编译原理实验报告

LAB1：基于有限自动机的词法分析器

朱宇翔

141250216

目录

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：

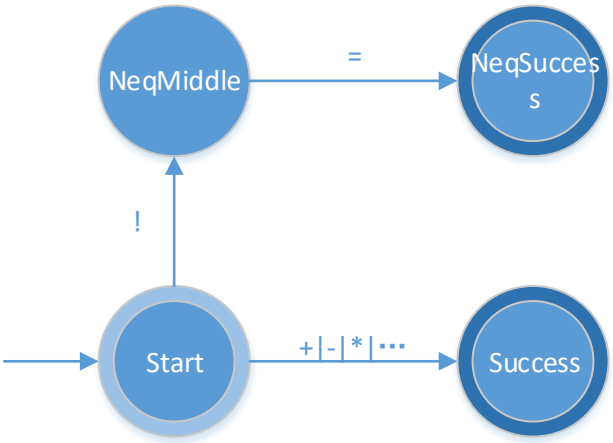
符号	Token type	符号	Token type
数字	NUM	标识符	ID
注释	COMMENT		

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。然而，符号!=的处理比较特殊，需要单独处理。

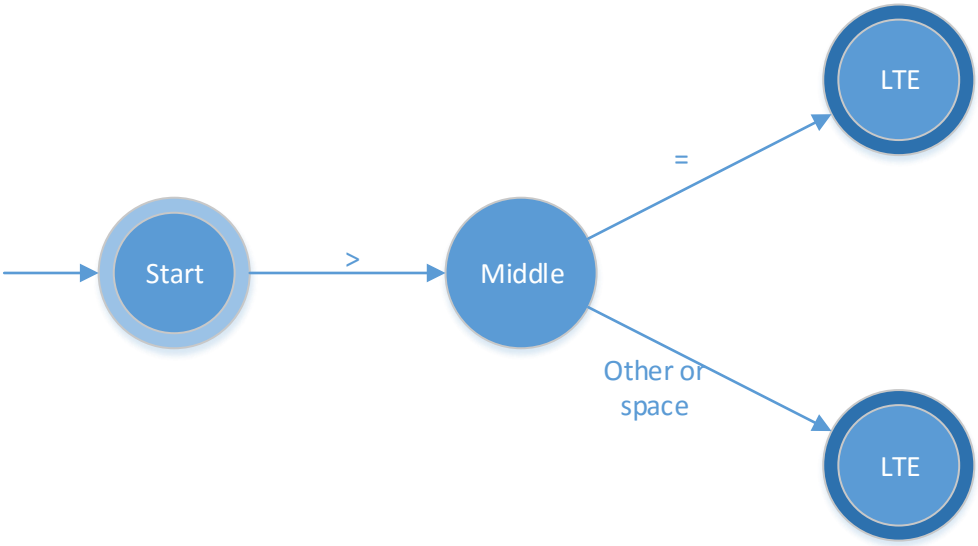


