**《编译系统设计实践》**

实验项目三：语法分析程序实验

学号： 152201106

姓名： 庄仲华

年级： 2022

学院： 计算机与大数据学院

专业： 软件工程

本组其它成员：学号 052201224 姓名 游志军

学号 072203218 姓名 庄邵锟

学号 052203130 姓名 邱哲晗

学号 092200201 姓名 蔡子航

学号 052207115 姓名 欧阳开源

学号 042201126 姓名 杨世铭

实验时间：2024－2025学年第二学期

任课教师：陈晖

## 一、实验目的与任务

### 目的

加深对LR(1)分析法的理解，掌握语法分析表的构造方法，并能够实现对输入符号串的语法分析。本次实验的目的主要是加深对 LR（1）分析法的理解。

### 任务

1. 实现LR(1)分析表的构造与输出功能
2. 实现基于分析表的语法分析过程，并输出分析栈内容
3. 测试程序对不同文法规则和输入串的分析能力

## 二、程序功能描述

本程序实现了一个完整的LR(1)语法分析器，具有以下功能：

1. 从给定的文法规则自动构造LR(1)分析表
2. 输出构造的ACTION和GOTO表
3. 对输入的符号串进行语法分析
4. 详细输出分析过程中的状态栈、符号栈和剩余输入信息
5. 支持ε产生式处理

## 三、程序结构描述

### 主要类结构

1. **Grammar类**
   * 表示文法的产生式
   * 包含左部(lhs)和右部(rhs)两个主要属性
   * 实现了equals和hashCode方法用于比较
2. **Item类**
   * 表示LR(1)项目
   * 包含Grammar对象、点的位置(dotPos)和展望符(lookahead)
   * 实现了getSymbolAfterDot()方法获取点后的符号
3. **State类**
   * 表示LR(1)分析状态
   * 包含状态ID和项目集(items)
   * 实现了equals和hashCode方法用于状态比较
4. **LR1Parser类**
   * 核心分析器类
   * 主要方法：
     + computeSymbolSets(): 计算终结符和非终结符集合
     + closure(): 计算项目集的闭包
     + goTo(): 计算转移函数
     + first(): 计算FIRST集合
     + buildLR1Table(): 构造LR(1)分析表
     + parse(): 执行语法分析
     + printTables(): 打印分析表
5. **TestCases类**
   * 包含三个测试用例，分别测试算术表达式、if语句和变量声明

### 关键方法说明

1. **closure(Set items)**
   * 参数：项目集
   * 返回值：闭包后的项目集
   * 功能：计算给定项目集的闭包
2. **goTo(Set items, String symbol)**
   * 参数：项目集和符号
   * 返回值：转移后的项目集
   * 功能：计算通过给定符号转移后的项目集
3. **first(List symbols)**
   * 参数：符号序列
   * 返回值：FIRST集合
   * 功能：计算符号序列的FIRST集合
4. **buildLR1Table()**
   * 功能：构造完整的LR(1)分析表
   * 过程：
     1. 添加增广文法
     2. 构造初始项目集
     3. 通过闭包和转移函数扩展状态
     4. 填充ACTION和GOTO表
5. **parse(List input)**
   * 参数：输入符号串
   * 返回值：分析是否成功
   * 功能：执行语法分析并输出详细过程

