Dept. of Computer Science and Engineering
Periyar Nagar, Vallam, Thanjavur-613 403, TamilNadu India
Phone +91 – 4362 264600, Fax +91– 4362 264660

MANUAL

XCS607 - COMPILER DESIGN LAB

Course Teacher
C.SNEHA (AP/CSE)

SYLLABUS

Subject Code	XCS607	L	T	P	C
Subject Name	Compiler Design Lab	0	0	3	2
		L	T	P	H
		0	0	3	3
LIST OF EXPERIMENTS					45

1. Construction of NFA
2. Construction of Minimized DFA
3. Implementation of Lexical Analyzer Using LexTool
4. Generation of Tokens for Given Lexeme
5. Conversion of Infix to Postfix Expression
6. Implementation of Symbol Table
7. Syntax Analysis using YACC
8. Implementation of Shift Reduce Parsing Algorithm
9. Construction of LR Parsing Table
10. Construction of Operator Precedence Parse Table
11. Implementation of Quadruples
12. Implementation of Triples
13. Implementation of Intermediate Code Generation
14. Implementation of Code Generation
15. Implementation of Code Optimization Techniques

Mini Project.
Total:45 Hrs
1.spoken-tutorial.org
2.http://vlab.co.in/cse08/

Ex.No:1 Construction of NFA

AIM : To write a program to Convert regular expression into NFA

ALGORITHM:

STEP1: Declare the necessary variables such as €,a, union, concatenation, kleen closure, parenthesis.

STEP2: Define the rule for €, and draw the transition table.

STEP3: Define the rule for a and draw the transition table.

STEP4: Define the rule for union and draw the transition table.

STEP5: Define the rule for concatenation and draw the transition table.

STEP6: Define the rule for kleen closure and draw the transition table.

STEP7: Define the rule for parenthesis(a) and draw the transition table.

STEP8: Combine the entire transition table.

STEP9: Display the result of NFA

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
int Fa[10][10][10],states[2][10],curr,row=0,col=0,sr=0,sc=0,th=0,in;
char *str;
int nfa(char *string,int state)
 {
   int i,j;
   for(i=0;i<=row;i++)
       {
        if(*string)
           {
         curr=Fa[state][*string-97][i];

           if (curr==-1)
             break;
           if(nfa(string+1,curr))
             return 1;
           }
          else
            {
           if(states[1][i]==-1)
          break;
```

```
            if(state==states[1][i])
           return 1;
            }
         }
         return 0;
     }
int main()
  {
    FILE *fp;
    int i,j,k,flag=0;
    char c,ch;
    clrscr();
    fp=fopen("Nfa_ip.txt","r");

    for(i=0;i<2;i++)
      for(j=0;j<10;j++)
    states[i][j]=-1;

    for(i=0;i<10;i++)
      for(j=0;j<10;j++)
     for(k=0;k<10;k++)
        Fa[i][j][k]=-1;

    while(fscanf(fp,"%d",&in)!=EOF)
      {
      fscanf(fp,"%c",&c);

       if(flag)
         {
        states[sr][sc++]=in;
        if(c=='\n')
           {
            sr++;
            sc=0;
           }
         }
       else if(c=='#')
         {
        flag=1;
        Fa[row][col][th]=in;
        printf("\nFa[%d][%d][%d]=%d",row,col,th,Fa[row][col][th]);
         }
        else if(!flag)
         {
        Fa[row][col][th]=in;
        printf("\nFa[%d][%d][%d]=%d",row,col,th,Fa[row][col][th]);
```

```
        if(c==',')
          {
           th++;
          }
        else if(c=='\n')
          {
            col=0;
            row++;
            th=0;
          }
        else if(c!=',')
         {
            col++;
            th=0;
         }
         }
     }
     printf("\n\nEnter the string : \n");
     scanf("%s",str);
     if(nfa(str,states[0][0]))
   printf("\nString Is Accepted");
      else
   printf("\nString Not Accepted");
    getch();
    return 0;
 }
```

SAMPLE OUTPUT:

Enter the string :
1,2 1
STRING ACCEPTED
Enter the string :
a 1 2 2
STRING NOT ACCEPTED

RESULT : Thus the program has been executed successfully and verified.

Ex.No:2 Construction of Minimized DFA

AIM : Write a program to Convert regular expression into DFA

ALGORITHM:

STEP1: Declare the necessary variables such as €,a, union, concatenation, kleen closure, parenthesis.

STEP2: Define the rule for €, and draw the transition table.

STEP3: Define the rule for a and draw the transition table.

STEP4: Define the rule for union and draw the transition table.

STEP5: Define the rule for concatenation and draw the transition table.

STEP6: Define the rule for kleen closure and draw the transition table.

STEP7: Define the rule for parenthesis(a) and draw the transition table.

STEP8: Combine the entire transition table.

STEP9: Display the result of DFA

PROGRAM

```
#include<stdio.h>
#include<conio.h>
int main()
{
 FILE * fp;
 int Fa[10][10],states[2][10],row=0,col=0,sr=0,sc=0,flag=0,i,j,in,curr;
 char k,*str;
 clrscr();
 fp = fopen("Dfa_ip.txt","r");
 if(fp==NULL)
 printf("file could not find\n");

 for(i=0;i<3;i++)
   for(j=0;j<10;j++)
     states[i][j]=-1;

 while(fscanf(fp,"%d",&in)!=EOF)
   {
     fscanf(fp,"%c",&k);

   if (flag)
    {
```

```
    states[sr][sc++]=in;
    if(k=='\n')
       {
      sr++;
      sc=0;
       }
  }
     else if(k=='#')
   {
     flag=1;
     Fa[row][col++]=in;
   }
  else if(!flag)
     {
       Fa[row][col++]=in;
       if(k=='\n')
         {
        row++;
        col=0;
         }
     }
  }

  printf("THE AUTOMATA IS : \n\n");
  for (i=0;i<=row;i++)
     {
     for (j=0;j<col;j++)
       {
      printf("%2d ",Fa[i][j]);
       }
       printf("\n");
     }

    printf("\n\nEnter the string : ");
    gets(str);
    curr=states[0][0];
    i=0;
    while(str[i]!='\0')
      {
     curr=Fa[curr][str[i]-97];
     if(curr==-1)
        break;
     i++;
      }

     flag=0;
```

```
    if(curr!=-1)
     {
   for(i=0;i<=sc&&!flag;i++)
     {
       if(curr==states[1][i])
      {
        printf("\n\nSTRING ACCEPTED\n");
        flag=1;
        break;
      }
     }
     }
     if(flag==0)
    printf("\n\nSTRING NOT ACCEPTED ");
 getch();
 return 0;

