## 实验二、自动生成词法分析程序（JFlex）

1. 以表格形式列出 Oberon-0 语言的词汇表

   根据词法单元的含义分类，我们将词法单元分为保留字、关键字、运算符、常量、标识符和标点符号六个大类，其含义分别为：

   保留字：由语言已经定义好的有确切语法含义的字，用户不能够再定义同名的标识符或变量

   关键字：语言预定义好的数据类型或过程，用户可以对其进行重定义

   运算符：与算数、逻辑、关系运算相关的标识某种运算的词法单元

   常量：数字常量

   标识符：用于唯一标识某个过程、类型、常量、变量等的名称

   标点符号：程序中用于分隔或作集合的部分

   事实上我们还需要考虑一种特殊的符号：空白符号，包括了换行、空格等。特别需要注意的是，数字常量中不能包含任何字母，即数字常量与标识符要用空白符好隔开。除此之外，我们往往直接忽略这些空白符号

   由此得到的Oberon-0语言的词汇表为

   | **类型** | **词法单元** |
   | --- | --- |
   | 保留字 | MODULE、BEGIN、END、CONST、TYPE、VAR、RECORD、ARRAY、OF、WHILE、DO、IF、THEN、ELSIF、ELSE、PROCEDURE |
   | 关键字 | INTEGER、BOOLEAN、READ、WRITE、WRITELN |
   | 运算符 | =、#、<、<=、>、>=、+、-、OR、*、DIV、MOD、&、~ |
   | 常量 | [1-9][0-9]*\|0[0-7]* |
   | 标点符号 | ';'、'.'、'('、')'、':'、','、':='、'['、']' |
   | 标识符 | letter(letter\|digit)* |

   保留字和关键字可以通过以下方法区分：由于用户无法定义名字与保留字相同的标识符，因此当词法分析器分析出保留字时，将不返回标识符类型，而是返回其独有的类型；而由于关键字可以被重定义，我们可以将其直接先将其视为标识符。在语法分析时，我们首先判断当前标识符是否在字符表上，不在时我们再查找关键字表，从而获得对应标识符的含义

2. 抽取 Oberon-0 语言的词法规则

Oberon-0词法规则的正则定义为

```
Octal -> 0|1|2|3|4|5|6|7
Digit -> 0|1|2|3|4|5|6|7|8|9
Nonzero -> 1|2|3|4|5|6|7|8|9
Constant -> Nonzero Digit* | 0 Octal*
Letter -> a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|
          A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
Identifier -> Letter (Letter|Digit)*
```

采用extended operators的正则定义

```
Octal -> [0-7]
Digit -> [0-9]
Nonzero -> [1-9]
Constant -> Nonzero Digit* | 0 Octal*
Letter -> [a-z]|[A-Z]
Identifier -> Letter (Letter|Digit)*
Comment -> "(*" [^*] ~"*)" | "(*" "*"+ ")"
```

