朱宇翔

141250216

编译原理实验报告

LAB2：基于LR的语法分析器

目录

[1 目标 2](#_Toc466887231)

[2 内容概述 2](#_Toc466887232)

[3 假设与依赖 2](#_Toc466887233)

[3.1 实验环境 2](#_Toc466887234)

[3.2 文法定义 2](#_Toc466887235)

[3.3 其他假设 3](#_Toc466887236)

[4 思路与方法 4](#_Toc466887237)

[5 主要数据结构描述 4](#_Toc466887238)

[5.1 输入队列，符号栈，状态栈 4](#_Toc466887239)

[5.2 分析表 5](#_Toc466887240)

[6 核心算法描述 5](#_Toc466887241)

[7 测试用例 6](#_Toc466887242)

[7.1 正常情况 6](#_Toc466887243)

[7.1.1 测试输入 6](#_Toc466887244)

[7.1.2 测试输出 6](#_Toc466887245)

[7.2错误情况 6](#_Toc466887246)

[7.2.1 测试输入 7](#_Toc466887247)

[i+(i+i)\* 7](#_Toc466887248)

[7.2.2测试输出 7](#_Toc466887249)

[8 困难与解决方案 7](#_Toc466887250)

[9 心得与感受 7](#_Toc466887251)

# 1 目标

本次实验较为简单，目的是模拟编译器的词法分析过程。我选择了较为简单有效的自底向上的词法分析器的分析过程。程序接受任何LR, SLR, LALR的文法作为输入，进行词法分析。

# 2 内容概述

本报告描述了构造语法分析器的过程，着重说明了实验模型的数据结构，核心算法及思路。并且还包括了最终产品的概要，配置文件的格式定义和个人的心得和体会。

# 3 假设与依赖

## 3.1 实验环境

|  |  |
| --- | --- |
| 操作系统 | Windows 10 |
| 编程语言 | Java |
| JDK版本 | Java 1.8.0 |
| IDE | Eclipse Neon |

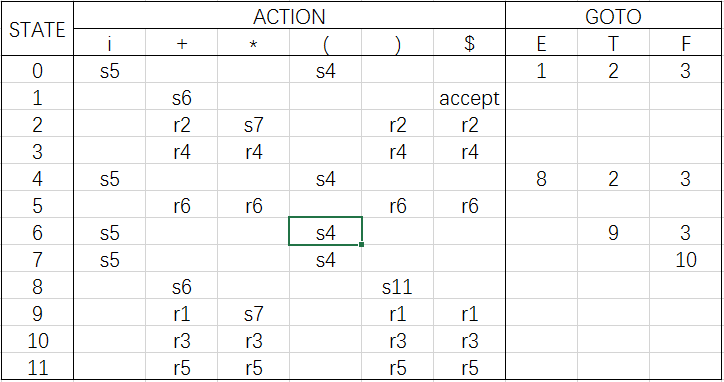
## 3.2 文法定义

由于本次实验选择什么文法并不重要，事实上程序被设计为可复用的，可以方便地将输入的文法定义在外部文件当中。因此我们选择了一个简单的LR语法用于分析：

*E=E+T E=T T=T\*F*

*T=F F=(E) F=i*

然后构建分析表，结果如下：



## 3.3 其他假设

1. 假设用户输入的文法和parsing table是**相匹配**的，程序不会再去验证它们的匹配性。

2. 假设用户输入的parsing table是无冲突项的，即该文法是符合LR或SLR或LALR的且无二义的。程序不会去验证parsing table有无冲突项。

3. 所有终结符用**单个小写字母**或**特殊符号**如+, \*（但不包括等号）等表示，所有非终结符用**单个大写字母**表示。推导符号使用=号表示。Parsing table中接受用accept表示

4. 用户的输入符合格式。格式如下：

在input.avaj中，定义了输入字符串：在第七节的测试用例中使用的实例测试字符串为i+(i+i)\*i+i

在Grammar.txt中，定义了文法。用户可以随意更改文法，但必须保证Parsing Table.txt中描述的分析表必须与文法一致。格式见文件示例。注意我们用等号代表推出符。

在Parsing Table.txt中定义了分析表。文件中的每一行都代表着分析表的一个表项，比如对于示例分析表的第一行：

*0 i s5*

第一个字符代表初始状态，即state。中间的字符代表ACTION或GOTO的表头，即任意一个终结符（包括$）或非终结符。第三个字符串代表表项。Accept代表接受，s后带上一个数字代表移入操作，r后带上一个数字代表规约操作，只有一个数字代表GOTO操作。

# 4 思路与方法

首先可以比较明确的一点是，我们希望程序有较高的可复用性。因此我决定把所有与语法相关的都定义在外部文件中。因此对于文法，我们从外部文件Grammar.txt中读取并分析。然而，从文法求分析表的难度较高，因此我们令用户自己求出分析表然后通过Parsing Table.txt传入程序。这样，程序就完全独立于文法和输入，用户可以很方便地通过修改外部文件来更改或测试文法。

