## **青岛科技大学实验报告**

姓名： 魏鹏超 专业： 计算机科学与技术 班级： 计算224

学号： 2208010423 课程：编译原理

实验项目： 实验2 LL(1)分析器的设计与实现

**实验名称：LL(1)语法**分析

实验目的和要求

编制一个能识别由词法分析给出的单词符号序列是否是给定文法的正确句子（程序），并输出对输入符号串的分析过程。

实验内容和步骤：

1. **实验内容**

实现一个LL(1)语法分析器，能够识别由词法分析器给出的单词符号序列是否符合给定的文法规则。

输出分析过程中每一步的状态，包括分析栈、剩余输入串以及使用的产生式。

**二、实验步骤**

1.定义文法规则：

使用HashMap存储文法规则。

初始化FIRST集合和FOLLOW集合。

构建分析表。

2.计算FIRST集合：

遍历文法规则，对每个非终结符计算其FIRST集合。

根据产生式不断更新FIRST集合直到不再变化。

3.计算FOLLOW集合：

遍历文法规则，对每个非终结符计算其FOLLOW集合。

根据产生式不断更新FOLLOW集合直到不再变化。

4.构建分析表：

根据FIRST和FOLLOW集合构建分析表。

分析表中的每个条目对应一个非终结符和终结符的组合。

5.输入字符串的分析：

使用分析表对输入字符串进行语法分析。

输出分析过程的每一步，包括当前栈状态、剩余输入串以及使用的产生式或匹配情况。

实验代码如下：

package org.example;  
  
import java.util.\*;  
  
public class LL1Parser {  
 private static final Map<String, List<String[]>> grammar = new HashMap<>();  
 private static final Map<String, Set<String>> first = new HashMap<>();  
 private static final Map<String, Set<String>> follow = new HashMap<>();  
 private static final Map<String, Map<String, String[]>> parsingTable = new HashMap<>();  
  
 public static void main(String[] args) {  
 initializeGrammar();  
 computeFirstSets();  
 computeFollowSets();  
 constructParsingTable();  
  
 // 指定输入字符串  
 String[] input = {"i", "+", "i", "\*", "i", "$"};  
  
 parseInput(input);  
 }  
  
 private static void initializeGrammar() {  
 grammar.put("E", Arrays.<String[]>asList(new String[]{"T", "E'"}));  
 grammar.put("E'", Arrays.<String[]>asList(new String[]{"+", "T", "E'"}, new String[]{"ε"}));  
 grammar.put("T", Arrays.<String[]>asList(new String[]{"F", "T'"}));  
 grammar.put("T'", Arrays.<String[]>asList(new String[]{"\*", "F", "T'"}, new String[]{"ε"}));  
 grammar.put("F", Arrays.<String[]>asList(new String[]{"(", "E", ")"}, new String[]{"i"}));  
 }  
  
 private static void computeFirstSets() {  
 for (String nonTerminal : grammar.keySet()) {  
 first.put(nonTerminal, new HashSet<>());  
 }  
  
 boolean changed;  
 do {  
 changed = false;  
 for (String nonTerminal : grammar.keySet()) {  
 for (String[] production : grammar.get(nonTerminal)) {  
 int originalSize = first.get(nonTerminal).size();  
 Set<String> firstSet = computeFirstOfString(production);  
 first.get(nonTerminal).addAll(firstSet);  
 if (first.get(nonTerminal).size() > originalSize) {  
 changed = true;  
 }  
 }  
 }  
 } while (changed);  
 }  
  
 private static Set<String> computeFirstOfString(String[] production) {  
 Set<String> resultSet = new HashSet<>();  
 for (String symbol : production) {  
 if (!grammar.containsKey(symbol)) {  
 resultSet.add(symbol);  
 break;  
 }  
 Set<String> firstSet = first.get(symbol);  
 resultSet.addAll(firstSet);  
 if (!firstSet.contains("ε")) {  
 break;  
 }  
 }  
 return resultSet;  
 }  
  
 private static void computeFollowSets() {  
 for (String nonTerminal : grammar.keySet()) {  
 follow.put(nonTerminal, new HashSet<>());  
 }  
 follow.get("E").add("$");  
  