运算符、保留字等词法单元内容都是确定的，可以直接识别。注意的是常量数字长度不超过12，标识符长度不超过24。

3. 与常见高级程序设计语言的词法规则相比，Oberon- 0 语言的词法规则有何异同

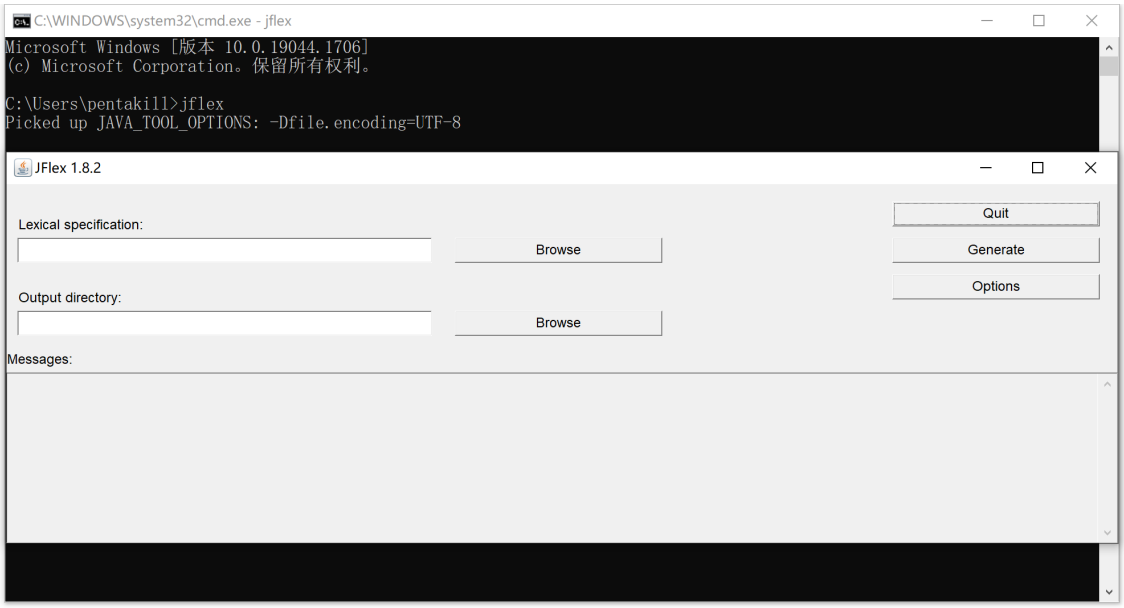
   具体的差异主要体现在：

   - Oberon-0的标识符中不允许有下划线_，而C这些高级语言标识符支持包含下划线
   - Oberon-0的注释为(**)，高级语言注释通常为//或/**/
   - Oberon-0中常量只包含数字常量，且数字常量只能为整数
   - Oberon-0中的部分运算符与高级语言不同，如除法为DIV
   - Oberon-0中只支持十进制和八进制数，高级语言还支持二进制和十六进制
   - Oberon-0语言是大小写无关的，允许的标识符和常量长度与高级语言不同

   相同点体现在(有些显而易见的不在这里列举)：

   - 常量与标识符之间都需要有空白符号分隔
   - 注释不允许嵌套
   - 标识符和常量长度具有一定的限制
   - 标识符都不允许数字开头，否则判定为常量错误
   - 两者都具有关键字和保留字的划分，但是高级语言的保留字通常包含了关键字
   - 两者未匹配的注释都会引发错误
4. 下载词法分析程序自动生成工具 JFlex

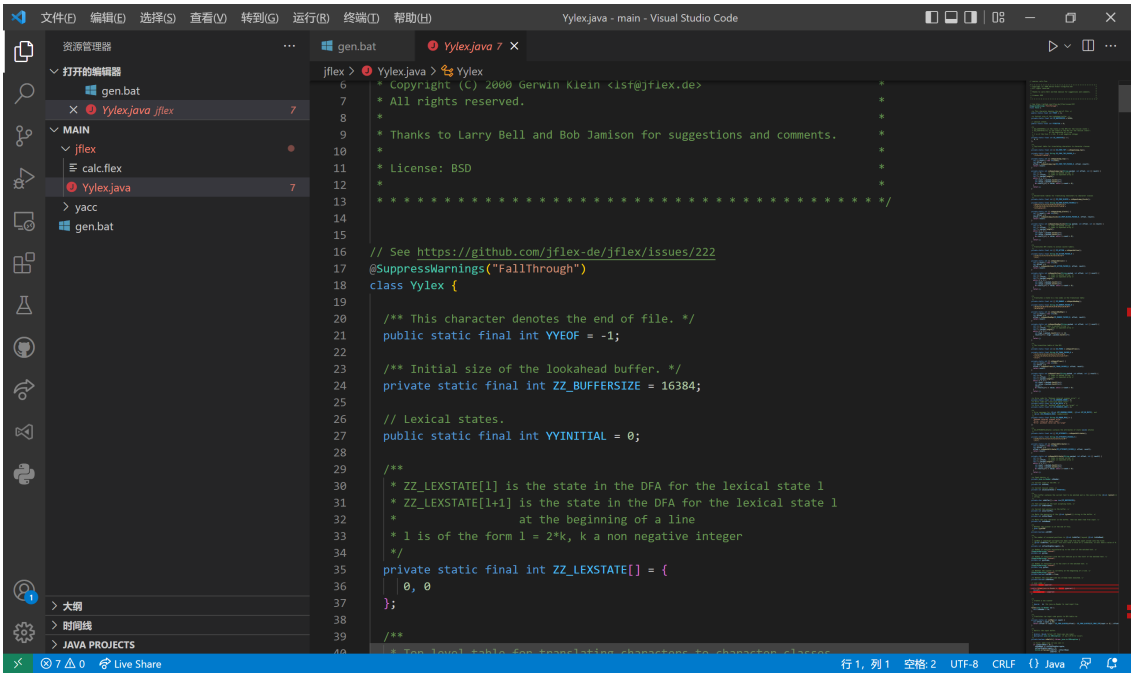
   配置好环境变量后，在命令行直接运行jflex指令，能够弹出对应的界面




尝试运行jflex给定的用例，检验jflex是否正确安装配置，并尝试使用命令行指令，生成对应的词法分析器，这里使用一个批处理指令，运行example中的byaccj(算术表达式的词法分析器)，查看运行结果。批处理指令为

```
@echo off
cd jflex
java -jar D:\JFLEX\jflex-1.8.2\lib\jflex-full-1.8.2.jar calc.flex
pause
```