对于此类特殊符号，扫描到就可以直接返回相应的 token，并将状态机状态重置为 start 继续扫描下一个符号。

5.2 第二类有限自动机

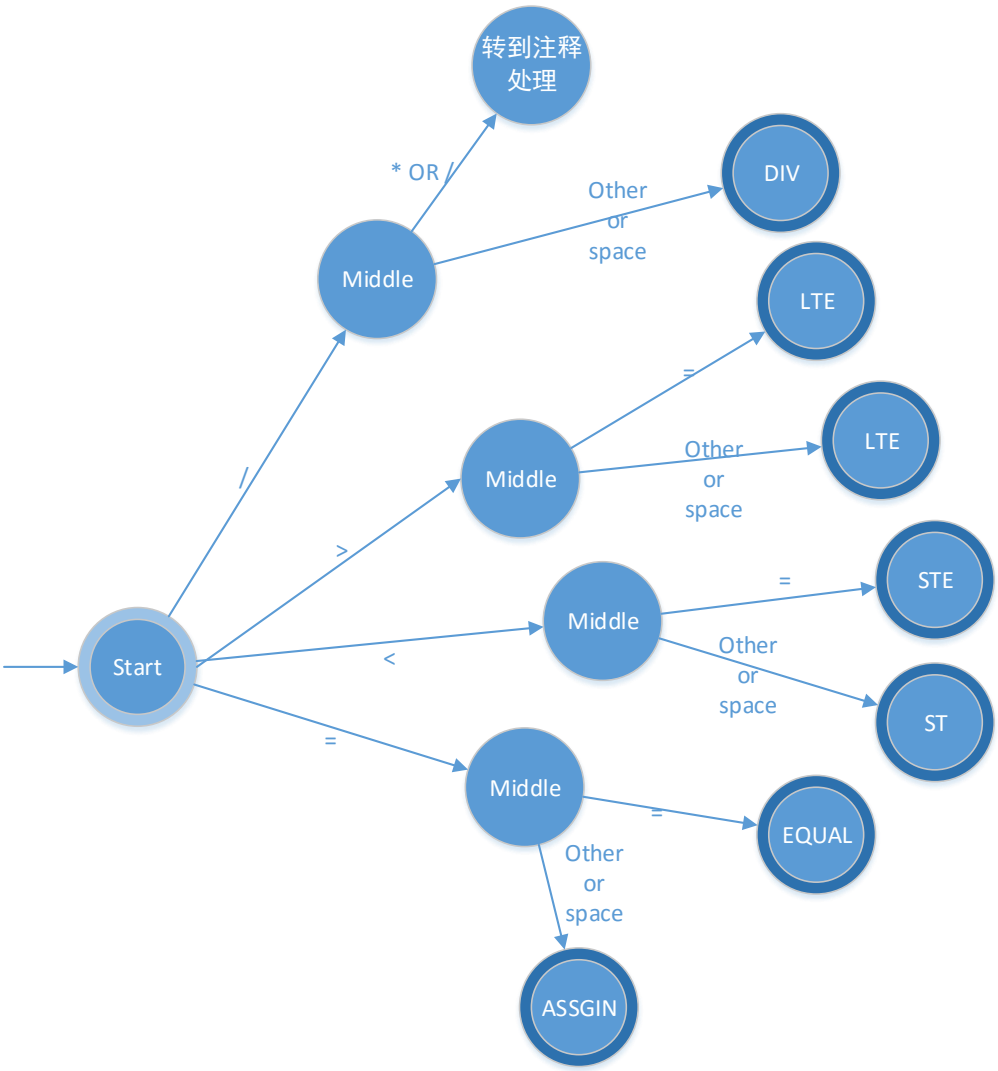
此类特殊符号存在与其他符号的前后缀关系，因此不能直接返回，要确定是同一族符号中的具体哪个。

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：



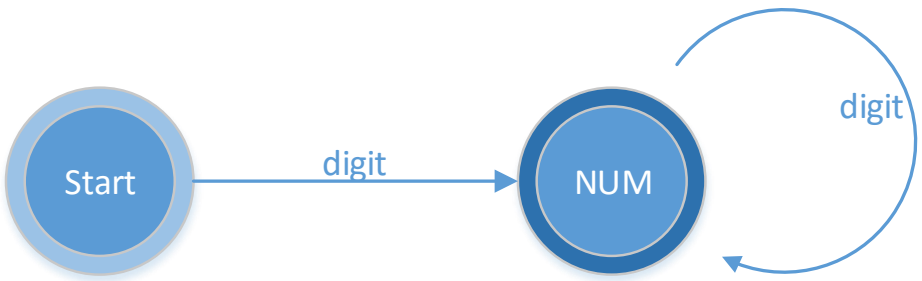
5.3 第三类有限自动机

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.

