**编译原理与设计**

**实验报告**

实验名称： Lab 3：词法分析实验

1. **实验目的和内容**

**实验目的：**

（1）熟悉C语言的词法规则，了解编译器词法分析器的主要功能和实现技术，掌握典型词法分析器构造方法，设计并实现C语言词法分析器；

（2）了解Flex工作原理和基本思想，学习使用工具自动生成词法分析器；

（3）掌握编译器从前端到后端各个模块的工作原理，词法分析模块与其他模块之间的交互过程。

**实验内容：**

根据C语言的词法规则，设计识别C语言所有单词类的词法分析器的确定有限状态自动机，并使用Java、C\C++或者Python其中任何一种语言，采用程序中心法或者数据中心法设计并实现词法分析器。词法分析器的输入为C语言源程序，输出为属性字流。可以选择手动编码实现词法分析器，也可以选择使用Flex自动生成词法分析器。需要注意的是，Flex生成的是C为实现语言的词法分析器，如果需要生成Java为实现语言的词法分析器，可以尝试JFlex或者ANTLR。由于框架是基于Java语言实现的，并且提供了相应的示例程序，建议学生使用 Java语言在示例的基础上完成词法分析器。

具体步骤如实验指导文档所示。

1. **实验环境**

设备：RedmiBook 14 锐龙版

操作系统：Windows 10 Pro, 64-bit (Build 19045.2604) 10.0.19045

Java IDE：IntelliJ IDEA 2022.2.2 (Community Edition)

