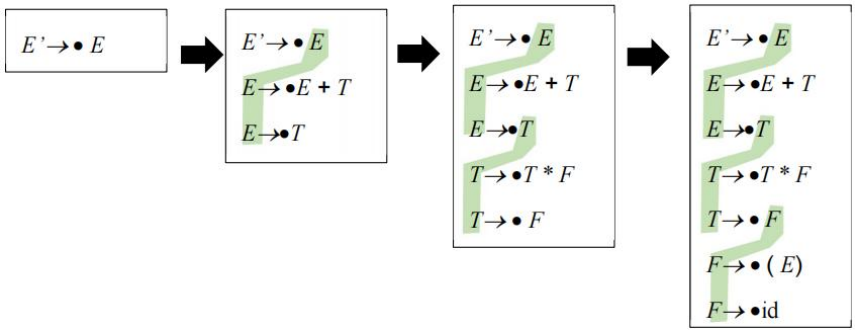
实现函数：

```java
public static void getClosure(ItemSet itemSet) {
    // 栈，用于保存未求其后续文法符的项目
    Stack<LR0Item> item = new Stack<>();
    // 将所有核心项推入栈中
    for (LR0Item lr: itemSet.getpItemTable()) {
        item.push(lr);
    }

    while (!item.isEmpty()) {
        LR0Item lr = item.pop();
        int pos = lr.getDotPosition();
        //判断圆点位置
        // pos在产生式最后面，说明为规约项目
        if (pos == lr.getProduction().getBodySize()) {
            continue;
        }
        else {
            // 找到后续文法符
            GrammarSymbol grammarSymbol = lr.getProduction().getpBodySymbolTable().get(pos);
            // 该文法符为非终结符，说明为待约项目 需要遍历非终结符（类型转换）
            if (grammarSymbol.getType() == SymbolType.NONTERMINAL) {
                // 遍历该非终结符的每个产生式
                for (Production production: ((NonTerminalSymbol)grammarSymbol).getpProductionTable()) {
                    if (!itemSet.containsItem(production, dot: 0)) {
                        // 新建一个非核心项，原点位置为0
                        LR0Item newItem = new LR0Item((NonTerminalSymbol)grammarSymbol, production, dot: 0,
                                ItemCategoy.NONCORE);
                        // 添加进闭包
                        itemSet.addItem(newItem);
                        // 推入栈中
                        item.push(newItem);
                    }
                }
            }
        }
    }
}
```

函数测试：求 E’ 的闭包 I0

(0) $E'\rightarrow \bullet E$
(1) $E\rightarrow E+T$
(2) $E\rightarrow T$
(3) $T\rightarrow T*F$
(4) $T\rightarrow F$
(5) $F\rightarrow (E)$
(6) $F\rightarrow$ id



验证结果如下：

```
ItemSet{
状态序号:0,
LR0项目表=[
item {非终结符: E', 产生式: E, 圆点位置: 0, 类别: CORE},
item {非终结符: E, 产生式: E+T, 圆点位置: 0, 类别: NONCORE},
item {非终结符: E, 产生式: T, 圆点位置: 0, 类别: NONCORE},
item {非终结符: T, 产生式: T*F, 圆点位置: 0, 类别: NONCORE},
item {非终结符: T, 产生式: F, 圆点位置: 0, 类别: NONCORE},
item {非终结符: F, 产生式: (E), 圆点位置: 0, 类别: NONCORE},
item {非终结符: F, 产生式: id, 圆点位置: 0, 类别: NONCORE}]
}
```

2）void exhaustTransition(ItemSet itemSet)

函数作用：穷举一个 LR（0）项集的变迁，其中中包括驱动符的穷举，下一项集的创建，下一项集中核心项的确定，下一项集是否为新项集的判断。

实现方法：首先找到所有驱动符，对每个驱动符创建一个项集，求该项集的核心项及其闭包，再判断该项集是否为新项集。最后创建一条变迁边连接两个项集。

实现函数：

1. 新维护了一个 ArrayList 来保存所有项集和之后判断新项集

```
private static ArrayList<ItemSet> allItemSet = new ArrayList<>();
4 个用法
public static void addItemSet(ItemSet set) { allItemSet.add(set); }
```

2. 新增了一个构造函数构建核心项

```
public LR0Item(LR0Item item) {
    this.nonTerminalSymbol = item.getNonTerminalSymbol();
    this.production = item.getProduction();
    this.dotPosition = item.getDotPosition()+1;
    this.type = ItemCategoy.CORE;

}
```

