成绩



2019-2020第2学期

《编译原理》实验二分组报告

|  |  |  |
| --- | --- | --- |
| **题 目：** | LL1语法分析器 | |
| **学 院：** | 计算机与软件学院 | |
| **专 业：** | 计算机科学与技术 | |
| **年级班级：** | 17级2班 | |
| **课题成员** | 陈佳欢 20171308065 |  |
|  | 何爽 20171308067 |  |
| **任课教师：** | 郑关胜 | |

**二Ｏ二Ｏ 年 五 月 二十七 日**

|  |  |
| --- | --- |
| **实践小组成员分工及职责** | |
| 学号姓名 | 任务分工 |
| 陈佳欢 | 语法分析器逻辑代码实现 |
| 何爽 | JavaFX界面设计 |

目 录

[1编译原理基本理论 1](#_Toc41916048)

[1.1编译程序的相关概念 1](#_Toc41916049)

[1.2编译过程 1](#_Toc41916050)

[1.2.1词法分析 1](#_Toc41916051)

[1.2.2语法分析 1](#_Toc41916052)

[1.2.3语法分析与中间代码产生 1](#_Toc41916053)

[1.2.4优化 2](#_Toc41916054)

[1.2.5目标代码生成 2](#_Toc41916055)

[2 LL(1)分析 2](#_Toc41916056)

[2.1环境介绍 2](#_Toc41916057)

[2.1.1开发环境——IDEA 2](#_Toc41916058)

[2.1.2可视化技术——JavaFX 2](#_Toc41916059)

[2.2分析技术 2](#_Toc41916060)

[2.3项目主要类介绍 3](#_Toc41916061)

[2.4分析过程 4](#_Toc41916062)

[2.4.1消除直接左递归 4](#_Toc41916063)

[2.4.2构造预测分析表 5](#_Toc41916064)

[2.4.2.1计算First集 6](#_Toc41916065)

[2.4.2.2计算Follow集 7](#_Toc41916066)

[2.4.2.3判断LL(1)文法 10](#_Toc41916067)

[2.4.2.4预测分析表 12](#_Toc41916068)

[2.4.3 字符串的预测分析 14](#_Toc41916069)

[3 实验结果 17](#_Toc41916070)

[4 小结 20](#_Toc41916071)

[参考文献 21](#_Toc41916072)

# 1编译原理基本理论

## 1.1编译程序的相关概念

编译程序：把高级语言（源语言）表示转换成等价的低级语言（目标语言：汇编或者机器语言）表示的翻译程序，即把面向人的高级抽象描述转换成等价的可执行程序的程序。

遍：对源程序或者源程序的中间结果进行从头到尾的扫描，并且做出相应的处理，生成新的中间结果或者目标程序的过程。

表格和表格管理：编译的每个阶段都会有相应的表格记录各类信息以及编译的进展状况。

出错处理：当编译的过程中有错误出现，应当及时进行报错处理，找出具体的出错地点、原因，并且使得范围尽量小，使源程序的其余部分继续执行。

文法：定义一组规则产生式的集合。一组产生式是一个四元式（VT,VN,S,P)，由终结符号集合、非终结符号集合、开始符号、一组产生式集合构成。

语言：定义一个文法中所有句子的集合。其中，句子表示能由文法的开始符号经过若干步推导出的、仅含终结符号的句型。

语法树：句型的推导过程就是一颗语法树。

二义性：是指对于一种推导有多中不同的结果，而不是分别最左推导和最右推导有不同的结果。有二义性的文法可形成不同的语法分析树。

## 1.2编译过程

编译的过程分为几个阶段：源程序->词法分析->语法分析->语义分析和中间代码生成->优化->目标代码的生成->目标代码。错误处理贯穿整个编译过程，每个阶段都可能产生错误，必须及时处理。

### 1.2.1词法分析

词法分析是输入语言源程序，对构成源程序的字符串从左到右进行扫描和分解，识别出若干单词，并转换成统一的内部表示（token），输出<单词的种类标示符，单词的值>形式的二元组，送给语法分析程序。

### 1.2.2语法分析

语法分析是在词法分析基础上，根据语言的语法规则，输入单词符号串，将其分解成各类语法单位，来判断输入的终结符号串是否为该语言中的句子。

### 1.2.3语法分析与中间代码产生

语义分析与中间代码产生阶段是对语法分析所识别出的各类语法范畴，分析其含义并进行初步翻译（产生中间代码）。通常包括两方面工作，即对静态语义检查、解释执行动态语义（初步翻译）并生成中间代码。

### 1.2.4优化

优化是指编译时刻为改进目标程序的质量而进行的各项工作。它对前段产生的中间代码再次扫描和翻译，经过加工变换去掉冗余的中间代码，以期在最后阶段能产出更为高效的目标代码。

### 1.2.5目标代码生成

目标代码生成是把中间代码（或经优化处理之后）变换成特定机器上的低级语言代码（绝对指令代码或可重定位的指令代码或汇编指令代码）。这个阶段实现了最后的翻译，它的工作有赖于硬件系统结构和机器指令含义。

