DATE:
EX. NO: 01

IMPLEMENTATION OF A SYMBOL TABLE:

AIM:

To write a program to implement the basic symbol table for a compiler using C.

SYMBOL TABLE:

A symbol table is an important data structure used in a compiler. It is used to store the information about the occurrence of various entities such as objects, classes, variable names, interfaces, function names etc. It is used by both the Analysis and Synthesis phases of a compiler. A symbol table can either be linear or a hash table.

- It is used to store the name of all the entities in a structured form in one place.
- It is used to verify if a variable has been declared already.
- It is used to determine the scope of a name.
- It is used to implement type checking by verifying assignments and expressions in the source code are semantically correct.

ALGORITHM:

1. Start
2. Open the file using a file pointer.
3. Copy the content of the file, line by line and store it in an array of strings.
4. Repeat the below-given steps for the number of lines scanned in the file.
5. Trace through each line of the array of string, until a white space (“ ”) is encountered. If it is not encountered, continue to the next iteration.
6. If white space is detected, copy the token to the type variable for the symbol table.
7. Now split the remaining tokens of the string with the occurrence of “commas (,), semi-colons (;), line-breaks (\n), and white spaces (‘ ’).”
8. Check whether the split token is a data type, if yes then copy it to the type variable. Else, print the variable identifier, data types and its address.
9. Stop.

SOURCE CODE:

DOCUMENT: **file.txt**

```
int a, b;
float c, d;
double e; char f; long g;
```

C DOCUMENT: **symbol_table.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   FILE *fp = fopen("file.txt", "r");

   char line[100][100];
   char data_types[10][10] = {"int", "float", "char", "double", "long"};
   char types[100];

   int n = 0;
   while (fgets(line[n], 100, fp))
      n++;

   int i = 0;
   printf("\n IDENTIFIERS \t ADDRESS LOCATIONS \t DATATYPES");
   for (i = 0; i < n; i++)
   {
      char *str = strtok(line[i], " ");

      if (strcmp(data_types[0], str) == 0 || strcmp(data_types[1], str) == 0 ||
      strcmp(data_types[2], str) == 0 || strcmp(data_types[3], str) == 0 || strcmp(data_types[4],
      str) == 0)
      {
         strcpy(types, str);

         while (str = strtok(NULL, " ,\n;"))
         {
                  if (strcmp(data_types[0], str) == 0 || strcmp(data_types[1], str) == 0 ||
                  strcmp(data_types[2], str) == 0 || strcmp(data_types[3], str) == 0 ||
                  strcmp(data_types[4], str) == 0)
                           strcpy(types, str);

                  else
                           printf("\n %s \t\t %p \t\t %s", str, str, types);
         }
      }
   }

   return 0;
}
```

SAMPLE OUTPUT:

IDENTIFIERS	ADDRESS LOCATIONS	DATATYPES
a	0061D804	int
b	0061D807	int
c	0061D86A	float
d	0061D86D	float
e	0061D8CF	double
f	0061D8D7	char
g	0061D931	long

RESULT:

Thus, the symbol table is implemented using C successfully.

DATE:
EX. NO: 02

IMPLEMENTATION OF A LEXICAL ANALYZER:

AIM:

To write a program to implement the basic lexical analyzer for a compiler using C.

LEXICAL ANALYSIS:

- Lexical Analysis is the very first phase in Compiler Designing.
- It takes modified source code from language pre-processors that are written in the form of sentences.

LEXICAL ANALYZER (OR) LEXER:

- A lexer takes the given source code and converts the available sequence of characters into a sequence of tokens.
- It is the interface between the source program and the compiler. It breaks the syntax into a series of tokens.
- It reads the source program one character at a time, carving the source program into a sequence of atomic units called, *tokens.*
- Each token represents a sequence of characters that can be treated as a single logical entity. Identifiers, Keywords, Constants, Operators, Punctuators, and Literals are typical tokens.

ALGORITHM:

1. Start
2. Open the file using a file pointer.
3. Copy the content of the file, line by line and store it in an array of strings.
4. Print the table headers.
5. Repeat the below given steps for each line scanned.
6. Set NULL as the initial value for all flags and start scanning until the given delimiter is found.
7. If '#' exists, it is a preprocessor statement. If ')' exisits, then it is a function header.
8. If it matches with an assumed keyword, it is a keyword.
9. If none is matched, check for operators, non-operators, and literals and mark them.
10. If it is not all of them, then mark it as an identifier.
11. Print the values.
12. Continue for the next iteration.
13. Stop.

SOURCE CODE:

DOCUMENT: **input.c**