3. 判断两项集是否相同：判断两者 LR0 项目是否相等

```java
public Boolean isSameItemSet(ItemSet itemSet) {
    if(pItemTable.size() !=itemSet.getpItemTable().size()) return false;

    //项目数量相同的情况下 判断元素是否相同
    for(int i = 0; i < pItemTable.size();i++) {
        if(!pItemTable.get(i).equals(itemSet.getpItemTable().get(i))) {
            return false;
        }
    }
    return true;
}
```

```java
public static ArrayList<TransitionEdge> exhaustTransition(ItemSet itemSet) {
    // 保存新创建的变迁边
    ArrayList<TransitionEdge> edges = new ArrayList<>();
    Map<GrammarSymbol, Vector<LR0Item>> map = new HashMap<>();
    // 穷举所有驱动符，并将其保存在map中
    for (LR0Item item: itemSet.getpItemTable()) {
        int pos = item.getDotPosition();
        // 当前项目为规约项目
        if (pos == item.getProduction().getBodySize()) {
            continue;
        }
        // 获得后续文法符
        GrammarSymbol grammarSymbol = item.getProduction().getpBodySymbolTable().get(pos);
        // 该文法符未出现，则新创建一个vector对象
        if (map.get(grammarSymbol) == null) {
            Vector<LR0Item> v = new Vector<>();
            v.add(item);
            map.put(grammarSymbol, v);
        }
        // 之前出现过，则在后面添加item
        else {
            map.get(grammarSymbol).add(item);
        }
    }
    // 遍历所有驱动符
    for (GrammarSymbol grammarSymbol: map.keySet()) {
        // 下一项集的建立,id为-1（防止重复导致的状态序号不连续）
        ItemSet newSet = new ItemSet( stateId: -1);
        for (LR0Item item: map.get(grammarSymbol)) {
            // 新建一个核心项，其type为CORE，pos加一
            LR0Item lr = new LR0Item(item);
            newSet.addItem(lr);
        }
        // 求该项集的闭包
        getClosure(newSet);
        // 判断下一项集是否为新项集
        Boolean isExist = false;
        for (ItemSet set: allItemSet) {
            if (newSet.isSameItemSet(set)) {
                isExist = true;
                newSet = set;
                break;
            }
        }
        // 该项集为新项集
        if (!isExist) {
            // 设置该项集为新的项集id，保证连续
            newSet.setStateId();
            allItemSet.add(newSet);
            //得到的新状态
            newItemSet.add(newSet);
        }
        // 创建一条变迁边
        TransitionEdge edge = new TransitionEdge(grammarSymbol, itemSet, newSet);
        edges.add(edge);
    }
    return edges;
}
```

函数测试：以项目集 I0 的变迁为例，先求出其核心项，再求其闭包。

(0) $E' \rightarrow \bullet E$

(1) $E \rightarrow E+T$

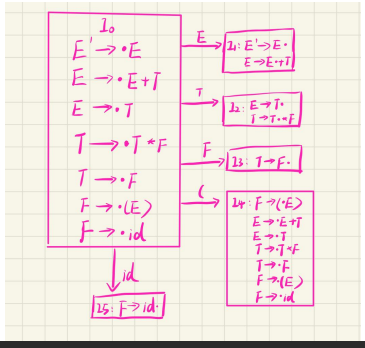
(2) $E \rightarrow T$

(3) $T \rightarrow T*F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow \mathrm{id}$



