

编译原理与设计

实验报告

实验名称: Lab 3: 词法分析实验

姓名/学号: 李昊阳/1120203053

一、 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驱动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。

二、 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

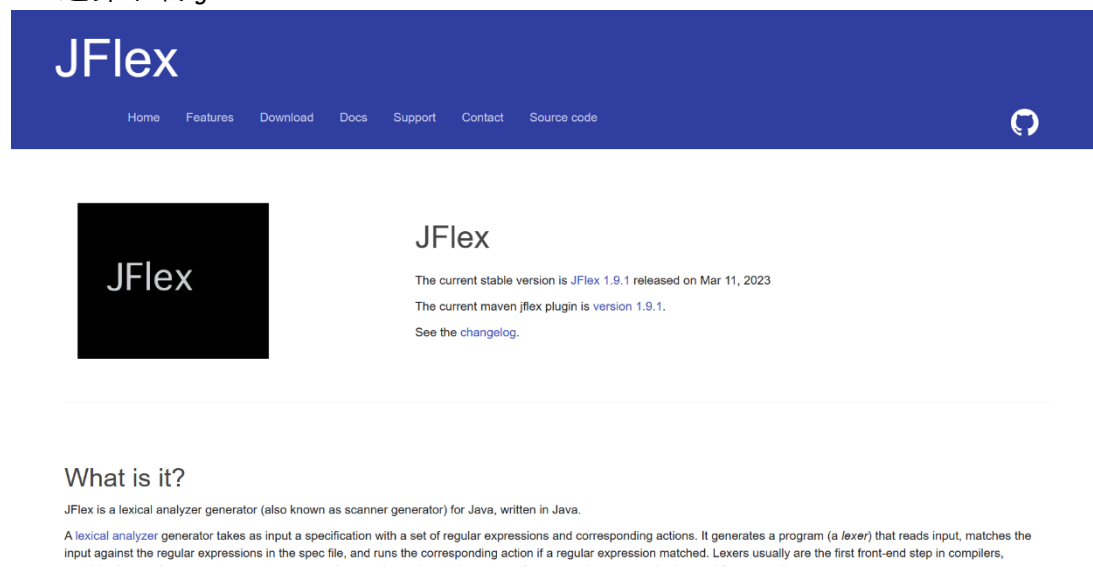
三、 实现的具体过程和步骤

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

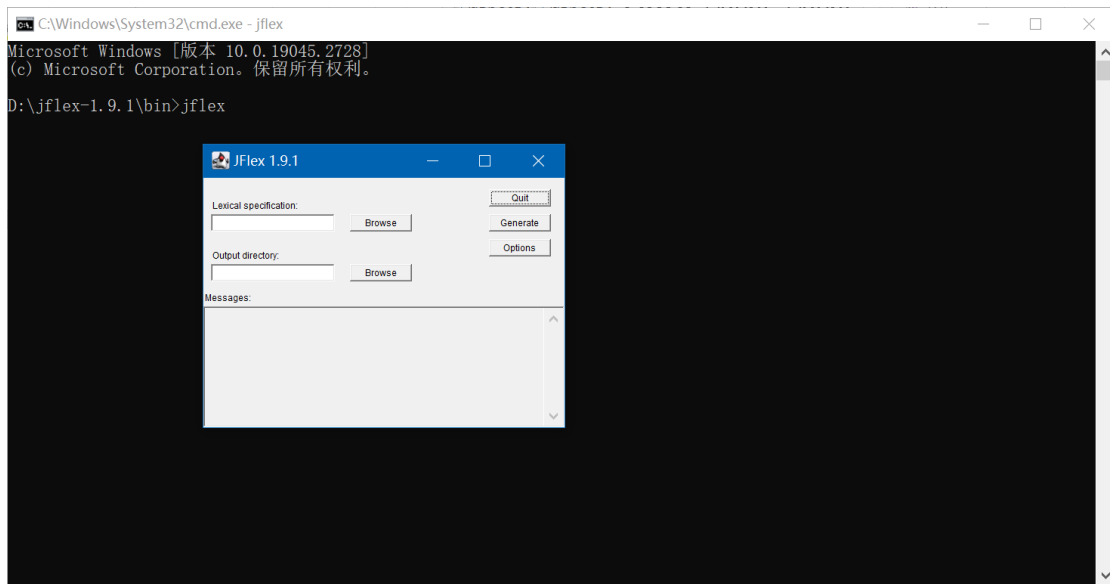
JFlex 是一款快速的词法分析程序生成器，可以根据 flex 文档自动生成对应语言的词法分析工具类(class)，实现语言（或称宿主语言）均为 Java。用户可通过调用该类的应用编程接口(API)，实现要求的词法分析效果与输出。此外，JFlex 还可与 Bison 结合，使用 JavaCUP 包，进一步实现语法分析。

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。



2. 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}
```


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}
```