```
#include<stdio.h>
void main()
{
   int a = 5;
   char str[] = "Srinivasan\0";
   float b = 7.0;
   printf("%f",a+b);
}
```

C DOCUMENT: **lexical_analyzer.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

int is_number(char buffer[])
{
   int i;
   for (i = 0; buffer[i] != '\0'; i++)
   {
      if (isdigit(buffer[i]))
         return 1;
   }
   return 0;
}

int main()
{
   FILE *fp = fopen("input.c", "r");

   if (fp == NULL)
   {
      printf("\n Error Opening the File");
      exit(0);
   }

   char line[100], temp[100], preprocessor[100], keyword[100], function[100],
identifier[100], operator[100], non_operators[100], literals[100];

   char operator_list[13][100] = {"+", "-", "*", "/", "=", "%", ".", ">", "<", "^", "|", "&", "!"};
   char non_operator_list[10][100] = {"{", "}", "(", ")", ";", ",", ":", "?", "_", "$"};
```

```
   char key_word[32][10] = {"auto", "break", "case", "char", "const", "continue", "default",
"do", "double", "else", "enum", "extern", "float", "for", "goto", "if", "int", "long", "register",
"return", "short", "signed", "sizeof", "static", "struct", "switch", "typedef", "union",
"unsigned", "void", "volatile", "while"};

   printf("----------------------------------------------------------------------------------------------------
-----------------------------\n");
   printf("LINES\t| PREPROCESSOR\t\t | KEYWORD\t| FUNCTION\t | IDENTIFIER\t |
OPERATOR\t | NON-OPERATOR\t | LITERALS\n");
   printf("----------------------------------------------------------------------------------------------------
-----------------------------\n");
   int i=0, j=0, k=0, l=0;

   while (fgets(line, 100, fp))
   {
      i++;
      char *str = strtok(line, " \n");
      while (str != NULL)
      {
         strcpy(preprocessor,"\t\t ");
         strcpy(keyword,"\t ");
         strcpy(function,"\t");
         strcpy(identifier,"\t");
         strcpy(operator,"\t");
         strcpy(non_operators,"\t");
         strcpy(literals,"\t");
         strcpy(temp,str);

         for (j = 0; j < 32; j++)
            if (strcmp(temp, key_word[j]) == 0)
               break;

         for (k = 0; k < 13; k++)
            if (strcmp(temp, operator_list[k]) == 0)
               break;

         for (l = 0; l < 10; l++)
            if (strcmp(temp, non_operator_list[l]) == 0)
               break;

         char *pre = strchr (str, '#');
         char *fun = strchr (str, ')');

         if (pre != NULL)
            strcpy (preprocessor,str);
```

```
        else if (fun != NULL)
           strcpy (function, str);

        else if (j != 32)
        {
           strcpy (keyword, str);
           strcat (keyword, "\t");
        }

        else if (k != 13)
        {
           strcpy (operator, str);
           strcat (operator, "\t");
        }

        else if (l != 10)
        {
           strcpy (non_operators, str);
           strcat (non_operators, "\t");
        }

        else if (is_number(temp))
        {
           strcpy (literals, str);
           strcat (literals, "\t");
        }

        else
        {
           strcpy (identifier, str);
           strcat (identifier, "\t");
        }
        printf(" %d\t| %s\t| %s\t| %s\t | %s\t | %s\t | %s\t | %s \n", i,preprocessor, keyword
,function,identifier ,operator ,non_operators ,literals);
        str = strtok(NULL, " \n");
      }
   }
   return 0;
}
```

SAMPLE OUTPUT:

LINES	PREPROCESSOR	KEYWORD	FUNCTION	IDENTIFIER	OPERATOR	NON-OPERATOR	LITERALS
1	#include<stdio.h>						
2		void					
2			main()				
3						{	
4		int					
4				a			
4					=		
4							5;
5		char					
5				str[]			
5					=		
5							"Srinivasan\0";
6		float					
6				b			
6					=		
6							7.0;
7			printf("%f",a+b);				
8						}	

RESULT:

Thus, the lexical analyzer is implemented using C successfully.