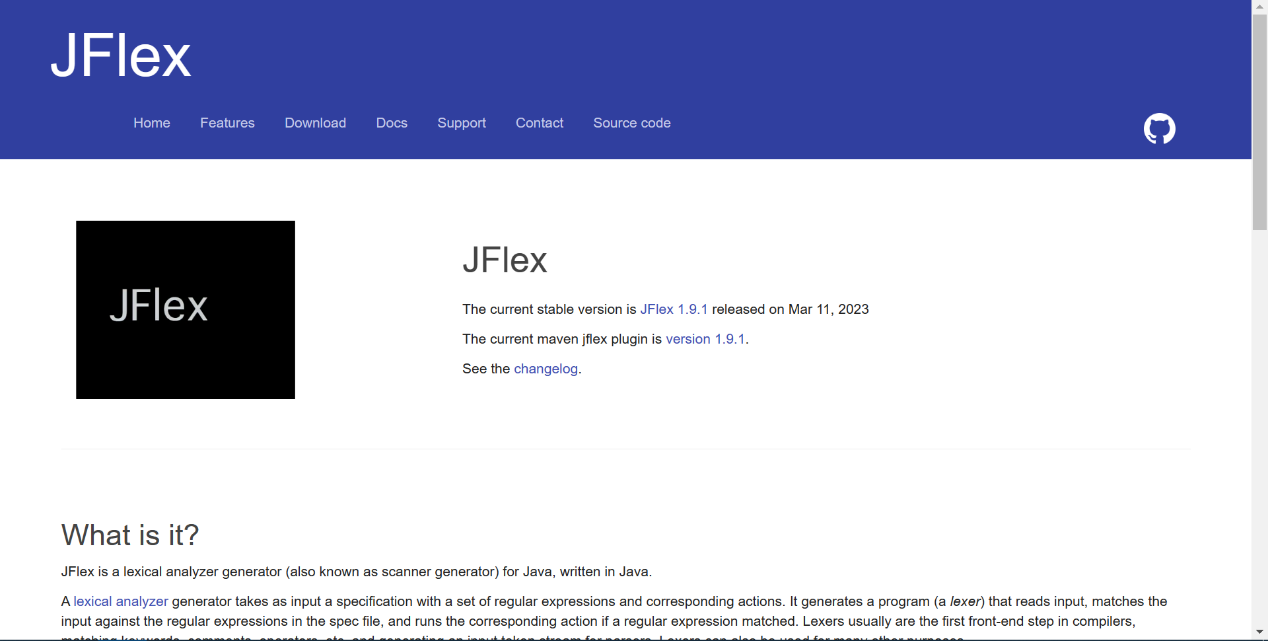
JFlex：jflex-1.9.1

1. **实现的具体过程和步骤**

按照实验文档的指示，在谨慎考虑后，选择以Java语言作为源语言，构建C语言的词法分析器。本次实验均在Windows系统上进行。下面按照实验步骤进行说明：

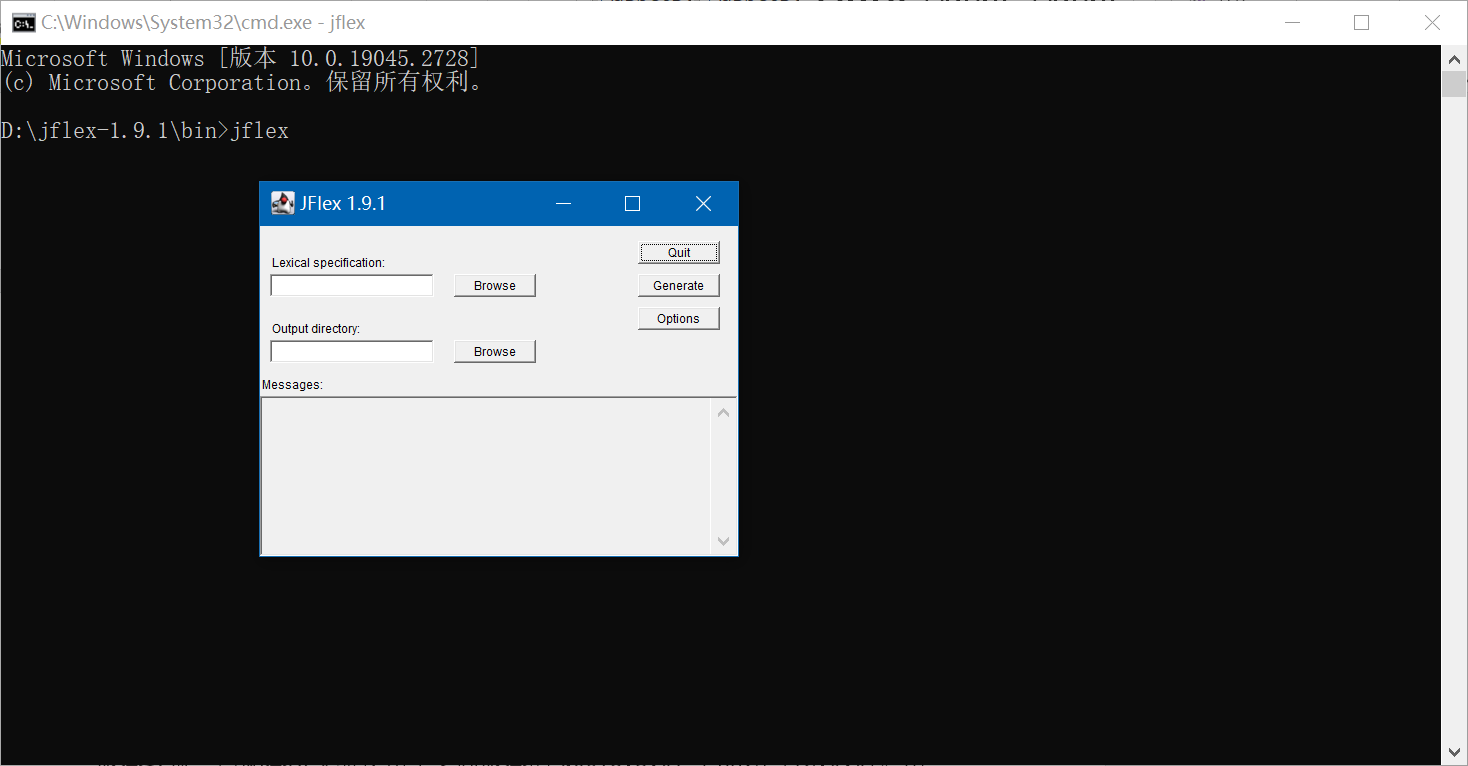
1. **下载并安装JFlex**

JFlex是一款快速的词法分析程序生成器，可以根据flex文档自动生成对应语言的词法分析工具类(class)，实现语言（或称宿主语言）均为Java。用户可通过调用该类的应用编程接口(API)，实现要求的词法分析效果与输出。此外，JFlex还可与Bison结合，使用JavaCUP包，进一步实现语法分析。