}
```

SAMPLE OUTPUT:

Enter the string : 1 3

STRING ACCEPTED

Enter the string : 5 5#

STRING ACCEPTED

Enter the string : a 1 3

STRING NOT ACCEPTED

RESULT : Thus the program has been executed successfully and verified.

Ex.no.3 Implementation of Lexical AnalyzerUsing LexTool

AIM: To write a program for lexical analyzer to using lex tool

ALGORITHM:

STEP 1: Open gedit text editor from accessories in applications

STEP 2: Specify the header files to be included inside the declaration part (i.e. between %{ and %})

STEP 3: Define the digits i.e. 0-9 and identifiers a-z and a-z 0-9

STEP 4: Using translation rule, we defined the regular expression for digit, if it is matched with the given input then store and display it as digit in yytext.

STEP 5: Using translation rule, we defined the regular expression for keywords, if it is matched with the given input then store and display it as keyword in yytext.

STEP 6: Using translation rule, we defined the regular expression for identifiers, if it is matched with the given input then store and display it as identifier in yytext.

STEP 7: Using translation rule, we defined the regular expression for operators, if it is matched with the given input then store and display it as operator in yytext.

STEP 8: Inside procedure main(),use yyin() to point the current file being parsed by the lexer.

STEP 9: The call yylex(), which starts the analysis.

PROGRAM:

```
%{
#include<math.h>
#include<stdlib.h>
%}

DIGIT [0-9]
ID       [a-z][a-z 0-9]*
%%

{DIGIT}+
        {
```

```
          printf("An integer:%s(%d)\n",yytext,atoi(yytext));
          }
{DIGIT}+"."{DIGIT}*
         {
          printf("A float:%s(%g)\n",yytext,atof(yytext));
          }

if|then|begin|end|procedure|function
          {
          printf("A keyboard:%s\n",yytext);
          }
{ID}
    printf("An identifier:%s\n",yytext);

"+"|"-"|"*"|"/"
    printf("An operator:%s\n",yytext);

[\t\n]+
    printf("Unrecognized chareacter:%s\n",yytext);
%%

main(argc,argv)
int argc;
char **argv;
{
          ++argv,--argc;
          if(argc > 0)
          yyin=fopen(argv[0],"r");
          else
          yyin=stdin;
          yylex();
}
```

FILE PROGRAM:

```
ram is a good boy
sita is a good girl
gowtham is a clever boy
geetha is a smart girl
a+b*c/d=e.
```

SAMPLE OUTPUT:

```
An identifier:ram is a good boy
An identifier:sita is a good girl
An identifier:gowtham is a clever boy
An identifier:geetha is a smart girl
An identifier:a
An operator:+
An identifier:b
An operator:*
An identifier:c
An operator:/
An identifier:d
Unrecognized chareacter:=
```

RESULT : Thus the program has been executed successfully and verified.

Ex.4. Generation of Tokens for Given Lexeme

AIM: To write a program to implement a lexical analyser in C.

ALGORITHM:

STEP1: Define header files, sub functions and declare the variables required.

STEP2: Inside main(), call the sub functions such as input(),operation(),output().

STEP3: Inside the sub function input (), read the source program from the user using while loop, i.e while (a[c]! ='A')

STEP4: Inside the sub-functionoperation (), do the following steps (i.e from step 5 to step 9)

STEP5: Check for blank space, tab space etc., if so then increment the variable line by else proceed with the next step.

STEP6: Check for digits, using the inbuilt function isdigit (), if so then increment the variable digit by 1, else proceed with the next step.

STEP7: Check for special symbols, using the inbuilt function punct (), if so then check for operators (ie. +,-,*,/) and increment the variable operator by 1 else increment the variable special by 1.

STEP8: Check for the words using the inbuilt function isalpha(), if so then assign it to the variable word and compare the word with the keywords specified and increment the variable keyword by 1.

STEP9: Inside the sub function output(), display all the tokens such as digits,special symbols,keywords,identifiers etc using appropriate variables and for loop.

PROGRAM:

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

void removeduplicate();
void final();
int Isiden(char ch);
int Isop(char ch);
int Isdel(char ch);
int Iskey(char * str);
```

```
int Isalpha(char *str);
int Isdigit(char *str);

char op[8]={'+','-','*','/','=','<','>','%'};
char del[8]={'}','{',';','(',')','[',']',','};
char *key[]={"int","void","main","char","float"};

//char *operato[]={"+","-","/","*","<",">","=","%","<=",">=","++"};

int idi=0,idj=0,k,opi=0,opj=0,deli=0,uqdi=0,uqidi=0,uqoperi=0,kdi=0,liti=0,ci=0;
int uqdeli[20],uqopi[20],uqideni[20],l=0,j;
char uqdel[20],uqiden[20][20],uqop[20][20],keyword[20][20];
char iden[20][20],oper[20][20],delem[20],litral[20][20],lit[20],constant[20][20];

void lexanalysis(char *str)
 {
  int i=0;
  while(str[i]!='\0')
   {
   if(Isiden(str[i]))     //for identifiers
     {
       while(Isiden(str[i]))
      {
         iden[idi][idj++]=str[i++];
      }
       iden[idi][idj]='\0';
       idi++;idj=0;
     }
     else
     if(str[i]=='"')         //for literals
       {
       lit[l++]=str[i];
       for(j=i+1;str[j]!='"';j++)
         {
          lit[l++]=str[j];
         }
       lit[l++]=str[j];lit[l]='\0';
       strcpy(litral[liti++],lit);
       i=j+1;
       }
     else
     if(Isop(str[i]))        // for operators
       {
       while(Isop(str[i]))
         {
          oper[opi][opj++]=str[i++];
```

```
            }
          oper[opi][opj]='\0';
          opi++;opj=0;
           }
        else
        if(Isdel(str[i]))     //for delemeters
           {
            while(Isdel(str[i]))
             {
               delem[deli++]=str[i++];
             }
            }
         else
            {
            i++;
         }
      }

   removeduplicate();
   final();
}

int Isiden(char ch)
 {
   if(isalpha(ch)||ch=='_'||isdigit(ch)||ch=='.')
   return 1;
   else
   return 0;
 }

int Isop(char ch)
 {
  int f=0,i;
  for(i=0;i<8&&!f;i++)
    {
     if(ch==op[i])
      f=1;
    }
 return f;
}

int Isdel(char ch)
 {
  int f=0,i;
  for(i=0;i<8&&!f;i++)
    {
```

```
   if(ch==del[i])
    f=1;
  }
 return f;
 }

int Iskey(char * str)
{
 int i,f=0;
 for(i=0;i<5;i++)
  {
   if(!strcmp(key[i],str))
     f=1;
   }
 return f;
}

void removeduplicate()
 {
  int i,j;
  for(i=0;i<20;i++)
   {
    uqdeli[i]=0;
    uqopi[i]=0;
    uqideni[i]=0;
   }
  for(i=1;i<deli+1;i++)  //removing duplicate delemeters
    {
     if(uqdeli[i-1]==0)
    {
         uqdel[uqdi++]=delem[i-1];
         for(j=i;j<deli;j++)
        {
            if(delem[i-1]==delem[j])
             uqdeli[j]=1;
        }
     }
    }

   for(i=1;i<idi+1;i++)  //removing duplicate  identifiers
      {
     if(uqideni[i-1]==0)
       {
       strcpy(uqiden[uqidi++],iden[i-1]);
       for(j=i;j<idi;j++)
        {
```

```
          if(!strcmp(iden[i-1],iden[j]))
            uqideni[j]=1;
        }
        }
    }

    for(i=1;i<opi+1;i++)  //removing duplicate  operators
    {
      if(uqopi[i-1]==0)
        {
        strcpy(uqop[uqoperi++],oper[i-1]);
        for(j=i;j<opi;j++)
          {
           if(!strcmp(oper[i-1],oper[j]))
             uqopi[j]=1;
          }
        }
    }

}
void final()
 {
  int i=0;
  idi=0;
  for(i=0;i<uqidi;i++)
   {
    if(Iskey(uqiden[i]))        //identifying keywords
    strcpy(keyword[kdi++],uqiden[i]);
    else
     if(isdigit(uqiden[i][0]))    //identifying constants
    strcpy(constant[ci++],uqiden[i]);
    else
    strcpy(iden[idi++],uqiden[i]);
   }

// printing the outputs

printf("\n\t Delimiter are : \n");
for(i=0;i<uqdi;i++)
printf("\t%c\n",uqdel[i]);

printf("\n\tOperators are : \n");
for(i=0;i<uqoperi;i++)
 {
 printf("\t");
 puts(uqop[i]);
```

```
 }

printf("\n\tIdentifiers are : \n");
for(i=0;i<idi;i++)
 {
  printf("\t");
  puts(iden[i]);
 }

printf("\n\tKeywords are : \n");
for(i=0;i<kdi;i++)
 {
  printf("\t");
  puts(keyword[i]);
 }

printf("\n\tConstants are :\n");
for(i=0;i<ci;i++)
  {
  printf("\t");
  puts(constant[i]);
  }

printf("\n\tLiterals are :\n");
for(i=0;i<liti;i++)
 {
   printf("\t");
   puts(litral[i]);
 }
}
void main()
{
  char str[50];
  //clrscr();
  printf("\nEnter the string : ");
  scanf("%[^\n]c",str);
  lexanalysis(str);
  //getch();
}
```