DATE:
EX. NO: 3

STUDY AND IMPLEMENTATION OF LEX PROGRAMMING

AIM:

To understand and learn to implement the LEX tool for lexical analysis.

LEX TOOL:

- Lex is a tool that automatically generates the lexical analyzer (Scanners or Lexers).
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.
- LEX tool is commonly used with YACC Parser Generator.

STRUCTURE OF THE LEX PROGRAM:

```
%{
      DECLARATIONS AND DEFINITIONS
%}
GLOBAL DECLARATIONS
%%
      TRANSLATION RULES SECTION
%%
USER SUBROUTINES (OR) AUXILIARY FUNCTIONS
```

DEFINITIONS AND DECLARATIONS:

The lex definitions section can contain any of several classes of items. The most critical are **external definitions, preprocessor statements like #include, and abbreviations.**

- **External Definitions**: The definitions section is where you usually place C definitions of objects accessed by actions in the rules section or by routines with external linkage.

- **Pre-processor Statements**: Pre-processor statements and C source code appear between a line of the form %{ and one of the form %}. This includes files that are important to the lexical analyzer program.

- **Abbreviations**: The definitions section can also contain abbreviations for regular expressions to be used in the rules section. The purpose of abbreviations is to avoid needless repetition in writing your specifications and to provide clarity in reading them. A definition for an "identifier" would often be placed in this section.
  Example: digit [0-9] (or) identifier [a-zA-Z][a-zA-z0-9]*

TRANSLATION RULES SECTION:

Each rule consists of a pattern to be searched for in the input, followed on the same line by an action to be performed when the pattern is matched.

Because lexical analyzers are often used in conjunction with parsers, as in programming language compilation and interpretation, the patterns can be said to define the classes of tokens that may be found in the input.

Pattern specification is organized as below.

```
PATTERN / REGULAR EXPRESSION        { Set Of C Statements That
                                    Define The Actions To Be Taken}
```

RE	DEFINITIONS	EXAMPLES
.	Match any character except '\n'.	
*	Matches 0 or more copies of the preceding expression	[a]* → 0 or more 'a'
+	Matches 1 or more copies of the preceding expression	[a]+ → 1 or more 'a'
?	Matches 0 or 1 copy of the preceding expression	[a]? → 0 or 1 'a'
[ ]	Character Classes – Matches anything inside it.	
[^]	Character Classes – Matches anything except it.	[^a-z] → no 'a' to 'z'
[-]	Character Classes – Indicates a range	[a-z] → between 'a' and 'z'
$	Matches the end of the line for Regular Expressions	
\	It is used to escape meta characters.	Escape Sequences
|	Matches the preceding or the following RE	[a|b] → 'a' or 'b'
" "	Matches the matching string	["aa"] → containing "aa"
( )	Groups the RE into a new Regular Expression	

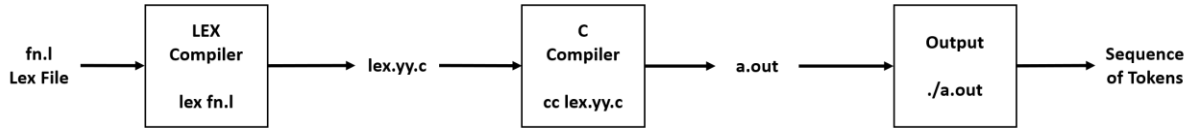
USER SUBROUTINE (OR) AUXILIARY FUNCTIONS:

- The user subroutine is optional.
- If the LEX tool is used alone, then the user sub-routine consists of the main function and the other user-defined methods.
- To start the LEX Scanning, the **yylex()** function should be called in the main function.
- When the LEX tool is used along with the YAAC tool, then the user subroutine section could be left blank.