3. 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;
```

4. 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:

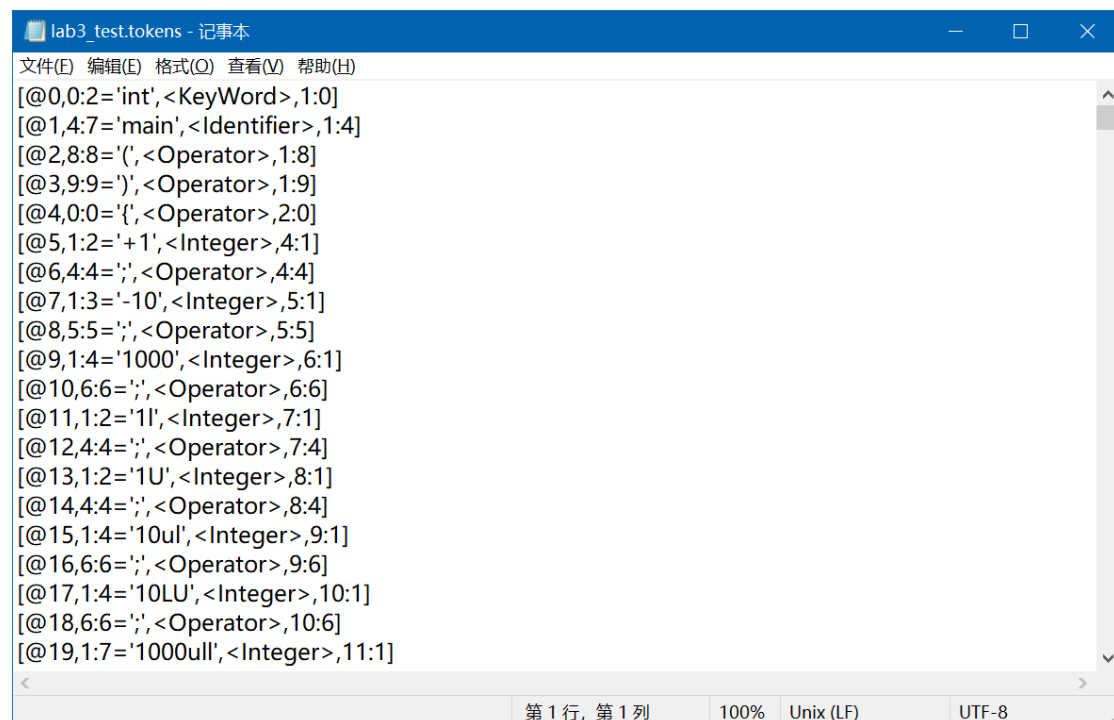
```
String strToken = "";

