Group B-ASSIGNMENT NUMBER: 03

TITLE : Lexical analysis to count number of words,lines and characters.

PROBLEM STATEMENT:

Working of a compiler with their phases and Java programming.
OBJECTIVES:

- Analyze source code
- Identify tokens required in the lexical analysis process.
- Count the lines, words and characters

OUTCOMES:
The students will be able to

- Count number of lines, words and characters

THEORY:

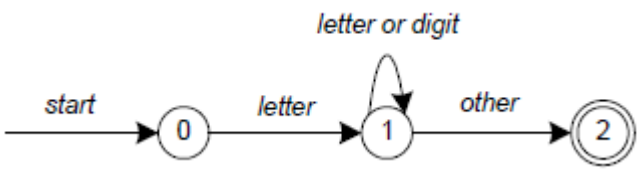
During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. 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

Any regular 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.



Finite State Automate

In Figure, 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. *Any* FSA 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 *input* character and *current state* the 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.

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

Table 1: Special Characters

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
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
abc	abc
abc*	ab abc abcc abccc ...
"abc*"	literal abc*
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc

Table 2: Operators

Pattern	Matches
[abc]	one of: a b c
[a-z]	any letter a-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

Table 3: Character Class

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

Table 4: Lex Predefined Variables

Regular expressions are used for pattern matching.
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 C statement, 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 lex program, functioned properly.

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 **yylineno**. The input file for lex is **yyin** and 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);
}
```

The definitions section is composed of substitutions, code, and start states. Code in the definitions section is simply copied as-is to the top of the generated C file and must be bracketed with "**%{**" and "**%}**" markers. Substitutions simplify pattern-matching rules. For example, we may define digits and letters:

```
digit [0-9]
letter [A-Za-z]
%{
int count;
%}
%%
/* match identifier */
{letter}({letter}|{digit})* count++;
%%
int main(void) {
yylex();
printf("number of identifiers = %d\n", count);
return 0;
}
```

Whitespace must separate the defining term and the associated expression. References to substitutions in the rules section are surrounded by braces (**{letter}**) to distinguish them from literals. When we have a match in the rules section the associated C code is executed. Here is a scanner that counts the number of characters, words, and lines in a file (similar to Unix wc):

```
%{
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;
}
```

CONCLUSION : By performing this assignment we have successfully counted the numbe of words,lines and characters in the given input file.