LEX PREDEFINED VARIABLES AND FUNCTIONS:

NAME	FUNCTION
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of the matched string
`yylval`	the value associated with the token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output-file
`FILE *yyin`	input file

EXECUTION OF THE LEX PROCEDURE:



1. Create a LEX file with the".l" extension using the following command. **vi file_name.l**
2. Write the content of the definition, rule and user subroutine section and save the file.
3. Now compile the LEX file using the LEX compiler. **lex file_name.l**
4. After compiling the LEX file, a new file named "lex.yy.c" will be created in the same directory.
5. Now compile the lex.yy.c file. **cc lex.yy.c**
6. Execute the compiled file. **./a.out**

RESULT:

Thus, the LEX tool is studied and learned to be implemented.

A. Write a LEX Program to determine whether the number is positive or negative.

AIM:

To implement a LEX Program to determine whether the integer is positive or negative.

ALGORITHM:

10. Start
11. Include the declarations in the Declaration sections.
12. Formulate the pattern rule to identify the integer lexemes and print their values.
13. In the main function, call the yylex() function to parse them.
14. Stop.

SOURCE CODE:

```
%option noyywrap
%{
#include <stdio.h>
%}

%%
[+][0-9]+  {printf ("Positive Number");}
[-][0-9]+  {printf ("Negative Number");}
\n return 0;
%%

main()
{
        yylex();
}
```

SAMPLE OUTPUT:

```
Ex_03>          lex integer.l
Ex_03>          gcc lex.yy.c
Ex_03>          ./a.out
        +56
        Positive Number
Ex_03>          ./a.out
        -23
        Negative Number
```

RESULT:

Thus, the LEX program to identify whether the integer is positive or negative is implemented successfully.

B. Write a LEX Program to determine whether the number is odd or even.

AIM:

To implement a LEX Program to determine whether the integer is odd or even.

ALGORITHM:

1. Start
2. Include the declarations in the Declaration sections.
3. Formulate the pattern rule to identify whether the integer lexeme is odd or even and print their values.
4. In the main function, call the yylex() function to parse them.
5. Stop.

SOURCE CODE:

```
%option noyywrap
%{
        #include <stdio.h>
%}

%%
[0-9]*[02468]           {printf ("Even Number!");}
[0-9]+                  {printf ("Odd Number!");}
\n return 0;
%%

main()
{
        yylex();
}
```

SAMPLE OUTPUT:

```
Ex_03>      lex odd_even.l
Ex_03>      gcc lex.yy.c
Ex_03>      ./a.out
        56
        Even Number
Ex_03>      ./a.out
        23
        Odd Number
```

RESULT:

Thus, the LEX program to determine whether the integer is odd or even is implemented successfully.

C. Write a LEX Program to count the number of vowels and consonants in the string.

AIM:

To implement a LEX Program to count the number of vowels and consonants in the string.

ALGORITHM:

1. Start
2. Include the declarations in the Declaration sections.
3. Formulate the pattern rule count the number of vowels and consonants.
4. In the main function, call the yylex() function to parse them.
5. Stop.

SOURCE CODE:

```
%option noyywrap
%{
        #include <stdio.h>
        int vowels = 0, consonants = 0;
%}

%%
[AEIOUaeiou]{vowels++;}
[a-zA-Z]                {consonants++;}
\n return 0;
%%

main()
{
        yylex();
        printf ("\n Vowels: %d", vowels);      printf ("\n Consonants: %d", consonants);
}
```

SAMPLE OUTPUT:

```
Ex_03>      lex alphabets.l
Ex_03>      gcc lex.yy.c
Ex_03>      ./a.out
        Hello World
        Vowels:     3
        Consonants: 7
```

RESULT:

Thus, the LEX program to count the number of vowels and consonants in the string is implemented successfully.

D. Write a LEX Program to check the validity of the identifier:

AIM:

To implement a LEX Program to check the validity of the identifiers.

ALGORITHM:

1. Start
2. Include the declarations in the Declaration sections.
3. Formulate the pattern rule to identify the perfect identifier.
4. In the main function, call the yylex() function to parse them.
5. Stop.

