EXPERIMENT - 6

CLASS: TE CMPN B

DATE: 31/01/ '22

NAME: Keegan Vaz

ROLL NO.: 28

AIM: STUDY AND THE IMPLEMENTATION OF LEX AND YACC TOOL

THEORY:

- Explain the role of Lexical analyzer in Compiler Design

Ans: Programs that perform Lexical Analysis in compiler design are called lexical analyzers or lexers. A lexer contains a tokenizer or scanner. If the lexical analyzer detects that the token is invalid, it generates an error. The role of Lexical Analyzer in compiler design is to read character streams from the source code, check for legal tokens, and pass the data to the syntax analyzer when it demands.

- Brief descriptions of tools used for lexical analysis

Ans:

1. Lex: This tool is a standard component on most UNIX operating systems. The GNU flex tool provides the same functionality.
2. Yacc: This tool is standard on most UNIX operating systems. The GNU bison tool provides the same functionality.
3. C compiler: Any standard C compiler, including Gnu CC, will be fine.
4. Make tool: This tool is required to use the sample Makefile to simplify building.

- Brief introduction to Lex tool

Ans: The lex tool (or the GNU tool, flex) uses a configuration file to generate C source core, which you can then use either to make a standalone application, or you can use it within your own application. The configuration file defines the character sequences you expect to find in the file that you want to parse, and what should happen when this sequence is discovered. The format of the file is straightforward, you specify the input sequence and the result, separated by a space (or tab).

- Brief introduction to Yacc tool

Ans: YACC stands for Yet Another Compiler Compiler. It provides a tool to produce a parser for a given grammar. It is a program designed to compile a LALR (1) grammar. It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar. The input of YACC is the rules or grammar and the output is a C program.

- Steps to execute LEX and YACC Program in Linux OS.

Ans: Execute lex -

1. Write the program within the rules of lex.
2. Running the file on the terminal by the command 'flex filename.l'.
3. We then need to run the command 'gcc lex.yy.c'.
4. After all the above commands are executed without any error then we need to run the a.exe file and we get the final output.

Execute yacc -

1. Write the program within the rules of yacc.
2. Running the file on the terminal by the command 'flex filename.l'.
3. Then we need to run the command 'bison -dy filename.y'.

4. We need to run the commands to compile the file in yacc by the command 'gcc lex.yy.c y.tab.h'.

After all the above commands are executed without any error then we need to run the a.exe file and we get the final output.

IMPLEMENTATION:

- LEX and YACC Program in Flex Tool for Windows or in UBUNTU

To install flex and Bison in Ubuntu→

sudo apt-get update
sudo apt-get install flex
sudo apt-get install bison

Steps to execute your LEX and YACC Program in UBUNTU→

For Compiling Lex file only:
lex hello.l
gcc lex.yy.c –ll
./a.out

For Compiling Lex and YACC both :
lex hello.l
yacc –d hello.y
gcc lex.yy.c y.tab.c -ll
./a.out

To install flex and Bison in WINDOWS →

Installing Softwares:
Download Flex 2.5.4a
Download Bison 2.4.1
Download DevC++
Install Flex at "C:\GnuWin32"
Install Bison at "C:\GnuWin32"
Install DevC++ at "C:\Dev-Cpp"
Open Environment Variables.
Add "C:\GnuWin32\bin; C:\Dev-Cpp\bin;" to path.

Compilation & Execution of your Program in WINDOWS→

1. Open Command prompt and switch to your working directory where you have stored your lex file (".l") and yacc file (".y")

2. Save your Lex files and YACC files be "hello.l" and "hello.y". Now, follow the preceding steps to compile and run your program.

For Compiling Lex file only:
flex hello.l
gcc lex.yy.c

For Compiling Lex & Yacc file both:
flex hello.l
bison -dy hello.y
gcc lex.yy.c y.tab.c

For Executing the Program

a.exe

OUTPUT (Program + Output Snapshots)

1. Write a LEX program for “Hello World” with output

Code:

```
%option noyywrap
%{
   #include<stdio.h>
%}
%%
\n {printf("Hello World");}
%%
int main()
{
   printf("Press Enter to view the output");
   yylex();
   return 0;
}
```

Output:

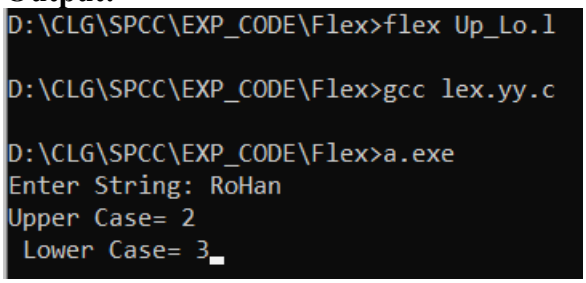
```
D:\CLG\SPCC\EXP_CODE\Flex>flex first.l

D:\CLG\SPCC\EXP_CODE\Flex>gcc lex.yy.c

D:\CLG\SPCC\EXP_CODE\Flex>a.exe
Press Enter to view the output
Hello World
```

