EXPERIMENT NO: 08

1. Title:

Write a program using Lex specifications to implement lexical analysis phase of compiler to total no's of words, chars and line etc. of given file.

2. Objectives:

- To understand LEX Concepts
- To implement LEX Program for nos of count
- To study about Lex & Java
- To know important about Lexical analyzer

3. Problem Statement:

Write a program using Lex specifications to implement lexical analysis phase of compiler to count nos. of words, chars and line of the given program/file.

4. Outcomes:

After completion of this assignment students will be able to:

- Understand the concept of LEX Tool
- Understand the lexical analysis part
- It can be used for data mining concepts.

5. Software Requirements:

- LEX Tool (flex)

6. Hardware Requirement:

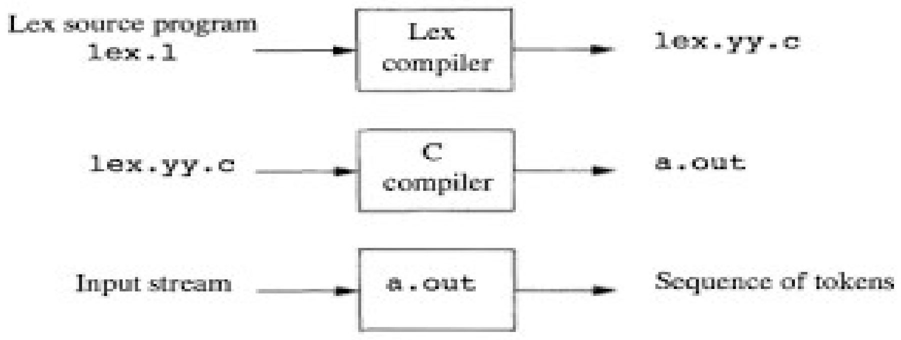
- M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

7. Theory Concepts:

Lex stands for Lexical Analyzer. Lex is a tool for generating Scanners. Scanners are programs that recognize lexical patterns in text. These lexical patterns (or regular Expressions) are defined in a particular syntax. A matched regular expression may have an associated action. This action may also include returning a token. When Lex receives input in the form of a file or text, it takes input one character at a time and continues until a pattern is matched, then lex performs the associated action (Which may include returning a token). If, on the other hand, no regular expression can be matched, further processing stops and Lex displays an error message.

Lex and C are tightly coupled. A .lex file (Files in lex have the extension .lex) is passed through the lex utility, and produces output files in C. These file(s) are coupled to produce an executable version of the lexical analyzer.

Lex turns the user‟s expressions and actions into the host general –purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.



Overview of Lex Tool

During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value.

The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

```
letter(letter|digit)*
```

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- repetition, expressed by the "*" operator
- alternation, expressed by the "|" operator
- concatenation

Anyregular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state and one or more final or accepting states.

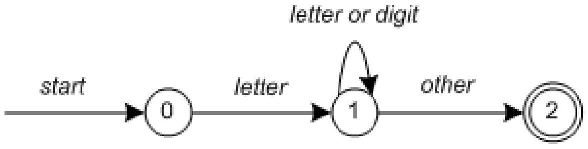


Figure 2: Finite State Automaton

In Figure 3 state 0 is the start state and state 2 is the accepting state. As characters are read we make a transition from one state to another. When the first letter is read we transition to state 1. We remain in state 1 as more letters or digits are read. When we read a character other than a letter or digit we transition to accepting state 2. AnyFSA may be expressed as a computer program. For example, our 3-state machine is easily programmed:

```
start: goto state0

state0: read c
    if c = letter goto state1
    goto state0

state1: read c
    if c = letter goto state1
    if c = digit goto state1
    goto state2

state2: accept string
```

This is the technique used by lex. Regular expressions are translated by lex to a computer program that mimics an FSA. Using the next inputcharacter and current statethe next state is easily determined by indexing into a computer-generated state table.

Now we can easily understand some of lex‟s limitations. For example, lex cannot be used to recognize nested structures such as parentheses. Nested structures are handled by incorporating a stack. Whenever we encounter a " (" we push it on the stack. When a ")" is encountered we match it with the top of the stack and pop the stack. However lex only has states and transitions between states. Since it has no stack it is not well suited for parsing nested structures. Yacc augments an FSA with a stack and can process constructs such as parentheses with ease. The important thing is to use the right tool for the job. Lex is good at pattern matching. Yacc is appropriate for more challenging tasks.