选择下载jflex-1.9.1：

可视化运行JFlex：

打开CMD，进入D:\jflex-1.9.1\bin路径后，运行jflex。通过指定源文件\*.flex与输出路径，单击Generate即可生成词法分析类。



1. **编写jflex文档c.flex**

接下来开始编写jflex文档，命名为c.flex。

由教材可知，Lex语言是对表示语言单词集的正规式的描述，以解决正规式规则输入问题。JFlex也使用了稍加修改的Lex语言来编写Lex源程序，该文档以.flex为扩展名，整个文档分为三个部分，使用%%划分：

用户代码段

%%

参数设置和声明段

%%

词法规则段

其中，**第一部分用户代码**，JFlex直接将这部分代码拷贝到生成词法分析器Java源文件中，通常只定义一些类注释信息以及package和import的引用。

在c.flex中，引入了自定义的MySymbol类，取代javaCUP的Symbol类，因为本次实验只需完成词法分析，不必过于繁琐。

import lab3.MySymbol;

在**第二部分参数设置和声明段**中，参数选项用来定制词法分析器，声明则是声明一些能够在第三部分（词法规则定义）使用的宏定义和词法状态，其中宏大多由正则表达式定义。

所有选项都要由一个“%“符号开头，下面逐个解释c.flex中选项含义：

%class Lab3Scanner  
%unicode  
%line  
%column  
%type MySymbol  
%eofval{  
 return symbol(MySymbol.EOF, "<EOF>");  
%eofval}

%class定义生成词法分析器Java文件的文件名，指定为Lab3Scanner。如果不定义该选项，则默认生成“Yylex.java”；

%unicode指定Unicode字符集；

%line %column为行、列计数器，yyline记录当前行数，yycolumn记录当前列数，分别指代已匹配字符串的首个符号的行数与列数；

%type指令用于设置扫描函数的返回类型，现设置为自定义类MySymbol。如果指定的类型不是java.lang.Object的子类那么应该使用%eofval指令或者<EOF>来指定其他文件结束值。

%eofval{用户代码%eofval}，其中用户代码部分直接被复制到扫描函数中，并且在每次文件结束时执行。在c.flex中，返回表示文件结束的MySymbol类。

在词法状态部分中，可以声明扫描器用到的成员变量和函数。可以加入Java代码，并放在 %{ 和 }% 之间。它们将被拷贝到生成的词法类源代码中。在c.flex的词法说明中，声明了两个私有成员函数，用于返回MySymbol实例：

%{  
private MySymbol symbol(int type){  
return new MySymbol(type, yyline, yycolumn, yytext());  
}  
private MySymbol symbol(int type, String value){  
return new MySymbol(type, yyline, yycolumn, value);  
}  
%}

下面进行宏定义的编写，宏定义的规则为：宏标示符＝正则表达式。

按照这种形式定义的宏标识符可以再第三部分引用，右边的正则表达式必须是合式，并且不能包含 ^ , / 或 $ 等运算符。

由C11标准为基准，参考实验指导文档，修改后可得c.flex的宏定义规则。单词大致分为KeyWord、Identifier、Integer、Float、ConstString、String、Operator七类与结束符EOF，以及空白字符WhiteSpace和注释Comment，但这两类识别后不做任何处理。前七类分别表示C语言的关键字、标识符、整型常量、浮点型常量、字符常量、字符串字面量、运算符和界限符。

可通过对BFN范式的化简、代入、转换，得到相应的正则表达式，注意消除递归定义，即不能成环。具体代码如下：

KeyWord = "auto"|"break"|"case"|"char"|"const"  
 |"continue"|"default"|"do"|"double"|"else"  
 |"enum"|"extern"|"float"|"for"|"goto"  
 |"if"|"inline"|"int"|"long"|"register"  
 |"restrict"|"return"|"short"|"signed"|"sizeof"  
 |"static"|"struct"|"switch"|"typedef"|"union"  
 |"unsigned"|"void"|"volatile"|"while"  
  
