EX NO :1 **IMPLEMENTATION OF SYMBOL TABLE**
DATE :

AIM :

To implement Symbol table using a C program.

ALGORITHM :

1. Start.
2. Open a file using a FILE pointer.
3. Copy the content of the file line by line and store it in a array of String.
4. Repeat the step 5 to step 9 for Number line scanned from the file.
5. Trace through the line[i] until a space is detected, if not just continue the next iteration.
6. If a space is detected copy that token to type variable for further use.
7. Now split the remaining tokens of the string with the occurrence of commas(‘,’), Semi-colon(‘;‘), line breaks(‘\n’) and white space(‘ ’).
8. Check whether the splitted token is a token is a data type, if YES then copy it to data type variable else print the variable identifier, data type and its address.
9. Stop.

SOURCE CODE:

```
#include<stdio.h>
#include<string.h>
void main()
{
FILE  *f;
char  line[100][200],data_types[100][100]={"int","float","double","char"},type[10];
int  i=0,j=0,k=0;
char  *str;
f=fopen("test.c" , "r");
while(fgets(line[i] ,200 , f)!=NULL )
      i++;
printf("\n\nIdent\t|Type\t|Address\n");
printf("--------------------------------\n");
for(j=0 ; j<I ; j++)


      {
                  str=strtok(line[j] , " ");
      if(strcmp(data_types[0],str)==0||strcmp(data_types[1],str)==0||strcmp(data_types[2],str)==0||strcmp(data_ty
pes[3],str)==0)
```

```
                    {
                              strcpy( type , str);
                              while( ( str = strtok( NULL , ",\n; " ) ) != NULL )
                              {

     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)
                                        { strcpy( type , str ); continue;}
                              else
                              printf("%s\t|%s\t|%p\n",str,type,str);
                              }
                    }
          }
     }
```

OUTPUT:

```
Ident   |  Type    |   Address
--------------------------------
   a      |  int       |  0061B210
   c      | double  |  0061B219
   d      | double  |  0061B21B
   a      | char      |  0061B2D9
   e      |  float     |  0061B3A2
   r      |  float     |  0061B3A4
   t      |  float     |   0061B3A6
  y      |  float     |   0061B3A8
```

RESULT:

Thus the given program has executed successfully to implement the symbol table.

EX NO :2 **IMPLEMENTATION OF LEXICAL ANALYZER**
DATE :

AIM :

To implement lexical analyzer using a C program.

ALGORITHM :

1. Start.
2. Open a file using a FILE pointer.
3. Copy the content of the file line by line and store it in a array of String.
4. Print the table headers and
5. Repeat the step 6 to step for Number line scanned from the file.
6. Set the NULL as initial value for all flags and start scanning the line until any given delimiter is found.
7. Once the delimiter is found check the string for following characters
8. If '#' exist then it is a preprocessor statement.
9. If ')' exist then it is a function header statement.
10. If it match with any of our assumed keyword list then it is a keyword.
11. If none of the above character found then check for non identifierlist ,if no non-identifier found then it is identifier.
12. Again resume with scanning the line for next occurrence of delimiter.
13. Print the stored values.
14. Stop.

SOURCE CODE:

LANALYSIS.C:

```
#include<stdio.h>
#include<string.h>
void  main()
{
        FILE  *f;
        char  line[100][200],*check_p,*check_f,*check_i,*check;
char  keys[100][100]={"int","float","double","char","void"},type[10],non_idents[5]={'{','}','[',']',':'};
        int  i=0,j=0,k=0,l;
        char  *str,pre[30],key[30],ident[10],func[30];
        f = fopen( "sample.c","r" );

        while(fgets(line[i],200,f) != NULL )
                i++;
        printf( "\n\n\tline\t|pre-processor\t  |Keyword|function|Identifier\n" );
        printf( "________________________________________________________________________\n" );
        for( j = 0 ; j < I ; j++)
        {
                str = strtok( line[j]," ;\n" );
                check_p = check_f = check_i = check = NULL;
                strcpy( pre,"\t\t  \0" );
                strcpy( key,"\t\0" );
                strcpy(ident,"\t\t\0" );
                strcpy(func,"\t   \0" );
                while( str != NULL )
                {
                for( k = 0 ; k < 5 ; k++)
if(strcmp( keys[k],str ) == 0 )
break;
                        check_p = strchr(str,'#' );
                        check_f = strchr(str,')' );
                        if(check_p != NULL )
                                check = strcpy( pre,str );
                        else
                        if( k != 5 )
                                check = strcpy( key,str );
                        else
                        if(check_f != NULL )
                                check = strcpy( func,str );
                        else
                        {
for( l=0 ; check_i == NULL && l < 5 ; check_i = strchr( str,non_idents[l] ) , l++ );
                                if(check_i == NULL )
                                check = strcpy( ident,str );
                        }
                        str = strtok( NULL," ;\n" );
                }
                if( check != NULL )
printf( "\t%d\t|%s|%s  |%s|%s\n" , j+1 , pre , key , func , ident );
        }
}
```

SAMPLE.C :

```
#include<stdio.h>
void  main ()
{
int  a;
float  ex;
float  b , c;
printf( "" );
}
```

OUTPUT:

```
      line    |    pre-processor      |  Keyword  |  Function    |  Identifier
____________________________________________________________________________
        1     |  #include<stdio.h>  |             |              |
        2     |                      |    void     |    main( )   |
        4     |                      |     int     |              |       a
        5     |                      |    float    |              |       ex
        6     |                      |    float    |              |       b,c
        7     |                      |             |  printf("")  |
```

RESULT:
Thus the given program has executed successfully to implement the lexical analyzer using c program.

STUDY EXPERIMENT (LEX AND YACC TOOL)

AIM :

To understand and learn to implement the LEX and YACC tools.

LEX TOOL

WHAT IS LEX TOOL ?

- **LEX** is a computer program that generates lexical analyzers ("**scanners**" or "**lexers**").
- **LEX** is commonly used with the **YACC parser generator**.

SPECIFICATION OF LEX TOOL:

...definition section ...

%%

... rules section ...

%%

... user subroutines ...

DEFINITION SECTION :

- Th

e **LEX** definitions section may contain any of several classes of items.

- The most important are external definitions, **#include** statements, and abbreviations.
- **#INCLUDE STATEMENTS :**
    - The purpose of the **#include** statement is the same as in C: to include files that are important to the lexical analyzer program.
    - This file can contain definitions for token names. **#include** statements and variable declarations must be placed between the delimiters **%{' and %}**
    - EXAMPLE :

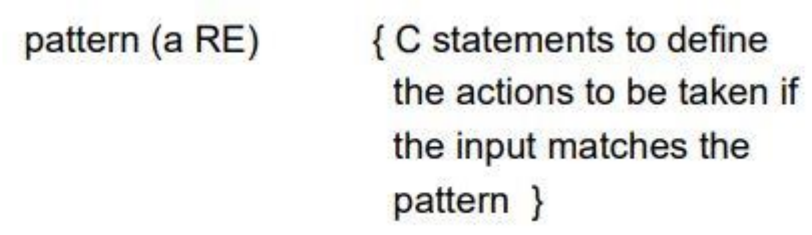
```
 %{
          #include " y.tab.h "
          extern  inttokval;
          intlineno;
%}
```

- **ABBREVIATION:**
  - 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 :

```
dig     [0-9]
ident   [a-zA-Z][a-zA-z0-9]*
   %%
-{dig}+              printf("negative
integer\n");
"+"?{dig}+            printf("positive
integer\n");
-0\.{dig}+            printf("negative real number, no whole number part\n");
```

