**Experiment No.: 1 Date: 13/10/2020**

**LEX**

**Aim**: To study LEX and write a LEX program to find if the input is an integer, real number, or word.

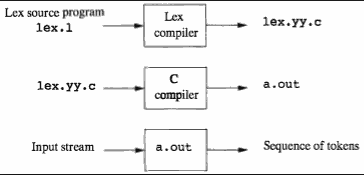
**Theory:**

A lexical analyzer takes input streams and tokenizes it, i.e. divides up into lexical tokens. This division of units (tokens) is known as lexical analysis. A token is a sequence of characters having a collective meaning. Pattern is a rule that describes a set of strings in the input for which the same token is produced as output. The pattern is said to match each string in the set. A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Lex is a software tool for building lexical analyzers/lexers/scanners. The tool is referred to as the Lex compiler and its input specification is referred to as the Lex language. The Lex compiler transforms the input patterns into a transition diagram and generates code that simulates this transition diagram.

Lex is a pattern-action language for specifying lexical analyzers. In this language, patterns are specified by regular expressions, and a compiler for Lex can generate an efficient finite-automaton recognizer for the regular expressions. A major advantage of a lexical analyzer generator is that it can utilize the best-known pattern matching algorithms and thereby create efficient lexical analyzers for people who are not experts in pattern-matching techniques.

Creating a lexical analyzer with Lex



A specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c. The program lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expressions of lex.l, together with a standard routine that uses the table to recognize the lexemes. The actions associated with regular expressions in lex.l are pieces of C code and are carried over directly to lex.yy.c. Finally lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

A Lex program has the following form:

Declarations

%%

Translation rules

%%

Auxiliary functions

The declaration section includes declarations of variables, manifest constants (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions.

The translation rules each have the form

Pattern {Action}

Each pattern is a regular expression (notation suitable for describing tokens) which may use the regular definitions of the declaration section. The actions are fragments of code which are to be executed whenever a token specified by the corresponding regular expression is recognized. An action passes an indication of the token found to the parser and makes an entry in the symbol table.

The third section holds whatever additional functions are used in the actions.

The lexical analyzer L created by Lex reads its input, one character at a time, until it has found the longest prefix of the input which matches one of the regular expressions. Once L has found that prefix, L removes it from the input and places it in a buffer called TOKEN. L then executes the action. If none of the regular expressions denoting the tokens matches any prefix of the input an error has occurred. If two or more patterns match the same longest prefix of the remaining input, L will break the tie in favor of that token which came first in the list of translation rules.

Whenever the scanner matches a token, the text of the token is stored in the null terminated string yytext and its length in yyleng. yylex() is an entry point to the scanner and is created from the rules section. All C codes in the rules section are copied into yylex.

**Task:**

Connect to the Linux server using PuTTY. Host Name = 172.16.40.10, Port = 22 (Save the session). Login with roll no. and password = pcce.

Create a folder for your work (optional). Create a \*.l file using the vi editor.

vi prog.l

Enter ‘i’  -> to edit the files

ESC :wq  -> to save and quit the editor

lex prog.l -> to compile the Lex program

ls -> to check if lex.yy.c is generated

gcc lex.yy.c  -> to compile the generated C program. [ gcc lex.yy.c –o lex1]

./a.out           -> to run the executable [./lex1]

 --------------------------------------------------------------------------------------------

**PROGRAM 1:**

root@kali:~# vi program1

%{

#include<stdio.h>

%}

%%

[0-9]+      printf("Number\n");

[a-zA-Z]\*   printf("Word\n");

%%

int main()

{

        yylex();

        return 0;

}

int yywrap(){}

root@kali:~# lex program1

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

**OUTPUT 1:**

root@kali:~# ./a.out

18

Number

17ce18

Number

Word

Number

Prajwal

Word

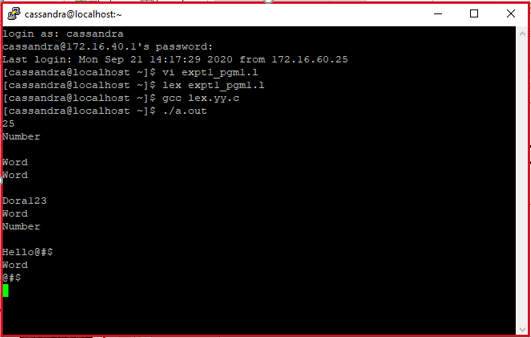
#####$$$$&&&

#####$$$$&&&

^Z

[1]+ Stopped ./a.out

**OUTPUT 2:**



 --------------------------------------------------------------------------------------------

**PROGRAM: 2**

root@kali:~# vi program2

%{

#include<stdio.h>

%}

%%

-?[0-9]+                           printf("Integer\n");

[a-zA-Z][a-zA-Z]\*           printf("Word\n");

-?[0-9]+["."]?[0-9]+       printf("Real Number\n");

%%

int main()

{

        yylex();

        return 0;

}

int yywrap(){}

root@kali:~# lex program2

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

OUTPUT 1:

root@kali:~# ./a.out

-10

Integer

-0.5

Real Number

himaam

Word

trying space

Word

Word

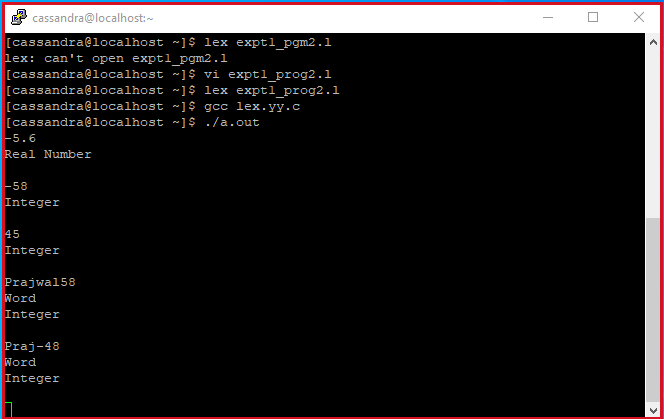
cassandra-maam-123456789

Word

-Word

Integer

**OUTPUT 2:**



**Conclusion:**  Lex programs were successfully implemented