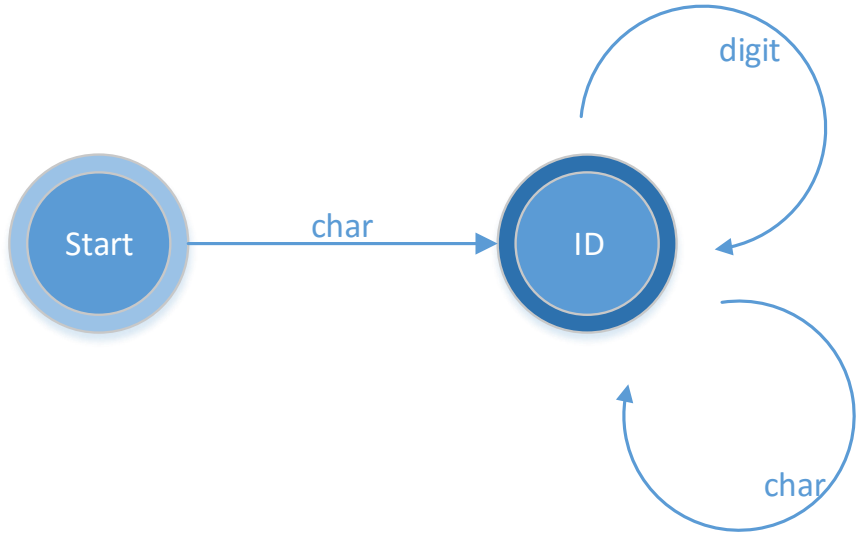
转化为 DFA 为



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



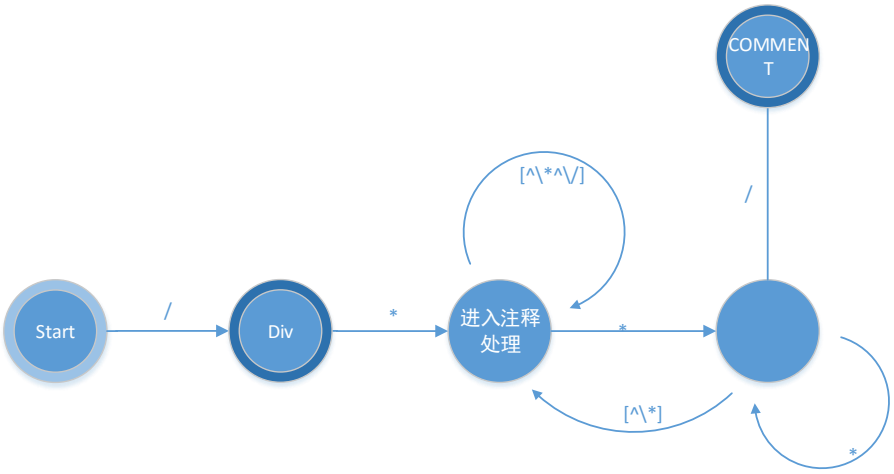
要注意，应该将识别后的字符串与保留字相比较，因为标识符不能和保留字重复。

5.3.3 注释

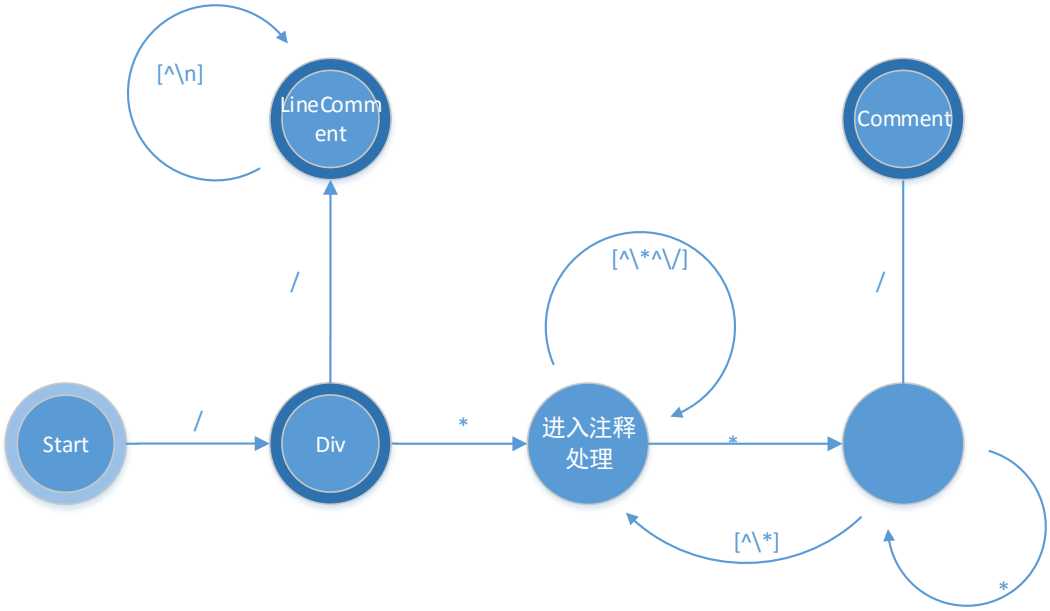
注释的正则表达式为

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。（由于第二类自动机的三个基本符号<,>,=都用共同点：后面加等号能变为两个长度的第二类符号，因此把他们合并为 Middle 和 MiddleSuccess 两个状态）。