验证

```
ItemSet{
状态序号:0,
LR0项目表=[
item {非终结符: E', 产生式: E, 圆点位置: 0, 类别: CORE},
item {非终结符: E, 产生式: E+T, 圆点位置: 0, 类别: NONCORE},
item {非终结符: E, 产生式: T, 圆点位置: 0, 类别: NONCORE},
item {非终结符: T, 产生式: T*F, 圆点位置: 0, 类别: NONCORE},
item {非终结符: T, 产生式: F, 圆点位置: 0, 类别: NONCORE},
item {非终结符: F, 产生式: (E), 圆点位置: 0, 类别: NONCORE},
item {非终结符: F, 产生式: id, 圆点位置: 0, 类别: NONCORE}]
},
ItemSet{
状态序号:1,
LR0项目表=[
item {非终结符: T, 产生式: F, 圆点位置: 1, 类别: CORE}]
},
ItemSet{
状态序号:2,
LR0项目表=[
item {非终结符: F, 产生式: id, 圆点位置: 1, 类别: CORE}]
},
ItemSet{
状态序号:3,
LR0项目表=[
item {非终结符: E, 产生式: T, 圆点位置: 1, 类别: CORE},
item {非终结符: T, 产生式: T*F, 圆点位置: 1, 类别: CORE}]
},
ItemSet{
状态序号:4,
LR0项目表=[
item {非终结符: E', 产生式: E, 圆点位置: 1, 类别: CORE},
item {非终结符: E, 产生式: E+T, 圆点位置: 1, 类别: CORE}]
},
ItemSet{
状态序号:5,
LR0项目表=[
item {非终结符: F, 产生式: (E), 圆点位置: 1, 类别: CORE},
item {非终结符: E, 产生式: E+T, 圆点位置: 0, 类别: NONCORE},
item {非终结符: E, 产生式: T, 圆点位置: 0, 类别: NONCORE},
item {非终结符: T, 产生式: T*F, 圆点位置: 0, 类别: NONCORE},
item {非终结符: T, 产生式: F, 圆点位置: 0, 类别: NONCORE},
item {非终结符: F, 产生式: (E), 圆点位置: 0, 类别: NONCORE},
item {非终结符: F, 产生式: id, 圆点位置: 0, 类别: NONCORE}]
}]
```

3）文法的 LR（0）型 DFA 求解

实现方法：在符号栈中，从状态 0 开始，穷举所有变迁。对于每一变迁的驱动文法符，求下一状态（即核心项闭包）。如果下一状态是一个新状态，则使用相同策略穷举。如此迭代下去，直到把所有的状态变迁都穷举出来。

实现函数：

```java
public static DFA getDFA(ItemSet start) {
    // 新建一个DFA
    DFA dfa = new DFA(start);
    // 保存未穷举状态的项集（队列先进先出的特点）
    Deque<ItemSet> queue = new ArrayDeque<>();
    queue.push(start);
    while (!queue.isEmpty()) {
        ItemSet cur = queue.pop();
        // 对当前项集进行穷举，得到变迁边
        ArrayList<TransitionEdge> edges = exhaustTransition(cur);
        //添加新的状态 继续穷举新的状态集
        queue.addAll(newItemSet);
        newItemSet.clear();
        // 添加所有变迁表到dfa中
        dfa.addEdges(edges);
    }
    return dfa;
}
```

函数测试：文法如上，对每个项集求其变迁即可

(0) $E' \rightarrow \bullet E$

(1) $E \rightarrow E+T$

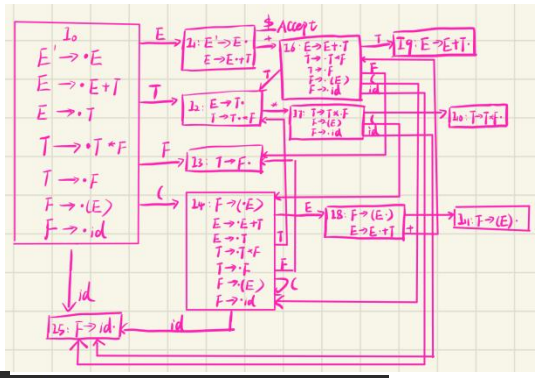
(2) $E \rightarrow T$

(3) $T \rightarrow T*F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow \mathrm{id}$



