ASSIGNMENT NO:1.

Title: Lexical analyzer for HLL using LEX.

Problem Statement:-

Implement a Lexical Analyzer using LEX for a subset of C. Cross check your output with Standford LEX.

Objective:

1. Appreciate the role of lexical analysis phase in compilation
2. Understand the theory behind design of lexical analyzers and lexical analyzer generator.
3. Be able to use LEX to generate lexical analyzers.

S/W packages & Hardware apparatus & p used

1. 64 bit latest Computer.
2. Keyboard.
3. Mouse.
4. 40 GB HDD.

Theory:-

Regular Expression in lex

"..." - Any string enclosed in double-quotes shall represent the characters within the double-quotes as themselves, except that backslash escapes.

<state>r, <state1, state2, ...>r - The regular expression r shall be matched only when the program is in one of the start conditions indicated by state, state1, and so on.

r/x - The regular expression r shall be matched only if it is followed by an occurence of regular expression x.

* - matches zero or more occurences of preceding expressions.

[] - a character class which matches any character within the brackets. If the first character is circumflex '^' it changes the meaning to match any character except the ones with brackets.

^ - matches the beginning of a line as the first characters of a regular expression. Also used for negation within square brackets.

{} - indicates how many times the previous pattern is allow to match, when containing one or two numbers.

$ - matches the end of line as the last character of a regular expressions.

\ - used to escape metacharacters & a part of usual c escape sequences eg. '\n' is a newline

'*' - is a literal asterik

+ - matches one more occurences of the preceding regular expression.

+ matches one or more occurences of the preceding regular expressions.

() - groups a series of regular expressions together, into a new regular expression.

Examples:

```
DIGIT [0-9]+
IDENTIFIER [a-z A-Z][a-z A-Z 0-9]*
```

The functions or macros that are accessible to user code:

int yylex(void)

Performs lexical analysis on the intput; this is the primary function generated by the lex utility. The function shall return zero when the end of input is reached; otherwise, it shall return non-zero values (token) determined by the actions that are selected.

int yymore(void)

When called, indicates that when the next input string is recognized, it is to be appended to the current value of yytext rather than replacing it; the value in yyleng shall be adjusted accordingly.

int yyless(int n)

Retains n initial characters in yytext, NUL-terminated & treats the remaining characters as if they had not been read; the value in yyleng shall be adjusted accordingly.

int yywrap(void)

Called by yylex() at end of file; the default yywrap() shall always return 1. If the application requires yylex() to continue processing with another source of input, then the application can include a function yywrap(), which associates another file with the external variable FILE * yyin & shall return a value of zero.

int main(int argc, char *argv[])

Calls yylex() to perform logical analysis, then exits. The user code can contain main() to perform appl^{n} specific operations, calling yylex() as applicable.

Algorithm:

1. Accept input filename (file is in HLL) as a command line argument.

2. Separate the tokens as identifiers, constants, keywords, etc, and fill the generic symbol table.

3. Check for lexical errors and give error messages if needed.

Test input:

```
#include<stdio.h>
void main()
{ int a,b;
  char bilateral;
  a=3;
  c=a+d;
}
```

Test o/p:

Lexeme	Token
#include<stdio.h>	Directive
void	Reserved
main()	Reserved
{	Punctuation
int	keyword
a	Identifier
b	Identifier
;	Delimiter
char	Reserved
bilateral	Lexical error
;	Delimiter
a	Identifier
=	Operator
3	Constant
;	Delimiter
c	Identifier
=	Operator
a	Identifier
+	Operator
d	Identifier
;	Delimiter
}	Punctuation

Symbol table:

Symbol name	Type	Value
a		
b		
c		
d		

Conclusion:-

Hence, implemented lexical analyzer for HLL using LEX.