SOURCE CODE:

```
%option noyywrap
%{
        #include <stdio.h>
%}

%%
[a-zA-Z_]+[a-zA-Z0-9_]*     {printf ("Valid Identifier");     return;}
.                           {printf ("Invalid Identifier");   return;}
%%

main ()
{
        yylex();
}
```

SAMPLE OUTPUT:

```
Ex_03>      lex identifier.l
Ex_03>      gcc lex.yy.c
Ex_03>      ./a.out
        alpha
        Valid Identifier
Ex_03>      ./a.out
        5av
        Invalid Identifier
```

RESULT:

Thus, the LEX program to identify the perfect identifier is implemented successfully.

E. Write a LEX Program to count the number of lines, characters and words in a file:

AIM:

To implement a LEX Program to count the number of lines, characters and words in a file.

ALGORITHM:

1. Start
2. Include the declarations in the Declaration sections.
3. Formulate the pattern rule to identify the line breaks, characters, words, numbers and symbols in a file.
4. In the main function, call the yylex() function to parse them.
5. Stop.

SOURCE CODE:

The input file: text.txt

```
Hello, This is the Input File.
The LEX File will invoke this file using the yyin file pointer.
It will then count the number of lines, characters, symbols and numbers used in the file.
This file contains 4 lines, and 46 words, 194 characters and 9 symbols.
```

The LEX file: count_1.l

```
%option noyywrap

%{
        #include <stdio.h>
        int lines=0, characters = 0, words = 0;
        int symbols = 0, numbers = 0;
%}

%%

[\n]             {lines++;      words++;}
[0-9]            {numbers++;}
[a-zA-Z]         {characters++;}
[' '\t]          {words++;}
.                {symbols++;}

%%
```

```
main ()
{
        yyin = fopen("abc.txt", "r");
        yylex();

        printf ("\n Lines: %d", ++lines);
        printf ("\n Characters: %d", characters);
        printf ("\n Words: %d", words);
        printf ("\n Symbols: %d", symbols);
        printf ("\n Numbers: %d", numbers);
}
```

SAMPLE OUTPUT:

```
Ex_03>      lex count_1.l
Ex_03>      gcc lex.yy.c
Ex_03>      ./a.out
        Lines: 4
        Characters: 194
        Words: 46
        Symbols: 9
        Numbers: 7
```

RESULT:

Thus, the LEX program to count the number of vowels and consonants in the string is implemented successfully.

F. Write a LEX Program to count the number of words starting with a vowel or a numeral in a file:

AIM:

To implement a LEX Program to count the number of words starting with a vowel or an integer or a floating-point number.

ALGORITHM:

1. Start
2. Include the declarations in the Declaration sections.
3. Formulate the pattern rule to count the number of words starting with a vowel or an integer or a floating-point number.
4. In the main function, call the yylex() function to parse them.
5. Stop.

SOURCE CODE:

The input file: text.txt

```
Input
Output
15.25ff
65f
75f
```

The LEX file: count_2.l

```
%option noyywrap

%{
        #include <stdio.h>
        int w = 0, n = 0, f = 0;
%}

%%

[aeiouAEIOU]+[a-zA-Z]*[\n" ",.]*        {w++;}
[0-9]+[a-zA-Z]*[\n" "]*                          {n++;}
[0-9]+[.][0-9]*[a-zA-Z]*[\n" "]*        {f++;}

%%
```

```
main ()
{
        yyin = fopen("abc.txt","r");
        yylex();
        printf ("\n Words starting with Vowels: %d", w);
        printf ("\n Words starting with Integers: %d", n);
        printf ("\n Words starting with Floating-Point Numerals: %d", f);
}
```

SAMPLE OUTPUT:

```
Ex_03>      lex count_2.l
Ex_03>      gcc lex.yy.c
Ex_03>      ./a.out
        Words starting with Vowels: 2
        Words starting with Integers: 2
        Words starting with Floating-Point Numerals: 1
```

RESULT:

Thus, the LEX program to count the number of words starting with a vowel or an integer or a floating-point number is implemented successfully.