朱宇翔

141250216

编译原理实验报告

LAB1：基于有限自动机的词法分析器

# 1 目标

本次实验中，我定义了一个简单的类似java语言的avaj语言。并且通过1.定义正则表达式2.通过正则表达式生成最小化的DFA 3.根据此DFA编程。最后生成一个avaj词法分析器。

# 2 内容概述

本报告描述了构造avaj词法分析器的过程。本报告着重说明了实验模型的数据结构，核心算法及理论推导。并且还包括了最终产品的概要，一些遇到的困难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 avaj语言定义

### 3.2.1 保留字

由于对保留字的实现十分简单，保留字的多少并不会大程度影响程序的复杂度，因此本实验仅挑选少量保留字进行实现。

程序分支控制，循环：while, if, else, break

数据，函数的类型声明：int, void

函数返回：return

输入输出，导入库：printf, scanf, import

### 3.2.2 特殊符号

|  |  |
| --- | --- |
| 符号类别 | 符号表达 |
| 算术运算符 | + - \* / = |
| 比较运算符 | < <= > >= == != |
| 分隔符号 | ; , |
| 空白符号 | Space \n \t |
| 括号 | () {} [] |

### 3.2.3 其他说明

|  |  |
| --- | --- |
| 标识符 | 由字母开头，以字母和数字组成的字符串，且不与保留字重复，长度至少为1的字符串 |
| 数据类型 | 只支持正整数，相当于C里的unsigned int。考虑到浮点数和负数处理方法大致相同。因此此处省略了机械性重复的工作，将复杂的工作安排到注释方面。 |
| 注释 | 支持两种情形的注释：/\*这是注释\*/ 以及 //这是注释 |

# 4 思路与方法

首先，依据需要解析的字符的复杂度，将上文中定义的特殊符号类型进行分类，可以分为三类：1.**不成为其他特殊字符前缀或不以其他特殊字符为前缀**的特殊字符。当词法分析器检测到这些字符后，可以直接判断就是这些字符，十分简单，如 [,{ 。2.长度有限的特殊字符，但不包括1中的，比较复杂，比如 >= 3.由长度不定的字符组成的，如标识符，注释。然后按照复杂度不同进行处理

对于类型1，我们可以为其定义枚举类型为：

|  |  |  |  |
| --- | --- | --- | --- |
| 符号 | Token type | 符号 | Token type |
| + | PLUS | { | LB |
| - | MINUS | } | RB |
| \* | MULT | [ | LM |
| ( | LS | ] | RM |
| ) | RS | != | NEQ |
| , | COMA | ; | SEMI |
| " | QUOTE |  |  |

对于类型2，我们可以为其定义枚举类型为：

|  |  |  |  |
| --- | --- | --- | --- |
| 符号 | Token type | 符号 | Token type |
| < | ST | <= | STE |
| > | LT | >= | LTE |
| = | ASSIGN | == | EQUAL |
| / | DIV |  |  |

对于类型3，我们可以为其定义枚举类型为：

|  |  |  |  |
| --- | --- | --- | --- |
| 符号 | Token type | 符号 | Token type |
| 数字 | NUM | 标识符 | ID |
| 注释 | COMMENT |  |  |

其次，我们定义一系列的状态变量用于控制自动机的转化的中间状态。程序从源文件中一次读取一行进行解析，忽略大块的空白，忽略多余的换行，对每一个字符用状态机进行匹配，返回识别的token，并报出错误提示。注意，注释可能能跨多行。

# 5 相关有限自动机描述

## 5.1 第一类有限自动机

对于此类特殊符号，其有限自动机表示非常简单，仅仅进行大略说明



对于此类特殊符号，扫描到就可以直接返回相应的token，并将状态机状态重置为start继续扫描下一个符号。

## 5.2 第二类有限自动机

此类特殊符号存在与其他符号的前后缀关系，因此不能直接返回，要确定是同一族符号中的具体哪个。

比如，对于<和<=的区分，应当使用DFA：



这样对于输入的<=或是<，都能将其识别为对应的适当状态。

将所有第二类有限自动机合并，得到：



## 5.3 第三类有限自动机

第三类有限自动机包含注释，标识符和数字，他们的长度不定，因此提高了复杂度。

### 5.3.1数字

数字的正则表达式为digit digit\*，其中，digit为0|1|2|3|4|5|6|7|8|9.

转化为DFA为



### 5.3.2 标识符和保留字

字母的正则表达式为,char (char|digit)\*，其中char为[a-zA-Z]

转化为DFA为



要注意，应该将识别后的字符串与保留字相比较，因为标识符不能和保留字重复。

### 5.3.3 注释

注释的正则表达式为

行级注释：//[^\n]\*

建立块级注释十分复杂，因此不如直接建立块级的DFA:



然后，将行级注释的DFA与之合并，就得到了注释的完整DFA：



### 5.4 总体有限自动机

将三类状态机合并起来，就得到了总体的有限自动机。（由于第二类自动机过于复杂，因此我们将第二类自动机部分缩略了大部分）。



事实上，以上所有自动机都忽略了错误的节点，比如，识别出 ^等没有定义的符号，就无法匹配任何自动机，此时自动机就会转入错误状态并且报错。此时就打印错误信息并继续扫描。此时的自动机应如下：



# 6 重要数据结构描述