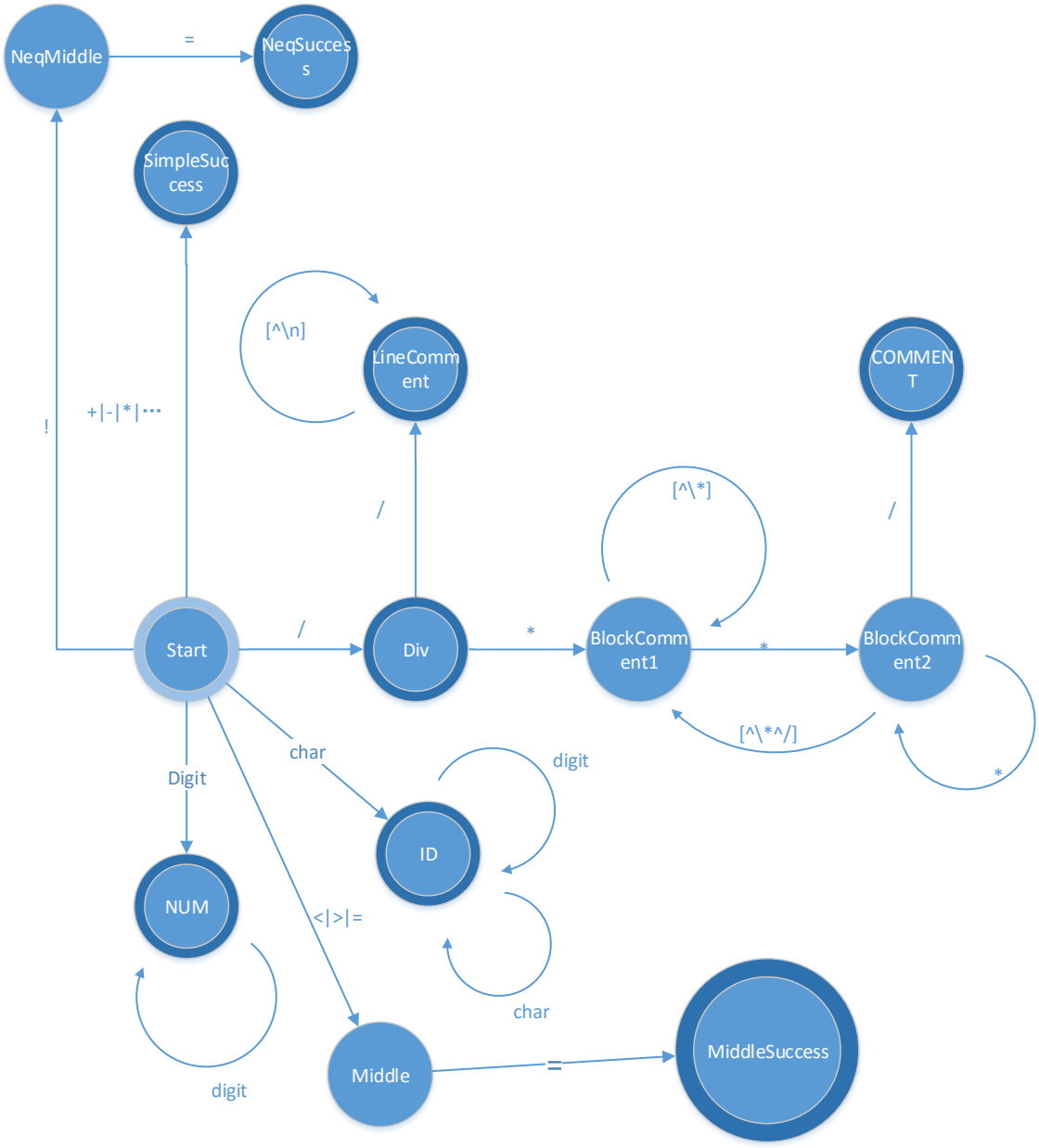
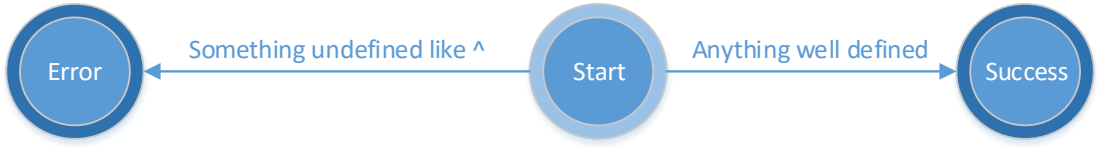