## 四、测试数据及结果

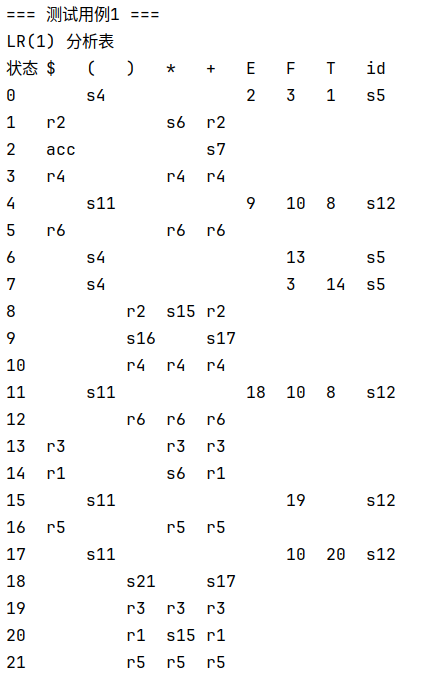
### 测试用例1：算术表达式

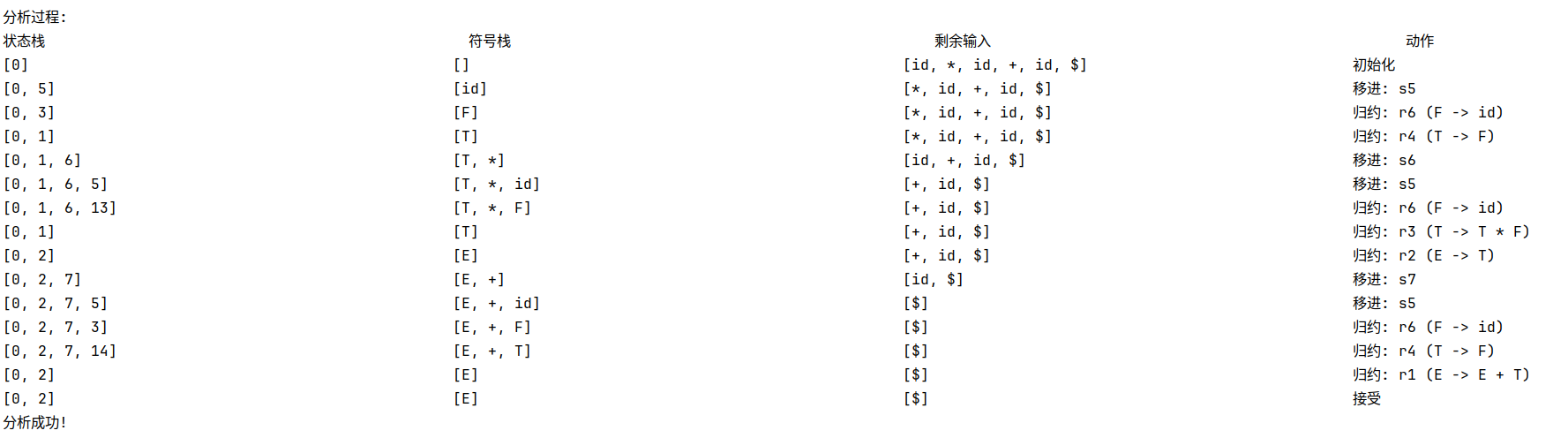
**文法规则**:

E → E + T  
E → T  
T → T \* F  
T → F  
F → ( E )  
F → id

**输入串**: id \* id + id

**输出结果**:





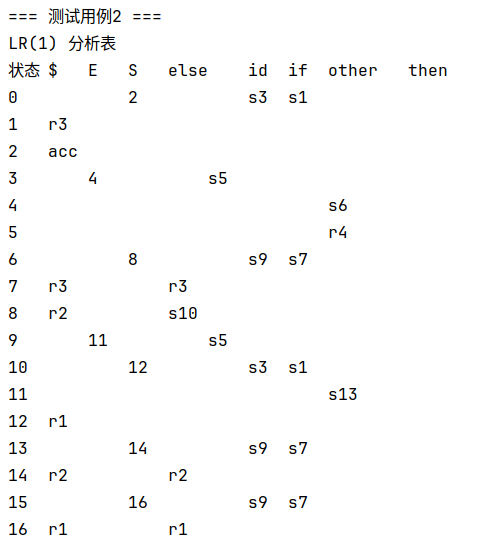
### 测试用例2：if语句

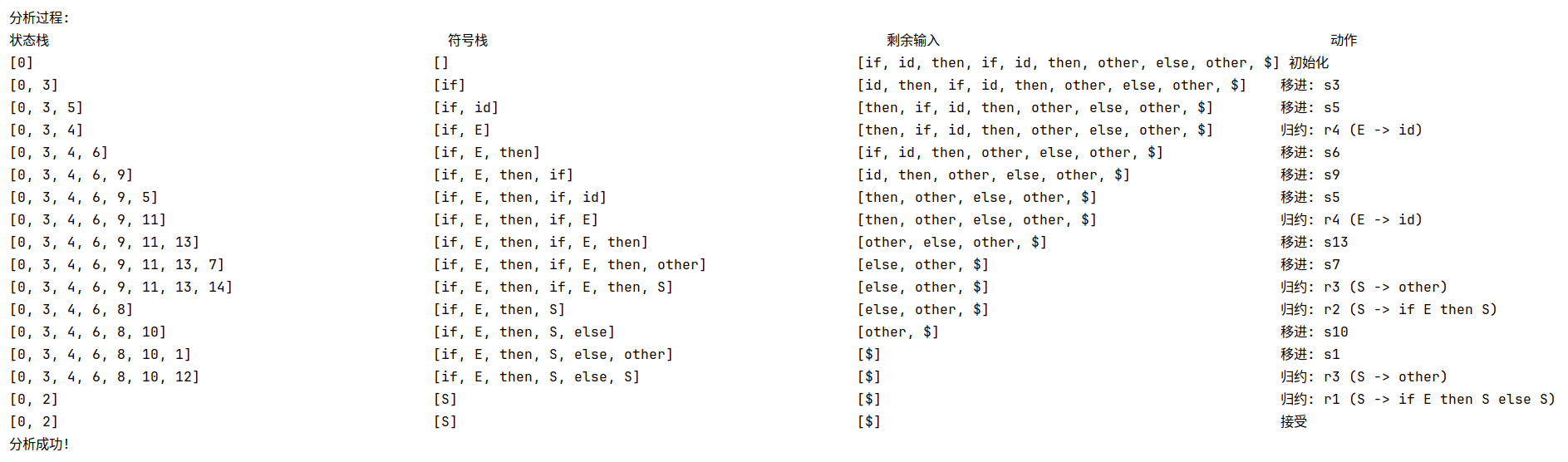
**文法规则**:

S → if E then S else S  
S → if E then S  
S → other  
E → id

**输入串**: if id then if id then other else other

**输出结果**:





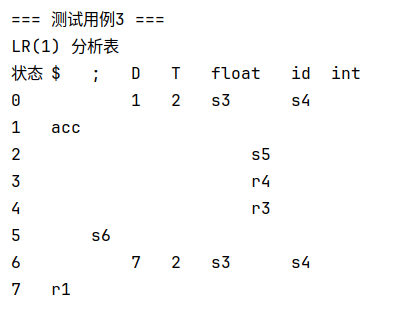
### 测试用例3：变量声明

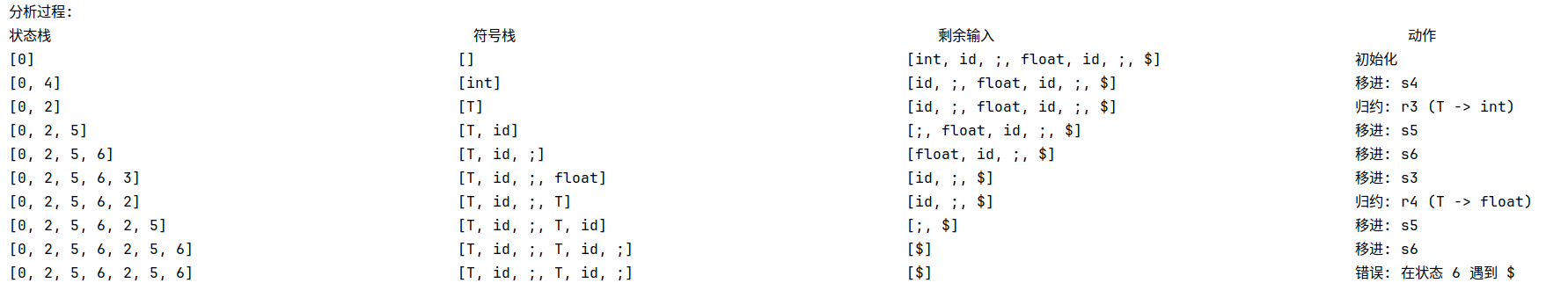
**文法规则**:

D → T id ; D  
D → ε  
T → int  
T → float

**输入串**: int id; float id;

**输出结果**:





## 五、符号表设计

程序中使用了以下主要数据结构：

1. **文法规则表示**:
   * 使用Grammar类存储产生式
   * lhs: 左部符号(String)
   * rhs: 右部符号列表(List)
2. **LR(1)项目表示**:
   * 使用Item类存储项目
   * grammar: 关联的文法规则
   * dotPos: 点的位置
   * lookahead: 展望符
3. **分析表结构**:
   * actionTable: Map<Integer, Map<String, String>>
     + 外层键: 状态ID
     + 内层键: 终结符
     + 值: 动作(sn/rn/acc)
   * gotoTable: Map<Integer, Map<String, Integer>>
     + 外层键: 状态ID
     + 内层键: 非终结符
     + 值: 转移状态

## 六、程序执行流程

1. 初始化LR1Parser对象，传入文法规则和开始符号
2. 计算终结符和非终结符集合
3. 构造LR(1)分析表:
   * 添加增广文法
   * 构造初始项目集
   * 通过闭包和转移函数扩展状态
   * 填充ACTION和GOTO表
4. 输出分析表
5. 对输入串进行语法分析:
   * 初始化状态栈和符号栈
   * 根据当前状态和输入符号查表执行动作
   * 输出每一步的分析状态
6. 输出最终分析结果

## 七、关键代码展示

### **1. Grammar类（文法表示）**

#### **关键代码**

public class Grammar {  
 public String lhs; // 产生式左部  
 public List<String> rhs; // 产生式右部  
  
 @Override  
 public String toString() {  
 return lhs + " -> " + String.join(" ", rhs); // 格式化输出，如 "E -> E + T"  
 }  
}

#### **功能说明**

* 表示一个上下文无关文法的产生式。
* lhs 和 rhs 分别存储左部和右部符号。
* toString() 用于直观展示文法规则。

### **2. Item类（LR(1)项目）**

#### **关键代码**

public class Item {  
 public Grammar grammar;  
 public int dotPos; // 点的位置  
 public String lookahead; // 展望符  
  
 public String getSymbolAfterDot() {  
 if (dotPos < grammar.rhs.size()) {  
 return grammar.rhs.get(dotPos); // 返回点后的符号  
 }  
 return null; // 点已在末尾  
 }  
  
 @Override  
 public String toString() {  
 // 格式化输出，如 "E -> E • + T, $"  
 StringBuilder sb = new StringBuilder();  
 sb.append(grammar.lhs).append(" -> ");  
 for (int i = 0; i < grammar.rhs.size(); i++) {  
 if (i == dotPos) sb.append("•");  
 sb.append(grammar.rhs.get(i)).append(" ");  
 }  
 if (dotPos == grammar.rhs.size()) sb.append("•");  
 sb.append(", ").append(lookahead);  
 return sb.toString();  
 }  
}

#### **功能说明**

* 表示LR(1)分析中的项目（如 A → α•β, a）。
* getSymbolAfterDot() 用于判断移进或归约。
* toString() 提供可读性输出，便于调试。

### **3. LR1Parser类（核心分析器）**

#### **关键方法1：闭包计算**

private Set<Item> closure(Set<Item> items) {  
 Set<Item> closureSet = new HashSet<>(items);  
 boolean changed;  
 do {  
 changed = false;  
 for (Item item : closureSet) {  
 String symbol = item.getSymbolAfterDot();  
 if (symbol != null && nonTerminals.contains(symbol)) {  
 // 计算展望符的FIRST集  
 Set<String> firstBeta = first(item.grammar.rhs.subList(item.dotPos + 1, item.grammar.rhs.size()));  
 for (Grammar g : grammars) {  
 if (g.lhs.equals(symbol)) {  
 for (String lookahead : firstBeta) {  
 Item newItem = new Item(g, 0, lookahead);  
 if (!closureSet.contains(newItem)) {  
 closureSet.add(newItem); // 扩展闭包  
 changed = true;  
 }  
 }  
 }  
 }  
 }  
 }  
 } while (changed);  
 return closureSet;  
}

#### **功能说明**

* 计算项目集的闭包，处理非终结符的展望符传播。
* 通过迭代扩展项目集，直到不再变化。

#### **关键方法2：分析表构建**

private void buildLR1Table() {  
 // 初始化增广文法  
 Grammar augmentedGrammar = new Grammar("S'", Arrays.asList(startSymbol));  
 Item initialItem = new Item(augmentedGrammar, 0, "$");  
 State initialState = new State(0, closure(Collections.singleton(initialItem)));  
  