# 2 LL(1)分析

## 2.1环境介绍

### 2.1.1开发环境——IDEA

IDEA全称 IntelliJ IDEA，是java编程语言开发的集成环境。IntelliJ是非常人性化的java开发工具，在智能代码助手、代码自动提示、重构、JavaEE支持、各类版本工具(git、svn等)、JUnit、CVS整合、代码分析、 创新的GUI设计等方面的功能尤其优异。

### 2.1.2可视化技术——JavaFX

JavaFX工具的基本思想类似于针对Swing的Matisse项目，它提供了与VB或Delphi非常相像的可视化编辑器，简化了 Swing的开发工作。JavaFX工作台工作时生成代码，而且可以查看生成的代码[1]。

当从调色板(palette）中拖拽一个JavaFX组件到设计区时，它可以绑定数据源。该插件支持JDBC数据源、HTTP (XML / JSON）、文件系统（FileSystem）、文件数据源以及内建过滤器。它还提供一些简单的转换器，因而可以方便地将某数据源（比如REST或 JSON数据源)绑定到一组组件上。

## 2.2分析技术

本实验采用LL(1)分析方法，这是一种不带回溯的非递归的自顶向下的分析技术。自顶向下分析是对任何输入串，试图用一切可能的办法，从文法开始符号（根结点）出发，自上而下地为输入串建立一棵语法树。或者说，为输入串寻找一个最左推导。这种分析过程本质上是一种试探过程，是反复使用不同产生式谋求匹配输入串的过程。自顶向下的分析技术存在两个关键的问题：文法的左递归问题和回溯问题。左递归问题是指文法中存在非终结符号P和符号串，有，这样的自上而下的分析过程陷入无限循环。回溯问题是指在推导过程中选择了错误的推导规则，必须进行回溯，才可能推出正确的句子。左递归问题使程序陷入无限循环，而回溯问题分析的效率低下。

L L(1)分析方法的基本思想是从左向右根据输入串当前输入的一个符号，确定唯一的规则进行推导。当输入的符号（终结符号）和推导规则右侧的第一个符号相同，再取下一个符号重复推导过程，直至推导出输入串或者出错。L L(1)中的第一个“L”表示自左而右地扫描输入，第二个“L”表示生成一个最左推导，“1”表示为决定分析动作，每一步利用一个向前看的符号。[2]

L L(1)分析方法使用等价的非左递归产生式改写左递归的规则，消除直接左递归；并以一定的顺序排列文法规则左侧的非终结符号，不断把该非终结符号能推导出的字符串带入规则右侧，因而实现了消除间接左递归。若规则右部有左因子，应先提取左因子。为进一步简化，还可以消除有害规则和省略多余规则。L L(1)分析方法既消除了左递归，又能巧妙地避免回溯，是自顶向下分析的有效方法之一。

## 2.3项目主要类介绍

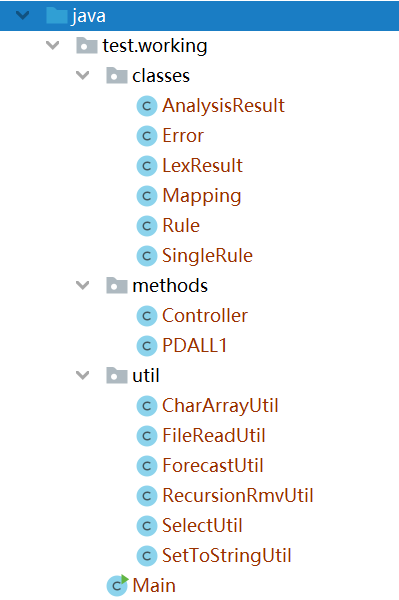


图1 主要类图

Mapping: 存放语法规则的映射。以一个非终结符号key（规则左部）为单位，存放key和其对应的value（所有规则右部的字符串）。其中的方法包括从文件中读取语法规则字符串、形成映射、计算所有key对应的first集和follow集、判断是否符合LL(1)文法要求。

Rule: 存放单个key对应和其对应的所有规则values。其中的方法包括计算单个key对应单个first集和follow集、判断单个key的规则是否符合LL(1)文法要求。

PDALL1: LL(1)的下推自动机，用来分析字符串是否是该语言文法中的句子。用栈结构以单个字符为单位实现对字符串的分析判断，并用字符串的列表形式记录下整个分析过程。