Figure Alpha

事实上，以上所有自动机都忽略了错误的节点，比如，识别出 ˆ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 核心算法描述

这个实验使用的方法是老师给的方案 1（自己手动求出 DFA 然后编程）和方案 2（编写 LEX，然后向 lex 输入 RE）的综合。我的方案是，自己通过自己定义的 RE 求出一个整体的 DFA，然后，编写一个**通用自动机生成器**，这个自动机生成器以**自定义的自动机描述**为输入，输出我自定义的自动机，然后我再将字符流输入这个**自动生成的自动机**，使这个自动机在一系列**定义在外界的状态**之间进行转换，获得 token 序列。

具体而言，我首先求出 avaj 语言的自动机，也就是上一章节的倒数第二幅图 Figure Alpha。然后，将 Figure Alpha 转化为文字描述语言，保存在文件夹根目录的 Description of Automata.txt 中，内容如下：

```
Start \s Start
Start ! NeqMiddle
Start simpleSign SimpleSuccess
Start digit Num
Start complexSign Middle
Start char ID
Start / Div
NeqMiddle = NeqSuccess
Num digit Num
ID digit ID
ID char ID
Div / LineComment
LineComment [^\n] LineComment
Div \* BlockComment1
BlockComment1 [^\*] BlockComment1
BlockComment1 \* BlockComment2
BlockComment2 \* BlockComment2
BlockComment2 [^\*^/] BlockComment1
BlockComment2 / Comment
Middle = MiddleSuccess
final Num SimpleSuccess Div ID Comment Middle MiddleSuccess NeqSuccess LineComment
```

可以看到，描述的每一行（除去最后一行）都是自动机的一条边，第一个字符串是边的开始状态，第二个字符串是转换条件的正则表达式（digit,char 等比较复杂的表达式我定义在程序中，作为默认的表达式），第三个字符串是边的结束状态。

最后一行以特殊字符串 final 开头，定义了结束状态（成功状态）。另外，开始状态默认为特殊状态 Start。

显然地，**开始状态，结束状态和边的集合唯一确定了一个自动机**。因此，系统在初始化自动机时（这一块代码位于 Analyzer.java 中），读取这个文件，生成边的集合 transformList，再读取最后一行，生成结束状态的集合 endStatusList，并将自动机的初始状态设为默认的“Start”。然后，将输入流逐字符地传入自动机（在 Automata.java 中实现），如果在 transformList 中能

找到**开始状态满足自动机当前状态，输入字符满足边上的正则表达式**的边，就认为这条边是正确的转换边，就将自动机的状态变为这条边的结束状态。

如果找不到满足条件的边，分为两种情况。情况 1：如果当前状态是结束状态，那么说明这一串输入的字符已经最大匹配了一个正则表达式，就认为匹配成功，根据处理的这一段字符和自动机的结束状态生成对应的 token，然后将自动机状态设回开始状态，准备处理下一个字符。情况 2：如果当前状态不是结束状态，说明系统处理了一个异常字符，就启动错误处理方案，具体错误处理方案这里略过，详见代码。

因此，自动生成的自动机不断地根据输入的字符和边的集合被动地改变自身的状态，达到触发条件时就输出相应的 token 并回到开始状态，并且对于异常情况，我们也定义了错误处理方案。

7 重要数据结构描述

在边的表示上，我们定义了 Transformation 类。使用三个成员变量 preStatus, edge, nextStatus 来表示这条边。然后使用 ArrayList<Transformation>来存储边的集合。