获得对应的词法分析器，说明jflex配置安装成功




5. 生成 Oberon-0 语言的词法分析程序

这里我的jflex引用了sample.flex，并根据需要进行了一定的增添，从而生成符合要求的词法分析程序。对sample.flex主要的修改在于：

- 用户代码部分我们只需要import对应的模块即可，为了便于后续使用，我们需要引用java-cup的包以及exceptions的包
- 选项与声明部分，我们需要将类名修改为OberonScanner，使得词法分析器不采用jflex的默认名字，并删除原有的一条不必要的指令%type java_cup.runtime.Symbol。删除的原因是在%cup指令中已经包含了这个定义。%cup为我们增加了如下指令

```
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
%eofval{
return newjava_cup.runtime.Symbol(<CUPSYM>.EOF);
%eofval}
%eofclose
```

由于我们的返回函数是自定义的，因此不采用这里的eofval，而是重新在下面定义eofval。程序中，我对每个指令都进行了注释

- 选项与声明中，我们依旧应用sample中的get_line等函数，这些函数帮助我们找到对应错误的位置，同时CREATE_SYM函数帮助我们生成对应的词法单元。这里最主要的修改在于增添一些必要的声明，如注释的声明以及字符常量的声明，便于后续定义词法单元时使用。这里填入我们对应的正则定义(改为了正则表达式)，并修改identifier的定义(原来是使用jletter，其中包含了下划线)

```
WhiteSpace  = " "|\t|\b|\f|\r|\n|\r\n
Identifier      = [:letter:]+([:letter:]|[:digit:])*
Constant = [1-9][0-9]* | 0 [0-7]*
Comment = "(*" ([^*] |[*]* [^)])* [*]+ ")"
```

- 修改词法定义表。这里我们按照我们先前规定好的词法表，将其填入Sym.java文件中。特别地，我们在词法分析部分将关键字视为普通标识符，在语法分析时再做区分。除了关键字外，我们都在Sym.java中定义对应的词法单元，其中常量和标识符都是作为整体，且具有对应的值。由于这里的词法定义过多，我们不在这赘述。
- 增加纠错功能。由于我们的词法规则中有不兼容的部分，因此我们需要为词法分析器增加纠错功能，使之能够发现对应的词法错误。

  我们词法分析器分出的错误大致分为两个大类：长度不正确和格式不正确。对于长度不正确的问题，我们只需要在识别出对应的词法单元时，检查其对应的长度即可。对于格式不正确的问题，我们添加一些错误的正则表达式对其进行匹配，在匹配到错误的正则表达式时，发生了我们预期的错误。具体实现为

```
//错误的表达式
IllegalConstant = [:digit:]+[:letter:]+
IllegalOctal = 0[0-7]*[89]
IllegalComment = "(*"
//纠错规则
{Identifier} {
    if (yylength() > 24)
        throw new IllegalIdentifierLengthException(yytext());
    return CreateSYM(Sym.IDENTIFIER, yytext());
}
{Constant} {
    if (yylength() > 12)
        throw new IllegalIntegerRangeException(yytext());
    return CreateSYM(Sym.DIGIT, yytext());
}
{IllegalConstant} {
    throw new IllegalIntegerException(yytext());
}
{IllegalOctal} {
    throw new IllegalOctalException(yytext());
}
```