 boolean changed;  
 do {  
 changed = false;  
 for (String nonTerminal : grammar.keySet()) {  
 for (String[] production : grammar.get(nonTerminal)) {  
 for (int i = 0; i < production.length; i++) {  
 String symbol = production[i];  
 if (!grammar.containsKey(symbol)) {  
 continue;  
 }  
 int originalSize = follow.get(symbol).size();  
 Set<String> followSet = computeFollowOfSymbol(nonTerminal, production, i);  
 follow.get(symbol).addAll(followSet);  
 if (follow.get(symbol).size() > originalSize) {  
 changed = true;  
 }  
 }  
 }  
 }  
 } while (changed);  
 }  
  
 private static Set<String> computeFollowOfSymbol(String nonTerminal, String[] production, int index) {  
 Set<String> resultSet = new HashSet<>();  
 for (int i = index + 1; i < production.length; i++) {  
 String symbol = production[i];  
 if (!grammar.containsKey(symbol)) {  
 resultSet.add(symbol);  
 break;  
 }  
 Set<String> firstSet = first.get(symbol);  
 resultSet.addAll(firstSet);  
 if (!firstSet.contains("ε")) {  
 resultSet.remove("ε");  
 break;  
 }  
 }  
 if (index == production.length - 1 || resultSet.contains("ε")) {  
 resultSet.addAll(follow.get(nonTerminal));  
 }  
 resultSet.remove("ε");  
 return resultSet;  
 }  
  
 private static void constructParsingTable() {  
 for (String nonTerminal : grammar.keySet()) {  
 parsingTable.put(nonTerminal, new HashMap<>());  
 for (String[] production : grammar.get(nonTerminal)) {  
 Set<String> firstSet = computeFirstOfString(production);  
 for (String terminal : firstSet) {  
 if (!terminal.equals("ε")) {  
 parsingTable.get(nonTerminal).put(terminal, production);  
 }  
 }  
 if (firstSet.contains("ε")) {  
 Set<String> followSet = follow.get(nonTerminal);  
 for (String terminal : followSet) {  
 parsingTable.get(nonTerminal).put(terminal, production);  
 }  
 }  
 }  
 }  
 }  
  
 private static void parseInput(String[] input) {  
 Stack<String> stack = new Stack<>();  
 stack.push("$");  
 stack.push("E");  
  
 int index = 0;  
 int step = 1;  
  
 System.out.printf("%-10s%-30s%-30s%-30s%n", "步骤", "分析栈", "剩余输入串", "推导所用产生式或匹配");  
  
 while (!stack.isEmpty()) {  
 String top = stack.peek();  
 String current = input[index];  
  
 System.out.printf("%-10d%-30s%-30s", step++, stack, Arrays.toString(Arrays.copyOfRange(input, index, input.length)));  
  
 if (top.equals(current)) {  
 stack.pop();  
 index++;  
 System.out.printf("%-30s%n", "匹配 " + current);  
 } else if (grammar.containsKey(top)) {  
 String[] production = parsingTable.get(top).get(current);  
 if (production == null) {  
 System.out.println("Error: No production found for " + top + " with input " + current);  
 return;  
 }  
 stack.pop();  
 for (int i = production.length - 1; i >= 0; i--) {  
 if (!production[i].equals("ε")) {  
 stack.push(production[i]);  
 }  
 }  
 System.out.printf("%-30s%n", top + " -> " + String.join(" ", production));  
 } else {  
 System.out.println("Error: Unexpected symbol " + current);  
 return;  
 }  
 }  
  
 if (index == input.length && stack.isEmpty()) {  
 System.out.println("接受");  
 } else {  
 System.out.println("拒绝");  
 }  
 }  
}