SAMPLE OUTPUT:

Enter the string : #include<stdio.h> void main() { printf(“hi”); }
Delimiters are:
(
)
{
;
}

Operators are :
<
>
Identifiers are :
include
stdio.h
printf

Keywords are :
Void
main

Constants are:

Literals are :
“hi”

RESULT : Thus the program has been executed successfully and verified.

5. Conversion of Infix to Postfix Expression

AIM: Write a program to implement the Infix to Postfix expression.

ALGORITHM:

STEP 1 : Scan the Infix Expression from left to right.

STEP 2 : If the scanned character is an operand, append it with final Infix to Postfix string.

STEP 3 : Else,

STEP 3.1 : If the precedence order of the scanned(incoming) operator is greater than the precedence order of the operator in the stack (or the stack is empty or the stack contains a '(' or '[' or '{'), push it on stack.

STEP 3.2 : Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

STEP 4 : If the scanned character is an '(' or '[' or '{', push it to the stack.

STEP 5 : If the scanned character is an ')'or ']' or '}', pop the stack and and output it until a '(' or '[' or '{' respectively is encountered, and discard both the parenthesis.

STEP 6 : Repeat steps 2-6 until infix expression is scanned.

STEP 7 : Print the output

STEP 8 : Pop and output from the stack until it is not empty.

PROGRAM:

```
#include<stdio.h>
char stack[20];
int top = -1;
void push(char x)
{
   stack[++top] = x;
}

char pop()
{
   if(top == -1)
      return -1;
   else
      return stack[top--];
}

int priority(char x)
```

```
{
   if(x == '(')
      return 0;
   if(x == '+' || x == '-')
      return 1;
   if(x == '*' || x == '/')
      return 2;
}

main()
{
   char exp[20];
   char *e, x;
   printf("Enter the expression :: ");
   scanf("%s",exp);
   e = exp;
   while(*e != '\0')
   {
      if(isalnum(*e))
         printf("%c",*e);
      else if(*e == '(')
         push(*e);
      else if(*e == ')')
      {
         while((x = pop()) != '(')
            printf("%c", x);
      }
      else
      {
         while(priority(stack[top]) >= priority(*e))
            printf("%c",pop());
         push(*e);
      }
      e++;
   }
   while(top != -1)
   {
      printf("%c",pop());
   }
}
```

SAMPLE OUTPUT:

```
Enter the expression : a+b*c

abc*+
```

Enter the expression : (a+b)*c+(d-a)

ab+c*da-+

RESULT : Thus the program has been executed successfully and verified.

Ex.No:6 Implementation of Symbol Table

AIM: To write a "C" program for the implementation of symbol table.

ALGORITHM:

STEP 1: Start the program.

STEP 2: Get the input from the user with the terminating symbol $.

STEP 3: Allocate memory for the variable by dynamic memory allocation function.

STEP 4: If the next character of the symbol is an operator then only the memory is allocated.

STEP 5: While reading, the input symbol is inserted into symbol table along with its memory address.

STEP 6: The steps are repeated till $ is reached.

STEP 7: To reach a variable, enter the variable to be searched and symbol table has been checked for corresponding variable, the variable along its address is displayed as result.

STEP8 : Stop the program.

