Lab No. 01 – Building a Lexical Analyzer

A Lexical Analyzer checks spelling and breaks a statement into streams of tokens. The checking is done by patterns specified by Regular Expressions or REGEX.

Take a look at the given code below:

Code Listing 1 - A simple lexical analyzer (tokens.l)

```
%{
int tokencount = 0;
%}

%%
[ \t\n]+ {printf("White spaces ignored\n");}
\\\\.*.[\n]? {printf("%s is a comment\n",yytext);}
[a-zA-Z]+ {printf("%s is a token\n",yytext);tokencount++;}
[<>\-\+\?\*\/!\^,\(\);] {printf("%s is a token\n",yytext);tokencount++;}
. {printf("Unexpected\n");}
%%

int main(){
      yylex();
      printf("Number of tokens is : %d\n",tokencount);
}
```

Now, let's explain the program.

1. **The coding environment**: We need the package FLEX to be installed for building a lexical analyzer.
2. **Naming convention**: The file extension must be (**any_name.l**), the **.l** extension is for lexical analyzer.
3. **Running the code**: In order to run the code, we need to run the following commands sequentially-
    a. Go to terminal and run the following:
        i. **`lex file_name.l`** [The name of the file that contains the code.]
        ii. **`cc lex.yy.c -o output_name -ll`**
        [**"cc**" is for selecting the C compiler, "**lex.yy.c**" is the output file from the previous command, "**-o**" is to specify an output file name, "**output_name**" can be given by user, "**-ll**" is to add the lex library to the compiler.]
        iii. **`./output_name`** [To run the output file.]
    b. Type anything on the terminal and it will show, if the token is a lexeme or not.
    c. To get out of the program press "**CTRL+D**" or "**CTRL+C**".

Breakdown of the Code:

- Any lex file contains three main regions of code:

```
1. %{
   //This part is to include any file or declare any global variable.
2. %}
3. %%
   //Here all the regular expressions or patterns are defined
4. %%
5. int main(){
   //all required execution code are specified in main function.
6. }
```

- **Line no. 02:** The **`int tokencount = 0`** is a variable to store the number of valid tokens after execution of the code.
- **Line no. 06:** **`[ \t\n]+`** is a regular expression that accepts all Spaces, Tabs and Newline characters for once or infinite times and prints a message whenever they are encountered. Beware that, there's a **SPACE** before the "\t" in the regular expression.
- **Line no. 07:** **`\\\\.*.[\n]?`** is a regular expression that accepts a comment line. Remember in C code a comment line follows the given structure ("\\This is a comment line."). As in flex "\" has a special function so to accept a "\" as a character we need to escape its functionality using another "\". So, to accept one "\" we need two "\\".
- **Line no. 08:** **`[a-zA-Z]+`** accepts all valid characters from small (a-z) and big (A-Z) and the pattern is accepted for 1 up to infinite times. Basically it accepts all valid strings that are made up of alphabets only. We are considering these as valid tokens and so increasing the tokencount if we encounter any token that has the specified pattern.
- **Line no. 09:** **`[<>\-\+\?\*\/!\^,\(\);]`** accepts a list of special characters. The special characters that have a function in flex are escaped first using a "\". We are also considering the operators as valid tokens and so increasing the tokencount.
- **Line no. 10:** All characters except a newline character are accepted by a **(.)**.
- **Line no. 14:** We are calling the **`yylex()`** function, which, runs the patterns that we just specified against all inputs that we provide.
- **Line no. 15:** When we exit the program, the total number of valid tokens i.e. lexemes are displayed.

Code Listing 2 - A simple lexical analyzer that ttakes input form a file called "code.c"

```
1  %{
2  int tokencount = 0;
3  %}
4
5  %%
6  [ ]+ {printf("White spaces ignored\n");}
7  \\\\.*.[\n]? {printf("%s is a comment\n",yytext);}
8  [a-zA-Z]+ {printf("%s is a token\n",yytext);tokencount++;}
9  [<>\-\+\?\*\/!\^,\(\);] {printf("%s is a token\n",yytext);tokencount++;}
10 \/\*(.|\n)+\*\/ {printf("Comment block found");}
11 . {printf("Unexpected\n");}
12 %%
13
14 int main(){
15       FILE *file;
16       file = fopen("code.c", "r") ;
17       if (!file) {
18             printf("couldnot open file");
19             exit (1);
20       }
21       else {
22             yyin = file;
23       }
24       yylex();
25       printf("Number of tokens is : %d\n",tokencount);
26 }
```

N.B.: In order to run this code you must create a file called "code.c" in the directory of the lex file and put some sample code in it.

Sample Input file: Suppose we have the sample "code.c" as follows:

```
This is a sample text.
a=10;
b=a+5;
```

Code Listing 3 – Sample "code.c" file

Sample Output: Here the output will be

```
This is a token.
White spaces ignored.
is is a token.
White spaces ignored.
a is a token.
White spaces ignored.
sample is a token.
White spaces ignored.
text is a token.
. Unexpected
White spaces ignored.
a is a token.
= Unexpected
10 Unexpected
; is a token.
White spaces ignored.
b is a token.
= Unexpected
a is a token.
+ is a token.
5 Unexpected
; is a token.

Number of tokens is 11.
```

Tasks for LAB 01:

1. Create regular expressions for accepting the following strings:
    a. Numbers or digits i.e. 123, 999 etc.
    b. Any string that starts with a capital letter, e.g., And, B04, Call etc.
    c. Any string that has an operator somewhere in it, e.g., a+b, a++, 2--, *b/ etc.
    d. Any double/float numbers, e.g., 23.343, 99.999 etc.
    e. Any string that starts and ends with a vowel, e.g., apostle, oscillate etc.
2. Create a lex program that will count the number of variables, variable types, operators and digits separately.

N.B.: Rules of regular expression can be found in the appendix.

Appendix

Regular Expressions:

General rules of regular expressions are as follows:

Table 1 – Rules of Regular Expressions

Expression	Pattern accepted
`[abc]`	Will accept all characters that are either a/b/c
`(abc)`	Will accept the string "abc"
`[abc]+`	Will accept the characters a/b/c either 1 or ∞ times, e.g., aa, bbbc, abc, a etc.
`[abc]*`	Will accept the characters a/b/c either 0 or ∞ times.
`(abc)?`	Will accept the string "abc" 0 or 1 time.
`.`	Will accept all characters except a newline character.
`(a|b)`	Will match the character "a" or "b"
`^(a)`	Matches the beginning of a line as the first character of a regular expression.
`[^a]`	Will match any character except "a".
`a$`	Matches "a" at the end of a line as the last character of a regular expression
`(abc){min,max}`	Will match the string "abc" from **min** to **max** times.
`(abc){,max}`	Will match the string "abc" from 0 times to max times.
`(abc){min,}`	Will match the string "abc" from min times to ∞ times.
`"abc"`	Will match everything literally within the quotation marks.

Check the book for extended usage and examples.