    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++;
}
```

5. 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：



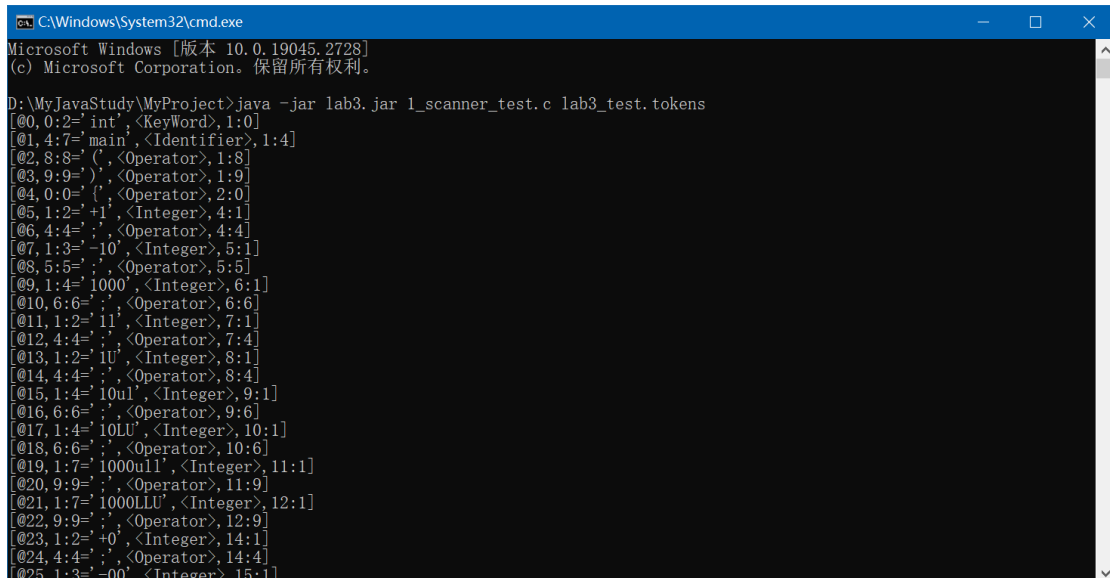
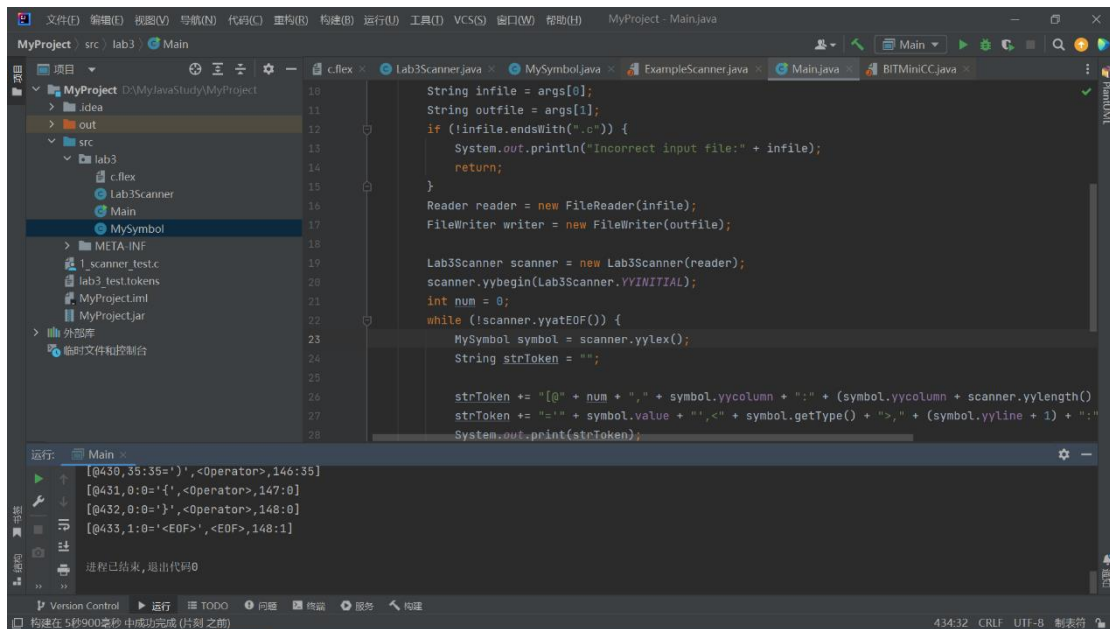
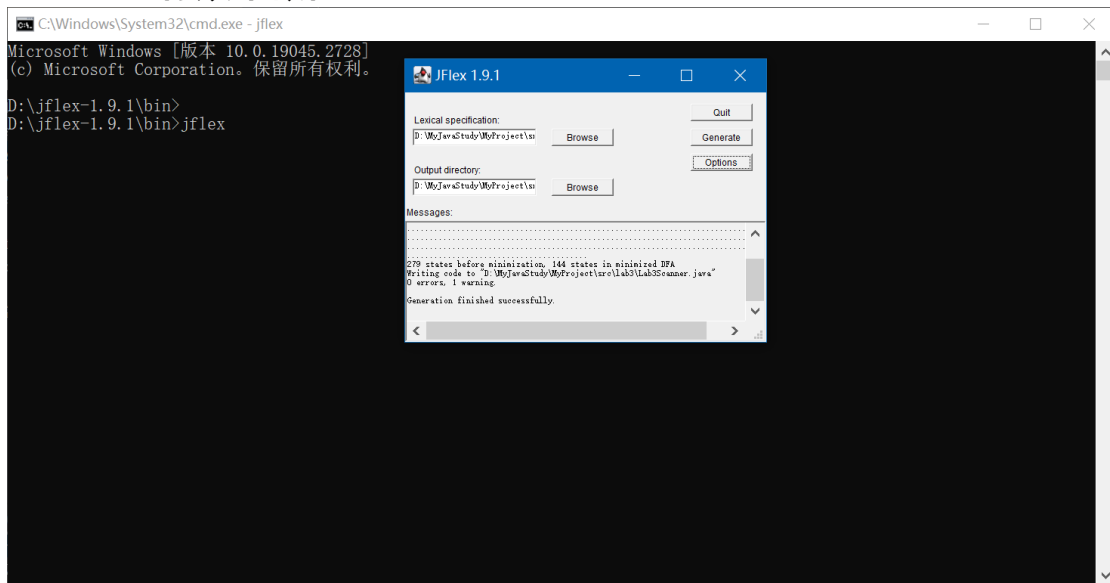
```
lab3_test.tokens - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
[0,0:2='int',<KeyWord>,1:0]
[1,4:7='main',<Identifier>,1:4]
[2,8:8='(',<Operator>,1:8]
[3,9:9=')',<Operator>,1:9]
[4,0:0='{',<Operator>,2:0]
[5,1:2='+1',<Integer>,4:1]
[6,4:4=';',<Operator>,4:4]
[7,1:3='-10',<Integer>,5:1]
[8,5:5=';',<Operator>,5:5]
[9,1:4='1000',<Integer>,6:1]
[10,6:6=';',<Operator>,6:6]
[11,1:2='1!',<Integer>,7:1]
[12,4:4=';',<Operator>,7:4]
[13,1:2='1U',<Integer>,8:1]
[14,4:4=';',<Operator>,8:4]
[15,1:4='10ul',<Integer>,9:1]
[16,6:6=';',<Operator>,9:6]
[17,1:4='10LU',<Integer>,10:1]
[18,6:6=';',<Operator>,10:6]
[19,1:7='1000ull',<Integer>,11:1]
```

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
```


四、 运行效果截图



五、 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。