PROGRAM :

```
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
void main()
{
 int i=0,j=0,x=0,n, flag=0;
 void *p,*add[5];
 char ch,srch,b[15],d[15],c;
 printf("Expression terminated by $:");
 while((c=getchar())!='$')
 {
  b[i]=c;
  i++;
 }
 n=i-1;
 printf("Given Expression:");
 i=0;
 while(i<=n)
 {
```

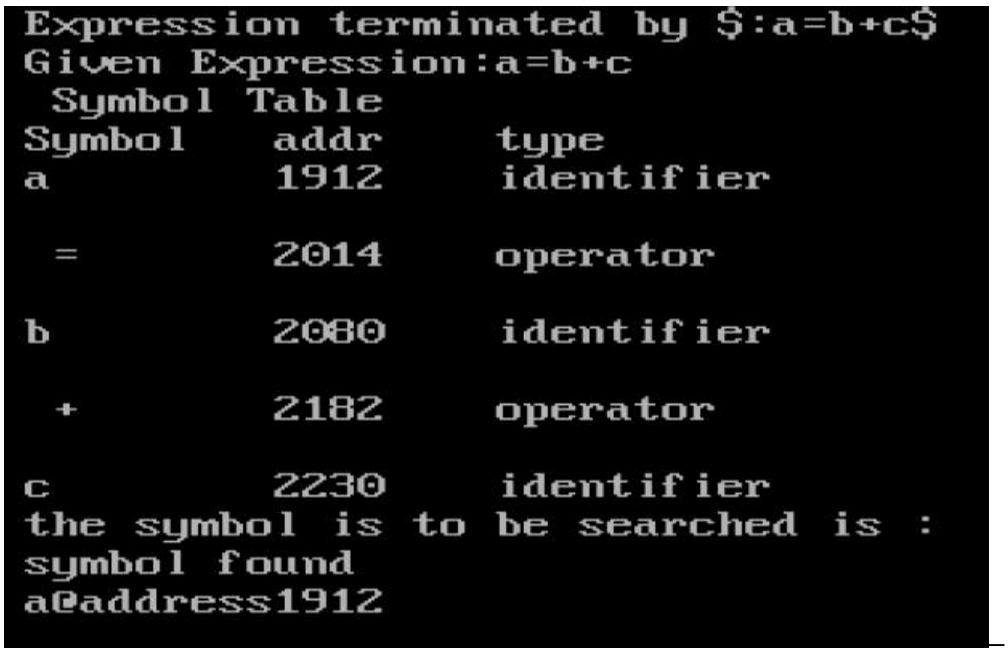
```
 printf("%c",b[i]);
 i++;
}
printf("\n Symbol Table\n");
printf("Symbol \t addr \t type");
while(j<=n)
{
 c=b[j];
 if(isalpha(toascii(c)))
 {
  p=malloc(c);
  add[x]=p;
  d[x]=c;
  printf("\n%c \t %d \t identifier\n",c,p);
  x++;
  j++;
 }
 else
 {
  ch=c;
  if(ch=='+'||ch=='-'||ch=='*'||ch=='=')
  {
   p=malloc(ch);
   add[x]=p;
   d[x]=ch;
   printf("\n %c \t %d \t operator\n",ch,p);
   x++;
   j++;
  }
 }
  }
printf("the symbol is to be searched is : \n");
srch= getch();
for(i=0;i<=x;i++)
{
if(srch==d[i])
{
printf("symbol found\n");
printf("%c%s%d\n",srch,"@address",add[i]);
flag=1;
  }
  }

if(flag==0)
printf("symbol not found\n");
getch();
```

}

-

SAMPLE OUTPUT:



RESULT : Thus the program has been executed successfully and verified.

Ex.No:7 Syntax Analysis using YACC

AIM: To implement Syntax Analysis using YACC

Algorithm :

STEP 1 : Start the program
STEP 2: Definition section include header files in LEX.
STEP 3: Define the regular expression
STEP 4: Define the regular expression for log,sin,cos,tan and return the values using yytext.
STEP 5: Using YAC, define the token.
STEP 6: Write the translation rule for addition, multiplication, division, subtraction
STEP 7: Define the LOG expression
$$=log($2)/log(10);|nLOG expression $$=log($2);
SINE expression { $$ = sin($2*3.141592654/180);
COS expression {$=cos($2*3.141592654/180);
|TAN expression {$$=tan($2 = tan($2*3.141592654/180);
|NUMBER {$$ = $1;}
|MEM {$$ = memvar;}
STEP 8: Define the main function and call the YYPARSE() and YYERROR();
STEP 9: Stop the program.

A. LEX SPECIFICATION

```
%{
#include"y.tab.h"
#include<math.h>
%}
%%
([0-9]+|([0-9]*\.[0-9]+)([eE][\-+]?[0-9]+)?){
yylval.dval=atof(yytext);
return NUMBER;
}
log |
LOG { return LOG; }
ln { return nLOG; }
sin |
SIN { return SINE; }
cos |
COS { return COS; }
tan |
TAN { return TAN; }
mem |
```

```
MEM { return MEM; }
[\t];
\$ { return 0; }
\n |
. { return yytext[0]; }
%%
```

B. YACC SPECIFICATION

```
%{
double memvar;
%}
%union
{
        double dval;
}
%token <dval> NUMBER;
%token <dval> MEM;
%token LOG SINE nLOG COS TAN
%left '-''+'
%left '*''/'
%right '^'
%left LOG SINE  nLOG  COS TAN
%nonassoc UMINUS
%type <dval>expression
%%
start:statement'\n'
| start statement'\n'
;
statement: MEM '=' expression { memvar=$3; }
| expression { printf("Answer=%g\n",$1); }
;
expression: expression '+' expression { $$=$1 + $3; }
|expression '-' expression { $$ = $1 - $3; }
|expression '*' expression { $$ = $1 * $3; }
|expression '/' expression
{
        if( $3 == 0)
        yyerror("Divide by zero");
        else
        $$ = $1 / $3;
}
|expression '^' expression { $$=pow($1,$3); }
expression:'-' expression %prec UMINUS ( $$=-$2; }
|'('expression')' { $$=$2; }
```

```
|LOG expression { $$=log($2)/log(10); }
|nLOG expression { $$=log($2); }
|SINE expression { $$=sin($2*3.141592654/180; }
|COS expression { $$=cos($2*3.141592654/180); }
|TAN expression { $$=tan($2=tan($2*3.141592654/180); }
|NUMBER { $$=$1; }
|MEM { $$=memvar; }
;
%%
main()
{
        printf("\n Enter expression:");
        yyparse();
}
int yyerror(char *error)
{
        fprintf("%s\n",error);
}
```

SAMPLE OUTPUT:

```
[user50@localhost~]$ lex ll.l
[user50@localhost~]$ yacc yy.y
[user50@localhost~]$ yacc –d yy.y
[user50@localhost~]$ cc y.tab.c lex.yy.c –ll –lc -lm
[user50@localhost~]$ ./a.out

Enter expression:5+3
Answer= 8
5*2
Answer= 10
cos 0
Answer= 1
sin 90
Answer= 1
tan 45
Answer= 1
```

RESULT : Thus the program has been executed successfully and verified.

Ex.No: 8 Implementation of Shift Reduce Parsing Algorithm

AIM : To write a C program to perform the shift reduce parsing.