- Regular Expression in Lex:-

A Regular expression is a pattern description using a meta language. An expression is made up of symbols. Normal symbols are characters and numbers, but there are other symbols that have special meaning in Lex. The following two tables define some of the symbols used in Lex and give a few typical examples.

- Defining regular expression in Lex:-

Character	Meaning
A-Z, 0-9,a-z	Character and numbers that form of the pattern.
.	Matches any character except \n.
-	Used to denote range. Example: A-Z implies all characters from A to Z.

[]	A character class. Matches any character in the brackets. If character is ^ then it indicates a negation pattern. Example: [abc] matches either of a,b and c.
*	Matches zero or more occurences of the preceiding pattern.
+	Matches one or more occurences of the preceiding pattern.
?	Matches zero or one occurences of the preceiding pattern.
$	Matches end of line as the last character of the pattern.
{}	Indicates how many times a pattern can be present. Example: A {1, 3} implies one or three occurences of A may be present.
\	Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table.
^	Negation
|	Logical OR between expressions.
“<some symbols>”	Literal meaning of characters. Meta characters hold.
/	Look ahead matches the preceding pattern only if followed by the succeeding expression. Example: A0/1 matches A0 only if A01 is the input.
()	Groups a series of regular expressions.

- Examples of regular expressions

Regular expression	Meaning
Joke[rs]	Matches either jokes or joker
A {1,2}shis+	Matches Aashis, Ashis, Aashi, Ashi.
(A[b-e])+	Matches zero or one occurrences of A followed by any character from b to e.

Tokens in Lex are declared like variable names in C.Every token has an associated expression.(Examples of tokens and expression are given in the following table). Using the examples in our tables, we‟ll build a word-counting program. Our first task will be to show how tokens are declared.

- Examples of token declaration

Token	Associated expression	Meaning
Number	([0-9])+	1 or more occurences of a digit
Chars	[A-Za-z]	Any character
Blank	“”	A blank space
Word	(chars)+	1 or more occurences of chars
Variable	(chars)+(number)*(chars)*(number)*	

Programming in Lex:-

Programming in Lex can be divided into three steps:

1. Specify the pattern-associated actions in a form that Lex can understand.

2. Run Lex over this file to generate C code for the scanner.

3. Compile and link the C code to produce the executable scanner.

Note:If the scanner is part of a parser developed using Yacc, only steps 1 and 2 should be performed.

A Lex program is divided into three sections: the first section has global C and Lex declaration, the second section has the patterns (coded in C), and the third section has supplement C functions. Main (), for example, would typically be founding the third section. These sections are delimited by %%.so, to get back to the word to the word-counting Lex program; let‟s look at the composition of the various program sections.

Table 1: Special Characters

Pattern	Matches
.	any character except newline
\.	literal .
\n	newline
\t	tab
^	beginning of line
$	end of line

Table 2: Operators

Pattern	Matches
?	zero or one copy of the preceding expression
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
a|b	a or b (alternating)
(ab)+	one or more copies of ab (grouping)
abc	abc
abc*	ab abc abcc abccc ...
"abc*"	literal abc*
abc+	abc abcc abccc abcccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc

Table 3: Character Class

Pattern	Matches
[abc]	one of: a b c
[a-z]	any letter a through z
[a\-z]	one of: a - z
[-az]	one of: - a z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: a b
[a^b]	one of: a ^ b
[a|b]	one of: a | b

Regular expressions are used for pattern matching. A character class defines a single character and normal operators lose their meaning. Two operators supported in a character class are the hyphen (" - ") and circumflex ("^"). When used between two characters the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string the longest match wins. In case both matches are the same length, then the first pattern listed is used.

```
... definitions

%%
.
.. rules..

%%

... subroutines
```

Input to Lex is divided into three sections with %% dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file:

```
%%
```

Input is copied to output one character at a time. The first %% is always required as there must always be a rules section. However if we don"t specify any rules then the default action is to match everything and copy it to output. Defaults for input and output are stdin and stdout, respectively. Here is the same example with defaults explicitly coded:

```
%%
/* match everything except newline */
. ECHO;
/* match newline */
\n  ECHO;

%%
```