Identifier = {nodigit} ({nodigit} | {digit})\*  
nodigit = [a-zA-Z\_]  
digit = [0-9]  
  
Integer = {sign}? ({decimal} | {octal} | {hexadecimal}) {integer\_suffix}?  
decimal = {nonzero\_digit} {digit}\*  
octal = 0 {octal\_digit}\*  
hexadecimal = {hexadecimal\_prefix} {hexadecimal\_digit}+  
hexadecimal\_prefix = 0x | 0X  
nonzero\_digit = [1-9]  
octal\_digit = [0-7]  
hexadecimal\_digit = [0-9a-fA-F]  
integer\_suffix = {unsigned\_suffix} ({long\_suffix} | {long\_long\_suffix})?  
 | ({long\_suffix} | {long\_long\_suffix}) {unsigned\_suffix}?  
unsigned\_suffix = u | U  
long\_suffix = l | L  
long\_long\_suffix = ll | LL  
  
Float = {sign}? ({decimal\_float} | {hexadecimal\_float})  
decimal\_float = ({fractional\_constant} | {digit\_sequence}) {exponent\_part}? {floating\_suffix}?  
hexadecimal\_float = {hexadecimal\_prefix} ({hexadecimal\_fractional\_constant} | {hexadecimal\_digit\_sequence})  
 {binary\_exponent\_part}? {floating\_suffix}?  
fractional\_constant = {digit\_sequence} "." {digit\_sequence}?  
exponent\_part = (e | E) {sign}? {digit\_sequence}  
sign = "+" | "-"  
digit\_sequence = {digit}+  
hexadecimal\_fractional\_constant = {hexadecimal\_digit\_sequence} "." {hexadecimal\_digit\_sequence}?  
binary\_exponent\_part = (p | P) {sign}? {digit\_sequence}  
hexadecimal\_digit\_sequence = {hexadecimal\_digit}+  
floating\_suffix = f | l | F | L  
  
ConstString = [LuU]?\'([^\'\\\r\n]|\\.)\*\'  
simple\_escape\_sequence = "\'" | "\"" | "\?" | "\\" | "\a" | "\b" | "\f" | "\n" | "\r" | "\t" | "\v"  
  
String = (u8 | u | U | L)?\"([^\"\\\r\n]|\\.)\*\"  
  
Operator = "["|"]"|"("|")"|"{"|"}"|"."|"->"  
 |"++"|"--"|"&"|"\*"|"+"|"-"|"~"|"!"  
 |"/"|"%"|"<<"|">>"|"<"|">"|"<="|">="  
 |"=="|"!="|"^"|"|"|"&&"|"||"  
 |"?"|":"|";"|"..."  
 |"="|"\*="|"/="|"%="|"+="|"-="|"<<="|">>="  
 |"&="|"^="|"|="|","|"#"|"##"|"<:"|":>"  
 |"<%"|"%>"|"%:"|"%:%:"  
  
Comment = {traditional\_comment} | {end\_of\_line\_comment}  
traditional\_comment = "/\*"([^\*]|\\*+[^\*/])\*\\*+"/"  
end\_of\_line\_comment = "//" [^\r\n]\* {line\_terminator}?  
  
line\_terminator = \r|\n|\r\n  
WhiteSpace = " "|{line\_terminator}|\t|\f

在**第三部分词法规则段**中，包含正则表达式和当扫描器匹配了相关的正则表达式后所要执行的动作。

需注意的是，扫描器会激活具有最大匹配的表达式。如果两个正则表达式完全相同，具有相同的长度，那么扫描器将匹配最先列在说明中的表达式。

每个正则表达式可以附带动作，每当扫描器匹配了正则表达式后就会激活该动作。此处动作即指Java代码片段。可以是进行输出，或是返回扫描器发现的标识符给处理器。

如对于{KeyWord}，将调用成员函数symbol，最终会返回一个带有匹配信息的MySymbol类。

{KeyWord} {return symbol(MySymbol.KeyWord);}

此外，JFlex允许程序员定义特殊的词法状态(lexical states)用作开始条件来细化说明。YYINITIAL是一个预定义的词法状态，是词法分析器初始扫描输入的状态。它是c.flex使用的唯一状态，所有的正则表达式都将从这个词法状态开始识别。但是也可以定义其他状态，以此来简化正则匹配。