验证

```
所有状态：
[
ItemSet{
状态序号:0,
LR0项目表=[
item {非终结符: E', 产生式: E, 圆点位置: 0, 类别: CORE},
item {非终结符: E, 产生式: E+T, 圆点位置: 0, 类别: NONCORE},
item {非终结符: E, 产生式: T, 圆点位置: 0, 类别: NONCORE},
item {非终结符: T, 产生式: T*F, 圆点位置: 0, 类别: NONCORE},
item {非终结符: T, 产生式: F, 圆点位置: 0, 类别: NONCORE},
item {非终结符: F, 产生式: (E), 圆点位置: 0, 类别: NONCORE},
item {非终结符: F, 产生式: id, 圆点位置: 0, 类别: NONCORE}]
},
ItemSet{
状态序号:1,
LR0项目表=[
item {非终结符: E', 产生式: E, 圆点位置: 1, 类别: CORE},
item {非终结符: E, 产生式: E+T, 圆点位置: 1, 类别: CORE}]
},
ItemSet{
状态序号:2,
LR0项目表=[
item {非终结符: F, 产生式: (E), 圆点位置: 1, 类别: CORE},
item {非终结符: E, 产生式: E+T, 圆点位置: 0, 类别: NONCORE},
item {非终结符: E, 产生式: T, 圆点位置: 0, 类别: NONCORE},
item {非终结符: T, 产生式: T*F, 圆点位置: 0, 类别: NONCORE},
item {非终结符: T, 产生式: F, 圆点位置: 0, 类别: NONCORE},
item {非终结符: F, 产生式: (E), 圆点位置: 0, 类别: NONCORE},
item {非终结符: F, 产生式: id, 圆点位置: 0, 类别: NONCORE}]
},
ItemSet{
状态序号:3,
LR0项目表=[
item {非终结符: T, 产生式: F, 圆点位置: 1, 类别: CORE}]
},
```

```
ItemSet{
状态序号:4,
LR0项目表=[
item {非终结符: F, 产生式: id, 圆点位置: 1, 类别: CORE}]
},
ItemSet{
状态序号:5,
LR0项目表=[
item {非终结符: E, 产生式: T, 圆点位置: 1, 类别: CORE},
item {非终结符: T, 产生式: T*F, 圆点位置: 1, 类别: CORE}]
},
ItemSet{
状态序号:6,
LR0项目表=[
item {非终结符: E, 产生式: E+T, 圆点位置: 2, 类别: CORE},
item {非终结符: T, 产生式: T*F, 圆点位置: 0, 类别: NONCORE},
item {非终结符: T, 产生式: F, 圆点位置: 0, 类别: NONCORE},
item {非终结符: F, 产生式: (E), 圆点位置: 0, 类别: NONCORE},
item {非终结符: F, 产生式: id, 圆点位置: 0, 类别: NONCORE}]
},
ItemSet{
状态序号:7,
LR0项目表=[
item {非终结符: F, 产生式: (E), 圆点位置: 2, 类别: CORE},
item {非终结符: E, 产生式: E+T, 圆点位置: 1, 类别: CORE}]
},
ItemSet{
状态序号:8,
LR0项目表=[
item {非终结符: T, 产生式: T*F, 圆点位置: 2, 类别: CORE},
item {非终结符: F, 产生式: (E), 圆点位置: 0, 类别: NONCORE},
item {非终结符: F, 产生式: id, 圆点位置: 0, 类别: NONCORE}]
},
ItemSet{
状态序号:9,
LR0项目表=[
item {非终结符: E, 产生式: E+T, 圆点位置: 3, 类别: CORE},
item {非终结符: T, 产生式: T*F, 圆点位置: 1, 类别: CORE}]
},
```

```
ItemSet{
状态序号:10,
LR0项目表=[
item {非终结符: F, 产生式: (E), 圆点位置: 3, 类别: CORE}]
},
ItemSet{
状态序号:11,
LR0项目表=[
item {非终结符: T, 产生式: T*F, 圆点位置: 3, 类别: CORE}]
}]
-----------------------------------------------------------
DFA {
ItemSet-Id= 0,
变迁边表=[
edge{ 驱动字符: E, 起始状态: 0, 到达状态: 1},
edge{ 驱动字符: (, 起始状态: 0, 到达状态: 2},
edge{ 驱动字符: F, 起始状态: 0, 到达状态: 3},
edge{ 驱动字符: id, 起始状态: 0, 到达状态: 4},
edge{ 驱动字符: T, 起始状态: 0, 到达状态: 5},
edge{ 驱动字符: +, 起始状态: 1, 到达状态: 6},
edge{ 驱动字符: E, 起始状态: 2, 到达状态: 7},
edge{ 驱动字符: (, 起始状态: 2, 到达状态: 2},
edge{ 驱动字符: F, 起始状态: 2, 到达状态: 3},
edge{ 驱动字符: id, 起始状态: 2, 到达状态: 4},
edge{ 驱动字符: T, 起始状态: 2, 到达状态: 5},
edge{ 驱动字符: *, 起始状态: 5, 到达状态: 8},
edge{ 驱动字符: (, 起始状态: 6, 到达状态: 2},
edge{ 驱动字符: F, 起始状态: 6, 到达状态: 3},
edge{ 驱动字符: id, 起始状态: 6, 到达状态: 4},
edge{ 驱动字符: T, 起始状态: 6, 到达状态: 9},
edge{ 驱动字符: ), 起始状态: 7, 到达状态: 10},
edge{ 驱动字符: +, 起始状态: 7, 到达状态: 6},
edge{ 驱动字符: (, 起始状态: 8, 到达状态: 2},
edge{ 驱动字符: F, 起始状态: 8, 到达状态: 11},
edge{ 驱动字符: id, 起始状态: 8, 到达状态: 4},
edge{ 驱动字符: *, 起始状态: 9, 到达状态: 8}]}
```