ALGORITHM:

STEP 1: Start the program
STEP 2: Define the main function
STEP 3: Declare array for string and stack and other necessary variables.
STEP 4: Get the expression from the user and store it as atring.
STEP 5: Append $ to the end of the string.
STEP 6: Store $ into the stack.
STEP 7: Print three columns as ‘stack’ , ‘String’ and ‘Action’ for the respective actions.
STEP 8: Use for loop from I as 0 till string length and check the string.
STEP 9: If string has some operator or id, push it to the stack.
STEP 10: Mark this action as ‘shift’.
STEP 11: Print the stack, string and action values.
STEP 12: If stack contains some production on shifting, reduce it.
STEP 13: Mark this action as ‘Reduce’.
STEP 14: Print the stack, string and action values.
STEP 15: Repeat steps 9 to 14 again and again till the for loop is valid.
STEP 16: Now check the string and the stack.
STEP 17: If the string contains only $ and the stack has only $E within it, then print that the ‘given string is valid’.
STEP 18:Else print the ‘given string is invalid’.
STEP 19: End of the program.

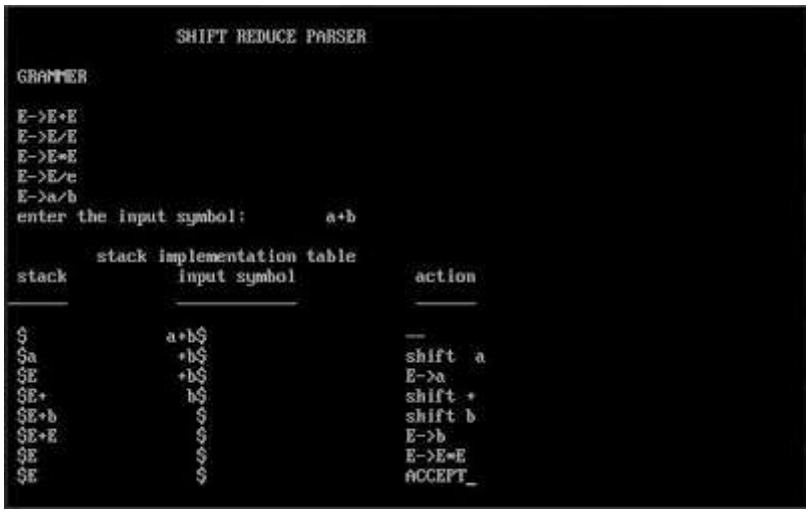
PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<string.h>
char ip_sym[15], stack[15];
int ip_ptr=0,st_ptr=0,len,i;
char temp[2],temp2[2];
char act[15];
void check();
void main()
{
clrscr();
printf("\n\t\t SHIFT REDUCE PARSER\n");
printf("\n GRAMMER\n");
printf("\n E->E+E\n E->E/E");
```

```
printf("\n E->E*E\n E->a/b");
printf("\n enter the input symbol:\t");
gets(ip_sym);
printf("\n\t stack implementation table");
printf("\n stack\t\t input symbol\t\t action");
printf("\n______\t\t _____________\t\t ______\n");
printf("\n $\t\t%s$\t\t\t--",ip_sym);
strcpy(act,"shift ");
temp[0]=ip_sym[ip_ptr];
temp[1]='\0';
strcat(act,temp);
len=strlen(ip_sym);
for(i=0;i<=len-1;i++)
{
stack[st_ptr]=ip_sym[ip_ptr];
stack[st_ptr+1]='\0';
ip_sym[ip_ptr]=' ';
ip_ptr++;
printf("\n $%s\t\t%s$\t\t\t%s",stack,ip_sym,act);
strcpy(act,"shift ");
temp[0]=ip_sym[ip_ptr];
temp[1]='\0';
strcat(act,temp);
check();
st_ptr++;
}
st_ptr++;
check();
}
void check()
{
int flag=0;
temp2[0]=stack[st_ptr];
temp2[1]='\0';
if((!strcmpi(temp2,"a"))||(!strcmpi(temp2,"b")))
{
 stack[st_ptr]='E';
 if(!strcmpi(temp2,"a"))
  printf("\n $%s\t\t%s$\t\t\tE->a",stack, ip_sym);
 else
  printf("\n $%s\t\t%s$\t\t\tE->b",stack,ip_sym);
 flag=1;
}
if((!strcmpi(temp2,"+"))||(strcmpi(temp2,"*"))||(!strcmpi(temp2,"/")))
{
 flag=1;
```

```
}
if((!strcmpi(stack,"E+E"))||(!strcmpi(stack,"E\E"))||(!strcmpi(stack,"E*E")))
{
strcpy(stack,"E");
st_ptr=0;
if(!strcmpi(stack,"E+E"))
printf("\n $%s\t\t%s$\t\t\tE->E+E",stack,ip_sym);
else
if(!strcmpi(stack,"E\E"))
printf("\n $%s\t\t %s$\t\t\tE->E\E",stack,ip_sym);
else
printf("\n $%s\t\t%s$\t\t\tE->E*E",stack,ip_sym);
flag=1;
}
if(!strcmpi(stack,"E")&&ip_ptr==len)
{
printf("\n $%s\t\t%s$\t\t\tACCEPT",stack,ip_sym);
getch();
exit(0);
}
if(flag==0)
{
printf("\n%s\t\t\t%s\t\t reject",stack,ip_sym);
exit(0);
}
return;
}
```

SAMPLE OUTPUT:



RESULT : Thus the program has been executed successfully and verified.

Ex.No: 9 Construction of LR Parsing Table

AIM: To write a program for construction of LR Parsing table using C.

ALGORITHM:

STEP 1: Get the input expression and store it in the input buffer.

STEP 2: Read the data from the input buffer one at the time and convert into corresponding Non Terminal using production rules available.

STEP 3: Perform push & pop operation for LR parsing table construction.

STEP 4: Display the result with conversion of corresponding input symbols

PROGRAM :

```
#include<stdio.h>
#include<conio.h>
char stack[30];
int top=-1;
void push(char c)
{
  top++;
  stack[top]=c;
}
char pop()
{
  char c;
  if(top!=-1)
 {
        c=stack[top];
top--;
       return c;
         }
         return'x';
}
void printstat()
{
         int i;
         printf("\n\t\t\t $");
         for(i=0;i<=top;i++)
         printf("%c",stack[i]);
}
void main()
{
         int i,j,k,l;
         char
         s1[20],s2[20],ch1,ch2,
         ch3;clrscr();
```