<YYINITIAL>  
{do something}

1. **编写自定义MySymbol类**

JavaCUP中提供了Symbol类，用于词法分析器的返回以及后续语法分析器的使用，但这超出了本次实验的范畴。因此编写了一个简易的Symbol类，命名为MySymbol.java，可记录每一个token的匹配信息。

类成员包括：

final int type, yyline, yycolumn;  
final String value;

public String getType();

使用静态常量定义了七类单词与EOF：

public static int *KeyWord* = 1001, *Identifier* = 1002, *Integer* = 1003, *Float* = 1004;  
public static int *ConstString* = 1005, *String* = 1006, *Operator* = 1007, *EOF* = 1008;

1. **编写Main类**

通过Main类来使用词法分析器，读取测试文件并将分析结果输出到屏幕与指定文件中。

运行时需带有两个参数，第一个为输入文件路径，第二个为输出文件路径。

首先初始化scanner：

Lab3Scanner scanner = new Lab3Scanner(reader);  
scanner.yybegin(Lab3Scanner.*YYINITIAL*);

然后循环扫描，匹配字符串，直至文件末尾：

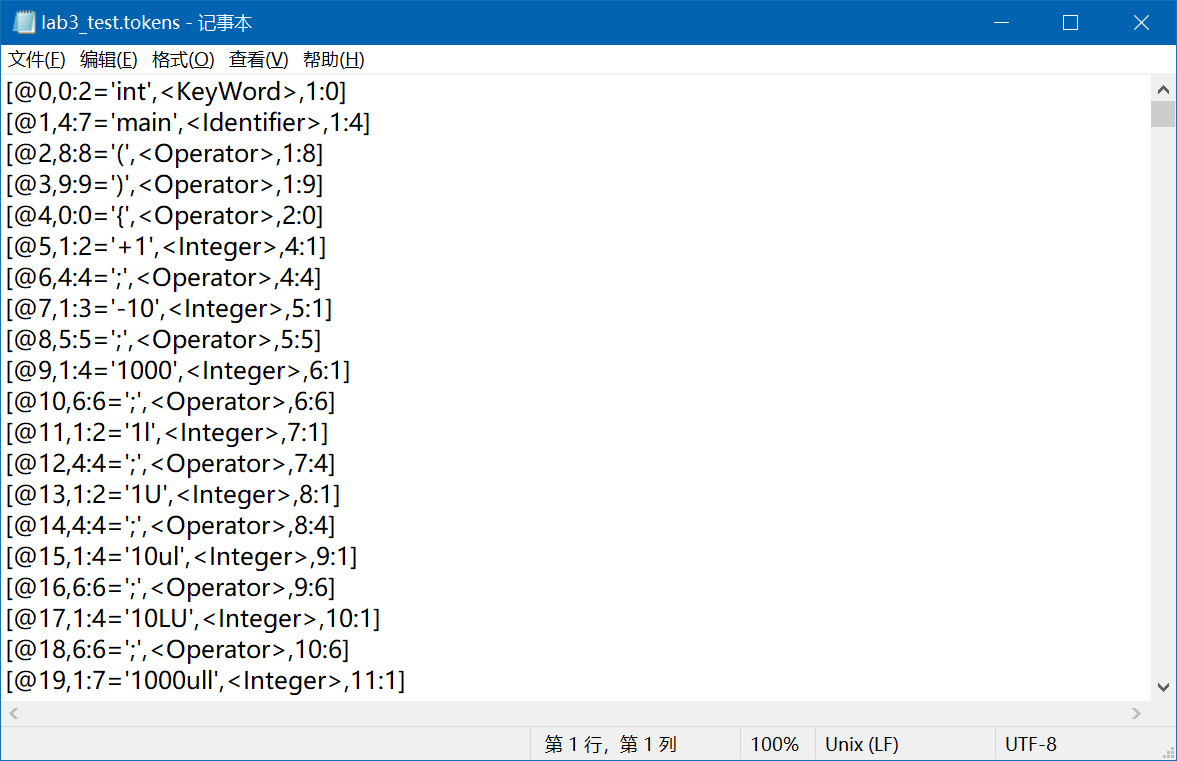
while (!scanner.yyatEOF()) {  
 MySymbol symbol = scanner.yylex();