ForecastUtil: 实现预测分析表的构造，结果用字符串类型的二维数组表示。

RecursionRmvUtil: 消除规则映射中的左递归，检测出规则中的左递归并用新的规则代替，从而消除左递归。

项目中的均用#代替。

## 2.4分析过程

图2 分析过程流程图

### 2.4.1消除直接左递归

直接左递归：

在语法规则中能找出其中存在某非终结符号A，最终会推导出来的句型里面包含以自己为最左符号的句型称为含有直接左递归的句型。即出现以下句型规则

的句型。

本实验实现的是消除直接左递归，用RecursionRmvUtil类中的removeDctRecur静态方法实现。实现过程如下：

public static Map<Character, ArrayList<String>> removeDctRecur  
 (Character key,Map<Character, ArrayList<String>> map){  
 Map<Character, ArrayList<String>> tmpMap = map;  
 // 若出现形如A->A或A->#则出现环路，出错  
 for(String value:map.get(key)){  
 if(value.equals(key) || value.equals("#")){  
 new Error(Error.Type.LoopError);  
 System.exit(0);  
 }  
 }  
 // 在字母表中找到未使用过的大写字母，作为新的语法规则左部  
 ArrayList<Character> letters = new ArrayList<>();  
 for(char ch = 90;ch >= 65;ch--){  
 letters.add(ch);  
 }  
 Character newKey = null;  
 for(Character ch:letters){  
 if(map.containsKey(ch) == false){  
 newKey = ch;  
 break;  
 }  
 }  
 // 用A->βZ替换A->Aα|β  
 ArrayList<String> values = map.get(key);  
 ArrayList<String> newValues = new ArrayList<>();  
 ArrayList<String> newKeyValues = new ArrayList<>();  
 for(String value:values){  
 Character fstChar = value.charAt(0);  
 if(fstChar.equals(key)){  
 newKeyValues.add(value);  
 continue;  
 }  
 String newValue = value+newKey;  
 newValues.add(newValue);  
 }  
 tmpMap.put(key,newValues);  
 // 新增Z->αZ|#  
 for(int i = 0;i < newKeyValues.size();i++){  
 String newKeyValue = newKeyValues.get(i);  
 newKeyValues.set(i,newKeyValue.substring(1)+newKey);  
 }  
 newKeyValues.add("#");  
 tmpMap.put(newKey,newKeyValues);  
  
 return tmpMap;  
}

### 2.4.2构造预测分析表

#### 2.4.2.1计算First集

First集：

令G是一个不含左递归的文法，对G的所有非终结符的每个候选α定义，它的First集为

特别的，若，则规定。即是α的所有可能推导的开头终结符或可能的。

本实验中，运用Mapping类中的findAllFirstSet方法和Rule类中的findFirstSet方法的结合，计算规则映射中所有非终结符号的first集。实现过程如下：

Mapping类中：

public Map<Character,ArrayList<Character>> findAllFirstSet(Map<Character,ArrayList<String>> map){  
 Map<Character,ArrayList<Character>> firstSets = new HashMap<>();  
 // 扫描映射中的每一个数据项<key,value>  
 for(Map.Entry<Character,ArrayList<String>> entry:map.entrySet()){  
 Character key = entry.getKey();  
 Rule rule = new Rule(map,key);  
 // 调用Rule类的方法获得该key对应的first集  
 ArrayList<Character> firstSet = rule.findFirstSet(key,map);  
 firstSets.put(key,firstSet);  
 }  
 return firstSets;  
}

Rule类中：

public ArrayList<Character> findFirstSet(Character key,Map<Character,ArrayList<String>> map){  
 boolean isAllNull = false;  
 // 找到当前key对应的first集  
 for(String value:map.get(key)){  
 char fstChar = value.charAt(0);  
 // 当前存在推导出空集，标志出来  
 if(fstChar == '#'){  
 isNull = true;  
 continue;  
 }  
 // 推导出的第一个元素是非终结符号，则加入firstSet  
 else if(Character.isUpperCase(fstChar) == false){  
 this.firstSet.add(fstChar);  
 }  
 // 是终结符号，递归计算下一个字符  
 else{  
 key = value.charAt(0);  
 firstSet = findFirstSet(key,map);  
  
 if(isNull){  
 isAllNull = true;  
 }  
 }  
 if(isAllNull || this.isNull){  
 if(this.firstSet.contains('#') == false){  
 this.firstSet.add('#');  
 }  
 }  
 }  
 return this.firstSet;  
}

#### 2.4.2.2计算Follow集

Follow集：

假定S是文法G的开始符号，对于G的任何非终结符A，定义

特别的，若，则规定。即是所有句型中出现在紧接A之后的终结符或。

本实验中，运用Mapping类中的findAllFollowSet方法和Rule类中的findFollowSet方法的结合，计算规则映射中所有非终结符号的follow集。实现过程如下：

Mapping类中：

public Map<Character,ArrayList<Character>> findAllFollowtSet(Map<Character,ArrayList<String>> map){  
 Map<Character,ArrayList<Character>> followSets = new HashMap<>();  
 // 扫描映射中的每一个数据项<key,value>  
 for(Map.Entry<Character,ArrayList<String>> entry:map.entrySet()){  
 Character key = entry.getKey();  
 Rule rule = new Rule(map,key);  
 // 调用Rule类的方法获得该key对应的follow集  
 ArrayList<Character> followSet = rule.findFollowSet(key,map);  
 followSets.put(key,followSet);  
 }  
 return followSets;  
}