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

int main(void) {
yylex();
return 0;
}
```

Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by whitespace (space, tab or newline) and an optional action associated with the pattern. The action may be a single Cstatement, or multiple C statements, enclosed in braces. Anything not starting in column one is copied verbatim to the generated C file. We may take advantage of this behavior to specify comments in our lex file. In this example there are two patterns, "   ." and "\n", with an ECHO action associated for each pattern. Several macros and variables are predefined by lex. ECHO is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, ECHO is defined as:

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

Variable yytext is a pointer to the matched string (NULL-terminated) and yyleng is the length of the matched string. Variable yyout is the output file and defaults to stdout. 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 . Some implementations of lex include copies of main and yywrap in a library thus eliminating the need to code them explicitly. This is why our first example, the shortest lexprogram, functioned properly.

Table 4: Lex Predefined Variables

Name	Function
int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
int yywrap(void)	wrapup, return 1 if done, 0 if not done
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN condition	switch start condition
ECHO	write matched string

Here is a program that does nothing at all. All input is matched but no action is associated with any pattern so there will be no output.

```
%%
.
\n
```

The following example prepends line numbers to each line in a file. Some implementations of lex predefine and calculate yylinenoThe input file for lex is yyinand defaults to stdin

```
%{
  int yylineno;
%}
%%
^(.*)\n printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
  yyin = fopen(argv[1], "r");
  yylex();
  fclose(yyin);
}
```

- Global C and Lex declaration

  In this section we can add C variable declaration. We will declare an integer variable here for our word-counting program that holds the number of words counted by the program. We‟ll also perform token declaration of Lex.

- Declaration for the word-counting program

```
%{
int wordcount=0;
%}
Chars [A-za-z\_\‟\.\"]
Number ([0-9]) +
Delim [“”\n\t]
Whitespace {delim} +
Words {chars} +
%%
```

  The double percent sign implies the end of this section and the beginning of the three sections in Lex programming.

- Lex rules for matching patterns:

  Let‟s look at the lex rules for describing the token that we want to match.(well use c to define to do when a token is matched).continuing with word counting program,here are the rules for matching token.

- Lex rules for word counting program:

{words}{word count++; /*
Increase the word count by one*/}
{whitespace}{/*do
Nothing*/}
{Number}{/*one may want to add some processing here*/}
%%

- C code

The third and final section of programming in lex cover c function declaration (and occasionally the main function) Note that this section has to include the yywrap() function. Lex has set the function and variable that are available to the user. One of them is yywrap.Typically, yywrap () is define as shown in the example below.

C code section for word counting program

```
void main()
{
Yylex();/*start the analysis*/
printf("No of words:%d\n",wordCount);
}
int yywrap()
{
return1;
}
```

In the processing section we have the basic element of lex programming.which should help in the writing simple lexical analysis programs.

- Putting it all together

The lex file is Lex scanner. It is represented to lex program as
$ lex <file name.lex>

This produce the lex.yy.c file which can be compiled using a C compile. It can also be used with parser to produce executable or you can include the Lex library in the link step with the option A-11.

- Here some of Lex's flags:
  - -c Indicate C action and is the default.
  - -t causes the lex.yy.c program to be written instead to standard output.
  - -v Provide a two-line summary of statistic.
  - -n will not print out the –v summary.

- Lex variable and Function

Lex has several functions and variable that provides different information and can be used to build programs that can perform complex function. Some of these variable and function along with their uses are listed in the following table.

Yyleng	Give the length of the match pattern

Yylineno	Provide the current line number information. (May or may not b supported by the lexer.)

- Lex variables:

Yyin	Of the type FILE*.This point to the current file being parsed by the lexer
Yyout	Of the type FILE*.This points location where the output of the lexer will be written. By default, both yyin and yyout point to standard input and output.
Yytext	The text of the matched pattern is stored in this variable (char*).

- Lex functions:

yylex()	The function that start the analysis. It is automatically generated by Lex.
Yywrap	This function is called when the file is encountered. If this function returned 1.The parsing stops. So, this can be used to parse multiple files .code can be written in the third section, which will allow multiple file to be parsed. The strategy is to make yyin file pointer (see the preceding table)point to different file until all file are parsed .At the end, yywrap() can return 1 to indicate end of the parsing
yyless(int n)	This function can be used to push back all but first „n" character of the read token.
yymore()	This function tells the lexer to append the next token to the current token

Examples:

The following example prepend line number to n each line in the file. Some implementations of the lex predefine &calculate yylineno. The input file for lex is yyin, and default to stdin.