然后，我们在程序中构造符号栈，状态栈和输入队列三个数据结构。首先根据输入的文法和分析表初始化分析程序，将这些数据存储到程序中并初始化这些原始数据。然后在将输入文件中的字符写入**输入队列**中。随后将输入队列中的第一个字符peek查看，根据**符号栈栈顶存储的当前状态**和**输入队列中的字符**查分析表得出的**表项**决定分析程序的行为：

1.表项开头为s，代表移入操作，那么我们将s后跟着的数字（代表移入的状态）压入状态栈，再将输入字符出列并压入符号栈。

2.表项开头为r，代表规约操作，那么我们根据用户提供的文法寻找规约的表达式，然后将符号栈顶符合此表达式的前n个字符全部出栈(n为规约字符串的长度)，然后将状态栈顶出栈并抛弃n个状态。然后在符号栈中压入规约表达式的右侧非终结符，在状态栈中压入GOTO的状态。

3.表项为accept，那么代表程序结束，打印结果，结束循环

4.找不到对应的表项，那么代表输入程序不符合语法。于是打印错误信息，程序结束。

# 5 主要数据结构描述

## 5.1 输入队列，符号栈，状态栈

输入队列使用Java的Java.util.Queue接口，以及Java自带的Java.util.concurrent.LinkedBlockingQueue实现。每一个输入字符使用Character。

符号栈，状态栈均使用Java原生的Java.util.Stack。 状态栈中只维护状态的编号（使用Integer）。符号栈中使用Character

## 5.2 分析表

分析表使用Java.util.HashMap<Integer, HashMap<Character, String>>存储。其中Integer为用于存储符号的整数。Character用于存储终结符和非终结符的表头，String则为表项，比如s5,accept等。不存储空白表项。

因此parsingTable中每个状态(Integer)都对应许多可以连接的下一个字符(Character),其中每个字符映射了一个表项。

# 6 核心算法描述

这次实验中，我首先读入三个输入文件：Parsing Table.txt， input.avaj， Grammar.txt，这些文件是用户根据格式定义在外界的，描述文法和输入字符串的文件。然后通过这些输入经过初始化程序（HandleRawData.java）得到格式化后的分析表和语法表。

初始化完毕后，在主函数中调用Analyzer.deal方法开始处理输入字符串。具体的处理是使用一个无限循环来让子函数一个个peek读取输入队列。算法如下：

**public** **void** deal(ArrayList<String> input) {

String firstLine = input.get(0) + '$';

**for**(**char** c: firstLine.toCharArray()){

inputQueue.add(c);

}

**while**(notEnd){

dealOneChar();

}

}

然后在逐字符读取输入队列的过程中，利用本文第四节中的方法不断进行分析表查找，然后进行判断，处理。

# 7 测试用例

本实验用了两个测试用例，第一个是正常程序的测试用例，第二个是包含错误代码的测试用例。

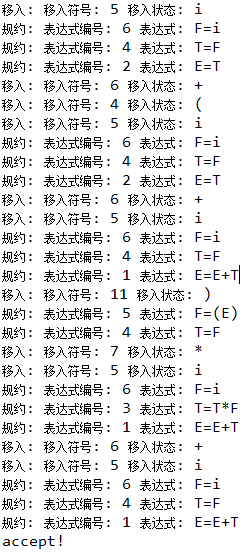
## 7.1 正常情况

### 7.1.1 测试输入

在这个输入中，笔者使用了文法的全部表达式构建了测试用例，也使状态栈遍历了所有的状态，因此具有完备性。

测试输入为：i+(i+i)\*i+i

### 7.1.2 测试输出



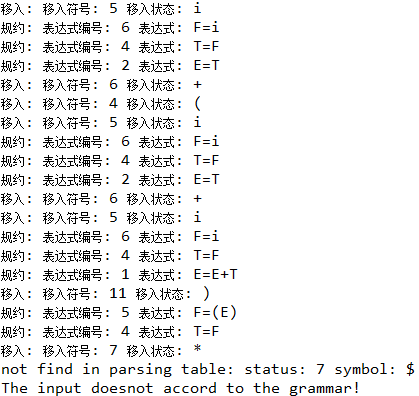
## 7.2错误情况

### 7.2.1 测试输入

在这个输入中，笔者使用了不符合文法的字符串：

### i+(i+i)\*

### 7.2.2测试输出



# 8 困难与解决方案

总体来说，此实验比实验一简单的多。本次实验中最难的部分还是在于通过文法求LR分析表这一步。如何仔细考虑而不出错，需要十分地细心。另外，通过已有的分析表如何结合分析栈，状态栈和输入字符流去构造分析过程，也是值得细细考虑的地方

# 9 心得与感受

这次实验笔者编写了一个高度可复用的程序，将输入和配置完全独立于程序外，使程序的逻辑更为抽象。笔者感受到这次实验虽然简单，但还是比较有锻炼意义的，这种将配置完全独立的思想被广泛运用在开源程序中。这次实验算是一个小小的锻炼吧。