RULE 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 (a RE)        { C statements to define
                        the actions to be taken if
                        the input matches the
                        pattern  }
```

- **REGULAR EXPRESSIONS** :
  - **.** Matches any single character except newline character.
  - ***** Matches zero or more copies of the preceding expression.
  - **[]** A character class -- matches any character within the brackets.
    - If the first character is a circumflex **("^")** it means any character except the ones within the bracket.
    - A dash inside the square bracket indicates a range.
  - ^ Matches the beginning of a line as the first character of a RE.
  - **$** Matches the end of the line as the last character of a RE.
  - \ Used to escape metacharacters.
  - + Matches one or more occurrences of the preceding RE.
  - **?** Matches zero or one occurrence of the preceding RE.
  - **|** Matches the preceding OR following RE.

- **“....”** Interprets everything within the quotation marks literally .
- **()** Groups the RE into a new regular expression.
- **[0-9]** A regular expression for a digit, i.e., any digit from 0 to 9.
- **[0-9]**+ A RE for an integer.
- **[0-9]*** A RE for an integer with possibly no digits.
- **-?[0-9]+** A RE for an integer with optional negative sign.
- **cow|pig|lamb** an RE that matches either cow, pig, or lamb.
- **(ab|cd)?ef** matches “abef”, “cdef”, or just “ef”.
- **“/*”** matches the two characters “/*”.
- ^verb matches the word “verb” when it happens at the beginning of a line.

- VARIABLES IN LEX PROGRAMS
  - **yytext -** whenever the scanner matches a token, the text of the token is stored in the null terminated string yytext.
  - **yyleng -** the length of the string yytext.
  - **yylex() -** the scanner created by the Lex has the entry point yylex().
  - you call **yylex()** to start or resume scanning.
  - if**LEX** action does a return to pass a value to the calling program, the next call to **yylex()** will continue from the point where it left off.

USER SUB ROUTINE :

- The user sub routine section is optional.
- If LEX tool is along used then the user sub routine consists of he main function and other user defined method.
- To start the LEX scanning the **yylex()** should be called in the main function.
- When **LEX** tool is used along with **YACC** tool then user sub routine section could be left blank.

EXECUTION PROCEDURE:

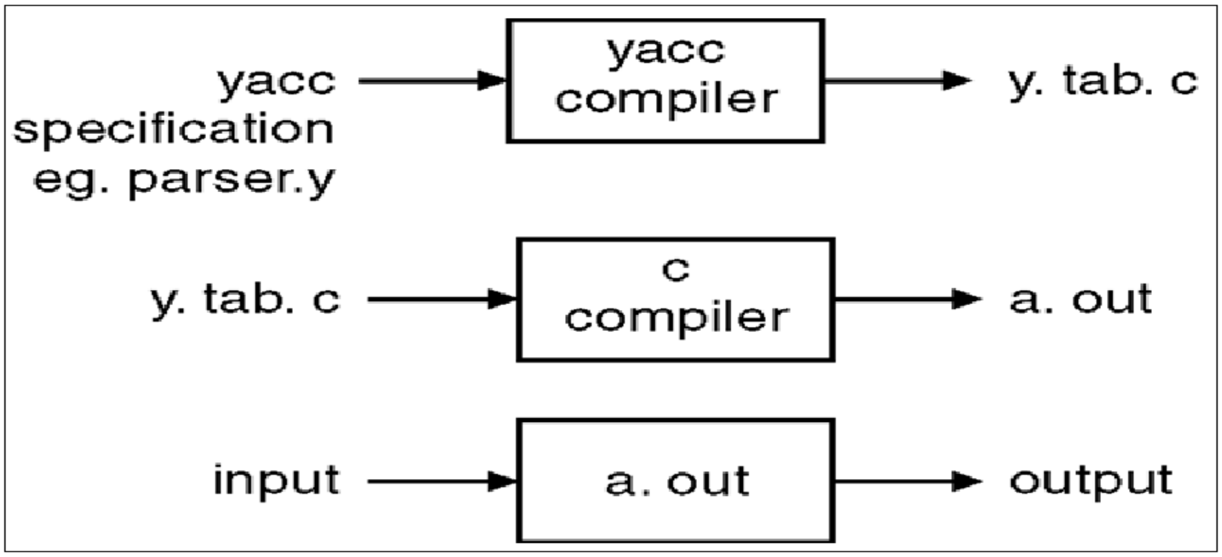
1. Create a LEX file with ”.l” extension using the following command
   - vi   file_name.l
2. Write the content of the definition , rule and user sub routine section and save the file.
3. Now compile the LEX file using the LEX compiler
   - lexfile_name.l
4. After compiling the LEX file ,a new file named “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

YACC TOOL

WHAT IS YACC TOOL ?

- YACC - **Y**et**A**nother **C**ompiler **C**ompiler.
- Yacc(for "yet another compiler compiler") is the standard parser generator for the Unixoperating system.
- An **open source** program, yacc generates **code** for the parser in the Cprogramming language.
- The acronym is usually rendered in lowercase but is occasionally seen as YACC or Yacc.



SPECIFICATION OF LEX TOOL:

- A **yacc** specification consists of a mandatory rules section, and optional sections for definitions and user subroutines..
- The declarations section for definitions, if present, must be the first section in the **yacc** program. The mandatory rules section follows the definitions; if there are no definitions, then the rules section is first.
- In both cases, the rules section must start with the delimiter **%%**. If there is a subroutines section, it follows the rules section and is separated from the rules by another **%%** delimiter.
- If there is no second **%%**delimiter, the rules section continues to the end of the file.

Declaration
%%
rules
%%
subroutines

DEFINITION SECTION :

- Declaration section may contain the following items.

- Declarations of tokens. Yacc requires token names to be declared as such using the keyword %token.
- Declaration of the start symbol using the keyword %start
- C declarations: included files, global variables, types.
- C code between %{ and %}.

RULE SECTION :

- A rule has the form:

```
Non-terminal : sentential form
               | sentential form
               .................
               | sentential form
               ;
```

- Actions may be associated with rules and are executed when the associated sentential form is matched.

LEX YACC INTERACTION :

- **yyparse()** calls **yylex()** when it needs a new token.

LEX	YACC
return(TOKEN)	%token TOKEN
	TOKEN is used in production

The external variable yylval is used for the following purposes :

- is used in a LEX source program to return values of lexemes,
- yylval is assumed to be integer if you take no other action.
- Changes related to yylval must be made
  - in the definitions section of YACC specification
    - by adding new types in the following way

```
  %union {

(type fieldname)
(type fieldname)
...............
}
```

    - and defining which token and non-terminals will use these types

```
%token <fieldname> token
%type <fieldname> non-terminal
```

- in LEX specification by using the fieldnames in the assignment as follows

```
yylval.fieldname= ...........
```

If you need a record type, then add it in the union. Example:

```
%union {
struct s {
doublefvalue;
intivalue;
   } t;
}
```

- in the LEX specification use the record name and record field in assignments:

```
yylval.t.ivalue= ......
```

- in the YACC rules specification use the record field only in the assignment:

```
$1.ivalue = ......
```

assuming that $1 has the appropriate type, whatever it denotes.

EXECUTION PROCEDURE:

1. Create a LEX file and fill the definition section and Rule section and user sub routine section not needed.

```
vi  lex_file_name.l
```

2. Create a YACC file with ".y" as extension and save it.

```
vi  yacc_file_name.y
```

3. Now compile the LEX file using LEX compiler to generate the "lex.yy.c" file.

```
lexlex_file_name.l
```

4. Now compile the YACC file with -d option using the YACC compiler to generate the "y.tab.c" file.

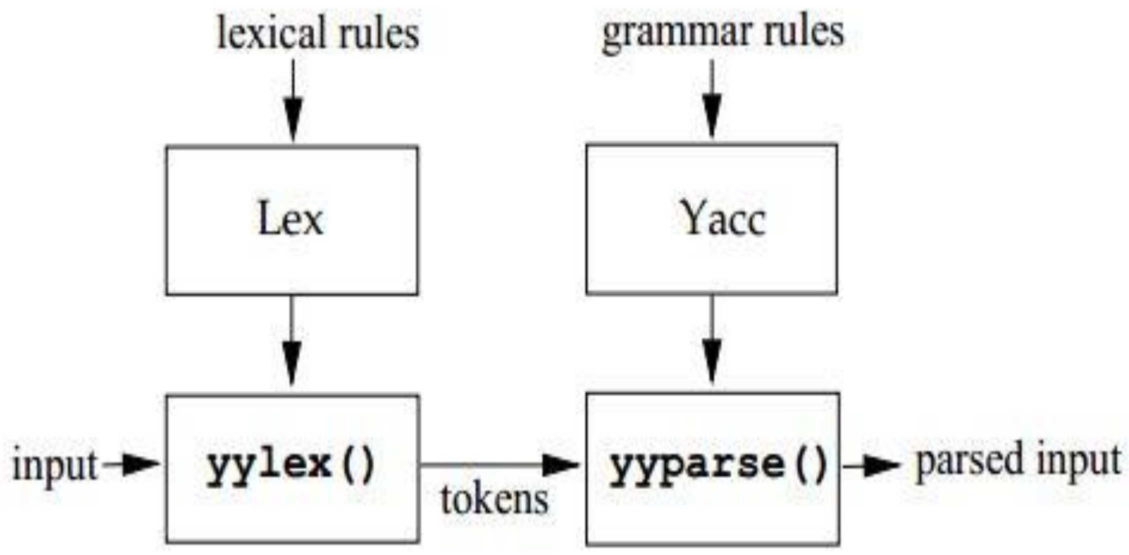
```
yacc  -d yacc_file_name.y
```

5. Now compile both the c file using the following command.

```
cc  lex.yy.cy.tab.c
```

6. After compiling both the file using the above command execute the compiled file.

```
./a.out
```



RESULT:

The usage and implementation of the LEX and YACC tool was learnt.

EXNO:3 **IMPLEMENTATION OF SIMPLE PROGRAMS USING LEX**

DATE:

A)TO FIND WHETHER THE GIVEN NUMBER IS POSITIVE OR NEGATIVE:

AIM: To find whether the given number is positive or negative using lex.

PROGRAM:

```
%option noyywrap
%{
#include<stdio.h>
%}
%%
[+][0-9]+ { printf("POSITIVE NUMBER");}
[-][0-9]+  { printf("NEGATIVE NUMBER");}
\n {return 0;}
%%
main()
{
printf("ENTER A NUMBER: ");
yylex();
}
```

SAMPLE OUTPUT:

```
ENTER A NUMBER:+10

POSITIVE NUMBER
```

RESULT:

Thus the program to find whether the given number is positive or negative using lexhad been executed successfully.

B)TO FIND WHETHER THE GIVEN NUMBER IS ODD OR EVEN:

AIM: Tofind whether the given number is odd or even using lex.

PROGRAM:

```
%option noyywrap
%{

#include<stdio.h>
%}
%%
[0-9]*[02468] { printf("ODD NUMBER");}
[0-9]*[13579] { printf("EVEN  NUMBER");}
\n {return 0;}
%%
main()
{
printf("ENTER A NUMBER: ");
yylex();
}
```

SAMPLE OUTPUT:

```
ENTER A NUMBER:20

EVEN NUMBER
```

RESULT:

Thus the program to find whether the given number is odd or even using lex had been executed successfully.

C)TO FIND THE NUMBER OF VOWELS AND CONSONANTS IN A GIVEN STRING :

AIM: Tofind the number of vowels and consonants in a given string using lex.

PROGRAM:

```
%option noyywrap
%{
#include<stdio.h>
int c=0,v=0;
%}
%%
[aeiouAEIOU] { v++;}
[a-z] { c++;}
\n {return 0;}
%%
main()
{
printf("ENTER A STRING: ");
yylex();
printf("CONSONANTS: %d \n VOWELS: %d\n",c,v);
}
```

SAMPLE OUTPUT:

```
ENTER A STRING: KOUSHIK

CONSONANTS: 4

VOWELS: 3
```

RESULT:

Thus the program toprint the number of vowels and consonants in a given string using lex had been executed successfully.

D)TO FIND WHETHER THE GIVEN IDENTIFIER IS VALID OR NOT:

AIM: Tofind whether the given identifier is valid or not using lex.

PROGRAM:

```
%option noyywrap
%{
#include<stdio.h>
%}
%%
[a-zA-Z]+[_0-9a-zA-Z]* {printf("VALID IDENTIFIER\n ");}
.* { printf(" INVALID ");}
\n {return 0;}
%%
main()
{
printf("ENTER AN IDENTIFIER:");
yylex();
}
```

SAMPLE OUTPUT:

```
ENTER AN IDENTIFIER :A_123BC

VALID IDENTIFIER
```

RESULT:

Thus the program to find whether the given identifier is valid or not using lexhad been executed successfully.

E)TO FIND THE NUMBER OF VOWELS,CONSONANTS,LINES AND SYMBOLS IN A GIVEN FILE :

AIM: To find the number of vowels,consonants,lines and symbols in a given file using lex.

PROGRAM:

```
%option noyywrap
%{
#include<stdio.h>
int v=0,c=0,s=0,l=0,n=0;
%}
%%
[aeiouAEIOU] {v++;}
[a-zA-Z] {c++;}
[0-9] {n++;}
\n {l++;}
[!@#$%^&*] {s++;}
. {return 0;}
%%
main()
{
yyin=fopen("file.txt","r");
yylex();
printf("VOWELS:%d\nCONSONANTS:%d\nNUMBERS:%d\nLINES:%d\nSYMBOLS:%d",v,c,n,l,s);
}
```

file.txt

```
KOUSHIK$

DHANUSH1010

RAM@

VIKASH23
```

SAMPLE OUTPUT:

```
VOWELS:8

CONSONANTS:15
NUMBERS:6
LINES:4
SYMBOLS:2
```

RESULT:

Thus the program to print the number of vowels,consonantslines and symbols in a given file using lex had been executed successfully.

F)TO FIND THE NUMBER OF WORDS STARTING WITH VOWELS, INTEGERS AND FLOATING POINT IN A GIVEN FILE USING LEX:

AIM: To find the number of words starting with vowels, integers and floating point in a given file using lex.

PROGRAM:

```
%option noyywrap
%{
#include<stdio.h>
int w=0,i=0,f=0;
%}
%%
[aeiouAEIOU][a-zA-Z]* {w++;}
[a-zA-Z]+ ;
[0-9]+ {i++;}
[0-9]*[.][0-9]* {f++;}
. {return 0;}
%%
main()
{
yyin=fopen("file.txt","r");
yylex();
printf("WORDS STARTING WITH VOWELS:%d\nINTEGERS:%d\nFLOATING POINTS:%d\n",w,i,f);
}
```

file.txt

```
ABINAND

8

9.8
```

SAMPLE OUTPUT:

```
WORDS STARTING WITH VOWELS:1

INTEGERS:1

FLOATINGPOINTS:1
```

RESULT:

Thus the program to print the number of words starting with vowels, integers and floating point in a given file using lex had been executed successfully.

EX.NO: 4 **IMPLEMENTATION OF LEXICAL ANALYSER USING LEX**
DATE:

AIM:

To implement lexical analyzer using lex tool which recognize COMMENT statements, KEYWORDS, IDENTIFIERS, FUNCTIONS, OPERATORS and PREPROCESSORS.

ALGORITHM:

1. Start
1. Create a lex file with extension .l example: myprog.l
2. Form regular expressions to recognize comment statements, preprocessors, keywords, identifiers and functions
3. Represent the regular expression in Rules section of lex file and specify the appropriate action in to be performed on recognizing the specified tokens.
4. use yytext variable to read the tokens from source file
5. In main function (C Code section), Call yylex() function to start reading tokens from the file or stdin
6. write the matching tokens in output.txt file
7. stop

Note:

A lex program consist of three sections

Definition section

any initial C code to be copied into the final program such as defining preprocessor directive, including header files can be represented here

Rules section

Two parts in rules section, pattern and corresponding action

Pattern: patterns to be searched for can be represented in regular expression

Action: This action will be triggered if specified pattern encountered in source file

Syntax:

<pattern or regular expression><action>

Example:

^[a-z]+<[a-z]+.h> {action}

The above RE will looks for preprocessors which inserts a header file in source program

C code section

yylex function should be invoked here to read tokens from the source and users can write their own c code here.

__Executing Lex file:__

lex -l <filename> or flex -l <filename>

This command will compile the lex file and create a equivalent c code file “lex.yy.c”

cc lex.yy.c

The above command will compile “lex.yy.c” and creates executable a.out

./a.out<source code file name>

Executes the file

Source Code:

```
%option noyywrap

%{

#include<stdio.h>

int l=1;

%}

%%

\n {l++;}

[#][a-zA-Z<>.]+ {printf(“preprocessor:%s line:%d\n”,yytext,l);}

[A-Za-z]+[()]+ {printf(“function:%s line:%d\n”,yytext,l);

“int”|”float”|”double”|”char”|”if”|”else” {printf(“keyword:%s line:%d\n”,yytext,l);}

[a-zA-Z_]+[A-Za-z0-9]* {printf(“identifier:%s line:%d\n”,yytext,l);}

%%

main()

{

yyin=fopen(“file.txt”,”r”);

yylex();

}
```

file.txt:

```
#include<stdio.h>

main()

{
```

```
int a;

}
```

Sample Output

```
Preprocessor:#include<stdio.h> line:1

Function:main() line:2

Keyword:int line:3

Identifier:a line:3
```

Result:

The implementation of lexical analyzer using lex tool has been executed successfully

EX.NO: 5a **Program to recognize a valid variable which starts with a letter**
DATE : **followed by any number of letters or digits.**

AIM:

To write a program using yacc tool to recognize a valid variable

ALGORITHM

1. Start
2. Create a yacc file with extension '.y' example myprog.y
3. Define tokens to be recognized by lexer in definition section of yacc
4. If the lexer returns var token then print the message "valid variable"
5. If the lexer returns nav token then print the message "not a valid variable"
6. In main function (C Code section), Call yyparse() function to start parsing tokens read byyylex from file or stdin
7. yyparse will invoke yylex() function whenever it needs token from input
8. Define yyerror() function to trace and notify syntax error in the input given
9. Compile the yacc file using the command yacc –d <filename.y>. On compiling yacc file two new files will be generated by yacc namely "y.tab.h" and "y.tab.c"

   example: yacc –d myprog.y
10. To read tokens from input, Create a lex file with extension .l example: myprog.l
11. Include the header file "y.tab.h" in definition section of lex file which establishes connection between lex and yacc files
12. Form regular expressions to recognize variables from the given input

    {[A-Za-z]} ({[A-Za-z]} | {[0-9]})*
13. If the token read from the input matches the specified regular expression then return token "var" to yacc else return "nav" token to yacc
14. Compile the C files generated by yacc and lex

    cc y.tab.clex.yy.c
15. Run the executable file ./a.out
16. Stop

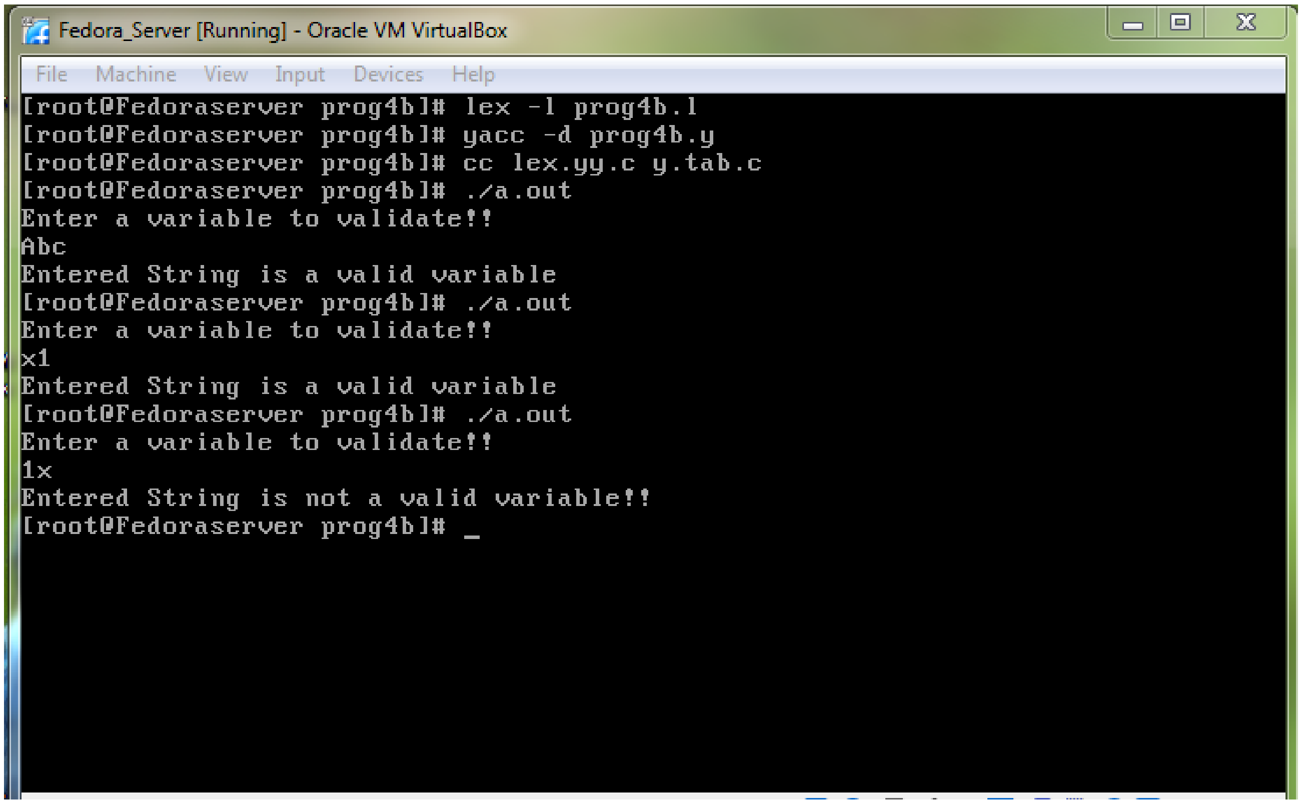
Prog5a.l

```
digit [0-9]
letter [A-Za-z]
%option noyywrap
%{
#include "y.tab.h"
%}

%%
{letter}({letter}|{digit})* { return var;}
.		{ return nav; }
%%
```

Prog5a.y

```
%{
#include<stdio.h>
intyylex();
voidyyerror(const char *);
%}

%token var nav

%%
term	: var	{ printf("Entered String is a valid variable\n"); exit(0); }
	| nav	{ yyerror("Entered String is not a valid variable!!\n"); exit(0); }
	;
%%
int main(void)
{
	printf("Enter a variable to validate!!\n");
	yyparse();
	return 0;
}
voidyyerror(const char *s)
{
	fprintf(stderr,"%s",s);
}
```

Sample Input and Output


Fedora_Server [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