Rule类中：

public ArrayList<Character> findFollowSet(Character key,Map<Character,ArrayList<String>> map){  
 // 若为文法开始符号，则先加入$  
 if(key == 'S'){  
 if(this.followSet.contains('$') == false){  
 this.followSet.add('$');  
 }  
 }  
 // 规则右部含有该非终结符号的子映射  
 Map<Character,ArrayList<String>> subMap = new HashMap<>();  
 for(Map.Entry<Character,ArrayList<String>> entry:map.entrySet()){  
 for(String nowValue:entry.getValue()){  
 if(nowValue.contains(String.valueOf(key))){  
 if(subMap.containsKey(entry.getKey()) == false){  
 ArrayList<String> validValues= new ArrayList<>();  
 validValues.add(nowValue);  
 subMap.put(entry.getKey(),validValues);  
 }  
 else{  
 subMap.get(entry.getKey()).add(nowValue);  
 }  
 }  
 }  
 }  
 // 在子映射中找到该非终结符号的后一个元素  
 for(Map.Entry<Character,ArrayList<String>> entry:subMap.entrySet()){  
 Character nowKey = entry.getKey();  
 ArrayList<String> nowValues = new ArrayList<>();  
 for(String s:entry.getValue()){  
 nowValues.add(s);  
 }  
 for(String nowValue:nowValues){  
 // 该非终结符号所在的字符串中的位置  
 int ind = nowValue.indexOf(key);  
 // 若该非终结符号是最后一个符号  
 if(ind == nowValue.length()-1){  
 if(nowKey.equals(key)){  
 continue;  
 }  
 followSet = findFollowSet(nowKey,map);  
 }  
 // 该非终结符号后面是终结符号，直接加入follow集  
 else if(Character.isUpperCase(nowValue.charAt(ind+1)) == false){  
 if(this.followSet.contains(nowValue.charAt(ind+1)) == false)  
 this.followSet.add(nowValue.charAt(ind+1));  
 }  
 // 该非终结符号后面仍然是非终结符号，递归计算follow集  
 else{  
 Character nextChar = nowValue.charAt(ind+1);  
 ArrayList<Character> firstValues = findFirstSet(nextChar,map);  
 for(Character ch:firstValues){  
 if(this.followSet.contains(ch) == false && ch.equals('#') == false)  
 this.followSet.add(ch);  
 }  
 if(firstValues.contains('#')){  
 ArrayList<Character> followValues = findFollowSet(nowKey,map);  
 for(Character ch:followValues){  
 if(this.followSet.contains(ch) == false)  
 this.followSet.add(ch);  
 }  
 }  
 }  
 }  
 }  
 return this.followSet;  
}

#### 2.4.2.3判断LL(1)文法

LL(1)文法：

如果文法满足以下条件：

①文法不含左递归；

②对于文法中每一个非终结符A的各个产生式的First集两两不相交。即若

则

③对文法中每个非终结符A，若存在某个First集包含，则

则该文法为LL(1)文法。

本实验中，运用Mapping类中的findAllSelectSets方法和Rule类中的findSelectSet方法、findSelectSets方法、findSelectMap方法、compare方法、compareSelectSet方法的结合，来判断该文法是否为LL1文法。几个主要方法的实现过程如下：

Mapping类中：

public Map<Character,ArrayList<Character[]>> findAllSelectSets(Map<Character,ArrayList<String>> map){  
 Map<Character,ArrayList<Character[]>> selectSets = new HashMap<>();  
 // 扫描映射中的每一个数据项<key,value>  
 for(Map.Entry<Character,ArrayList<String>> entry:map.entrySet()){  
 Character key = entry.getKey();  
 Rule rule = new Rule(map,key);  
 // 调用Rule类的方法获得该key对应的select集  
 ArrayList<Character[]> selectSet = rule.findSelectSets(key,map);  
 selectSets.put(key,selectSet);  
 }  
 return selectSets;  
}

Rule类中：

public ArrayList<Character> findSelectSet(Character key,String value,Map<Character,ArrayList<String>> map){  
 char fstChar = value.charAt(0);  
 // 存在推导出空集，则寻找该key对应的follow集  
 if(fstChar == '#'){  
 selectSet = findFollowSet(key,map);  
 }  
 // 推导出的第一个元素是非终结符号，加入select集  
 else if(Character.isUpperCase(fstChar) == false){  
 ArrayList<Character> tmpChar = new ArrayList<>();  
 tmpChar.add(fstChar);  
 selectSet = tmpChar;  
 }  
 // 推导出的第一个元素是非终结符号，找到对应的first集  
 else{  
 selectSet = findFirstSet(fstChar,map);  
 }  
 return this.selectSet;  
}

public boolean compare(ArrayList<Character> charArray){  
 int times = 0;  
 for(Character[] chs:selectSets){  
 ArrayList<Character> nowArray = new ArrayList<>(Arrays.asList(chs));  
 // 去除与该规则本身的比较  
 if(nowArray.equals(charArray)){  
 times++;  
 continue;  
 }  
 if(times > 1)  
 return false;  
 // 判断是否含有交集  
 // retainAll(String)保存String中与当前字符串相同的部分  
 nowArray.retainAll(charArray);  
 // 有交集则返回false  
 if(nowArray.size() != 0)  
 return false;  
 }  
 return true;  
}

#### 2.4.2.4预测分析表

预测分析表M的构造：

1. 对文法G的每个产生式执行第二步和第三步
2. 对每个终结符,把加至M[A,a]中
3. 若,则对任何把加至M[A,b]中
4. 把所有无定义的M[A,a]标上出错标志