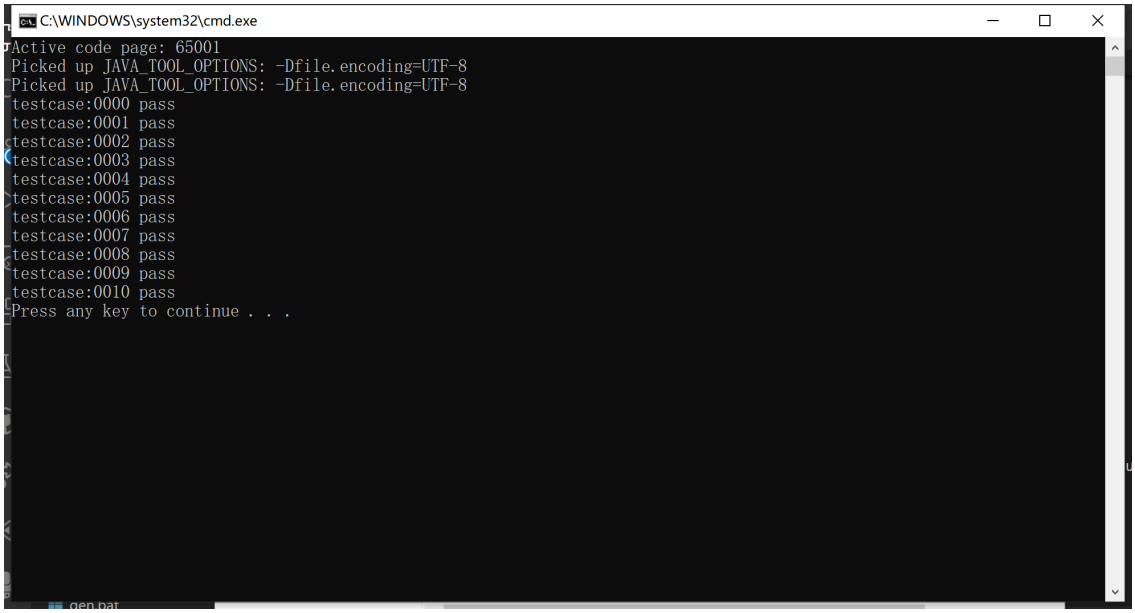
```
{IllegalComment} {
    throw new MismatchedCommentException(yytext());
}
```

6. 测试生成的词法分析程序

首先撰写词法分析的测试程序，参考ComplexityDemo2的写法，得到一个能够自动运行testcases的测试程序，具体的calculate函数依据LexicalMainDemo中的main函数，得到如下的测试程序。这里测试异常情况采用try……catch的方式，对于不同的异常产生不同的输出

```
public String calculate(String arg0) {
    try {
        String code = arg0;
        StringReader reader = new StringReader(code);
        // 创建由JFlex生成的Scanner类实例
        OberonScanner scanner = new OberonScanner(reader);
        StringBuilder builder = new StringBuilder();
        while (!scanner.yyatEOF()) {// 获取Scanner输出的token
            java_cup.runtime.Symbol s = scanner.next_token();
            // 记录token
            if (s.value != null) {
                builder.append("(" + s.toString() + ")[" + (String) s.value
+ "]");
            } else {
                builder.append("(" + s.toString() + ")");
            }
            builder.append(" ");
        }
        // 比较Scanner的输出和正确结果
        String result = builder.toString().trim();
        return result;
    } catch (Exception e) {
        return e.getMessage();
    }
}
```

接下来进行测试用例的测试，首先将实验软装置中的测试用例添加到testcases中，并根据对应的词法单元预期获得的结果，添加到结果中，检测词法单元生成是否符合预期。接下来再将实验一中我写的所有有关词法错误和正确的程序加入到testcases中(包含PDF中的用例程序)，检测查错是否正确。具体测试用例详见testcases中的testcase文件，用内部的注释说明了出现的词法错误，运行结果如下：




7. 不同 lex 族工具差异比较
    - 在整体定义的框架上，JFlex和JLex都将输入源文件划分为用户代码、选项与声明、词法规则部分，其中用户代码中的定义将直接添加入生成的词法分析器中；GNU Flex将输入源文件划分为定义、规则、用户代码部分。在源文件的组织方式上有所不同。但是用户代码部分，三者都是直接将输入代码copy到对应的词法分析器中，选项与声明部分与定义部分类似，规则与词法规则部分类似
    - JFlex和JLex都是面向java语言的自动化工具，而GNU Flex是面向C语言的自动化工具，在词法定义上因语言的不同而会有差异，但是三者采用的注释格式是相同的
    - 在词法规则的定义部分，三者都采用了pattern action的模式，pattern为对应的正则表达式，action为当识别为该正则表达式时，所需要采取的动作。但是JFlex和JLex还支持在每块定义中有一个State部分，表示字符串必须以state部分的字符串作为开头
    - JFlex可以定义--jlex使其采用jlex的解释规范，两者的定义上没有太大的差别