ASSIGNMENT3

Name – Masood Ismail Tamboli
Roll no – 223081
GR.No - 22020190
Batch – C3

AIM: Write a program using LEX specifications to implement lexical analysis phase of compiler to count no of words, lines and characters of given input file.

Theory:

Lex is a tool for building lexical analyzers or lexers. A lexer takes an arbitrary input stream and tokenizes it, i.e., divides it up into lexical tokens. This tokenized output can then be processed further, usually by yacc, or it can be the "end product." In Chapter 1 we demonstrated how to use it as an intermediate step in our English grammar. We now look more closely at the details of a lex specification and how to use it; our examples use lex as the final processing step rather than as an intermediate step which passes information on to a yacc-based parser.

When you write a lex specification, you create a set of patterns which lex matches against the input. Each time one of the patterns matches, the lex program invokes C code that you provide which does something with the matched text. In this way a lex program divides the input into strings which we call tokens. Lex itself doesn't produce an executable program; instead it translates the lex specification into a file containing a C routine called yylex( ). Your program calls yylex( ) to run the lexer.

Using your regular C compiler, you compile the file that lex produced along with any other files and libraries you want. (Note that lex and the C compiler don't even have to run on the same computer. The authors have often taken the C code from UNKX lex to other computers where lex is not available but C is.)

Regular Expressions

Before we describe the structure of a lex specification, we need to describe regular expressions as used by lex. Regular expressions are widely used within the UNIX environment, and lex uses a rich regular expression language.

A regular expression is a pattern description using a "meta" language, a language that you use to describe particular patterns of interest. The characters used in this metalanguage are part of the standard ASCII character set used in UNIX and MS-DOS, which can sometimes lead to confusion. The characters that form regular expressions are

$ Matches the end of a line as the last character of a regular expression

* Matches zero or more copies of the preceding expression.

[] A character class which matches any character within the brackets. If the first character is a circumflex ("^") it changes the meaning to match any character except the ones within the brackets. A dash inside the square brackets indicates a character range, e.g., "[0-91" means the same thing as "[0123456789In. A "-" or "I" as the first character after the "[" is interpreted literally, to let you include dashes and square brackets in character classes. POSIX introduces other special square bracket constructs useful when handling
non-English alphabets. See Appendix H, POSZX Lex and Yacc, for details. Other metacharacters have no special meaning within square brackets except that C escape sequences starting with "\" are recognized. A Matches the beginning of a line as the first character of a regular expression. Also used for negation within square brackets.

$ Matches the end of a line as the last character of a regular expression.

{} Indicates how many times the previous pattern is allowed to match when containing one or two numbers

\ Used to escape metacharacters, and as pan of the usual C escape sequences, e.g., "\nn is a newline character, while "*" is a literal asterisk.

+ Matches one or more occurrence of the preceding regular expression

? Matches zero or one occurrence of the preceding regular expression.

" . . . " Interprets everything within the quotation marks literally-metacharacters other than C escape sequences lose their meaning.

() Groups a series of regular expressions together 'into a new regular expression.

A lex specification consists of three sections: a definition section, a rules section, and a user subroutines section. The first section, the definition section, handles options lex will be using in the lexer, and generally sets up the execution environment in which the lexer operates

The section bracketed by "%{" and "%I" is C code which is copied verbatim into the lexer. It is placed early on in the output code so that the data definitions contained here can be referenced by code within the rules section. In our example, the code block here declares three variables used within the program to track the number of characters, words, and lines encountered.

The rules section contains the patterns and actions that specify the lexer. The beginning of the rules section is marked by a "%%". In a pattern, lex replaces the name inside the braces {} with substitution, the actual regular expression in the definition section.

When our lexer recognizes a newline, it will increment both the character count and the line count. Similarly, if it recognizes any other character it increments the character count. For this lexer, the only "other characters" it could recognize would be space or tab; anything else would match the first regular expression and be counted as a word.

The lexer always tries to match the longest possible string, but when there are two possible rules that match the same length, the lexer uses the earlier rule in the lex specification.

The third and final section of the lex specification is the user subroutines section. Once again, it is separated from the previous section by "%%". The user subroutines section can contain any valid C code. It is copied verbatim into the generated lexer. Typically this section contains support routines.