**三、实验过程记录：**

1. computeFollowSets 方法

在计算 follow 集合时，代码没有正确处理非终结符右侧的符号，当产生式右侧符号的 first 集合中包含 ε 时，没有正确传递 follow 集合。

源代码：

private static Set<String> computeFollowOfSymbol(String nonTerminal, String[] production, int index) {  
 Set<String> resultSet = new HashSet<>();  
 for (int i = index + 1; i < production.length; i++) {  
 String symbol = production[i];  
 if (!*grammar*.containsKey(symbol)) {  
 resultSet.add(symbol);  
 break;  
 }  
 Set<String> firstSet = *first*.get(symbol);  
 resultSet.addAll(firstSet);  
 if (!firstSet.contains("ε")) {  
 break;  
 }  
 resultSet.remove("ε");  
 }  
 if (resultSet.isEmpty() || resultSet.contains("ε")) {  
 resultSet.addAll(*follow*.get(nonTerminal));  
 }  
 return resultSet;  
}

修改后代码：

private static Set<String> computeFollowOfSymbol(String nonTerminal, String[] production, int index) {  
 Set<String> resultSet = new HashSet<>();  
 for (int i = index + 1; i < production.length; i++) {  
 String symbol = production[i];  
 if (!*grammar*.containsKey(symbol)) {  
 resultSet.add(symbol);  
 break;  
 }  
 Set<String> firstSet = *first*.get(symbol);  
 resultSet.addAll(firstSet);  
 if (!firstSet.contains("ε")) {  
 break;  
 } else {  
 resultSet.remove("ε");  
 }  
 }  
 if (index + 1 >= production.length || resultSet.contains("ε")) {  
 resultSet.addAll(*follow*.get(nonTerminal));  
 }  
 return resultSet;  
}