```
[root@Fedoraserver prog4b]# lex -l prog4b.l
[root@Fedoraserver prog4b]# yacc -d prog4b.y
[root@Fedoraserver prog4b]# cc lex.yy.c y.tab.c
[root@Fedoraserver prog4b]# ./a.out
Enter a variable to validate!!
Abc
Entered String is a valid variable
[root@Fedoraserver prog4b]# ./a.out
Enter a variable to validate!!
x1
Entered String is a valid variable
[root@Fedoraserver prog4b]# ./a.out
Enter a variable to validate!!
1x
Entered String is not a valid variable!!
[root@Fedoraserver prog4b]# _
```

Result

Thus the yacc program to validate a string for a valid C variable has been executed successfully

EX.NO: 5b

Program to recognize a valid arithmetic expression

DATE:

AIM:

To write a program using yacc tool to recognize a valid arithmetic expression

ALGORITHM

1. Start
2. Create a yacc file with extension '.y' example myprog.y
3. Define tokens and its types to be recognized by lexer in definition section of yacc
4. Represent the yacc grammar's production rules in rules section and specify corresponding action to be performed
5. In main function (C Code section), Call yyparse() function to start parsing tokens read by yylex from file or stdin
6. yyparse will invoke yylex() function whenever it needs token from input
7. Define yyerror() function to trace and notify syntax error in the input given
8. Compile the yacc file using the command yacc –d <filename.y>. On compiling yacc file two new files will be generated by yacc namely "y.tab.h" and "y.tab.c"

   example: yacc –d myprog.y
9. To read tokens from input, Create a lex file with extension .l example: myprog.l
10. Include the header file "y.tab.h" in definition section of lex file which establishes connection between lex and yacc files
11. Form regular expressions to recognize values from the given arithmetic expression
12. Recognized tokens will be available in yytext variable, convert the text to integer and copy the integer value in yyval so that yacc parser will make use of the value recognized.
13. Compile the C files generated by yacc and lex

    cc y.tab.clex.yy.c
14. Run the executable file ./a.out
15. Stop

Prog5b.l

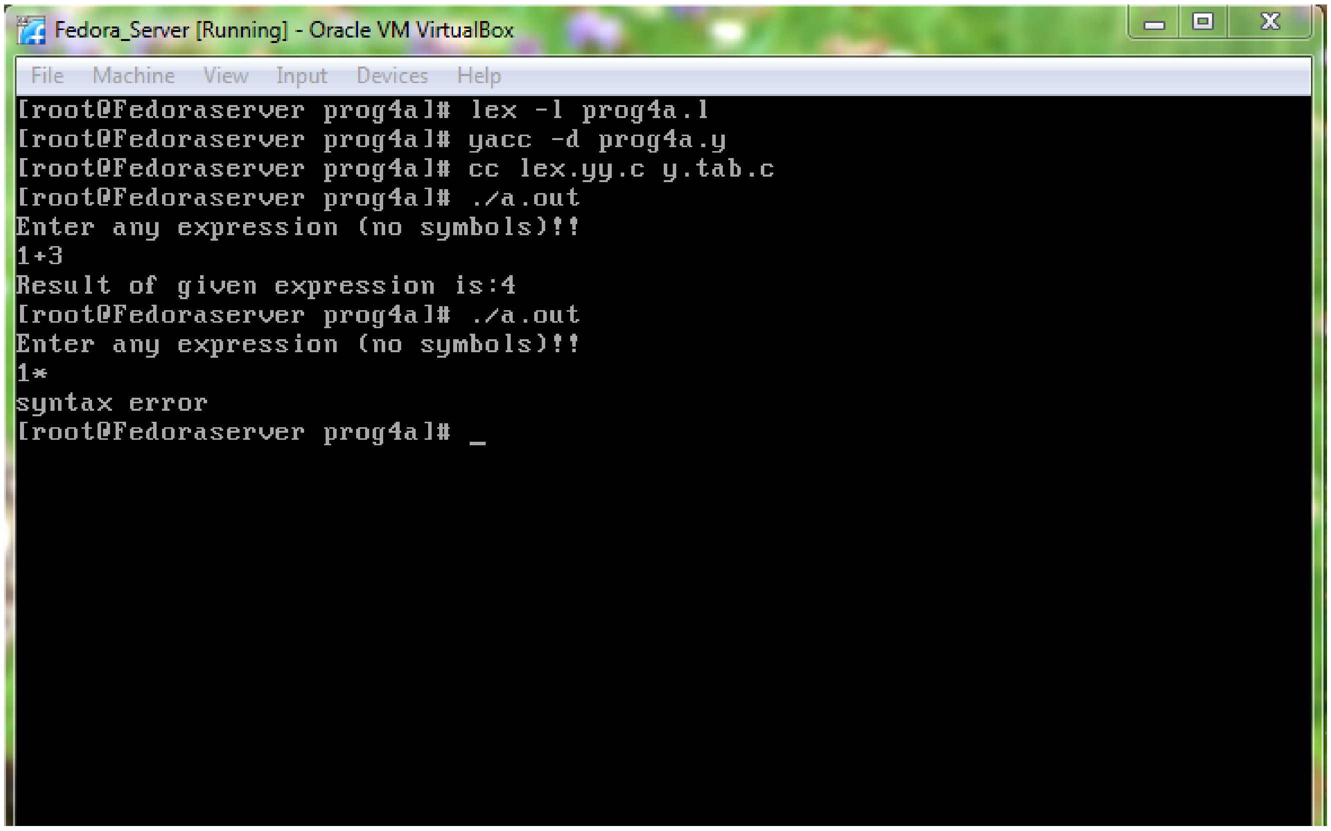
```
%option noyywrap
%{
#include"y.tab.h"
externintyylval;
%}

%%
[0-9]+ { yylval = atoi(yytext); return number;}
[ \t] ;
\n return 0;
. returnyytext[0];

%%
```

Prog5b.y

```
%{
#include<stdio.h>
intyylex();
voidyyerror(const char *s);
%}

%token number
%left '+' '-'
%left '*' '/' '%'
%nonassoc UMINUS

%%
stmt	: exp {printf("Result of given expression is:%d\n",$1); }
	;

exp	: exp '+' exp	{ $$ = $1 + $3; }
	| exp '-' exp	{ $$ = $1 - $3; }
	| exp '*' exp	{ $$ = $1 * $3; }
	| exp '/' exp
		{
			if($3!=0)
				$$ = $1 / $3;
			else
			{
				yyerror("Divide by Zero!!");
				exit(0);
			}
		}
	| exp '%' exp
		{
			if($3!=0)
				$$ = $1 % $3;
			else
				yyerror("Divide by Zero!!");
		}
```

```
        | '-' exp %prec UMINUS          { $$ = -$2; }
        | '(' exp ')'                   { $$ = $2; }
        | number                        { $$ = $1; }
        ;
%%
int main(void)
{
        printf("Enter any expression (no symbols)!!\n");
        returnyyparse();
}
voidyyerror(const char *s)
{
        fprintf(stderr, "%s\n", s);
}
```

Sample Input and Output



Fedora_Server [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

```
[root@Fedoraserver prog4a]# lex -l prog4a.l
[root@Fedoraserver prog4a]# yacc -d prog4a.y
[root@Fedoraserver prog4a]# cc lex.yy.c y.tab.c
[root@Fedoraserver prog4a]# ./a.out
Enter any expression (no symbols)!!
1+3
Result of given expression is:4
[root@Fedoraserver prog4a]# ./a.out
Enter any expression (no symbols)!!
1*
syntax error
[root@Fedoraserver prog4a]# _
```

Result

Thus the yacc program to validate a C expression has been executed successfully

EX.NO: 6 **Implementation of Calculator using LEX and YACC**
DATE :

AIM:

To implement calculator application using lexyacc tool

ALGORITHM

1. Start
2. Create a yacc file with extension '.y' example myprog.y
3. Define tokens to be recognized by lexer in definition section of yacc
4. Represent the yacc grammar's production rules in rules section and specify corresponding action to be performed
5. If lexer returns print token then print the value of the symbol following print token
6. If lexer returns close token then stop the execution of the program
7. If lexer returns identifier token then add the symbol and its corresponding value in symbol table
8. If lexer return any arithmetic symbols (+,-,*,/,%) perform the corresponding operation on value of the symbol
9. In main function (C Code section), Call yyparse() function to start parsing tokens read byyylex from file or stdin
10. yyparse will invoke yylex() function whenever it needs token from input
11. Define yyerror() function to trace and notify syntax error in the input given
12. Compile the yacc file using the command yacc –d <filename.y>. On compiling yacc file two new files will be generated by yacc namely "y.tab.h" and "y.tab.c"

    example: yacc –d myprog.y
13. To read tokens from input, Create a lex file with extension .l example: myprog.l
14. Include the header file "y.tab.h" in definition section of lex file which establishes connection between lex and yacc files
15. Form regular expressions to recognize identifier, number, yacc tokens "print and exit" and arithmetic operators from the given input and return appropriate token to yacc
16. If the token read from the input matches the specified regular expression then return token "var" to yacc else return "nav" token to yacc
17. Compile the C files generated by yacc and lex

    cc y.tab.clex.yy.c
18. Run the executable file ./a.out
19. Stop

calc.l

```
%option noyywrap
%{
#include "y.tab.h"
#include <stdio.h>
#include <stdlib.h>
voidyyerror (const char *);
%}

%%

"print"            { return print; }
"exit"             { return close; }
[a-zA-Z]+[0-9]*         { yylval.id = strdup(yytext); return identifier; }
[0-9]+             { yylval.num = atoi(yytext); return number; }
[ \t]              ;
[-+=/*%\n]         { returnyytext[0]; }
.                  { ECHO; yyerror("Not a Legal Character!!\n"); exit(0); }

