**Experiment No.:2 Date: 13/10/2020**

**LEX**

Aim: To study LEX and write a LEX program to add line numbers for given text.

Theory:

**Lex - A Lexical Analyzer Generator**

*M. E. Lesk and E. Schmidt*

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

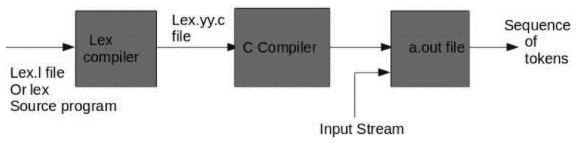
Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

**Flex, A fast scanner generator**

*Vern Paxson*

flex is a tool for generating scanners: programs which recognized lexical patterns in text. flex reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. flex generates as output a C source file, `lex.yy.c', which defines a routine `yylex()'. This file is compiled and linked with the `-lfl' library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.Reference: http://dinosaur.compilertools.net/

Creating a lexical analyzer with Lex



**Programs 1**

%{

int line\_number=1; //initializing line number to 1

%}  //”.” Matches any single character except the new line character

line  .\*\n   //declare 0 or more ASCII characters followed by a “\n” as a line

%%

/\* simple name definitions to simplify the scanner specification name  definition of line\*/

{line}     {printf("%5d %s", line\_number++,yytext);}

%%

int yywrap(){}

int main()

{

 extern FILE \*yyin;  //global variable to read the file. If local then standard input is read

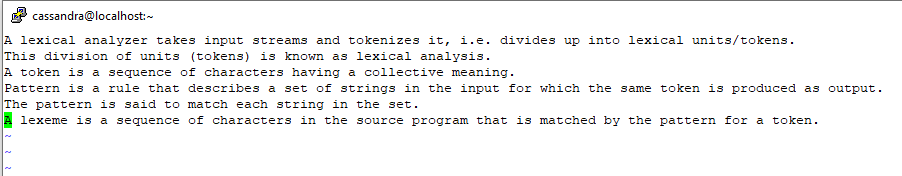
 yyin=fopen("source.txt","r");

 yylex();

 return 0;

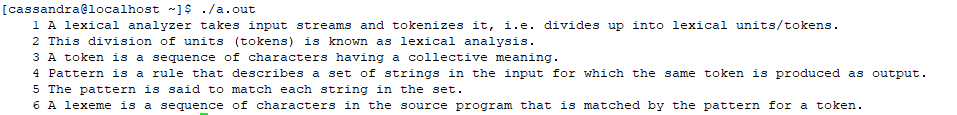
}

**INPUT :**





**OUTPUT:**



**Program 2**

root@kali:~# vi expt2prgm2

//Lex program to identify valid tokens such as identifier, keywords, integer and real //number.

%{

#include<stdio.h>

%}

%%

printf|while                printf("<keyword,%s>\n",yytext);

[a-zA-Z\_][a-zA-Z0-9\_]\*    printf("<id,%s>\n",yytext);

-?[0-9]+                    printf("<num,%s>\n",yytext);

-?[0-9]+["."]?[0-9]+       printf("<float,%s>\n",yytext);

\".\*\"   printf("<literal,%s>\n",yytext);

[,|(|)|;]                     printf("<delim,%s>\n",yytext);

%%

int yywrap(){}

int main()

{

 extern FILE \*yyin;

 yyin=fopen("srclng.txt","r");

 yylex();

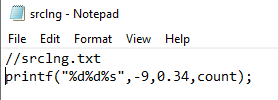
 return 0;

}

root@kali:~# lex expt2prgm2

root@kali:~# gcc lex.yy.c

Input:



**OUTPUT:**

 <keyword, printf>

<delim, (>

<literal, “%d%d%s”>

<delim, ,>

<num,-9>

<delim, ,>

<float, 0.34>

<delim, ,>

<id, count>

<delim, ,>

<delim, ;>

**Conclusion:**LEX program to add line numbers for given text was implemented successfully