4）SLR（1）文法的判断

实现思路：对于每个项集，找到它的移入终结符集合和规约项目集合，

①规约项目 FOLLOW 集合与移入终结符集合有冲突 ==> 移入-规约冲突

②规约项目 FOLLOW 集合之间有冲突 ==> 规约-规约冲突

上述两种情况都不发生，则为 SLR（1）文法。

实现函数：

```
public static Boolean isSLR1() {
    // 遍历每个项集
    for (ItemSet set: allItemSet) {
        // 设置规约项目
        ArrayList<LR0Item> reduce = new ArrayList<>();
        //移入终结符集合
        ArrayList<TerminalSymbol> shiftList = new ArrayList<>();
        // 遍历项集里面的每个项目
        for (LR0Item item: set.getpItemTable()) {
            //产生式--移入项
            Production production = item.getProduction();
            int pos = item.getDotPosition();
            // 圆点在最后---该项目为规约项目，添加该项目到规约项目集合
            if (pos == production.getBodySize()) {
                reduce.add(item);
            }
            // 该项目为移入项目，添加终结符到移入终结符集合
            else if (production.getpBodySymbolTable().get(pos).getType() == SymbolType.TERMINAL) {
                shiftList.add((TerminalSymbol) production.getpBodySymbolTable().get(pos));
            }
        }
        // 判断规约-移入冲突
        if (shiftList.size() > 0 && reduce.size() > 0) {
            // 遍历每个规约项目
            for (LR0Item item: reduce) {
                // 求其产生式左部的非终结符的FOLLOW集合
                Set<TerminalSymbol> follow = item.getNonTerminalSymbol().getpFollowSet();
                // FOLLOW集合不能与移入终结符集合相交，否则不为SLR(1)文法
                for (TerminalSymbol terminalSymbol: follow) {
                    if (shiftList.contains(terminalSymbol)) {
                        return false;
                    }
                }
            }
        }
        // 判断规约-规约冲突
        if (reduce.size() > 1) {
            Set<TerminalSymbol> list = new HashSet<>();
            for (LR0Item item: reduce) {
                // 求其产生式左部的非终结符的FOLLOW集合
                Set<TerminalSymbol> follow = item.getNonTerminalSymbol().getpFollowSet();
                for (TerminalSymbol terminalSymbol: follow) {
                    // 若该FOLLOW集合中的元素在其他规约项目的FOLLOW集中出现过，则存在规约-规约冲突
                    if (list.contains(terminalSymbol)) {
                        return false;
                    }
                    // 若未出现，则加入
                    else {
                        list.add(terminalSymbol);
                    }
                }
            }
        }
```