实验还存储了结束状态的集合：ArrayList<String> endStatusList，从文件中读取。

此外，程序还定义了一些常量，比如 remainedWordList 保留字列表，用于存储所有保留字，以及特殊符号与 token 名称对应的 Map，还有一些默认的正则表达式。（事实上，更好地方法是把这些常量和保留字都定义在外界文件中，能获得更大的可复用性。但是这些方法对于个人的实验学习没有太大的帮助。因此程序为了减少复杂度，减少工作量而采取了折中的方法）

程序定义的默认的正则表达式如下：

```
public static final HashMap<String,String> reExplainer = new HashMap<>();
static{
    reExplainer.put("simpleSign", "\\+|-|\\*|\\(|\\)|,|\\{|\\}|\\[|\\];|\"");
    reExplainer.put("digit", "[0-9]");
    reExplainer.put("complexSign", "=|==|>|>=|<|<=");
    reExplainer.put("char", "[a-zA-Z]");
}
```

8 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。

8.1 正常情况

8.1.1 测试输入

在这个输入中，笔者使用了全部的语法符号和保留字，也使自动机遍历了所有的状态，因此具有完备性。

```
/* au6*/

import calc;

void main(){
        scanf("something");// get an input <
        printf(calc());

        int list[6];
        if(list[0] < 3){
                if(list[0] >= 4){
                        //do nothing != doing nothing
                }
        }
}

/* hah
test !
int i = 6;
*/

int calc(){
        int i = 2, j;
        j = i + 4 * 3;

        //make a judgment
        if( j <= 2016){
                printf("Little than 2016");
        }else if( j == 6){
                j = i/2;
                if(i == 0)
                        break;
                else if (i != 2016)
                        i = i - 1;
        }|
        while(j > 0){
                j = j - 1;
        }
        return i;
}
```

8.1.2 测试输出

```
( keyword_import , "import" )
( ID , "calc" )
( SEMI , ";" )
( keyword_void , "void" )
( ID , "main" )
( LS , "(" )
( RS , ")" )
( LB , "{" )
( keyword_scanf , "scanf" )
( LS , "(" )
( QUOTE , """ )
( ID , "something" )
( QUOTE , """ )
( RS , ")" )
( SEMI , ";" )
( keyword_printf , "printf" )
( LS , "(" )
( ID , "calc" )
( LS , "(" )
( RS , ")" )
( RS , ")" )
( SEMI , ";" )
( keyword_int , "int" )
( ID , "list" )
( LM , "[" )
( Num , "6" )
( RM , "]" )
( SEMI , ";" )
( keyword_if , "if" )
( LS , "(" )
( ID , "list" )
( LM , "[" )
( Num , "0" )
( RM , "]" )
( ST , "<" )
( Num , "3" )
( RS , ")" )
( LB , "{" )
( keyword_if , "if" )
( LS , "(" )
( ID , "list" )
( LM , "[" )
( Num , "0" )
( RM , "]" )
( LTE , ">=" )
( Num , "4" )
( RS , ")" )
( LB , "{" )
( RB , "}" )
( RB , "}" )
( RB , "}" )
( keyword_int , "int" )
( ID , "calc" )
( LS , "(" )
( RS , ")" )
( LB , "{" )
( keyword_int , "int" )
( ID , "i" )
( ASSIGN , "=" )
( Num , "2" )
( COMMA , "," )
( ID , "j" )
( SEMI , ";" )
( ID , "j" )
( ASSIGN , "=" )
( ID , "i" )
( PLUS , "+" )
( Num , "4" )
( MULT , "*" )
( Num , "3" )
( SEMI , ";" )
( keyword_if , "if" )
( LS , "(" )
( ID , "j" )
( STE , "<=" )
( Num , "2016" )
( RS , ")" )
( LB , "{" )
( keyword_printf , "printf
( LS , "(" )
( QUOTE , """ )
( ID , "Little" )
( ID , "than" )
( Num , "2016" )
( QUOTE , """ )
( RS , ")" )
( SEMI , ";" )
( RB , "}" )
( keyword_else , "else" )
( keyword_if , "if" )
( LS , "(" )
( ID , "j" )
( EQUAL , "==" )
( Num , "6" )
( RS , ")" )
( LB , "{" )
( ID , "j" )
( ASSIGN , "=" )
( ID , "i" )
( Div , "/" )
( Num , "2" )
( SEMI , ";" )
( keyword_if , "if" )
( LS , "(" )
( ID , "i" )
( EQUAL , "==" )
( Num , "0" )
( RS , ")" )
( keyword_break , "break" )
( SEMI , ";" )
( keyword_else , "else" )
( keyword_if , "if" )
( LS , "(" )
( ID , "i" )
( Neq , "!=" )
( Num , "2016" )
( RS , ")" )
( ID , "i" )
( ASSIGN , "=" )
( ID , "i" )
( MINUS , "-" )
( Num , "1" )
( SEMI , ";" )
( RB , "}" )
( keyword_while , "while" )
( LS , "(" )
( ID , "j" )
( LT , ">" )
( Num , "0" )
( RS , ")" )
( LB , "{" )
( ID , "j" )
( ASSIGN , "=" )
( ID , "j" )
( MINUS , "-" )
( Num , "1" )
( SEMI , ";" )
( RB , "}" )
( keyword_return , "return" )
( ID , "i" )
( SEMI , ";" )
( RB , "}" )
```