The **yylex**() as written by **Lex** reads characters from a FILE * file pointer called yyin. If you do not set yyin, it defaults to standard input. It outputs to yyout, which if unset defaults to stdout. You can also modify yyin in the yywrap() function which is called at the end of a file.

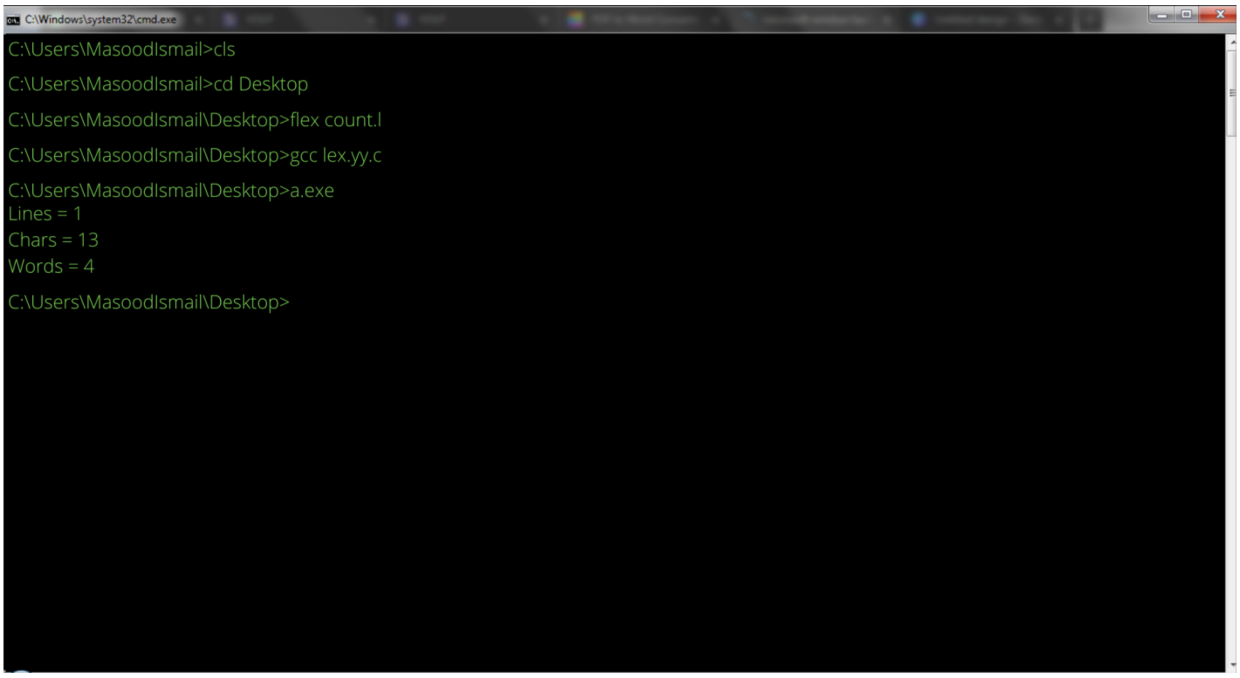
Function **yywrap** is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required. Every C program requires a main function. In this case we simply call yylex that is the main entry-point for lex .

yyin is a variable of the type FILE* and points to the input file. **yyin** is defined by LEX automatically. If the programmer assigns an input file to **yyin** in the auxiliary functions section, then **yyin** is set to point to that file.

Code: -

```
%{
int nlines,nwords,nchars;
%}
%%
\n {
nchars++;nlines+
+;
}
[^ \n\t]+ {nwords++, nchars=nchars+yyleng;}
. {nchars++;}
%%
int yywrap(void)
{
return 1;
}
int main(int argc, char*argv[])
{yyin=fopen("Line.txt","r"
); yylex();
printf("Lines =
%d\nChars=%d\nWords=%d\n",nlines,nchars,nwords); return 0;
}
```

Output: -



```
C:\Users\MasoodIsmail>cls

C:\Users\MasoodIsmail>cd Desktop

C:\Users\MasoodIsmail\Desktop>flex count.l

C:\Users\MasoodIsmail\Desktop>gcc lex.yy.c

C:\Users\MasoodIsmail\Desktop>a.exe
Lines = 1
Chars = 13
Words = 4

C:\Users\MasoodIsmail\Desktop>
```