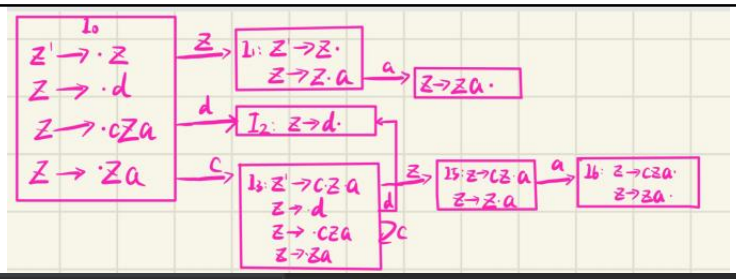
函数测试：

(0) Z'→Z

(1) Z→d

(2) Z→cZa

(3) Z→Za



验证

```
所有状态:
[
ItemSet{
状态序号:0,
LR0项目表=[
item {非终结符: Z', 产生式: Z, 圆点位置: 0, 类别: CORE},
item {非终结符: Z, 产生式: d, 圆点位置: 0, 类别: NONCORE},
item {非终结符: Z, 产生式: cZa, 圆点位置: 0, 类别: NONCORE},
item {非终结符: Z, 产生式: Za, 圆点位置: 0, 类别: NONCORE}]
},
ItemSet{
状态序号:1,
LR0项目表=[
item {非终结符: Z, 产生式: cZa, 圆点位置: 1, 类别: CORE},
item {非终结符: Z, 产生式: d, 圆点位置: 0, 类别: NONCORE},
item {非终结符: Z, 产生式: cZa, 圆点位置: 0, 类别: NONCORE},
item {非终结符: Z, 产生式: Za, 圆点位置: 0, 类别: NONCORE}]
},
ItemSet{
状态序号:2,
LR0项目表=[
item {非终结符: Z', 产生式: Z, 圆点位置: 1, 类别: CORE},
item {非终结符: Z, 产生式: Za, 圆点位置: 1, 类别: CORE}]
},
ItemSet{
状态序号:3,
LR0项目表=[
item {非终结符: Z, 产生式: d, 圆点位置: 1, 类别: CORE}]
},
ItemSet{
状态序号:4,
LR0项目表=[
item {非终结符: Z, 产生式: cZa, 圆点位置: 2, 类别: CORE},
item {非终结符: Z, 产生式: Za, 圆点位置: 1, 类别: CORE}]
},
ItemSet{
状态序号:5,
LR0项目表=[
item {非终结符: Z, 产生式: Za, 圆点位置: 2, 类别: CORE}]
},
ItemSet{
状态序号:6,
LR0项目表=[
item {非终结符: Z, 产生式: cZa, 圆点位置: 3, 类别: CORE},
item {非终结符: Z, 产生式: Za, 圆点位置: 2, 类别: CORE}]
}]
```

```
DFA {
ItemSet-Id= 0,
变迁边表=[
edge{ 驱动字符: c, 起始状态: 0, 到达状态: 1},
edge{ 驱动字符: Z, 起始状态: 0, 到达状态: 2},
edge{ 驱动字符: d, 起始状态: 0, 到达状态: 3},
edge{ 驱动字符: c, 起始状态: 1, 到达状态: 1},
edge{ 驱动字符: Z, 起始状态: 1, 到达状态: 4},
edge{ 驱动字符: d, 起始状态: 1, 到达状态: 3},
edge{ 驱动字符: a, 起始状态: 2, 到达状态: 5},
edge{ 驱动字符: a, 起始状态: 4, 到达状态: 6}]}
不是SLR(1)文法
```

5）LR 语法分析表的填写

实现思路：是从 0 状态开始，逐行填写。对于 DFA 中的每个状态，它的每条出边都要在语法分析表中对应填写一格。

①如果出边的驱动符为终结符，就填到 ACTION 部分，在目标状态序号前加 s，表示移入(shift)。

②如果出边为非终结符，就填到 GOTO 部分，直接填上目标状态序号即可。

③如果包含规约项目，对该产生式头部非终结符的 FOLLOW 集合中的每个终结符，都要在其对应格中填上规约项的产生式序号，并在产生式序号前加 r，表示规约(reduce)。

④如果包含接受项目，就填到 ACTION 部分，在非终结符“#”下填 a，表示接受。

实现函数：

1. 语法分析表

```
public static void getCell(DFA dfa) {
    // 遍历所有状态集合
    for (ItemSet set: allItemSet) {
        // 遍历每个项集下的所有项目
        for (LR0Item item: set.getpItemTable()) {
            //产生式
            Production production = item.getProduction();
            // 原点所处位置
            int pos = item.getDotPosition();
            // 该项目为规约项目，找到FOLLOW集合填r
            if (pos == production.getBodySize()) {
                // 其中该项目为接受项目，在“$”上填a
                if (item.getProduction().getProductionId() == 0) {
                    // 创建对应ACTION
                    ActionCell cell = new ActionCell(set.getStateId(), terminalSymbolName: "$",
                            ActionCategory.a, id: 0);
                    pActionCellTable.add(cell);
                }

                // 求其产生式左部的非终结符的FOLLOW集合
                Set<TerminalSymbol> follow = item.getNonTerminalSymbol().getpFollowSet();
                // 遍历FOLLOW集合中的每个终结符
                for (TerminalSymbol symbol: follow) {
                    // 创建对应ACTION
                    ActionCell cell = new ActionCell(set.getStateId(), symbol.getName(),
                            ActionCategory.r, item.getProduction().getProductionId());
                    pActionCellTable.add(cell);
                }
                continue;
            }

            // 移入-----原点后的文法符
            GrammarSymbol symbol = production.getpBodySymbolTable().get(pos);
            // 找到该文法符驱动的下一个状态
            ItemSet nextSet = dfa.findNextSet(set, symbol);
            // 该项目为移入项目，找到终结符填s
            if (symbol.getType() == SymbolType.TERMINAL) {
                // 创建对应ACTION
                ActionCell cell = new ActionCell(set.getStateId(), symbol.getName(),
                        ActionCategory.s, nextSet.getStateId());
                pActionCellTable.add(cell);
            }
            // 该项目为待约项目，找到非终结符在GOTO填状态序号
            else {
                // 创建对应GOTO
                GotoCell cell = new GotoCell(set.getStateId(), symbol.getName(),
                        nextSet.getStateId());
                pGotoCellTable.add(cell);
            }
        }
    }
```