本实验中，运用Mapping类中的tableInit方法和analyzeTableMap方法的结合，来实现预测分析表的构造。实现过程如下：

public Character[][] tableInit(Map<Character,ArrayList<String>> map){  
 ArrayList<Character> left = new ArrayList<>();  
 ArrayList<Character> right = new ArrayList<>();  
 for(Map.Entry<Character,ArrayList<String>> entry:map.entrySet()){  
 Character key = entry.getKey();  
 ArrayList<String> values = entry.getValue();  
 // 第一列，所有的非终结符号  
 if(left.contains(key) == false){  
 left.add(key);  
 }  
 // 第一行，所有的非终结符号和$  
 for(String value:values){  
 char[] arrChar = value.toCharArray();  
 for(Character ch:arrChar){  
 if(right.contains(ch) == false &&  
 Character.isUpperCase(ch) == false &&  
 ch.equals('#') == false){  
 right.add(ch);  
 }  
 }  
 }  
 }  
 right.add('$');  
 // 放入table数组  
 table = new Character[left.size()+1][right.size()+1];  
 for(int i = 0;i < left.size();i++){  
 table[i+1][0] = left.get(i);  
 }  
 for(int i = 0; i < right.size();i++){  
 table[0][i+1] = right.get(i);  
 }  
 return table;  
}

public Map<Integer[],SingleRule> analyzeTableMap  
 (Map<Character,ArrayList<String>> map,Character[][] table){  
 tableMap = new HashMap<>();  
 ArrayList<Character> keys = new ArrayList<>();  
 for(Map.Entry<Character,ArrayList<String>> entry:map.entrySet()){  
 Character key = entry.getKey();  
 keys.add(key);  
 }  
 Character[] charKeys = keys.toArray(new Character[keys.size()]);  
  
 int row = 1,col;  
 for(Character key:charKeys){  
 // 获得所有select集合的映射  
 Map<SingleRule,ArrayList<Character>> selectMap =  
 new Rule(map,key).findSelectMap(key,map);  
 for(Map.Entry<SingleRule,ArrayList<Character>> entry:selectMap.entrySet()){  
 SingleRule srule = entry.getKey();  
 ArrayList<Character> selectSet = entry.getValue();  
 // 得到select集合中元素在table中的位置  
 for(Character ch:selectSet){  
 if(ch.equals('$')){  
 col = table[0].length-1;  
 }  
 else{  
 col = new CharArrayUtil(table[0]).getIndex(ch);  
 }  
 Integer[] ptr = {row,col};  
 tableMap.put(ptr,srule);  
 }  
 }  
 row++;  
 }  
 return tableMap;  
}

### 2.4.3 字符串的预测分析

预测分析程序的总控程序在任何时候都是根据Stack栈顶符号X和当前的输入符号a决定下一步做什么的，对于任何(X,a)，总控程序每次都执行下述三种可能的动作之一：

若X是终结符号，且，则分析成功，停止分析过程

若X是终结符号，且，则把X从Stack栈顶弹出，让字符串指针指向a的下一个输入符号

若X是一个非终结符，则查看分析表M。若M[X,a]中存放着关于X的一条规则，则先把X弹出Stack栈顶，然后把规则的右部符号串按反序一一推进Stack栈（若右部符号为ε，则不推进栈）。若M[X,a]中为空，则表示出错，进行错误处理。

本实验中，运用PDALL1类中的analyze方法，来实现预测分析过程。以操作步骤、Stack栈中内容、剩余字符串和当前选用的预测分析表中的规则作为分析结果，把分析结果用String数组的列表形式返回。实现过程如下：