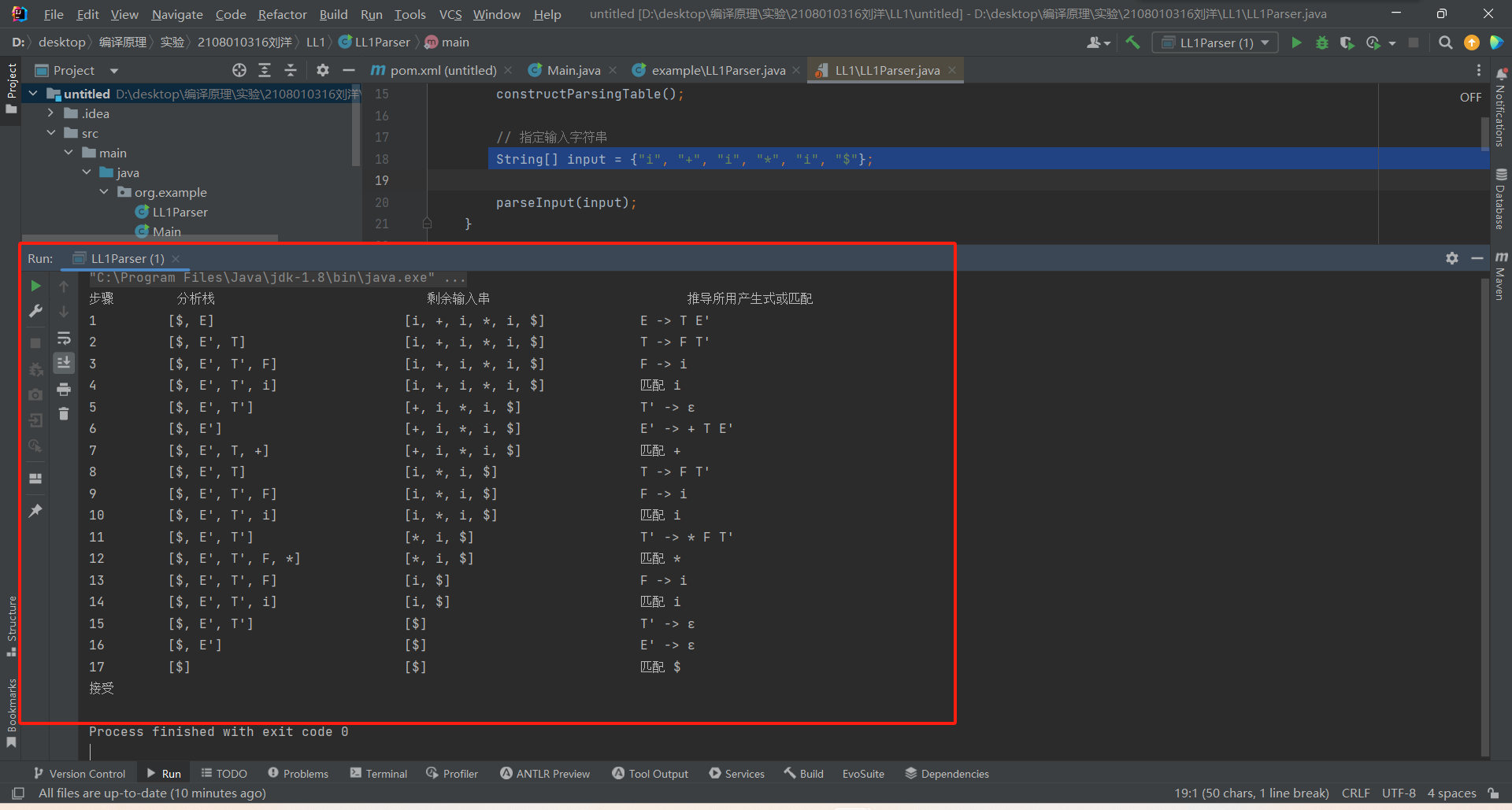
2. 在构造分析表时，代码没有正确处理 ε 产生式的情况

改正代码：

private static void constructParsingTable() {  
 for (String nonTerminal : *grammar*.keySet()) {  
 *parsingTable*.put(nonTerminal, new HashMap<>());  
 for (String[] production : *grammar*.get(nonTerminal)) {  
 Set<String> firstSet = *computeFirstOfString*(production);  
 for (String terminal : firstSet) {  
 if (!terminal.equals("ε")) {  
 *parsingTable*.get(nonTerminal).put(terminal, production);  
 }  
 }  
 if (firstSet.contains("ε")) {  
 Set<String> followSet = *follow*.get(nonTerminal);  
 for (String terminal : followSet) {  
 *parsingTable*.get(nonTerminal).put(terminal, production);  
 }  
 }  
 }  
 }  
}

**四、实验结果**

**对于该文法，当输入符号串为 i+i\*i#时，对应的结果为**



**五、实验总结**

在本次实验中，成功实现了一个基于LL(1)文法的语法分析器。通过编写该分析器，加深了对语法分析原理的理解，特别是对于FIRST集合和FOLLOW集合的计算方法，以及如何利用这些集合构建预测分析表。本实验的具体总结如下：

实现LL(1)语法分析器的基本步骤

文法规则定义：首先，我们定义了文法规则。这一步骤非常重要，因为文法规则的准确性直接影响后续的FIRST集合、FOLLOW集合的计算以及分析表的构建。

FIRST集合和FOLLOW集合的计算：这是构建预测分析表的基础。通过递归计算和迭代更新，确保了FIRST集合和FOLLOW集合的准确性。

分析表的构建：根据FIRST集合和FOLLOW集合，成功构建了LL(1)预测分析表。这个分析表用于指导语法分析过程。

输入串的语法分析：通过模拟分析栈的操作，我们实现了对输入符号串的逐步解析，并输出了每一步的状态，方便观察和调试。

关键技术点：

FIRST集合的计算：在计算FIRST集合时，需要考虑不同符号的性质，终结符直接加入FIRST集合，而非终结符则需要进一步展开其产生式。

FOLLOW集合的计算：FOLLOW集合的计算相对复杂，需要考虑符号在产生式中的不同位置，以及对应的上下文。

递归与迭代相结合：在计算FIRST和FOLLOW集合时，递归与迭代相结合的方式确保了集合的完整性和准确性。

错误处理与调试：在实现过程中，我遇到了一些问题，如FOLLOW集合计算错误等。通过逐步调试和修正，我们保证了程序的正确性。

通过本实验，我深入理解了LL(1)文法的核心概念，包括FIRST集合、FOLLOW集合、预测分析表等，提升了我在编写语法分析器方面的能力。