2. 产生式概述表

```
1 个用法
public static ArrayList<ProductionInfo> createInfo(NonTerminalSymbol symbol) {
    //产生式表
    ArrayList<ProductionInfo> productionInfoTable = new ArrayList<>();
    //非终结符的所有产生式
    for (Production production: symbol.getpProductionTable()) {

        ProductionInfo info = new ProductionInfo(symbol.getName(), production.getBodySize());
        productionInfoTable.add(info);
    }
    return productionInfoTable;
}
```

函数测试：文法如上，求出 FIRST 和 FOLLOW 集，求出项集及变迁边，创建 DFA，基于 DFA 填写下列表格。

(0) $E' \rightarrow \bullet E$

(1) $E \rightarrow E+T$

(2) $E \rightarrow T$

(3) $T \rightarrow T*F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow \text{id}$

状态	ACTION						GOTO		
	id	+	*	(	)	#	E	T	F
0	s2			s5			3	1	4
1		r2	s6		r2	r2			
2		r6	r6		r6	r6			
3		s7				acc			
4		r4	r4		r4	r4			
5	s2			s5			8	1	4
6	s2			s5					9
7	s2			s5				10	4
8		s7			s11				
9		r3	r3		r3	r3			
10		r1	s6		r1	r1			
11		r5	r5		r5	r5			