```
            printf("\n\n\t\t LR PARSING");
            printf("\n\t\t ENTER THE EXPRESSION");
            scanf("%s",s1);
            l=strlen(s1);
j=0;
           printf("\n\t\t $");
           for(i=0;i<l;i++)
           {
            if(s1[i]=='i' && s1[i+1]=='d')
            {
                        s1[i]=' ';
                        s1[i+1]='E';
                        printstat();
                        printf("id");
                        push('E');
                        printstat();
            }
            else if(s1[i]=='+'||s1[i]=='-'||s1[i]=='*' ||s1[i]=='/' ||s1[i]=='d')
            {
                    push(s1[i]);printstat();
            }
           }
            printstat(); l=strlen(s2);while(l)
            {
           ch1=pop();
           if(ch1=='x')
           {
           printf("\n\t\t\t $");break;
           }
           if(ch1=='+'||ch1=='/'||ch1=='*'||ch1=='-')
           {
           ch3=pop();
           if(ch3!='E')
           {
             printf("errror");
             exit();
           }
           else
           {
             push('E');
             printstat();
           }
         }
 ch2=ch1;
  }
  getch();
```

```
}
```

SAMPLE OUTPUT:

```
LR PARSING
ENTER THE EXPRESSION
id+id*id-id
$
$id
$E
$E+
$E+id
$E+E
$E+E*
$E+E*id
$E+E*E
$E+E*E-
$E+E*E-id
$E+E*E-E
$E+E*E-E
$E+E*E
$E
$
```

RESULT : Thus the program has been executed successfully and verified.

Ex.No:10 Construction of Operator Precedence Parse Table

AIM: To write a program for construction of Operator Precedence Parse Tableusing C.

ALGORITHM:

1. Get the input as expression.
2. Check the preference of each symbol in the given expression with all other symbols in the input expression.
3. After checking display the Operator Precedence matrix with corresponding symbol comparison and specify it using the relational operators< (Less than), > (Greater than) and =(equals).
4. Also display the postfix expression format of the given expressionas tree structure.

PROGRAM:

```
include<stdio.h>
#include<conio.h>
void main(){
int i,j,k,n,top=0,col,row;
clrscr();
for(i=0;i<10;i++)
{
 stack[i]=NULL;
 ip[i]=NULL;
 for(j=0;j<10;j++)
 {
 opt[i][j][1]=NULL;
 }
}
printf("Enter the no.of terminals :\n");
scanf("%d",&n);
printf("\nEnter the terminals :\n");
scanf("%s",&ter);
printf("\nEnter the table values :\n");
for(i=0;i<n;i++)
{
 for(j=0;j<n;j++)
 {
 printf("Enter the value for %c %c:",ter[i],ter[j]);
 scanf("%s",opt[i][j]);
 }
}
printf("\n**** OPERATOR PRECEDENCE TABLE ****\n");
for(i=0;i<n;i++)
{
```

```
printf("\t%c",ter[i]);
}
printf("\n");
for(i=0;i<n;i++){printf("\n%c",ter[i]);
for(j=0;j<n;j++){printf("\t%c",opt[i][j][0]);}}
stack[top]='$';
printf("\nEnter the input string:");
scanf("%s",ip);
i=0;
printf("\nSTACK\t\t\tINPUT STRING\t\t\tACTION\n");
printf("\n%s\t\t\t%s\t\t\t",stack,ip);
while(i<=strlen(ip))
{
for(k=0;k<n;k++)
{
if(stack[top]==ter[k])
col=k;
if(ip[i]==ter[k])
row=k;
}
if((stack[top]=='$')&&(ip[i]=='$')){
printf("String is accepted\n");
break;}
else if((opt[col][row][0]=='<') ||(opt[col][row][0]=='='))
{ stack[++top]=opt[col][row][0];
stack[++top]=ip[i];
 printf("Shift %c",ip[i]);
 i++;
  }
else{
 if(opt[col][row][0]=='>')
{
while(stack[top]!='<'){--top;}
top=top-1;
printf("Reduce");
}
else
{
printf("\nString is not accepted");
break;
}
}
printf("\n");
for(k=0;k<=top;k++)
{
printf("%c",stack[k]);
```

```
}
printf("\t\t\t");
for(k=i;k<strlen(ip);k++){
printf("%c",ip[k]);
}
printf("\t\t\t");
}
getch();
}
/*
```

SAMPLE OUTPUT:

```
Enter the value for * *:>
Enter the value for * $:>
Enter the value for $ i:<
Enter the value for $ +:<
Enter the value for $ *:<
Enter the value for $ $:accept

**** OPERATOR PRECEDENCE TABLE ****
      i     +     *     $

i     e     >     >     >
+     <     >     <     >
*     <     >     >     >
$     <     <     <     a
*/

Enter the input string:
i*i

STACK              INPUT STRING              ACTION

$                  i*i                  Shift i
$<i                 *i                   Reduce
$                  *i                   Shift *
$<*                 i                    Shift i
$<*<i

String is not accepted
```

RESULT : Thus the program has been executed successfully and verified.

11. Implementation of Quadruples

AIM: To write a C program to generate a Quadruples.

ALGORITHM:

STEP 1 : Start the program.

STEP 2 : Enter the expression.

STEP 3 : Define the table for operator, temp1, temp2, result

STEP 4: Each string is compared with an operator, if any operator seen the previous string as well as next string are concatenated and stored in a first temporary value and the three address code expression is printed.

STEP 5: Suppose if another operand is seen then first temporary value is concatenated to expression is printed.

STEP 6: End of the program

PROGRAM:

```
#include<stdio.h>
#include<string.h>
main()
{
char line[20];
int s[20];
int t=1;
int i=0;
printf("Enter the string  :");
gets(line);
for(i=0;i<20;i++)
s[i]=0;
printf("op\ta1\ta2\tres\n");
for(i=2;line[i]!='\0';i++)
 {
  if(line[i] == '/' || line[i] =='*')
   {
          printf("\n");
      if(s[i]==0)
       {
        if(s[i+1] ==0)
```

```
        {
         printf(:=\t%c\t\t t%d\n",line[i+1],t);
         s[i+1] = t++;
        }
       printf("%c\t",line[i]);
       (s[i-1]==0)?printf("%c\t",line[i-1]):printf("t%d\t",s[i-1]):
       printf("t%d\t t%d",s[i+1],t);
       s[i-1]=s[i+1]=t++;
       s[i]=1;
      }
}
}
for(i=2;line[i]!='\0';i++)
{
if(line[i]=='+'||line[i]=='-')
{
printf("\n");
if(s[i]==0)
{
if(s[i+1]==0)
{
printf(:=\t%c\t\t t%d\n",line[i+1],t);
s[i+1] = t++;
}
printf("%c\t",line[i]);
(s[i-1]==0)?printf("%c\t",line[i-1]):printf("t%d\t",s[i-1]):
printf("t%d\t\t%d",s[i+1],t);
s[i-1]=s[i+1]=t++;
s[i]=1;
}
}
}
printf("\n:=\tt%d\t\t%c",t-1,line[0]);
getch();
}
```

SAMPLE OUTPUT

Enter a String : a = b*c – b*c

op	a1	a2	res
:=	c		t1
*	b	t1	t2
:=	c		t3
*	b	t3	t4
-	t2	t4	t5
:=	t5		a

RESULT : Thus the program has been executed successfully and verified.

12. Implementation of Triples

AIM: To write a C program to generate a triple

ALGORITHM:

STEP 1: Start the program
STEP 2: Define the data in definition section
STEP 3: Define the quadruple table and triple table
STEP 4: Write requlars expression for NUMBER and LETTER
STEP 5: Define the three address code using array
STEP 6: Define the quadruple using array
STEP 7: Get the input and do the pattern matching concept and seprate the code in the form triples,
STEP 8: Call the yylex (), yyerror, yywrap () function.
STEP 9: End of the program

PROGRAM:

```
* LEX FILE */

%{
#include "y.tab.h"
extern char yyval;
%}

NUMBER [0-9]+
LETTER [a-zA-Z]+

%%

{NUMBER} {yylval.sym=(char)yytext[0]; return NUMBER;}
{LETTER} {yylval.sym=(char)yytext[0];return LETTER;}

\n {return 0;}
. {return yytext[0];}

%%

/* yacc file */

%{

#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void ThreeAddressCode();
```

```
void triple();
void qudraple();
char AddToTable(char ,char, char);

int ind=0;
char temp='A';
struct incod
{
char opd1;
char opd2;
char opr;
};
%}

%union
{
char sym;
}

%token <sym> LETTER NUMBER
%type <sym> expr
%left '-"+'
%right '*"/'

%%

statement: LETTER '=' expr ';' {AddToTable((char)$1,(char)$3,'=');}
| expr ';'
;

expr: expr '+' expr {$$ = AddToTable((char)$1,(char)$3,'+');}
| expr '-' expr {$$ = AddToTable((char)$1,(char)$3,'-');}
| expr '*' expr {$$ = AddToTable((char)$1,(char)$3,'*');}
| expr '/' expr {$$ = AddToTable((char)$1,(char)$3,'/');}
| '(' expr ')' {$$ = (char)$2;}
| NUMBER {$$ = (char)$1;}
| LETTER {$$ = (char)$1;}
;

%%

yyerror(char *s)
{
printf("%s",s);
exit(0);
}
```

```
struct incod code[20];

int id=0;

char AddToTable(char opd1,char opd2,char opr)
{
code[ind].opd1=opd1;
code[ind].opd2=opd2;
code[ind].opr=opr;
ind++;
temp++;
return temp;
}

void ThreeAddressCode()
{
int cnt=0;
temp++;
printf("\n\n\t THREE ADDRESS CODE\n\n");
while(cnt<ind)
{
printf("%c : = \t",temp);
if(isalpha(code[cnt].opd1))
printf("%c\t",code[cnt].opd1);
else
{printf("%c\t",temp);}

printf("%c\t",code[cnt].opr);

if(isalpha(code[cnt].opd2))
printf("%c\t",code[cnt].opd2);
else
{printf("%c\t",temp);}

printf("\n");
cnt++;
temp++;
}
}

void quadraple()
{
int cnt=0;
temp++;
printf("\n\n\t QUADRAPLE CODE\n\n");
while(cnt<ind)
```

```
{
//printf("%c : = \t",temp);
printf("%d",id);
printf("\t");
printf("%c",code[cnt].opr);
printf("\t");
if(isalpha(code[cnt].opd1))
printf("%c\t",code[cnt].opd1);
else
{printf("%c\t",temp);}

//printf("%c\t",code[cnt].opr);

if(isalpha(code[cnt].opd2))
printf("%c\t",code[cnt].opd2);
else
{printf("%c\t",temp);}

printf("%c",temp);

printf("\n");
cnt++;
temp++;
id++;

}
}

void triple()
{
int cnt=0,cnt1,id1=0;
temp++;
printf("\n\n\t TRIPLE CODE\n\n");
while(cnt<ind)
{
//printf("%c : = \t",temp);

if(id1==0)
{
printf("%d",id1);
printf("\t");
printf("%c",code[cnt].opr);
printf("\t");
if(isalpha(code[cnt].opd1))
printf("%c\t",code[cnt].opd1);
else
```

```
{printf("%c\t",temp);}

//printf("%c\t",code[cnt].opr);
cnt1=cnt-1;
if(isalpha(code[cnt].opd2))
printf("%c",code[cnt].opd2);
else
{printf("%c\t",temp);}
}
else
{
printf("%d",id1);
printf("\t");
printf("%c",code[cnt].opr);
printf("\t");
if(isalpha(code[cnt].opd1))
printf("%c\t",code[cnt].opd1);
else
{printf("%c\t",temp);}

//printf("%c\t",code[cnt].opr);
cnt1=cnt-1;
if(isalpha(code[cnt].opd2))
printf("%d",id1-1);
else
{printf("%c\t",temp);}
}

printf("\n");
cnt++;
temp++;
id1++;

}

}

main()
{
printf("\nEnter the Expression: ");
yyparse();
temp='A';
ThreeAddressCode();
quadraple();
triple();
}
```

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

SAMPLE OUTPUT

```
administrator@ubuntu:~/Desktop$ flex th.l
administrator@ubuntu:~/Desktop$ yacc -d th.y
administrator@ubuntu:~/Desktop$ gcc lex.yy.c y.tab.c -ll -lm
administrator@ubuntu:~/Desktop$ ./a.out