8.2 错误情况

8.2.1 测试输入

在这个输入中，笔者在第 6 行和第 13 行加入了两个不符合语法规范的错误。第六行中只有！出现，但是语法只支持！和=连用代表不等于。第 13 行笔者则加入了莫名其妙的两个冒号。

```
/* au6*/

import calc;

void main(){
      !scanf("something");// get an input <
        printf(calc());

        int list[6];
        if(list[0] < 3){
                if(list[0] >= 4){
                        //do nothing != doing nothing
                ::}
        }
}
```

8.2.2 测试输出

```
( keyword_import , "import" )
( ID , "calc" )
( SEMI , ";" )
( keyword_void , "void" )
( ID , "main" )
( LS , "(" )
( RS , ")" )
( LB , "{" )

Error! Error occurs in Line 6 ,which is
        !scanf("something");// get an input <

( keyword_printf , "printf" )
( LS , "(" )
( ID , "calc" )
( LS , "(" )
( RS , ")" )
( RS , ")" )
( SEMI , ";" )
( keyword_int , "int" )
( ID , "list" )
( LM , "[" )
( Num , "6" )
( RM , "]" )
( SEMI , ";" )
( keyword_if , "if" )
( LS , "(" )
( ID , "list" )
( LM , "[" )
( Num , "0" )
( RM , "]" )
( ST , "<" )
( Num , "3" )
( RS , ")" )
( LB , "{" )
( keyword_if , "if" )
( LS , "(" )
( ID , "list" )
( LM , "[" )
( Num , "0" )
( RM , "]" )
( LTE , ">=" )
( Num , "4" )
( RS , ")" )
( LB , "{" )

Error! Error occurs in Line 13 ,which is
                ::}

( RB , "}" )
( RB , "}" )
```

9 困难与解决方案

在实验中，笔者由于十分缺乏经验，遇到了很多困难。包括大方案的选择，“！=”符号的处理，以及匹配失败后的处理。

9.1 大方案的选择

老师介绍的方案一未免太过具体，最后总是要归结于用许多 switch case 来表达自动机状态的转换，一是十分复杂，自己写太多状态转换的代码容易出错，二是缺乏可变性，若语言成分稍作更改，则要从底至上全部重来。三是十分平庸，缺乏创造性。而方案二又未免太过复杂。

因此我选择将两个方案结合，把方案一最麻烦的部分使用统一的方式进行编写，把方案二比较复杂的部分让自己来推导。最后获得了较满意的方案。

9.2 ！＝的困境

笔者一开始处理!=时，是将!=与其他简单符号比如+ －（一同处理。直到上机跑的时候才发现，单个字符流无法匹配!=。于是笔者修改第一类状态机，将!=的情况单独写作两个状态，成功解决了问题。

9.3 匹配失败后的处理

匹配失败后，如何将自动机归位，如何处理错误都是较难的点，尤其是自动机的输入流需要回退一个字符，是比较难想到的。最后笔者结合回退方案，画出具体流程图成功解决了问题。

10 心得与感受

这次实验笔者编写了一个类似冯诺依曼机的模拟机器，将程序（自动机描述）和输入（字符流）同时输入程序，获得结果。感触非常深刻，体会到编写具备通用性的机器具备更大的价值。也感到编写自己的词法分析器十分有趣。