%%
```

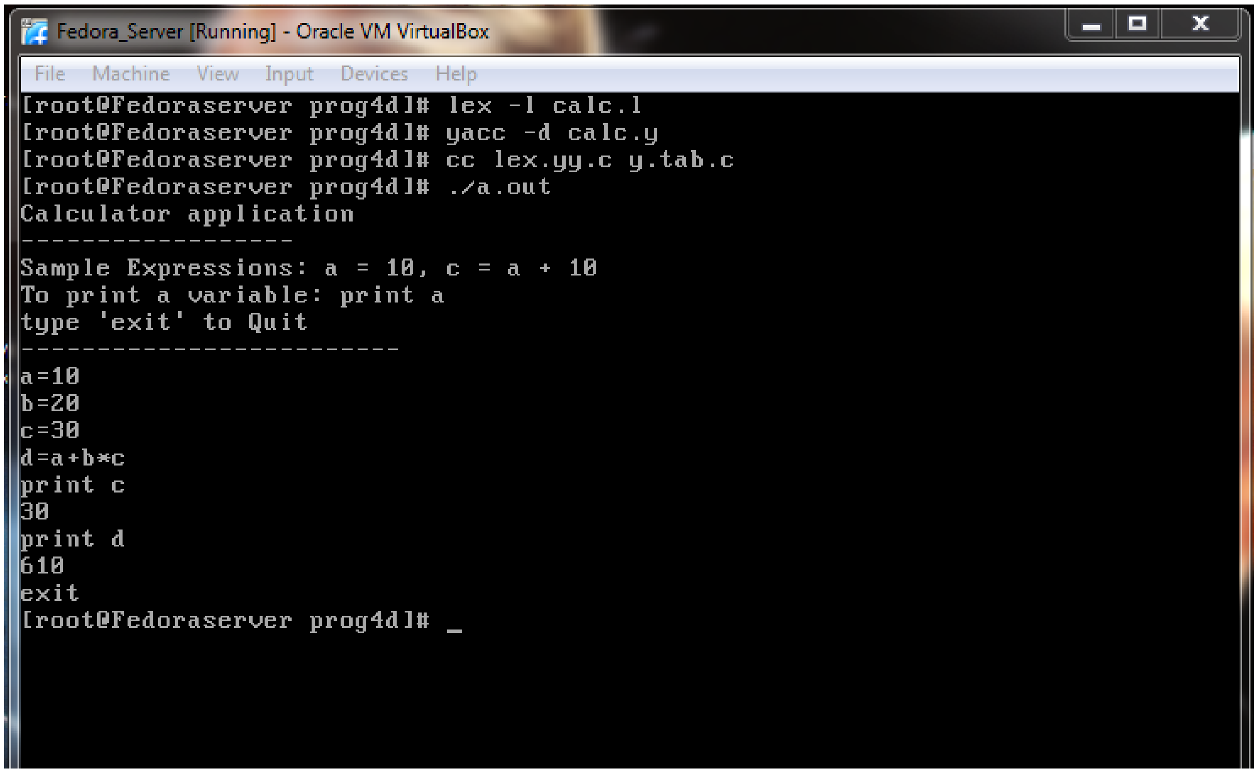
calc.y

```
%{
intyylex();
voidyyerror (char *);
#include <stdio.h>
#include <string.h>
#define MAX 100
#define YYERROR_VERBOSE
charsym[MAX][MAX];
intsymval[MAX], n=0;
intSearchSymtab(char *);
voidUpdateSymtab(char *, int);
%}

%union {int num; char *id;}
%start line
%token print
%token close
%token <num> number
%token <id> identifier
%type <num> line exp term
%type <id> assignment
%left '+' '-'
%left '*' '/' '%'
```

```
%nonassoc UMINUS
%%

line      : assignment '\n'          {;}
          | close '\n'               { exit(0); }
          | print exp '\n'  { printf("%d\n", $2); }
          | line assignment '\n'    {;}
          | line print exp '\n'      { printf("%d\n", $3); }
          | line close '\n'  { exit(0); }
          ;

assignment : identifier '=' exp { UpdateSymtab($1,$3); }
             ;
exp       : term                     { $$ = $1; }
          | exp '+' exp              { $$ = $1 + $3; }
          | exp '-' exp              { $$ = $1 - $3; }
          | exp '*' exp              { $$ = $1 * $3; }
          | exp '/' exp              { if ($3!=0)
                                               $$ = $1 / $3;
                                       else
                                         {
                                               yyerror("Divide by zero!!\n");
                                               exit(0);
                                         }
                                       }
          | exp '%' exp              { if ($3!=0)
                                               $$ = $1 % $3;
                                       else
                                         {
                                               yyerror("Divide by zero!!\n");
                                               exit(0);
                                         }
                                       }
          | '-' exp %prec UMINUS             { $$ = -$2; }
          | '(' exp ')'              { $$ = $2; }
          ;

term      : number                   { $$ = $1; }
          | identifier               { $$ = SearchSymtab($1); }
          ;
%%
int main(void)
{
          printf("Calculator application\n------------------\n");
          printf("Sample Expressions: a = 10, c = a + 10\n");
          printf("To print a variable: print a \n");
          printf("type 'exit' to Quit\n-------------------------\n");
          yyparse();
          return 0;
}
```

```
void yyerror(char *s)
{
	fprintf(stderr,"%s",s);
}

intchecksym(char *symbol)
{
	int i;
	for (i=0;i<=n;i++)
		if (strcmp(symbol,sym[i])==0)
			return i;
	return -1;
}

intSearchSymtab(char *symbol)
{
	int index = checksym(symbol);
	returnsymval[index];
}

voidUpdateSymtab(char *symbol, intval)
{
	int index = checksym(symbol);
	if (index != -1)
		symval[index] = val;
	else
	{
		strcpy(sym[n],symbol);
		symval[n] = val;
		n++;
	}
}
```

Sample Input and Output


Fedora_Server [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

```
[root@Fedoraserver prog4d]# lex -l calc.l
[root@Fedoraserver prog4d]# yacc -d calc.y
[root@Fedoraserver prog4d]# cc lex.yy.c y.tab.c
[root@Fedoraserver prog4d]# ./a.out
Calculator application
------------------
Sample Expressions: a = 10, c = a + 10
To print a variable: print a
type 'exit' to Quit
-------------------------
a=10
b=20
c=30
d=a+b*c
print c
30
print d
610
exit
[root@Fedoraserver prog4d]# _
```

Result:

Thus the calculator application using lex and yacc tool has been implemented and executed successfully

EX NO: 7 **Construction Of Abstract Syntax Tree**
DATE:

Aim:

To construct AST using LEX and YACC.

Algorithm:

1. Start.
2. Store the numeric constants, identifiers, operators and whitespaces in the LEX file.
3. Store the value of the constant and identifiers in yylval.
4. Define the expression grammar to get multiple expressions one at a time.
5. Generate the tree for each sub-expression.
6. Display the tree for valid expression, else call yyerror() to print error statement.
7. End.

Source Code:

Lex.l

```
%option noyywrap
%{
#include "y.tab.h"
extern YYSTYPE  yylval;
%}
%%
[_a-zA-Z][a-zA-Z0-9]* {yylval.name=strdup(yytext);return id;}
[0-9]+ {yylval.name=strdup(yytext);return num;}
[\n] {return 0;}
. {return yytext[0];}
%%
```

Yacc.y

```
%{
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int flag=1,i=0;
struct astnode
{
        char *data;
        struct astnode *left,*right;
}*root;
%}
```

```
%union{ struct astnode *node;
        char *name;
}
%token <name>  num id
%type <node> e start s
%left '+''-'
%left '*''/'
%right '='
%%
start:s '=' e  {$$=addnode($1,$3,"=");root=$$;}
| e            {$$=$1;root=$$;}
;
e:s            {$$=$1;}
|e '+' e       {$$=addnode($1,$3,"+");}
|e '*' e       {$$=addnode($1,$3,"*");}
|e '-' e       {$$=addnode($1,$3,"-");}
|e '/' e       {$$=addnode($1,$3,"/");}
|'('e')'       {$$=$2;};
;
s:id                   {$$=makenode($1);}
|num                   {$$=makenode($1);}
;
%%
int yyerror()
{
int flag=0;
printf("invalid expr!!");
return 0;
}
main()
{
        void printasttree(struct astnode *);
        printf("enter expr:");
        yyparse();
        printf("\nPreorder traversal of the AST tree for the
                given expression is:");
        printasttree(root);
        printf("\n");
}
struct astnode *addnode(struct astnode *l,struct astnode *r,char *sym)
{
```

```
struct astnode *temp;
        temp = (struct astnode *)malloc(sizeof(struct astnode));
        temp->data=strdup(sym);
        temp->left=l;
temp->right=r;
return temp;
}

struct astnode *makenode(char *data)
{
        struct astnode *temp;
        temp = (struct astnode *)malloc(sizeof(struct astnode));
        temp->data=strdup(data);
        temp->left=NULL;
        temp->right=NULL;
        return temp;
}
void printasttree(struct astnode *s)
{
        if(s!=NULL)
        {
                printf("%s",s->data);
        }
        if(s->left!=NULL)
                printasttree(s->left);
        if(s->right!=NULL)
                printasttree(s->right);
}
```

Output:

```
12+10
+ 12 10
12*10
* 12 10
12-10
```

Result:

Thus the LEX and YACC program to generate AST is implemented and output is verified.

EX NO: 8 **IMPLEMENTATION OF TYPE CHECKING**

DATE:

Aim:

To check statements for valid data type using LEX and YACC.

Algorithm:

1. Start.
2. Store the numeric constants, identifiers, operators and whitespaces in the LEX file.
3. Store the value of the constant and identifiers in yylval.
4. Define the expression grammar to get multiple expressions one at a time.
5. Do the type checking for the variables stored in the sample input file.
6. If any type mismatch occurs print the line of mismatch and prompt user whether to print the table or not if input is 'y' then goto step 7 else goto step 8 else no

   mismatch occurs then goto to step 7.

7.Print the table

8. End.

Source Code:

Typechk.l :

```
%option noyywrap
%{
        #include "y.tab.h"
        extern char *toktype;
        int lineno=1;
%}
key "int"|"char"|"float"|"double"|"long"
letter [_a-zA-Z]
digit [0-9]
%%
\'.\'      {yylval=strdup(yytext);return CH;}
{key} {toktype=strdup(yytext);yylval=strdup(yytext);return TYPE;}
{letter}({letter}|{digit})* {yylval=strdup(yytext);return ID;}
{digit}+(\.{digit}+)? {yylval=strdup(yytext);return NUM;} [ \t]+ ;
\n       lineno++;
. return *yytext;
%%
```

Typechk.y :

```
%{
	#include<stdio.h>
	#include<stdlib.h>
	struct symtab
	{
		char *name;
		char *type;
		union
		{
			char cval;
			int ival;
			double dval;
		}value;
		struct symtab *next;
	};
	#define YYSTYPE char *
	extern FILE *yyin;
	char *toktype;
	struct symtab *hashtab;
	struct symtab* initialize_symtab();
	char* gettype(char*);
	void addsymb(char *,char *, char *);
	extern int lineno;
%}
%token TYPE ID NUM CH
%left '+' '-'
%left '*' '/'
%%
stmt1	:	stmt stmt1
	|	stmt
;
stmt	:	TYPE idlist ';'
	|	asgnexpr
;
idlist	:	idlist ',' iden
	|	iden
;
iden	:	ID '=' NUM		{




					if(search($1)!=1)
```

```
                                                addsymb($1,toktype,$3);
                                        else
                                                printf("\nLine no:%d Multiple Declaration of
Symbol:%s!!!",lineno,$1);
                                        }
        |       ID                      {
                                        if(search($1)!=1)
                                                addsymb($1,toktype,"-999");
                                        else
{