```
%{
Int yylineno;
%}
%%
 ^ (.*)\n printf (“%s”,++yylineno, yytext);
%%
I nt main (int argc,char *argv[]) {
 yyin=fopen (argv [1],”r”);
 yylex ();
 fclose (yyin);
}
```

Here is a scanner that the number of character, words, and lines in a file

```
%{
```

```
int nchar,nword,nline;
%}
%%
\n{nline++;nchar++ ;}
[^\t\n]+ {nword++, nchar+=yyleng ;}
. {nchar++ ;}
%%
int main(void){
yylex ();
Printf (“%d\t%d\t%d\n”, nchar, nword,nline);
Return 0;
}
```

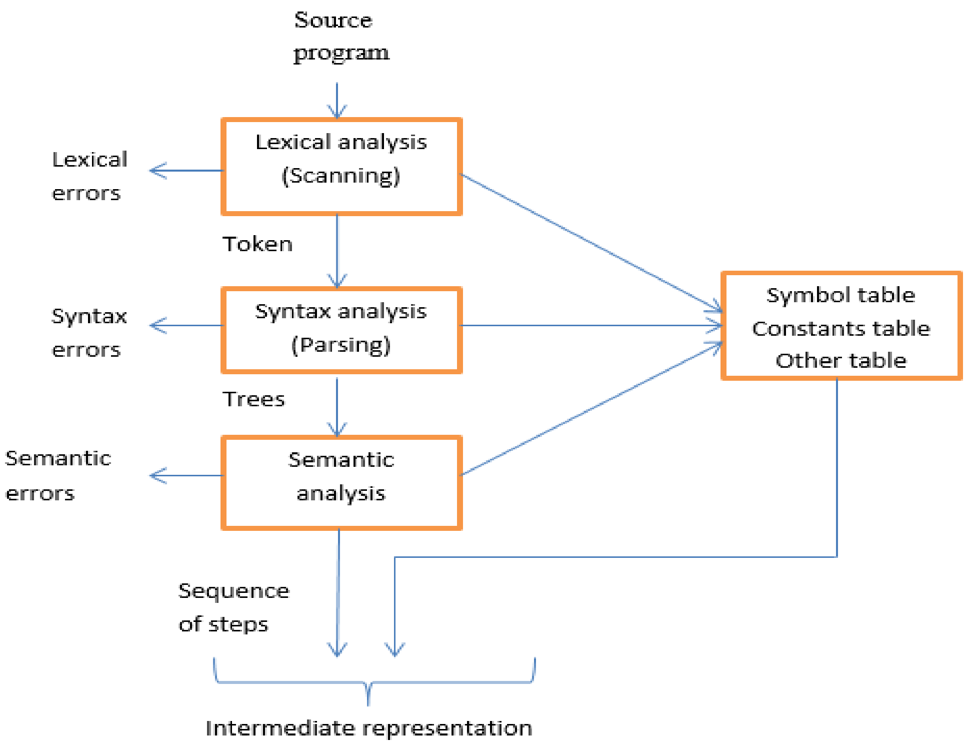
HOW THE INPUT IS MATCHED

When the generated scanner is run, it analyzes its input looking for strings, which match any of its patterns. If it finds more than one match, it takes the one matching the most text. If it finds two or more matches of the same length, the rule listed first in the flex input file is chosen.

Once the match is determined, the text corresponding to the match(called the token)is made available in the global character pointer „yytext‟,and its length in the global integer „yyleng‟.The actioncorresponding to the matched pattern is then executed,and then the remaining input is scanned for another match.

If no match is found, then thedefault rule executed: the next character in the input is considered matched and copied to the standard output.

8. Design (architecture) :



Front end of toy compiler

9. Algorithms(procedure) :

Note: you should write algorithm & procedure as per program/concepts

10.Flowchart :

Note: you should draw flowchart as per algorithm/procedure

11.Conclusion:

Thus, I have studied lexical analyzer and implemented an application for lexical analyzer to count total number of words, chard and line etc

References :

https://en.wikipedia.org/wiki/Lex_(software)
http://epaperpress.com/lexandyacc/prl.html
https://www.ibm.com/developerworks/library/l-lexyac/index.html

Oral Questions: [Write short answer]

1. What is Lex.
2. What is Compiler and phases of compiler.
3. What is Lex specification.
4. What is the difference between Lex and YACC.
5. What is Regular Expression.
6. How to run a Lex program.
7. What is yytext, yyin, yyout.
8. What is yywrap().
9. What is yylex().
10. token, lexemes, pattern?