Enter the Expression: a=((b+c)*(d+e))
syntax error

administrator@ubuntu:~/Desktop$ ./a.out

Enter the Expression: a=((b+c)*(d/e));
THREE ADDRESS CODE

B : = b + c
C : = d / e
D : = B * C
E : = a = D
QUADRAPLE CODE

0 + b c G
1 / d e H
2 * B C I
3 = a D J
TRIPLE CODE

0 + b c
1 / d 0
2 * B 1
3 = a 2
```

RESULT : Thus the program has been executed successfully and verified.

13. Implementation of Intermediate Code Generation

AIM : To write a C program to generate a three address code (intermediate code) for a given expression.

ALGORITHM:

STEP 1: Start the program

STEP 2: The expression is read from the file using a file pointer

STEP 3: Each string is read and the total no. of strings in the file is calculated.

STEP 4: Each string is compared with an operator; if any operator seen the previous string as well as next string are concatenated and stored in a first temporary value and the three address code expression is printed

STEP 5: Suppose if another operand is seen then the first temporary value is concatenated to the next string using the operator and the expression is printed.

STEP 6: The final temporary value is replaced to the left operand value.

STEP 7: End the program.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
struct three
{
char data[10],temp[7];
}s[30];
void main()
{
char d1[7],d2[7]="t";
int i=0,j=1,len=0;
FILE *f1,*f2;
clrscr();
f1=fopen("sum.txt","r");
f2=fopen("out.txt","w");
while(fscanf(f1,"%s",s[len].data)!=EOF)
len++;
itoa(j,d1,7);
strcat(d2,d1);
strcpy(s[j].temp,d2);
strcpy(d1,"");
strcpy(d2,"t");
if(!strcmp(s[3].data,"+"))
{
fprintf(f2,"%s=%s+%s",s[j].temp,s[i+2].data,s[i+4].data);
```

```
j++;
}
else if(!strcmp(s[3].data,"-"))
{
fprintf(f2,"%s=%s-%s",s[j].temp,s[i+2].data,s[i+4].data);
j++;
}
for(i=4;i<len-2;i+=2)
{
itoa(j,d1,7);
strcat(d2,d1);
strcpy(s[j].temp,d2);
if(!strcmp(s[i+1].data,"+"))
fprintf(f2,"\n%s=%s+%s",s[j].temp,s[j-1].temp,s[i+2].data);
else if(!strcmp(s[i+1].data,"-"))
fprintf(f2,"\n%s=%s-%s",s[j].temp,s[j-1].temp,s[i+2].data);
strcpy(d1,"");
strcpy(d2,"t");
j++;
}
fprintf(f2,"\n%s=%s",s[0].data,s[j-1].temp);
fclose(f1);
fclose(f2);
getch();
}
```

SAMPLE INPUT:

```
sum.txt
out = in1 + in2 + in3 - in4
```

SAMPLE OUTPUT:

```
out.txt
t1=in1+in2
t2=t1+in3
t3=t2-in4
out=t3
```

RESULT : Thus the program has been executed successfully and verified.

14. Implementation of Code Generation

AIM: To implement the code generation using C.

ALGORITHM:

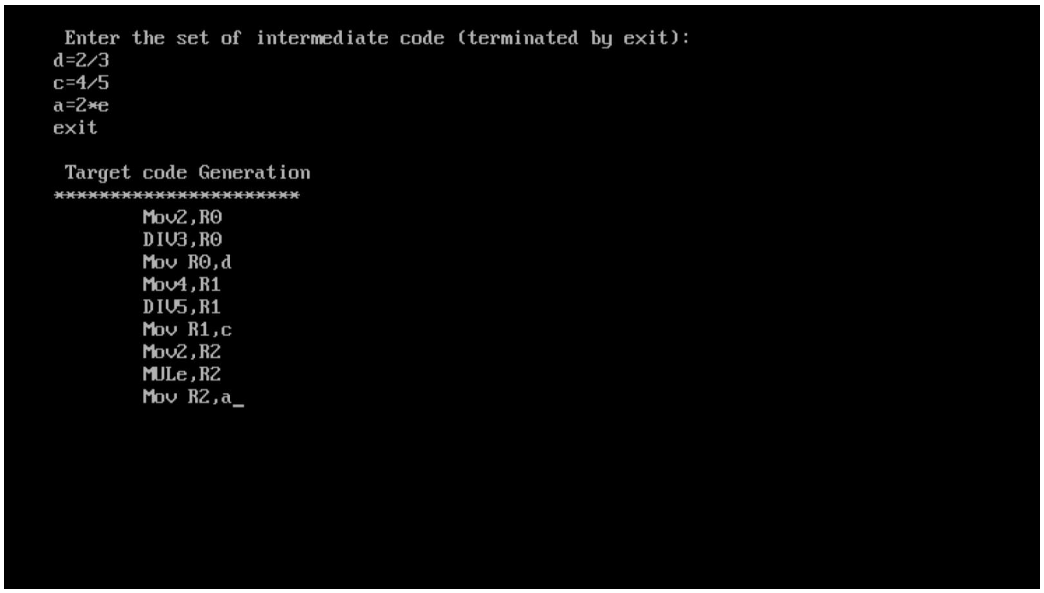
STEP 1: Start the program
STEP 2: Open the source file and store the contents as quadruples
STEP 3: Check for operators, in quadruples, if it is an arithmetic operator generator it or if assignment operator generates it, else perform unary minus on register c.
STEP 4: Write the generated code into output definition of the file in outp.c
STEP 5: Print the output
STEP 6: Stop the program

PROGRAM: (Back end of the compiler)

```
#include <stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char icode[10][30],str[20],opr[10];
int i=0;
clrscr();
printf("\n Enter the set of intermediate code (terminated by exit):\n");
do
{
scanf("%s",icode[i]);
}
while(strcmp(icode[i++],"exit")!=0);
printf("\n Target code Generation");
printf(\n**********");
i=0;
do
{
strcpy(str,icode[i]);
switch(str[3])
{
case'+':
strcpy(opr,"ADD");
break;
case'-':
strcpy(opr,"SUB");
break;
case'*':
strcpy(opr,"MUL");
break;
case'/':
strcpy(opr,"DIV");
break;
```

```
}
printf("\n\tMov%c,R%d",str[2],i);
printf("\n\t%s%c,R%d,opr,str[4],i);
printf("\n\tMov R%d,%c",i,str[0]);
}
while(strcmp(icode[++i],"exit")!=0);
//getch();
}
```

SAMPLE OUTPUT



RESULT : Thus the program has been executed successfully and verified.

15. Implementation of Code Optimization Techniques

AIM: To write a c program for Code Optimization.

ALGORITHM:

STEP 1: Start the program
STEP 2: Declare the variables and functions.
STEP 3: Enter the expression and state it in the variable a, b, c.
STEP 4: Calculate the variables b & c with 'temp' and store it in f1 and f2.
STEP 5: if(f1=null && f2=null)then expression could not be optimized.
STEP 6: Print the results.
STEP 7: Stop the program.

PROGRAM:

Before:

Using for loop:

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int n, i;
   int fact = 1;
   printf("Enter an integer: ");
   scanf("%d", &n);
   for (i = 1; i <= n; ++i)
      {
         fact *= i;
      }
   printf("Factorial of %d = %d", n, fact);
   clrscr();
   return 0;
}
```

SAMPLE OUTPUT:

```
Enter the number: 5
The factorial value is: 120
```

After:

Using do-while:

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int n, f;
   f = 1;
   printf("Enter an integer: ");
   scanf("%d", &n);
   do
      {
         f = f*n;
         n--;
      }
   while(n>0);
   printf("Factorial of %d = %d", n, fact);
   getch();
   clrscr();
}
```

SAMPLE OUTPUT:

```
Enter the number: 5
The factorial value is: 120
```

RESULT: Thus the program for implementation of Code Optimization technique is executed and verified.