                                                printf("\nLine no:%d: Multiple Declaration of
Symbol:%s!!!",lineno,$1);
                                        }
        |       ID '=' CH               {
                                        if(search($1)!=1)
                                                addsymb($1,toktype,$3);
                                        else
                                                printf("\nLine no:%d: Multiple Declaration of
Symbol:%s!!!",lineno,$1);
                                        }
;
asgnexpr:       ID '=' expr ';'         {
                                        if(search($1)!=1)
                                                printf("\nLine no%d:'%s' Symbol
Undefined!!",lineno,$1);
                                        else if(strcmp($3,gettype($1))!=0)
                                                        printf("\nLine no%d: Type mismatch!!",lineno);
                                        }
;
expr    :       expr '+' expr           {
                                                if(strcmp($1,$3)==0)
                                                        $$=strdup($1);
                                                     else
                                                        $$=strdup("Err");
                                        }
        |       expr '-' expr           {
                                                if(strcmp($1,$3)==0)
                                                        $$=strdup($1);
                                                     else
                                                        $$=strdup("Err");

                                        }
        |       expr '*' expr           {
                                                if(strcmp($1,$3)==0)
```

```
                                                                  $$=strdup($1);
                                                               else
                                                                  $$=strdup("Err");
                                               }
        |       expr '/' expr                  {
                                                       if(strcmp($1,$3)==0)
                                                                  $$=strdup($1);
                                                               else
                                                                  $$=strdup("Err");
                                               }
        |       '('expr')'                     {$$=strdup($2);}
        |       CH                             {$$=strdup("char");}
        |       NUM                            {if(strstr($1,".")!=NULL)$$=strdup("float"); else
$$=strdup("int");}
        |       ID                             {
                                                       if(search($1)!=1)
                                                                          printf("\nLine no:%d: '%s' Symbol
Undefined!!",lineno,$1);
                                                               else
                                                                  $$=gettype($1);
                                               }
;
%%
char* gettype(char *name)
{
        struct symtab *temp;
        int i=0;
        temp=(struct symtab *)malloc(sizeof(struct symtab));
        for(i=0;i<3;i++)
        {
                temp=(hashtab+i)->next;
                while(temp!=NULL)
                {
                                if(strcmp(temp->name,name)==0)
                                        return(temp->type);
                        temp=temp->next;
                }
        }
}




void main()
```

```
{
        struct symtab *temp;
        int i;
        char ch;
        hashtab=initialize_symtab();
        yyin=fopen("input.c","r");
        yyparse();
        printf("\nList symbol table[y/n]:");
        ch=getchar();
        if(ch=='y')
        {
                printf("\n\n\n----------------Symbol Table------------------");



                printf("\n Symbol Name \t Symbol Type \t Symbol Value"); printf("\n---
                ------------------------------------------");



                for(i=0;i<3;i++)
                {
                        temp=(hashtab+i)->next;
                        while(temp!=NULL)
                        {
                                printf("\n%s\t\t%s\t\t",temp->name,temp->type);
                                if(strcmp(temp->type,"int")==0)
                                                printf("%d\n",temp->value.ival);
                                           else if (strcmp(temp->type,"float")==0)
                                                printf("%lf\n",temp->value.dval);
                                else if (strcmp(temp->type,"char")==0)
                                        printf("%c\n",temp->value.cval);
                                temp=temp->next;
                        }
                }
        }
}

void addsymb(char *name,char *type,char *value)
{
        struct symtab *temp;
        temp=(struct symtab *) malloc(sizeof(struct symtab));
        temp->name=strdup(name);
        temp->type=strdup(type);
        if(strcmp(type,"int")==0)
```

```
	{
		temp->value.ival=atoi(value);
		temp->next=hashtab->next;
		hashtab->next=temp;
	}
	else if(strcmp(type,"float")==0||strcmp(type,"double")==0)
	{
		temp->value.dval=atof(value);
		temp->next=(hashtab+1)->next;
		(hashtab+1)->next=temp;
	}
	else if(strcmp(type,"char")==0)
	{
		if(strcmp(value,"-999")!=0)
			temp->value.cval=*(value+1);
		else
			temp->value.cval='$';
		temp->next=(hashtab+2)->next;
		(hashtab+2)->next=temp;
	}
}

int yyerror()
{
	printf("Error!!!\n");
	return 0;
struct symtab* initialize_symtab()
{
	struct symtab *node;
	node=(struct symtab *)malloc(100*sizeof(struct symtab)); return
	node;
}
int search(char *name)
{
	struct symtab *t1;
	int i=0;


	t1=(struct symtab *) malloc(sizeof(struct symtab));
	for(i=0;i<3;i++)
	{
		t1=(hashtab+i)->next;
		while(t1!=NULL)
		{
```

```
                                if(strcmp(t1->name,name)==0)
                                          return 1;
                                t1=t1->next;
                      }
            }
            return 0;
}
```

SAMPLE INPUT:

input.c:

```
float x,y,z=9.5;

int a,b,c=10;

char s,v,t='a';

int a;

a=c;

v='3';

b=c+10.5;

b=x+s;

c=w+t;
```

SAMPLE OUTPUT:

```
Line no:4: Multiple Declaration of Symbol:a!!!

Line no7: Type mismatch!!

Line no8: Type mismatch!!

Line no:9: 'w' Symbol Undefined!!

Line no9: Type mismatch!!

List symbol table[y/n]:y
```

```
---------------     Symbol Table------------------

 Symbol Name     Symbol Type   Symbol Value

----------------------------------------------

c              int           10

b              int           -999

a              int           -999

z              float         9.500000

y              float         -999.000000

x              float         -999.000000

t              char          a

v              char          $

s              char          $
```

RESULT:

Thus the given program for type checking has been compiled and executed successfully.

EX.NO: 9 IMPLEMENTATION OF CONTROL FLOW ANALYSIS
DATE:

Aim:

To implement control flow analysis using LEX.

Algorithm:

1. Start.
2. Make the first line instruction, instruction with labels, branch instructions as individual nodes.
3. Print the individual blocks along with the line numbers.
4. Construct the control flow graph.
5. Print the blocks, graph and the exit block for the given code.
6. Stop.

Source Code:

cfa.l:

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

void addblock(char *);
void append_code(char *);
int lnc=1,bno=1,prev=0;
char str[500]="",lineno[500]="",t[5];

struct graph{
	char code[500];
}blarr[1000];
%}
space	[ \t]
rop	">"|"<"|"=="|"<="|">="|"<>"
aop	"+"|"-"|"*"|"/"|"%"
asignop		"="
digit	[0-9]
num	{digit}+
letter	[a-zA-Z]
iden	{letter}({letter}|{digit})*
```

```
exp     ({iden}|{num}){space}*({aop}{space}*({iden}|{num}))?
aexp    {iden}{space}*{asignop}{space}*{exp}
rexp    {iden}{space}*{rop}{space}*({iden}|{num})
%%
{aexp}                              {append_code(yytext);prev=0;}
"if "{rexp}(" goto L"{digit}+)              {addblock(yytext);}
L{digit}+:.+                        {
                                            if(strcmp(str,"")!=0)
                                            {
                                                    printf("Block No:%d\tLine no's:%s\n",bno,lineno);
                                                    strcpy(blarr[bno].code,str);
                                            }
                                            if(prev==0)
                                                    bno++;
                                            strcpy(str,yytext);
                                            sprintf(t,"%d,",lnc);
                                            strcpy(lineno,t);
                                    }
("goto L"{digit}+)                  {addblock(yytext);}
\n                                  {lnc++;;}
{space}*                            ;
.                                   ;
%%
void main()
{
        int i,j,count;
        int blgraph[200][200]={0};
        char *substr,*s;
        yyin=fopen("cfain.txt","r");
        yylex();
        for(i=1;i<bno;i++)
        {
                if((substr=strstr(blarr[i].code,"goto"))==NULL)
                        blgraph[i][i+1]=1;
                else if(((substr=strstr(blarr[i].code,"goto"))!=NULL) && ((s=strstr(blarr[i].code,"if"))!=NULL))
                        blgraph[i][i+1]=1;
                if((substr=strstr(blarr[i].code,"goto"))!=NULL)
                {
                        s=strdup(substr+5);
                        strcat(s,":");
                        for(j=1;j<bno;j++)
                                if(strstr(blarr[j].code,s)!=NULL)
                                        blgraph[i][j]=1;
                }
```

```
	}
	printf("\nThe Control Flow Graph for the given code is:\n");
	printf("Block");
	for(i=1;i<bno;i++)
		printf("\t%d",i);
	for(i=1;i<bno;i++)
	{
		printf("\n%d",i);
		for(j=1;j<bno;j++)
			printf("\t%d",blgraph[i][j]);
	}
	printf("\nThe Exit block(s):");
	for(i=2;i<bno;i++)
	{
		count=0;
		for(j=1;j<bno;j++)
			count=count+blgraph[i][j];
		if(count==0)
			printf("B%d",i);
	}
	printf("\n");
}
void addblock(char *s)
{
	append_code(s);
	printf("Block No:%d\tLine no's:%s\n",bno,lineno);
	strcpy(blarr[bno].code,str);
	strcpy(str,"");
	strcpy(lineno,"");
	bno++;
	prev=1;
}
void append_code(char *s)
{
	strcat(str,s);
	sprintf(t,"%d,",lnc);
	strcat(lineno,t);
}
```

Input:
cfain.txt:

```
a = 1
b = 2
L0: c = a + b
```

```
d = c - a
if c < d goto L2
L1: d = b + d
if d < 1 goto L3
L2: b = a + b
e = c – a
if e == 0 goto L0
a = b + d
b = a – d
goto L4
L3: d = a + b
e = e + 1
goto L1
L4: return
```

Sample Output:

```
Block No:1     Line no's:1,2,
Block No:2     Line no's:3,4,5,
Block No:3     Line no's:6,7,
Block No:4     Line no's:8,9,10,
Block No:5     Line no's:11,12,13,
Block No:6     Line no's:14,15,16,

The Control Flow Graph for the given code is:
Block  1      2      3      4      5      6
1      0      1      0      0      0      0
2      0      0      1      1      0      0
3      0      0      0      1      0      1
4      0      1      0      0      1      0
5      0      0      0      0      0      0
6      0      0      1      0      0      0

The Exit block(s):B5
```

Result:

Thus the program for control flow analysis has been compiled and executed successfully.