在循环中计数并输出结果：

String strToken = "";  
  
 strToken += "[@" + num + "," + symbol.yycolumn + ":" + (symbol.yycolumn + scanner.yylength() - 1);  
 strToken += "='" + symbol.value + "',<" + symbol.getType() + ">," + (symbol.yyline + 1) + ":" + symbol.yycolumn + "]\n";  
 System.*out*.print(strToken);  
 writer.write(strToken);  
 num++;  
}

1. **生成词法分析器**

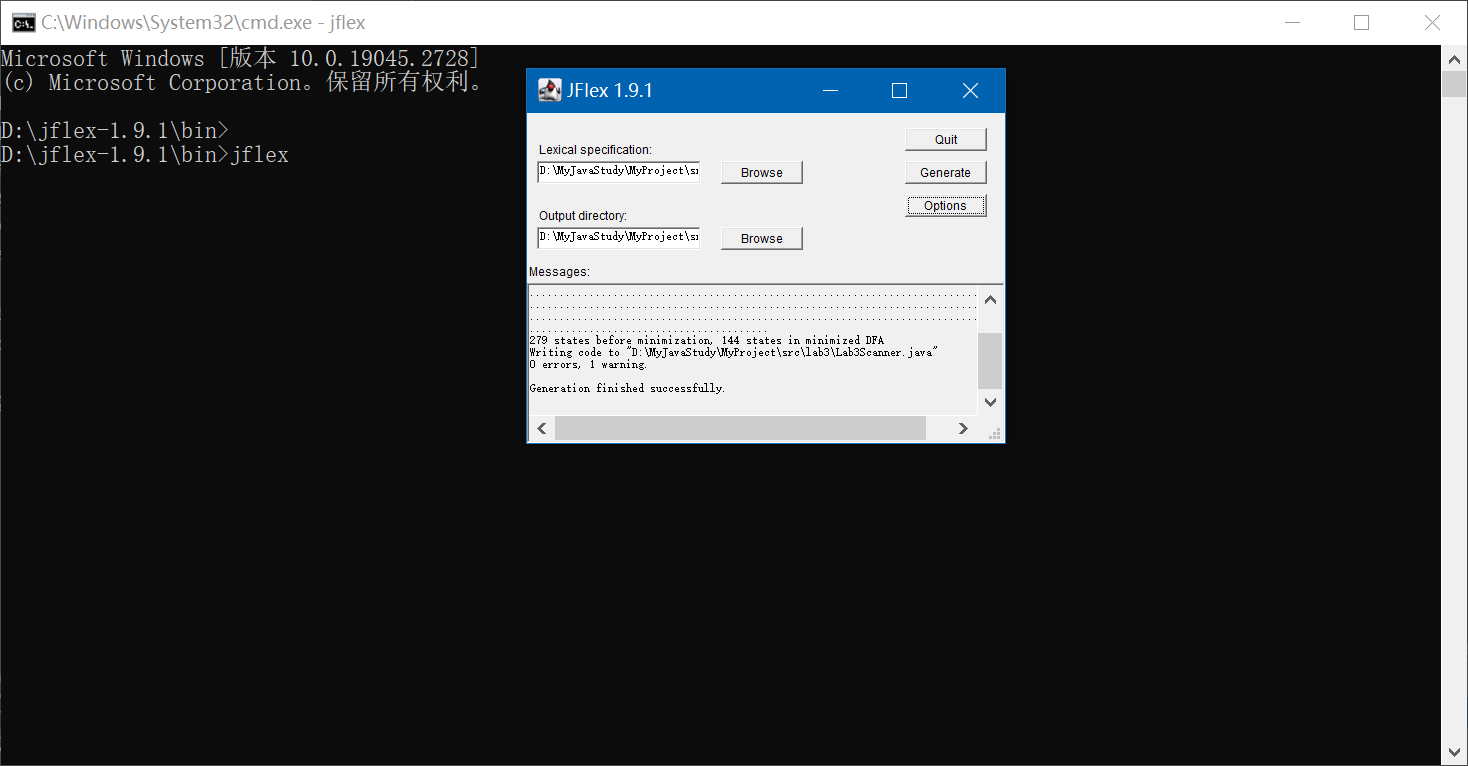
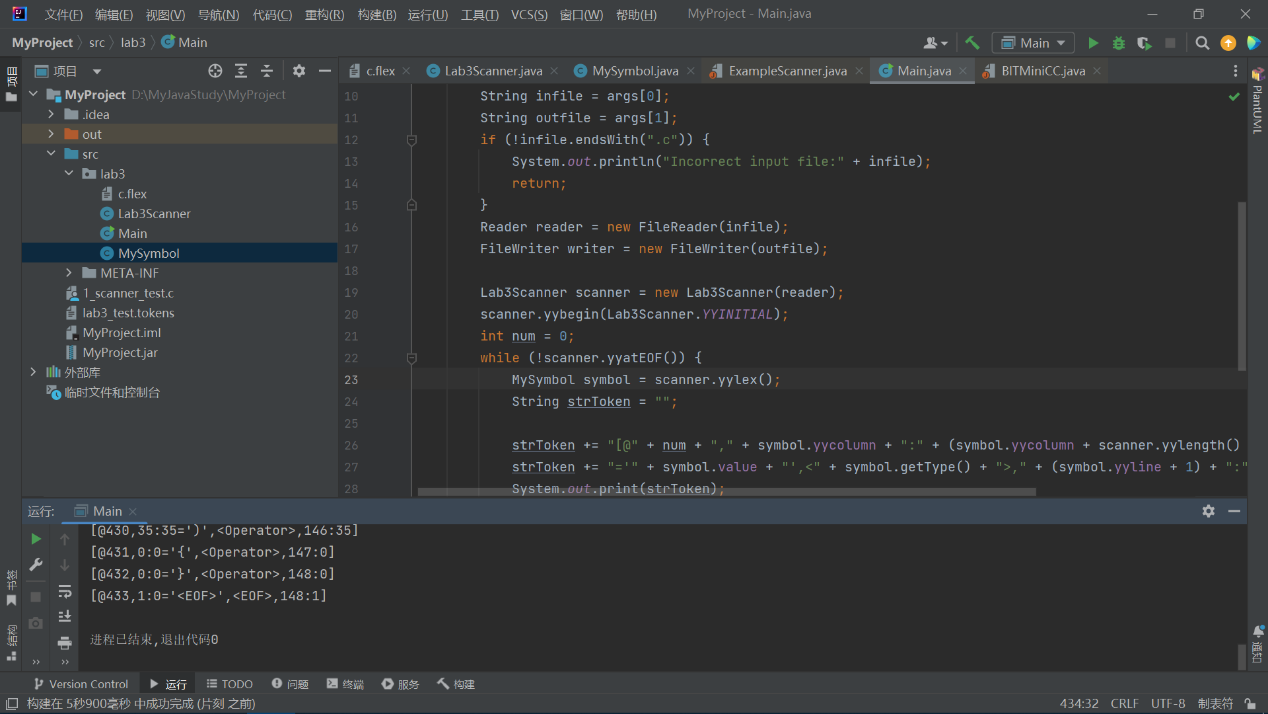
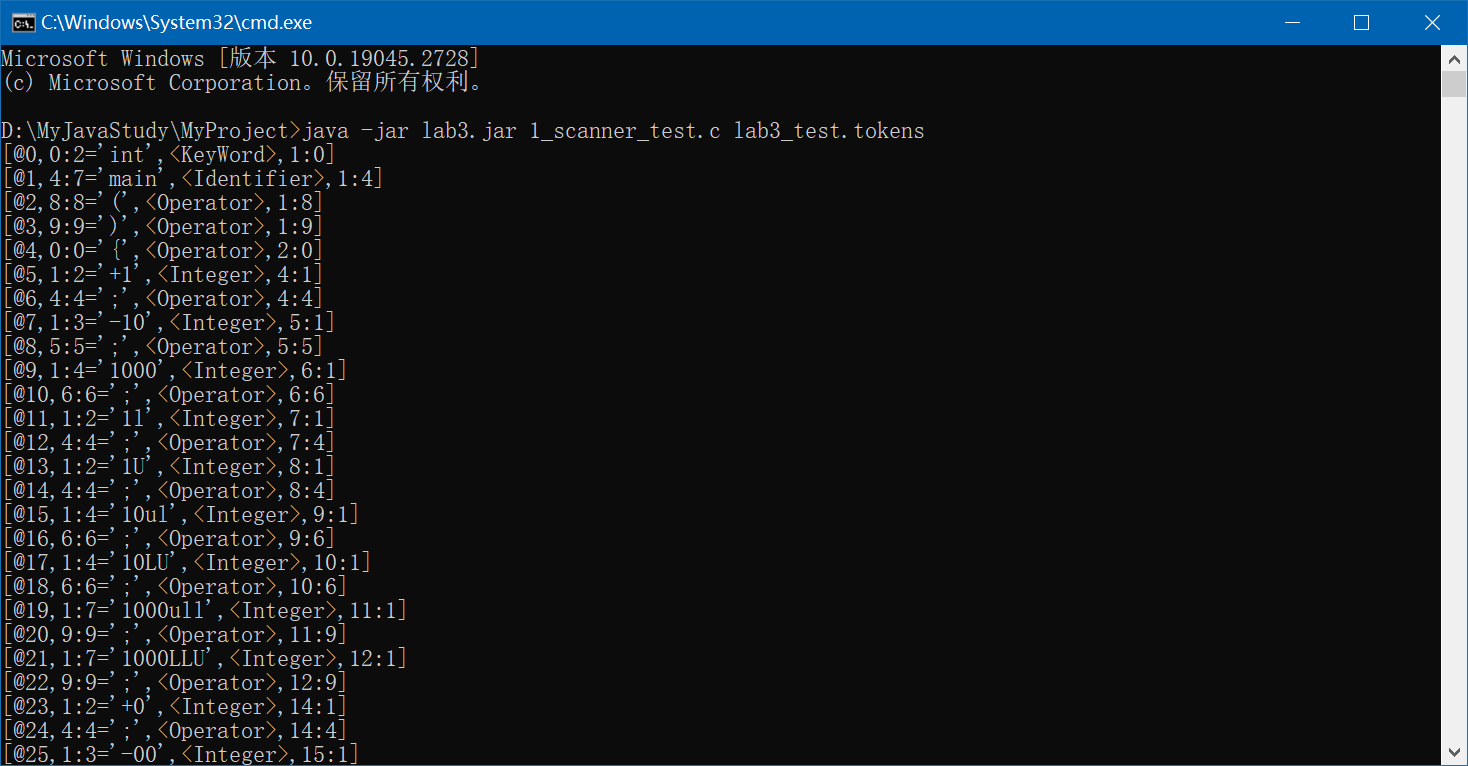
使用JFlex将c.flex生成为Lab3Scaneer，打包在lab3中。

使用实验指导文件中提到的BIT-MINICC框架中的词法分析器测试用例1\_scanner\_test.c。运行Main，参数为：1\_scanner\_test.c lab3\_test.tokens。得到测试结果：

最终将项目打包为jar包，命名为lab3.jar。

***Jar包运行方式：***

*java -jar lab3.jar 1\_scanner\_test.c lab3\_test.tokens*

1. **运行效果截图**
2. **实验心得体会**

本次实验是编译原理与设计的第三次实验，我首先从github下载了BIT-MINICC框架，安装了Eclipse，观察该框架下的词法分析器是如何编写的；后编写了一个简易的测试程序，使用该框架的词法分析器对输入进行测试，观察词法分析器的输入和输出；然后选择了Java作为实现的语言，编写flex文档，使用JFlex辅助生成，最终设计实现了自己的C语言词法分析器。

本次实验，让我进一步认识到了词法分析器的运行过程，并初步学会了编写自己的词法分析器，为后续编写编译器铺垫了良好的基础。

需要注意的是，JFlex使用人数远少于C语言实现的Flex，因此查阅资料时参考较少，必须得仔细参悟用户手册，并加以自己动手实践，才能不断解决问题，完成实验。