Getting Started with JLex

EXERCISE 1: (A) Try the following JLex Program 1 **to recognize an identifier**. (B) Understand the program by trying out with different inputs and by tweaking the code and seeing the effect and also by going through the generated Java code. (C) Finally, add rules to recognize numbers (NUM) and reals (REAL).

```
-------------------------Program 1: IdLexer----------------------------
import java.io.*;

class Main {
        public static void main(String args[]) throws IOException {
                IdLexer lex = new IdLexer(System.in);
                Token token = lex.yylex();

                while ( token.text != null ) {
                        token = lex.yylex(); //get next token
                }
        }
}

class Token {
    String text;
    Token(String t) { text = t; }
}
%%

%public
%class IdLexer
%type void

digit      = [0-9]
letter     = [a-zA-Z]
whitespace = [ \t\n]

%type Token
%eofval{
return new Token(null);
%eofval}
%%

{letter}({letter}|{digit})*  { System.out.print("<ID," + yytext() + "> "); }
{whitespace}+                { /* skip white spaces */ }

------------------------------End of Program------------------------------
```

```
Here is how we generate code and run it on input:

CMD> jlex IdLexer        ==> Generates IdLexer.java
CMD> javac IdLexer.java  ==> Generates IdLexer.class, Main.class, Token.class
CMD> java Main           // Type strings at the terminal and ctrl-D to exit
My first JLex program
alpha123numeric 1234 hello
```

EXERCISE 2: (A) Try the following JLex Program 2 that includes rule **to recognize the keyword** ”if” in addition to identifiers. (B) Move the ”if” rule below that of identifier rule and check the effect on your input. Do you see any difference in the output? (C) Add other keywords, operators and all types of parantheses in a similar fashion and try with several inputs to convince yourself of its working. (D) Go through the JLex manual and try few more things based on whatever you can grasp.

```
-------------------------Program 2: IdKwdLexer-------------------------
import java.io.*;

class Main {
        public static void main(String args[]) throws IOException {
                IdKwdLexer lex = new IdKwdLexer(System.in);
                Token token = lex.yylex();

                while ( token.text != null ) {
                        token = lex.yylex(); //get next token
                }
        }
}

class Token {
    String text;
    Token(String t) { text = t; }
}
%%

%public
%class IdKwdLexer
%type void

digit      = [0-9]
letter     = [a-zA-Z]
whitespace = [ \t\n]

%type Token
%eofval{
return new Token(null);
%eofval}
%%

"if"                         { System.out.print("<IF," + yytext() + "> "); }
{letter}({letter}|{digit})*  { System.out.print("<ID," + yytext() + "> "); }
{whitespace}+                { /* skip white spaces */ }

------------------------------End of Program------------------------------
```

EXERCISE 3: (A) Try the following JLex Program 3 that includes a rule **to recognize comments** of the type "// xxxx". (B) Remove {space} from the rule and see what happens with the input "// This is a single line comment". (C) Add rule(s) to recognize block comment i.e. comments of type /* xxxx */. (D) Add rule to recognize string literal. i.e. Given input "compiler", the output should be $< STRING, compiler >$.

```
-------------------------Program 3: IdKwdCmtLexer-------------------------
import java.io.*;

class Main {
        public static void main(String args[]) throws IOException {
                IdKwdCmtLexer lex = new IdKwdCmtLexer(System.in);
                Token token = lex.yylex();

                while ( token.text != null ) {
                        token = lex.yylex(); //get next token
                }
        }
}

class Token {
    String text;
    Token(String t) { text = t; }
}
%%

%public
%class IdKwdCmtLexer
%type void

digit      = [0-9]
letter     = [a-zA-Z]
space      = [ ]
tab        = [\t]
newline    = [\n]
whitespace = [ \t\n]

%type Token
%eofval{
return new Token(null);
%eofval}
%%

"//"({letter}|{digit}|{space})*{newline}        { /* skip comments */ }
"if"                           { System.out.print("<IF," + yytext() + "> "); }
{letter}({letter}|{digit})*  { System.out.print("<ID," + yytext() + "> "); }
{whitespace}+                { /* skip white spaces */ }

------------------------------End of Program------------------------------
```

EXERCISE 4: (A) Add the following **error handler** at the bottom-end of the JLex file. When a lexeme does not match any token type, this error is thrown. Test it with illegal input. (B) Check the difference between the errors thrown with and without this error handler. (C) If you have a valid lexeme after an invalid lexeme, does the program process the valid lexeme? i.e. say input is @@@@ xyz, does the program process xyz after throwing the error for @@@@?

```
.|\n    { throw new Error("Illegal character <" + yytext()+">"); }
```

EXERCISE 5: Write your own Jlex program to recognize the following. Test it thoroughly with examples.

1. Domain names of the form: http://www.google.co.in or ftp://aaa.bbbbb.com
2. Email ids of the form: student@am.amrita.edu
3. Names of the form: lastname, firstname[, middlename]. Note that middlename is optional and can have any number of subnames.
4. Phone numbers of the form: +91-476-2801280 or 0476-2801280 or 2801280.
5. Mobile number of the form: +91xxxxxxxxxx or 0xxxxxxxxxx or xxxxxxxxxx (10-digits exactly).

Note: Your rules should be very generic which can accept any depth of subdomains i.e. $ftp://abc.defg.-----.xyz.com$. Similar with email too.

EXERCISE 6: Write your own Jlex program to recognize the following. Test it thoroughly with examples.

1. All strings that start with P and end with !.
2. All strings that start with a number and end with an alphabet.
3. All strings with atmost three consecutive 0's as a substring.
4. All strings with every 1 followed by two 0's.

EXERCISE 7: *%char* macro can be added to the second section to activate character counting. The integer variable *yychar* can be used to print the number of characters processed in the input string so far. Similarly, *%line* macro can be added to the second section to activate character counting. The integer variable *yyline* can be used to print the number of characters processed in the input string so far.