EX NO: 10 **IMPLEMENTATION OF DATA FLOW ANALYSIS**
DATE:

AIM:
To write a C program to implement data flow analysis.

ALGORITHM:

1. START.
2. Get the number of statements and the statements.
3. For the first read statement tokenize definition and expression, store the expression value in

av[0] and initialize kill [0] to null.
4. for the next statements, tokenize definition and expression. (i) Check if the definition is a part of av[i-1] or a part of expression,if so add av[i-1] to kill.
0 (ii) Check if expression is not a part of definition, check if it has been killed already then make
1 (iii) If expression is part of definition, check if it has been killed already then make kill null
2

kill null else add to kill[i].
else add to kill[i].
5. Repeat 5 till all statements are processed.
6. Display statements, available statement and kill.
7. STOP

SOURCE CODE:

```
#include<stdio.h>
#include<string.h>
struct block
{
   char l[5];
   char r[20];
}b[10];

struct result
{
   char res[10];
}r[10];

int check(int val, int ind)
{
   int i, len;
   len = strlen(r[ind].res);
   for(i=0;i<len;i+=2)
   {
      if(r[ind].res[i] == val)
         return 0;
   }
```

```
    return 1;
}
void main()
{
    int n, i, j, k, len;
    printf("Enter no of values :: ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("\nLeft :: ");
        scanf("%s",b[i].l);
        printf("Right :: ");
        scanf("%s",b[i].r);
    }

    printf("\nCode :: \n");
    for(i=0;i<n-1;i++)
    {
        printf("\t%s = ",b[i].l);
        printf("%s\n",b[i].r);
    }
    printf("\treturn %s",b[i].r);

    //split
    r[n-1].res[0] = b[i].r[0];
    for(i=n-2;i>=0;i--)
    {
        k=0;
        len = strlen(b[i].r);
        for(j=0;j<len;j+=2)
        {
            if(check(b[i].r[j], i))
            {
                r[i].res[k++] = b[i].r[j];
                r[i].res[k++] = ',';
            }
        }

        //k = j;
        len = strlen(r[i+1].res);
        for(j=0;j<len;j+=2)
        {
            if(b[i].l[0] != r[i+1].res[j])
            {
                if(check(r[i+1].res[j], i))
                {
                    r[i].res[k++] = r[i+1].res[j];
                    r[i].res[k++] = ',';
                }
            }
        }
        r[i].res[k-1] = '\0';
        //printf("\n%s", r[i].res);
    }
    printf("\n\nData flow - Liveliness Analysis :: \n");
    for(i=0;i<n-1;i++)
    {
        len = strlen(r[i].res);
        if(len == 1 && (r[i].res[0] >= '0'  && r[i].res[0] < '9'))
            printf("\n\t%s = %s\t\t--", b[i].l, b[i].r);
        else
            printf("\n\t%s = %s\t\t%s", b[i].l, b[i].r, r[i].res);
    }
     printf("\n\t%s %s\t\t%s", b[i].l, b[i].r, r[i].res);
}
```

OUTPUT:

```
Enter no of values :: 6
Left :: a
Right :: 7

Left :: b
Right :: a+a

Left :: c
Right :: b+a

Left :: d
Right :: a+b

Left :: e
Right :: d+c

Left :: return
Right :: e

Code:
        a = 1
        b = a+a
        c = b+a
        d = a+b
        e = d+c
        return e

Data flow - Liveliness Analysis ::

        a = 1           --
        b = a+a                 a
        c = b+a                 b,a
        d = a+b                 a,b,c
        e = d+c                 d,c
        return e                e
```

RESULT:

Thus a program to implement data flow analysis have been written and executed successfully.

EX NO: 11 IMPLEMENTATION OF STORAGE ALLOCATION STRATEGIES

DATE:

AIM:

To implement storage allocation using stack.

ALGORITHM:

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

Step 1 − Checks if the stack is full.

Step 2 − If the stack is full, produces an error and exit.

Step 3 − If the stack is not full, increments top to point next empty space.

Step 4 − Adds data element to the stack location, where top is pointing.

Step 5 − Returns success.

Pop operation

Step 1 − Checks if the stack is empty.

Step 2 − If the stack is empty, produces an error and exit.

Step 3 − If the stack is not empty, accesses the data element at which top is pointing.

Step 4 − Decreases the value of top by 1.

Step 5 − Returns success.

SOURCE CODE:

```
#include <stdio.h>
int MAXSIZE = 8;
int stack[8];
int top = -1;
int isempty() {
if(top == -1)
return 1;
else
return 0;
}
int isfull() {
if(top == MAXSIZE)
return 1;
else
return 0;
}
int peek() {
return stack[top];
}
int pop() {
int data;
if(!isempty()) {
data = stack[top];
top = top - 1;
return data;
} else {
printf("Could not retrieve data, Stack is empty.\n");

}
}
int push(int data) {
if(!isfull()) {
top = top + 1;
stack[top] = data;
} else {
printf("Could not insert data, Stack is full.\n");
}
}

int main() {
/     push items on to the stack push(3);
push(5);
push(9);
push(1);
push(12);
push(15);

printf("Element at top of the stack: %d\n" ,peek());

printf("Elements: \n");

/     print stack data while(!isempty()) { int data = pop();
printf("%d\n",data);
}

printf("Stack full: %s\n" , isfull()?"true":"false");
printf("Stack empty: %s\n" , isempty()?"true":"false");

return 0;
}
```

OUTPUT:

```
Element at top of the stack: 15

Elements:
15
12
1
9
5
3
Stack full: false
Stack empty: true
```

RESULT:

Thus the given program for storage allocation strategy has been compiled and executed successfully.

EX.NO:12 **IMPLEMENTATION OF DIRECTED ACYCLIC GRAPH**
DATE:

Aim:

To implement directed acyclic graph using LEX and YACC.

Algorithm:

Lex.l:

1. Start.
2. Add the option noyywrap in the definition section.
3. Include the header files in the definition section.
4. In the rule section create the Regular expression along their actions to tokenize the string from the program file.
5. Stop

Yacc.y:

1. Start.
2. Include the header file and also the tokens that are returned by the LEX and also give precedence to the operators.
3. In the rule section write a grammar to find the index, left and right nodes of the current node in the expression.
4. Then print the DAG as a table with index, right and left nodes with the current node.
5. Call **yyparse()** to get more tokens from the LEX.
6. Define a **yyerror()** function to display a error message if the token is invalid.
7. Also have addsym() to add symbols and addop() to add operators.
8. Stop

Source Code:

dag.l

```
%option noyywrap
%{
  #include "y.tab.h"
  extern YYSTYPE  yylval;
%}
%%
[_a-zA-Z][a-zA-Z0-9]* {yylval.name=strdup(yytext);return id;}
[0-9]+                {yylval.name=strdup(yytext);return num;}
[ \t]*                {}
[\n]                  {return 0;}
.                             {return yytext[0];}
%%
```

dag.y

```
%{
        #include<stdio.h>
        #include<ctype.h>
        #include<stdlib.h>
        #include<string.h>
        struct DAG
        {
                char *sym;
                int left,right;
        }table[100];
        char subexp[20][20];
        int sub_exp_tbl_idx[20];
        int tbl_entry=0,idx,sub_exp_count=0;
        int search(char *);
        int issubexp(char *);
        int search_subexp(char *);
        void add_subexp(char *);
%}
%union{char *name;}
%token <name>  num id
%type <name> e start
%left '+' '-'
%left '*' '/'
%right '='
%%
start   :       id '=' e {
                                        idx = search($1);
                                        if(idx == -1)
                                                addsym($1,45,45);
                                        if(issubexp($3))
                                        addsym("=",search($1),search_subexp($3));
                                else
                                        addsym("=",search($1),search($3));
                        }
        |       e               {}
;
e       :       e '+' e         {$$ = strdup($1); strcat($$,"+"); strcat($$,$3); addop("+",$1,$3,$$);}
        |       e '*' e         {$$ = strdup($1); strcat($$,"*"); strcat($$,$3); addop("*",$1,$3,$$);}
        |       e '-' e         {$$ = strdup($1); strcat($$,"-"); strcat($$,$3); addop("-",$1,$3,$$);}
        |       e '/' e         {$$ = strdup($1); strcat($$,"/"); strcat($$,$3); addop("/",$1,$3,$$);}
        |       '(' e ')'   {$$ = strdup($2);}
        |       id              {
                                idx = search($1);
                                if(idx == -1)
                                        addsym($1,45,45);
                        }
        |       num             {
                                idx = search($1);
                                if(idx == -1)
                                        addsym($1,45,45);
                        }
;
%%
int yyerror()
{
        printf("\nInvalid expr!!");
        return 0;
}
main()
{
        int i;
        printf("\nEnter an expr:");
        yyparse();
        printf("\nDAG for the given expr is:");
        printf("\n-----------------------------");
        printf("\nIndex\tNode\tLeft\tRight");
        printf("\n-------- ---------------------\n");
        for(i=0;i<tbl_entry;i++)
        {
                if(isalnum(table[i].sym[0]))
                        printf("%d\t%s\t%c\t%c\n",i,table[i].sym,table[i].left,table[i].right);
                else
                        printf("%d\t%s\t%d\t%d\n",i,table[i].sym,table[i].left,table[i].right);
        }
}
int search(char *sym)
{
        int i;
        for(i=0;i<tbl_entry;i++)
                if(strcmp(table[i].sym,sym)==0)
                        return i;
        return -1;
}
void addsym(char *sym, int l, int r)
{
        table[tbl_entry].sym=strdup(sym);
        table[tbl_entry].left = l;
        table[tbl_entry].right = r;
        tbl_entry++;
}
void addop(char *sym, char *l, char *r, char *exp)
{
        int l_index,r_index;

        if(!issubexp(l) && !issubexp(r))
        {
                if(search_subexp(exp)==-1)
                {
                        addsym(sym,search(l),search(r));
                        add_subexp(exp);
                }
        }
        else
        {
                if(issubexp(l))
                        l_index = search_subexp(l);
                else
                        l_index = search(l);
                if(issubexp(r))
                        r_index = search_subexp(r);
                else
                        r_index = search(r);
                addsym(sym,l_index,r_index);
                add_subexp(exp);
        }
}
int search_subexp(char *exp)
{
        int i;
        for(i=0;i<sub_exp_count;i++)
        {
                if(strcmp(subexp[i],exp)==0)
                        return sub_exp_tbl_idx[i];
        }
        return -1;
}
int issubexp(char *exp)
{
        int i,len=strlen(exp);
        for(i=0;i<len;i++)
                if(exp[i]=='+' || exp[i] == '-' || exp[i] == '*' || exp[i] == '/')
                        return 1;
        return 0;
}
void add_subexp(char *exp)
{
        strcat(subexp[sub_exp_count],exp);
        sub_exp_tbl_idx[sub_exp_count] = tbl_entry - 1;
        sub_exp_count++;
}
```

Sample Output:

```
Enter an expr:(a+b)*(a+b)

DAG for the given expr is:
------------------------------
Index   Node   Left     Right
-------- ----------------------
```

```
0        a        -        -
1        b        -        -
2        +        0        1
3        *        2        2
```

```
Enter an expr:(a+b)*(a-b)

DAG for the given expr is:
------------------------------
Index   Node   Left    Right
-------- ----------------------
0        a        -        -
1        b        -        -
2        +        0        1
3        -        0        1
4        *        2        3
```

Result:

Thus the program for directed acyclic graph has been compiled and executed successfully.