public ArrayList<String[]> analyze(){  
 ArrayList<String[]> analyzeTbl;  
 analyzeTbl = new ArrayList<>();  
 // 存放步骤、栈中内容、剩余字符串、对应的规则信息  
 String[] item;  
 int step = 0;  
 StringBuilder stackInfo;  
 String nowStr;  
 SingleRule srule;  
 // 记录是否分析结束  
 boolean isEnd = false;  
 while(isEnd == false){  
 // 各项信息初始化  
 item = new String[4];  
 step++;  
 stackInfo = new StringBuilder();  
 nowStr = string.substring(ptr);  
 srule = null;  
 // 将栈中内容放入StringBuilder  
 for(Character ch:stack){  
 stackInfo.append(ch);  
 }  
// System.out.println(stackInfo.toString());  
 // 记录当前输入符号和栈顶  
 Character nowChar = string.charAt(ptr);  
 Character stkTop = stack.peek();  
 // 栈顶为$，即栈空，则判断当前输入符是否也为$  
 // 若是，则分析成功，结束分析；若不是，则出错  
 if(stkTop.equals('$')){  
 if(nowChar.equals('$')){  
 state = State.SUCCEED;  
 isEnd = true;  
 }  
 else{  
 state = State.ERROR;  
 isEnd = true;  
 }  
 }  
 // 当前栈顶是非终结符号，和当前输入符匹配  
 // 匹配则出栈，输入符指针后移；否则出错  
 else if(Character.isUpperCase(stkTop) == false){  
 if(nowChar.equals(stkTop)){  
 stack.pop();  
 ptr++;  
 }  
 else{  
 state = State.ERROR;  
 isEnd = true;  
 }  
 }  
 // 当前栈顶是非终结符号  
 else{  
 boolean canFind = false;  
 for(Map.Entry<Integer[],SingleRule> entry:tableMap.entrySet()){  
 int row = new CharArrayUtil(table).getIndex2(stkTop);  
 int col = new CharArrayUtil(table[0]).getIndex(nowChar);  
 Integer[] tableLoc = entry.getKey();  
 Integer[] nowLoc = {row,col};  
 srule = entry.getValue();  
 // 利用预测分析表找到相应规则，即把规则右部对应的字符串中字符依次出栈  
 if(Arrays.equals(tableLoc,nowLoc)){  
 canFind = true;  
 Character key = srule.getKey();  
 String value = srule.getValue();  
 stack.pop();  
 for(int i = value.length()-1;i >= 0;i--){  
 Character ch = value.charAt(i);  
 if(ch.equals('#')){  
 break;  
 }  
 stack.push(ch);  
 }  
 break;  
 }  
 }  
 // 否则出错  
 if(canFind == false){  
 state = State.ERROR;  
 isEnd = true;  
 }  
 }  
 // 保存各项信息  
 item[0] = String.valueOf(step);  
 item[1] = stackInfo.toString();  
 item[2] = nowStr;  
 if(srule == null)  
 item[3] = null;  
 else  
 item[3] = srule.toString();  
 analyzeTbl.add(item);  
 }  
 return analyzeTbl;  
 }