验证

```
E' {
产生式个数: 1,
产生式集:[{ 产生式序号: 0, 文法符个数: 1, 产生式体文法符: E}],
FIRST集:{id, (, },
FOLLOW集：{},
FOLLOW集依赖的非终结符:{}
}
E {
产生式个数: 2,
产生式集:[{ 产生式序号: 1, 文法符个数: 3, 产生式体文法符: E+T}, { 产生式序号: 2, 文法符个数: 1, 产生式体文法符: T}],
FIRST集:{id, (, },
FOLLOW集：{$, +, ), },
FOLLOW集依赖的非终结符:{E' }
}
T {
产生式个数: 2,
产生式集:[{ 产生式序号: 3, 文法符个数: 3, 产生式体文法符: T*F}, { 产生式序号: 4, 文法符个数: 1, 产生式体文法符: F}],
FIRST集:{id, (, },
FOLLOW集：{$, +, ), *, },
FOLLOW集依赖的非终结符:{E }
}
F {
产生式个数: 2,
产生式集:[{ 产生式序号: 5, 文法符个数: 3, 产生式体文法符: (E)}, { 产生式序号: 6, 文法符个数: 1, 产生式体文法符: id}],
FIRST集:{id, (, },
FOLLOW集：{$, +, ), *, },
FOLLOW集依赖的非终结符:{T }
}
```

```
产生式概述表 {产生式序号: 1, 产生式头部非终结符: 'E', 产生式体文法符个数: 3},
产生式概述表 {产生式序号: 2, 产生式头部非终结符: 'E', 产生式体文法符个数: 1},
产生式概述表 {产生式序号: 3, 产生式头部非终结符: 'T', 产生式体文法符个数: 3},
产生式概述表 {产生式序号: 4, 产生式头部非终结符: 'T', 产生式体文法符个数: 1},
产生式概述表 {产生式序号: 5, 产生式头部非终结符: 'F', 产生式体文法符个数: 3},
产生式概述表 {产生式序号: 6, 产生式头部非终结符: 'F', 产生式体文法符个数: 1}]
```

```
Action部分 {状态序号:0, 终结符:'(', 操作类型/ID:s5},
Action部分 {状态序号:0, 终结符:'id', 操作类型/ID:s3},
Action部分 {状态序号:1, 终结符:'$', 操作类型/ID:r2},
Action部分 {状态序号:1, 终结符:'+', 操作类型/ID:r2},
Action部分 {状态序号:1, 终结符:')', 操作类型/ID:r2},
Action部分 {状态序号:1, 终结符:'*', 操作类型/ID:s6},
Action部分 {状态序号:2, 终结符:'$', 操作类型/ID:r4},
Action部分 {状态序号:2, 终结符:'+', 操作类型/ID:r4},
Action部分 {状态序号:2, 终结符:')', 操作类型/ID:r4},
Action部分 {状态序号:2, 终结符:'*', 操作类型/ID:r4},
Action部分 {状态序号:3, 终结符:'$', 操作类型/ID:r6},
Action部分 {状态序号:3, 终结符:'+', 操作类型/ID:r6},
Action部分 {状态序号:3, 终结符:')', 操作类型/ID:r6},
Action部分 {状态序号:3, 终结符:'*', 操作类型/ID:r6},
Action部分 {状态序号:4, 终结符:'$', 操作类型/ID:a0},
Action部分 {状态序号:4, 终结符:'+', 操作类型/ID:s7},
Action部分 {状态序号:5, 终结符:'(', 操作类型/ID:s5},
Action部分 {状态序号:5, 终结符:'id', 操作类型/ID:s3},
Action部分 {状态序号:6, 终结符:'(', 操作类型/ID:s5},
Action部分 {状态序号:6, 终结符:'id', 操作类型/ID:s3},
Action部分 {状态序号:7, 终结符:'(', 操作类型/ID:s5},
Action部分 {状态序号:7, 终结符:'id', 操作类型/ID:s3},
Action部分 {状态序号:8, 终结符:')', 操作类型/ID:s11},
Action部分 {状态序号:8, 终结符:'+', 操作类型/ID:s7},
Action部分 {状态序号:9, 终结符:'$', 操作类型/ID:r3},
Action部分 {状态序号:9, 终结符:'+', 操作类型/ID:r3},
Action部分 {状态序号:9, 终结符:')', 操作类型/ID:r3},
Action部分 {状态序号:9, 终结符:'*', 操作类型/ID:r3},
Action部分 {状态序号:10, 终结符:'$', 操作类型/ID:r1},
Action部分 {状态序号:10, 终结符:'+', 操作类型/ID:r1},
Action部分 {状态序号:10, 终结符:')', 操作类型/ID:r1},
Action部分 {状态序号:10, 终结符:'*', 操作类型/ID:s6},
Action部分 {状态序号:11, 终结符:'$', 操作类型/ID:r5},
Action部分 {状态序号:11, 终结符:'+', 操作类型/ID:r5},
Action部分 {状态序号:11, 终结符:')', 操作类型/ID:r5},
Action部分 {状态序号:11, 终结符:'*', 操作类型/ID:r5}]
```

```
Goto部分{状态序号=0, 非终结符='E', 下一状态=4},
Goto部分{状态序号=0, 非终结符='E', 下一状态=4},
Goto部分{状态序号=0, 非终结符='T', 下一状态=1},
Goto部分{状态序号=0, 非终结符='T', 下一状态=1},
Goto部分{状态序号=0, 非终结符='F', 下一状态=2},
Goto部分{状态序号=5, 非终结符='E', 下一状态=8},
Goto部分{状态序号=5, 非终结符='E', 下一状态=8},
Goto部分{状态序号=5, 非终结符='T', 下一状态=1},
Goto部分{状态序号=5, 非终结符='T', 下一状态=1},
Goto部分{状态序号=5, 非终结符='F', 下一状态=2},
Goto部分{状态序号=6, 非终结符='F', 下一状态=9},
Goto部分{状态序号=7, 非终结符='T', 下一状态=10},
Goto部分{状态序号=7, 非终结符='T', 下一状态=10},
Goto部分{状态序号=7, 非终结符='F', 下一状态=2}]
```

收获与体会：

（1）对 LL 语法分析有了进一步的理解，并且在编码实现的过程中可以逐渐熟练掌握消除左递归和提取左公因子的方法。

（2）对于求 FIRST 函数和 FOLLOW 函数也有了进一步的了解，但是如何判断 FOLLOW 依赖环仍没有完全实现。

（3）对于 LR(0)项目的闭包求解和变迁能够较好地掌握,在求解 DF 的同时掌握了语法分析的基本步骤。

（4）对于判断 LL(1)文法和 SLR(1)文法有了更加深入的了解，能够区分 LL(1)语法分析表和 LR 语法分析表的差异，并且完成构造。

实验成绩	