EX. NO : 13 **IMPLEMENTATION OF CODE GENERATION**

DATE:

AIM:

To implement code generation using LEX and YACC.

ALGORITHM:

Lex.l:

1. Start.
2. Add the option noyywrap and yylval as extern variable in the definition section.
3. Include the header files in the definition section.
4. In the rule section create the Regular expression along their actions to tokenize the string from the program file.
5. Stop

Yacc.y:

1. Start.
2. Include the header file and also the tokens that are returned by the LEX and also give precedence to the operators.
3. In the rule section write a grammar to generate the assembly code.
4. Call **yyparse()** to get more tokens from the LEX.
5. Define a **yyerror()** function to display a error message if the token is invalid.
6. Also have swapreg() to add symbols and inc_timer() and other required functions.
7. Stop.

SOURCE CODE:

codegen.l:

```
	%option noyywrap
	%{
		#include "y.tab.h"
		extern YYSTYPE yylval;
	%}
	%%
	[a-z][a-z]*	{yylval.str=strdup(yytext);return id;}
	[1-9][0-9]*	{yylval.str=strdup(yytext);return num;}
	[ \t]*		{}
	[=+-/*\n]	{return *yytext;}
	.		{}
	%%
```

codegen.y:

```
	%{
		#include<stdio.h>
		#include<string.h>
		char* getreg(char *);
		void clr(char *);
		void mk_instr(char*, char*, char*);
		char *code;
		enum reg {AX,BX,CX,DX};
		char regname[4][3]={"AX","BX","CX","DX"};
		char regcont[4][100] = {"$","$","$","$"};
		char * swapreg_content(char*);
		int regcont_timer[4]={0,0,0,0};
		int yylex();
		extern FILE *yyin;
%}
%union {char *str;}
%left '+' '-'
%left '*' '/'
%right '='
%type <str> e start line
%token <str> id num
%%
start	:	line '\n' {printf("\n");inc_timer();}
	|	start line '\n'	{printf("\n");inc_timer();}
```

```
;
line    :       id '=' e {printf("\nMOV %s,%s",$1,$3);clr($3);}
;
e       :       e '+' e         {mk_instr("ADD ",$1,$3);}
        |       e '-' e         {mk_instr("SUB ",$1,$3);}
        |       e '*' e         {mk_instr("MUL ",$1,$3);}
        |       e '/' e         {mk_instr("DIV ",$1,$3);}
        |       id              {$$=getreg($1);}
        |       num             {$$=strdup($1);strcat($$,"H");}
;
%%
void main()
{
        yyin = fopen("intrcode.txt","r");
        yyparse();
        fclose(yyin);
}
char* getreg(char *iden)
{
        int i,result=regchk(iden);
        char *temp;
        if(result==-1)
        {
                for(i=0;i<4;i++)
                        if(strcmp(regcont[i],"$")==0)
                                break;
                if(i>4)
                {
                        temp = swapreg_content(iden);
                        printf("\nMOV %s,%s",temp,iden);
                        return temp;
                }
                else
                {
                        strcpy(regcont[i],iden);
                        printf("\nMOV %s,%s",regname[i],iden);
                        result = i;
                }
        }
        return regname[result];
}
```

```
void mk_instr(char *instr, char *op1, char *op2)
{
	char *code;
	code = strdup(instr);
	strcat(code,op1);
	strcat(code,",");
	strcat(code,op2);
	printf("\n%s",code);
}
void clr(char *reg)
{
	int i;
	for(i=0;i<4;i++)
	{
		if(strcmp(regname[i],reg)==0)
		{
			strcpy(regcont[i],"$");
			regcont_timer[i] = 0;
			break;
		}
	}
}
int yyerror(char *str)
{
	printf("%s",str);
	return -1;
}
int regchk(char *iden)
{
	int i;
	for(i=0;i<4;i++)
	{
		if(strcmp(regcont[i],iden)==0)
			return i;
	}
	return -1;
}
void inc_timer(void)
{
	int i=0;
	while(i)
```

```
        {
                regcont_timer[i]++;
                i++;
        }
}
char* swapreg_content(char *iden)
{
        int i,max=0;
        for(i=1;i<4;i++)
        {
                if(max<regcont_timer[i])
                        max = i;
        }
        printf("MOV %s,%s",regcont[max],regname[max]);
        regcont_timer[max] = 0;
        return regname[max];
}
```

SAMPLE INPUT:

intrcode.txt :

```
        a = b + c
        x = 2
        a = a * x
        d = b *10
        y = d/c
        y = e -5
```

SAMPLE OUTPUT:

```
MOV AX,b
MOV BX,c
ADD AX,BX
MOV a,AX

MOV x,2H

MOV AX,a
MOV CX,x
MUL AX,CX
```

```
MOV a,AX

MOV AX,b
MUL AX,10H
MOV d,AX

MOV AX,d
DIV AX,BX
MOV y,AX

MOV AX,e
SUB AX,5H
MOV y,AX
```

RESULT:

Thus the program for intermediate code generation has been compiled and executed successfully.

EX NO. : 14 IMPLEMENTATION OF CODE OPTIMISATION
DATE :

AIM:-

To write a C program to implement the code generation algorithm.

ALGORITHM:-

1. Start.
2. Invoke a function getreg to determine the location L where the result of the computation y op z should be stored. L will usually be a register, but it could also be a memory location. We shall describe getreg shortly.
3. Consult the address descriptor for y to determine y, (one of) the current location(s) of y. Prefer the register for y if the value of y is currently both in memory and a register. If the value of y is not already in l, generate the instruction MOV y, L to place a copy of y in L.
4. Generate the instruction op z, L where z is a current location of z. Again, prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If L is a register, update its descriptor to indicate that it contains the value of x, and remove x from all other register descriptors.
5. If the current values of y and/or z have no next users, are not live on exit from the block, and are in register descriptor to indicate that, after execution of x:=y op z, those registers no longer will contain y and/or z, respectively.
6. Stop.

PROGRAM:-

```
#include<stdio.h>
#include<string.h>
struct op
{
   char l[5];
   char r[20];
}op[10], pr[10];
void main()
{
   int n, i, j, k=0, m, a, b;
   char temp[10], t[10], *p, *q;
   printf("Enter no of values :: ");
   scanf("%d",&n);
   for(i=0;i<n;i++)
   {
      printf("\nLeft :: ");
      scanf("%s",op[i].l);
      printf("Right :: ");
      scanf("%s",op[i].r);
   }

   printf("\nIntermediate Code :: \n");
   for(i=0;i<n;i++)
```

```
{
    printf("\t%s = ",op[i].l);
    printf("%s\n",op[i].r);
}

for(i=0;i<n-1;i++)
{
    strcpy(temp, op[i].l);
    for(j=0;j<n;j++)
    {
        p = strstr(op[j].r, temp);
        if(p)
        {
            strcpy(pr[k].l, op[i].l);
            strcpy(pr[k].r, op[i].r);
            k++;
        }
    }
}
strcpy(pr[k].l, op[n-1].l);
strcpy(pr[k].r, op[n-1].r);
k++;

printf("\nAfter dead code elimination :: \n");
for(i=0;i<k;i++)
{
    printf("\t%s = ",pr[i].l);
    printf("%s\n",pr[i].r);
}

for(i=0;i<k;i++)
{
    strcpy(temp,pr[i].r);
    for(j=i+1;j<k;j++)
    {
        p = strstr(temp, pr[j].r);
        if(p)
        {
            strcpy(t, pr[j].l);
            strcpy(pr[j].l, pr[i].l);
            for(m=0;m<k;m++)
            {
                q = strstr(pr[m].r,t);
                if(q)
                {
                    a = q-pr[m].r;
                    pr[m].r[a] = pr[i].l[0];
                }
            }
        }
    }
}

printf("\nAfter common sub expression elimination :: \n");
for(i=0;i<k;i++)
{
    printf("\t%s = ",pr[i].l);
    printf("%s\n",pr[i].r);
}

//duplicate production elimination
for(i=0;i<k;i++)
{
    for(j=i+1;j<k;j++)
    {
        a=strcmp(pr[i].l,pr[j].l);
        b=strcmp(pr[i].r,pr[j].r);
        if(!a&&!b)
        {
            strcpy(pr[i].l,"\0");
            strcpy(pr[i].r,"\0");
        }
    }
}

printf("\nOptimized code ::\n");
for(i=0;i<k;i++)
{
    if((strcmp(pr[i].l,"\0")) != 0)
    {
        printf("\t%s = ",pr[i].l);
        printf("%s\n",pr[i].r);
    }
}
}
```

OUTPUT :

```
Enter no of values :: 5
Left :: a
Right :: 7

Left :: b
```

```
Right :: c+d

Left :: e
Right :: c+d

Left :: f
Right :: b+e

Left :: r
Right :: f

Intermediate Code ::
        a = 7
        b = c+d
        e = c+d
        f = b+e
        r = f

After dead code elimination ::
        b = c+d
        e = c+d
        f = b+e
        r = f
After common sub expression elimination ::
        b = c+d
        b = c+d
        f = b+b
        r = f

Optimized code ::
        b = c+d
        f = b+b
        r = f
```

RESULT :

Thus the program for code optimization is executed successfully and implemented.