# 3 实验结果

初始界面：

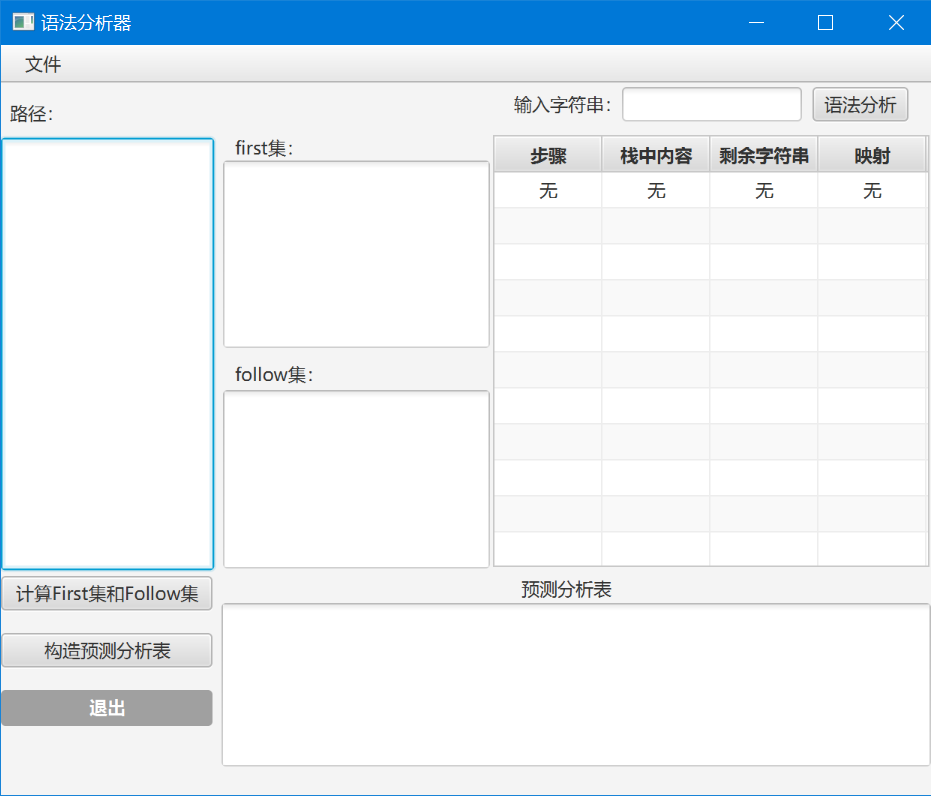


图3 初始界面展示图

打开文件：

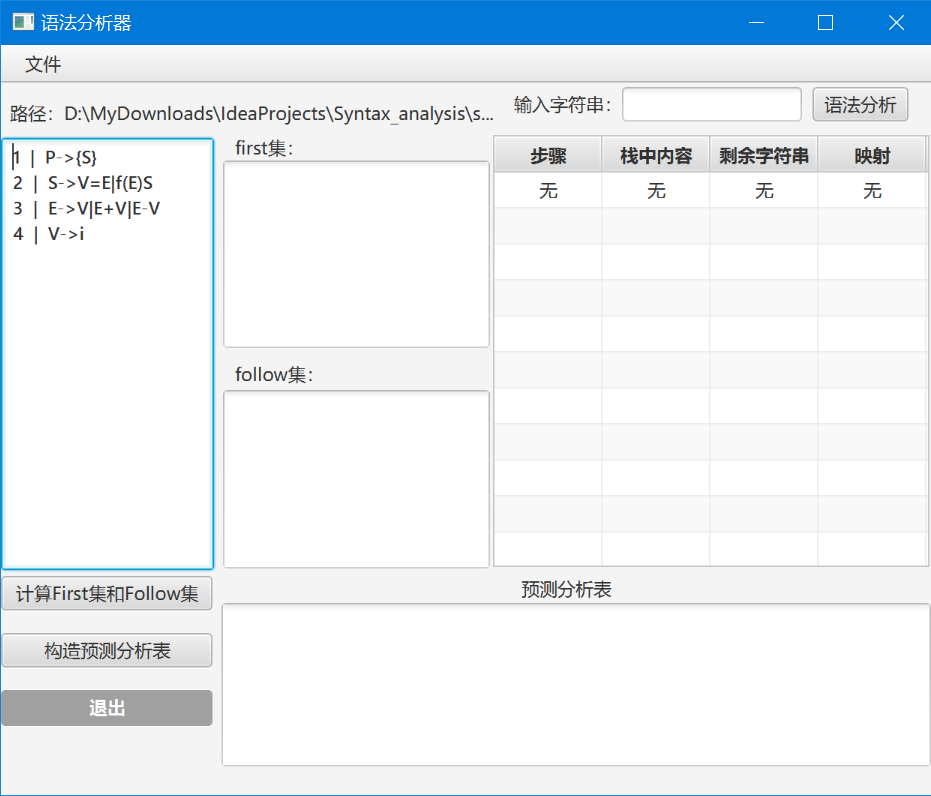


图4 打开文件展示图

计算First集和Follow集：

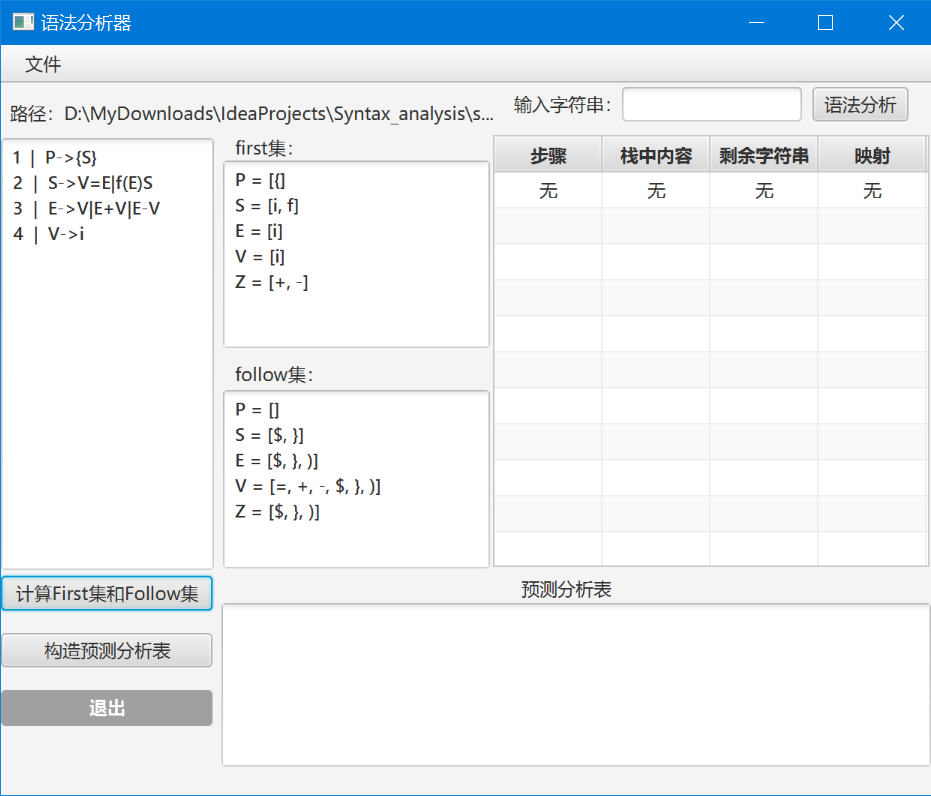


图5 计算First和Follow集展示图

构造预测分析表：

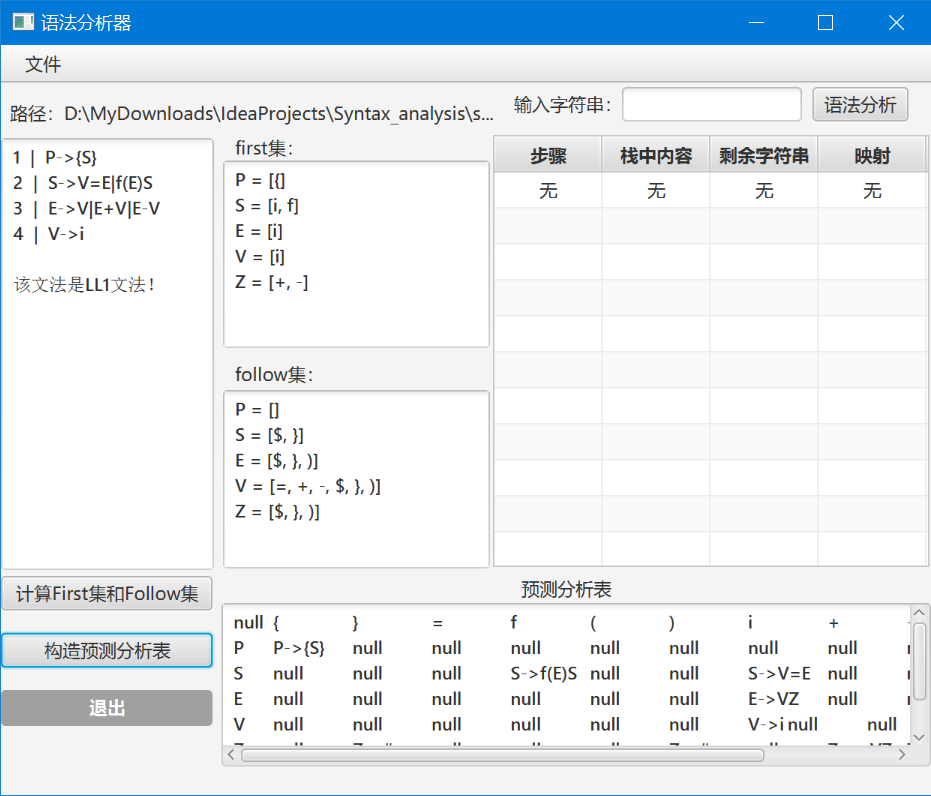


图6 构造预测分析表展示图1

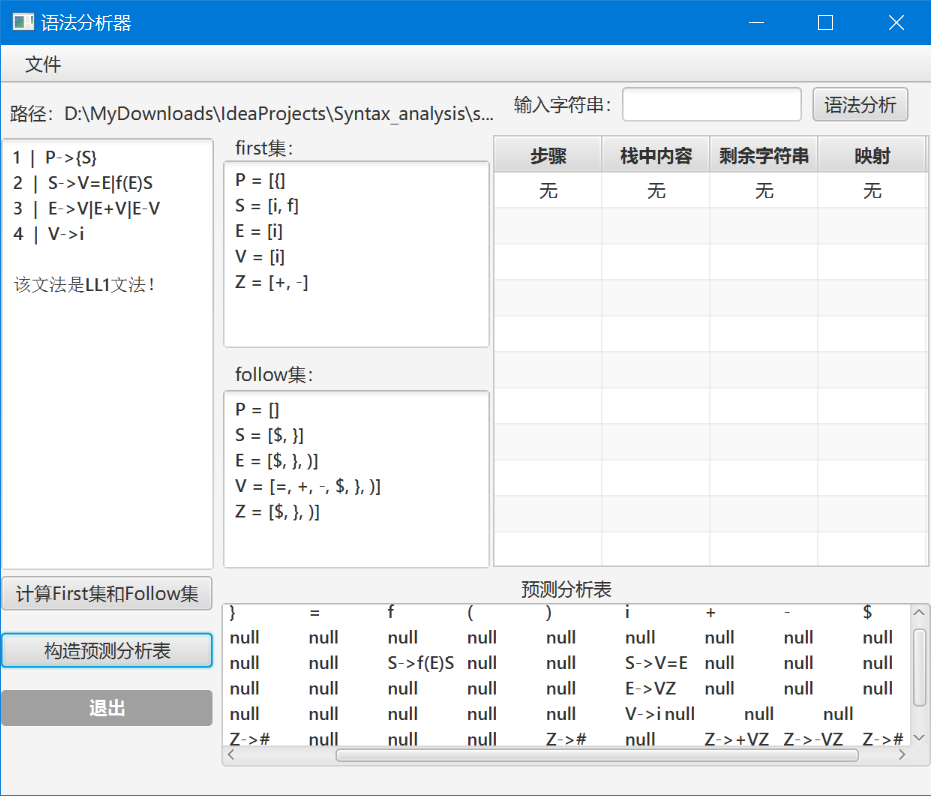


图7 构造预测分析表展示图2

输入字符串f(i)：



图8 输入字符串展示图1

输入字符串f(i)i=i：



图9 输入字符串展示图2



图10 输入字符串展示图3

# 4 小结

本次实验实现了LL1语法分析器，先从原理上深入理解了LL(1)分析方法消除左递归和避免回溯的思想，再利用Map类、ArrayList类等数据结构存放语法分析各个阶段结果，然后实现从文本中读取文件、计算First集和Follow集、构造预测分析表、判断LL(1)文法、对字符串进行分析的算法，最后用javaFX技术显示界面。通过实验，我对LL(1)分析方法的原理理解得更加透彻，也对语法分析器的分析过程有了进一步认识，还增强了实际的编程能力，收获颇丰。以后要更进一步理解编译过程的其他阶段，对编译器的整体工作原理有更深的理解。

# 参考文献

[1]宋晏,杨国兴,胡倩茹,等.Java程序设计及应用开发[M].机械工业出版社:北京市,2016:1-2.

[2]陈火旺,刘春林,谭庆平.程序设计语言编译原理[M].国防工业出版社:北京市,2020:2-81.