 // 状态转移处理  
 for (State state : states) {  
 for (String symbol : terminals) {  
 Set<Item> newItems = goTo(state.items, symbol);  
 if (!newItems.isEmpty()) {  
 actionTable.get(state.id).put(symbol, "s" + newState.id); // 移进动作  
 }  
 }  
 for (String symbol : nonTerminals) {  
 Set<Item> newItems = goTo(state.items, symbol);  
 if (!newItems.isEmpty()) {  
 gotoTable.get(state.id).put(symbol, newState.id); // GOTO动作  
 }  
 }  
 }  
  
 // 归约和接受动作  
 for (State state : states) {  
 for (Item item : state.items) {  
 if (item.getSymbolAfterDot() == null) {  
 if (item.grammar.lhs.equals("S'")) {  
 actionTable.get(state.id).put("$", "acc"); // 接受  
 } else {  
 actionTable.get(state.id).put(item.lookahead, "r" + grammarIndex); // 归约  
 }  
 }  
 }  
 }  
}

#### **功能说明**

* 构建LR(1)的ACTION和GOTO表。
* 处理移进、归约和接受动作，解决冲突。

#### **关键方法3：语法分析**

public boolean parse(List<String> input) {  
 Stack<Integer> stateStack = new Stack<>();  
 Stack<String> symbolStack = new Stack<>();  
 stateStack.push(0); // 初始状态  
  
 while (true) {  
 int currentState = stateStack.peek();  
 String currentSymbol = input.get(ip);  
 String action = actionTable.get(currentState).get(currentSymbol);  
  
 if (action.startsWith("s")) {  
 // 移进动作  
 stateStack.push(Integer.parseInt(action.substring(1)));  
 symbolStack.push(currentSymbol);  
 ip++;  
 } else if (action.startsWith("r")) {  
 // 归约动作  
 Grammar grammar = grammars.get(Integer.parseInt(action.substring(1)));  
 for (int i = 0; i < grammar.rhs.size(); i++) {  
 stateStack.pop();  
 symbolStack.pop();  
 }  
 symbolStack.push(grammar.lhs);  
 stateStack.push(gotoTable.get(stateStack.peek()).get(grammar.lhs));  
 } else if (action.equals("acc")) {  
 return true; // 接受  
 } else {  
 return false; // 错误  
 }  
 }  
}

#### **功能说明**

* 模拟LR(1)分析过程，根据ACTION和GOTO表执行移进、归约和接受动作。
* 使用状态栈和符号栈跟踪分析进度。

### **4. 测试用例（TestCases类）**

#### **关键代码**

public class TestCases {  
 public static void testArithmeticExpression() {  
 List<Grammar> grammars = Arrays.asList(  
 new Grammar("E", Arrays.asList("E", "+", "T")),  
 new Grammar("E", Arrays.asList("T")),  
 new Grammar("T", Arrays.asList("T", "\*", "F")),  
 new Grammar("T", Arrays.asList("F")),  
 new Grammar("F", Arrays.asList("(", "E", ")")),  
 new Grammar("F", Arrays.asList("id"))  
 );  
 LR1Parser parser = new LR1Parser(grammars, "E");  
 parser.parse(Arrays.asList("id", "\*", "id", "+", "id"));  
 }  
}

#### **功能说明**

* 提供典型文法测试（算术表达式、if语句、变量声明）。
* 验证分析器的正确性和鲁棒性。

## 八、实验总结

通过本次实验，我们深入理解了LR(1)分析法的原理和实现过程，掌握了以下内容：

1. LR(1)项目集的构造方法
2. 闭包和转移函数的计算方法
3. LR(1)分析表的构造算法
4. 基于分析表的语法分析过程

实验中遇到的主要难点是ε产生式的处理和展望符的计算，通过仔细阅读教材和调试程序最终解决了这些问题。程序的测试结果表明，它能够正确分析不同类型的文法规则和输入串。

本次实验让我们对自底向上的语法分析方法有了更深刻的认识，为后续的编译器开发工作打下了坚实的基础。