2. Write a LEX program to count and identify upper case and lower case letter with output

Code:

```
%option noyywrap
%{
   #include<stdio.h>
   int UC=0;
   int LC=0;
%}
%%
[A-Z] {UC++;}
[a-z] {LC++;}
\n {printf("Upper Case= %d \n Lower Case= %d",UC,LC);}
%%
int main()
{
printf("Enter String: ");
yylex();
return 0;
}
```

Output:

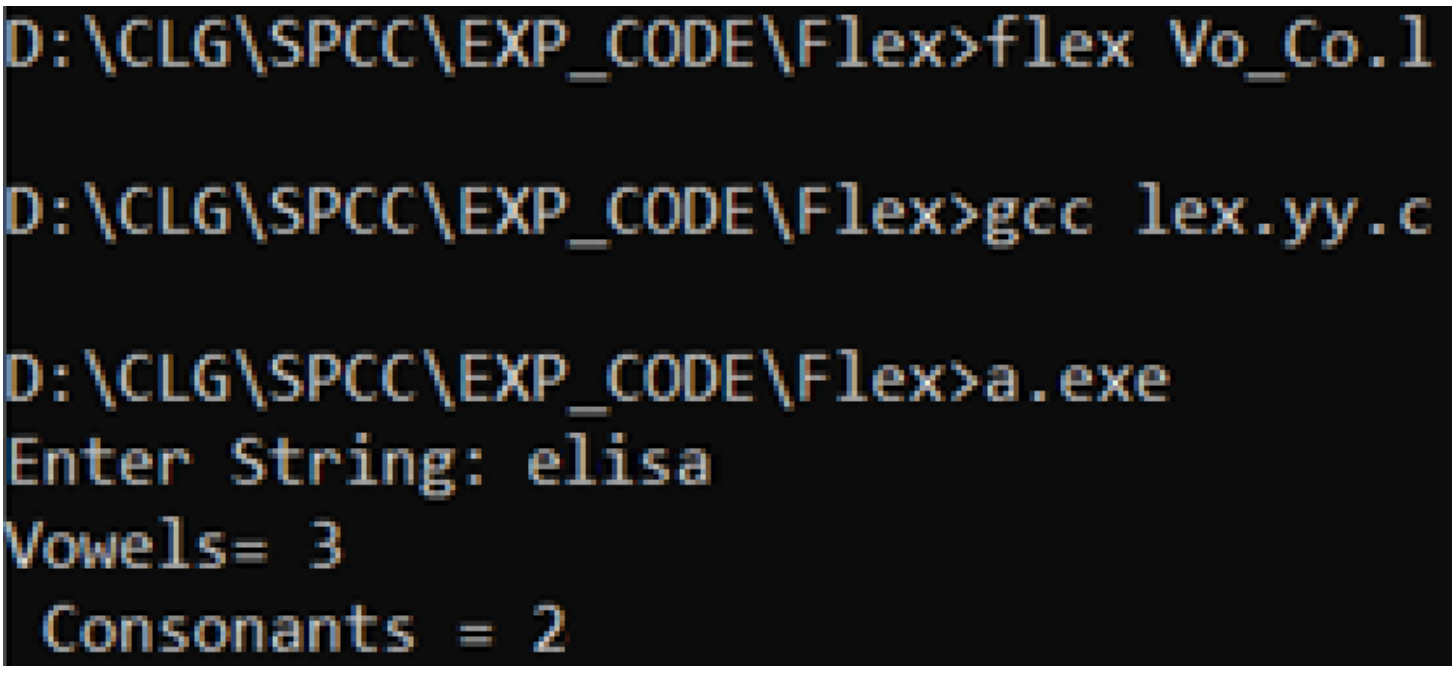


3. Write a LEX program to count and identify Vowels and consonants with output

Code:

```
%option noyywrap
%{
   #include<stdio.h>
   int V=0;
   int C=0;
%}
%%
[AEIOUaeiou] {V++;}
[^AEIOUaeiou\n] {C++;}
\n {printf("Vowels= %d \n Consonants = %d",V,C);}
%%
int main()
{
   printf("Enter String: ");
   yylex();
   return 0;
}
```

Output:



4. Write a LEX program to count and identify tokens with output

Code:

```
%option noyywrap
%{
int n = 0 ;
%}

%%
"while"|"if"|"else" {n++;printf("\t keywords : %s", yytext);}
"int"|"float" {n++;printf("\t keywords : %s", yytext);}
[a-zA-Z_][a-zA-Z0-9_]* {n++;printf("\t identifier : %s", yytext);}
"<="|"=="|"="|"++"|"-"|"*"|"+" {n++;printf("\t operator : %s", yytext);}
[(){}|, ;] {n++;printf("\t separator : %s", yytext);}
[0-9]*"."[0-9]+ {n++;printf("\t float : %s", yytext);}
[0-9]+ {n++;printf("\t integer : %s", yytext);}
. ;
%%

int main()
{
    yylex();
    printf("\n total no. of token = %d\n", n);
}
```

Output:

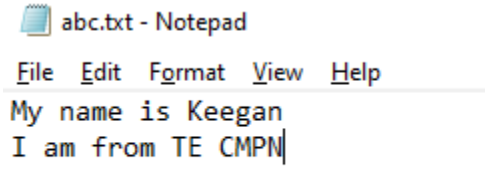
```
D:\CLG\SPCC\EXP_CODE\Flex>flex token.l

D:\CLG\SPCC\EXP_CODE\Flex>gcc lex.yy.c

D:\CLG\SPCC\EXP_CODE\Flex>a.exe
[illegible] A+B A*B
        identifier : [illegible]a       separator :      identifier : A  operator : +     identifier : B  separator :      identifier : A  operator : *     identifier : B

 total no. of token = 9
```

5. Write a program to count number of characters, words, sentences, lines, tabs, numbers and blank spaces present in input using LEX.

Code:

```
%{
#include<stdio.h>
int nlines=1,nwords,nchars,sc=0, tc=0, ch=0;
%}

%%
\n {
    nchars++;nlines++;
    }
([ ])+ sc++;
\t tc++;
. ch++;
[^ \n\t]+ {nwords++, nchars=nchars+yyleng;}
%%

int yywrap(void)
{
 return 1;
}


int main()
{
    yyin= fopen("abc.txt","r");
    yylex();
    printf("\nNo. of lines=%d", nlines);
    printf("\nNo. of words=%d", nwords);
    printf("\nNo. of characters=%d", nchars);
    printf("\nNo. of spaces=%d", sc);
    printf("\nNo. of tabs=%d", tc);
    return 0;
}
```

Output:




```
D:\CLG\SPCC\EXP_CODE\Flex>flex count.l

D:\CLG\SPCC\EXP_CODE\Flex>gcc lex.yy.c

D:\CLG\SPCC\EXP_CODE\Flex>a.exe

No. of lines=2
No. of words=8
No. of characters=30
No. of spaces=7
No. of tabs=0
```

6. Write a program to recognize valid arithmetic expression that uses operators +, -, * and / and design calculator using YACC

Code:

arithmetic.l

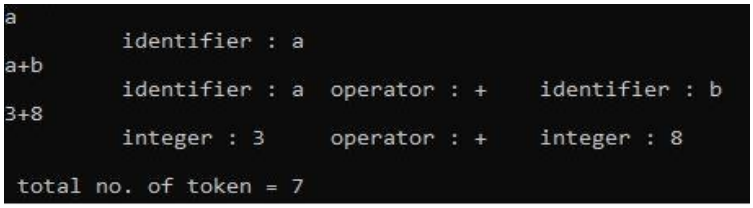
```
%{
   #include "y.tab.h"
%}

%%
[a-zA-Z_][a-zA-Z_0-9]* return id;
[0-9]+(\.[0-9]*)?      return num;
[+/*]                    return op;
.                       return yytext[0];
\n                       return 0;
%%

int yywrap()
{
return 1;
}
```

arithmetic.y

```
%{
   #include<stdio.h>
   int valid=1;
%}

%token num id op

%%
start : id '=' s ';'
s :     id x
      | num x
      | '-' num x
      | '(' s ')' x
      ;
x :     op s
      | '-' s
      |
      ;
%%

int yyerror()
```

```
{
   valid=0;
   printf("\nInvalid expression!\n");
   return 0;
}

int main()
{
   printf("\nEnter the expression:\n");
   yyparse();
   if(valid)
   {
      printf("\nValid expression!\n");
   }
}
```

Output:



CONCLUSION:

From the above experiment we are able to learn and write logics in lex language with the extension of '.l'. **Importance of Lex-** Lex can perform simple transformations by itself but its main purpose is to facilitate lexical analysis, the processing of character sequences such as source code to produce symbol sequences called tokens for use as input to other programs such as parsers.