S NO	LIST OF EXPERIMENT
1	1. Write a C program to identify different types of Tokens in a given Program.
2	2. Write a Lex Program to implement a Lexical Analyzer using Lex tool.
3	Write a C program to Simulate Lexical Analyzer to validating a given input String.
4	Write a C program to implement the Brute force technique of Top down Parsing.
5	Write a C program to implement a Recursive Descent Parser.
6	Write C program to compute the First and Follow Sets for the given Grammar.
7	Write a C program for eliminating the left recursion and left factoring of a given grammar
8	Write a C program to check the validity of input string using Predictive Parser.
9	Write a C program for implementation of LR parsing algorithm to accept a given input string.
10	Write a C program for implementation of a Shift Reduce Parser using Stack Data Structure to accept a given input string of a given grammar.
11	Simulate the calculator using LEX and YACC tool.
12	Generate YACC specification for a few syntactic categories.
13	Write a C program for generating the three address code of a given expression/statement.
14	Write a C program for implementation of a Code Generation Algorithm of a given expression/statement.

PROGRAM-1:

AIM: Write a C program to identify different types of tokens in a given program.

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

// Returns 'true' if the character is a DELIMITER.
bool isDelimiter(char ch)
{
   if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
      ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
      ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
      ch == '[' || ch == ']' || ch == '{' || ch == '}')
      return (true);
   return (false);
}
 // Returns 'true' if the character is an OPERATOR.
bool isOperator(char ch)
{
   if (ch == '+' || ch == '-' || ch == '*' ||
      ch == '/' || ch == '>' || ch == '<' ||
      ch == '=')
      return (true);
   return (false);
}

// Returns 'true' if the string is a VALID IDENTIFIER.
bool validIdentifier(char* str)
{
   if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
      str[0] == '3' || str[0] == '4' || str[0] == '5' ||
      str[0] == '6' || str[0] == '7' || str[0] == '8' ||
      str[0] == '9' || isDelimiter(str[0]) == true)
      return (false);
   return (true);
}
 // Returns 'true' if the string is a KEYWORD.
bool isKeyword(char* str)
{
   if (!strcmp(str, "if") || !strcmp(str, "else") ||
      !strcmp(str, "while") || !strcmp(str, "do") ||
      !strcmp(str, "break") ||
       !strcmp(str, "continue") || !strcmp(str, "int")
      || !strcmp(str, "double") || !strcmp(str, "float")
      || !strcmp(str, "return") || !strcmp(str, "char")
      || !strcmp(str, "case") || !strcmp(str, "char")
      || !strcmp(str, "sizeof") || !strcmp(str, "long")
      || !strcmp(str, "short") || !strcmp(str, "typedef")
      || !strcmp(str, "switch") || !strcmp(str, "unsigned")
```

```
       || !strcmp(str, "void") || !strcmp(str, "static")
        || !strcmp(str, "struct") || !strcmp(str, "goto"))
        return (true);
    return (false);
}
 // Returns 'true' if the string is an INTEGER.
bool isInteger(char* str)
{
    int i, len = strlen(str);

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' || (str[i] == '-' && i > 0))
            return (false);
    }
    return (true);
}

// Returns 'true' if the string is a REAL NUMBER.
bool isRealNumber(char* str)
{
    int i, len = strlen(str);
    bool hasDecimal = false;

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' && str[i] != '.' ||
            (str[i] == '-' && i > 0))
            return (false);
        if (str[i] == '.')
            hasDecimal = true;
    }
    return (hasDecimal);
}
 // Extracts the SUBSTRING.
char* subString(char* str, int left, int right)
{
    int i;
    char* subStr = (char*)malloc(

  sizeof(char) * (right - left + 2));
    for (i = left; i <= right; i++)
```

```
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
    return (subStr);
}
 // Parsing the input STRING.
void parse(char* str)
{
    int left = 0, right = 0;
    int len = strlen(str);
    while (right <= len && left <= right) {
        if (isDelimiter(str[right]) == false)
            right++;

        if (isDelimiter(str[right]) == true && left == right) {
            if (isOperator(str[right]) == true)
                printf("'%c' IS AN OPERATOR\n", str[right]);

            right++;
            left = right;
        } else if (isDelimiter(str[right]) == true && left != right
                || (right == len && left != right)) {
            char* subStr = subString(str, left, right - 1);

            if (isKeyword(subStr) == true)
                printf("'%s' IS A KEYWORD\n", subStr);

            else if (isInteger(subStr) == true)
                printf("'%s' IS AN INTEGER\n", subStr);

            else if (isRealNumber(subStr) == true)
                printf("'%s' IS A REAL NUMBER\n", subStr);

            else if (validIdentifier(subStr) == true
                    && isDelimiter(str[right - 1]) == false)
                printf("'%s' IS A VALID IDENTIFIER\n", subStr);

            else if (validIdentifier(subStr) == false
                    && isDelimiter(str[right - 1]) == false)
                printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
            left = right;
        }
    }
    return;
}
 // DRIVER FUNCTION
int main()
{
     // maximum length of string is 100 here
    char str[100] = "int a = b + 1c; ";
    parse(str); // calling the parse function
```

```
   return (0);
}
```

OUTPUT:

```
'int' IS A KEYWORD
'a' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'b' IS A VALID IDENTIFIER
'+' IS AN OPERATOR
'1c' IS NOT A VALID IDENTIFIER.
```

PROGRAM-2

AIM:Write a Lex Program to implement a Lexical Analyzer using Lex Tool.

```
/* program name is lexp.l */
%{
/* program to recognize a c program */
int COMMENT=0;
int cnt=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* { printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);}
int |
float |
char |
double |
while |
for |
do |
if |
break |
continue |
void |
switch |
case |
long |
struct |
const |
typedef |
return |
else |
goto {printf("\n\t%s is a KEYWORD",yytext);}
"/*" {COMMENT = 1;}
"*/" {COMMENT = 0; cnt++;}
{identifier}\( {if(!COMMENT)printf("\n\nFUNCTION\n\t%s",yytext);}
```

```
\{ {if(!COMMENT) printf("\n BLOCK BEGINS");}
\} {if(!COMMENT) printf("\n BLOCK ENDS");}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
\".*\" {if(!COMMENT) printf("\n\t%s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n\t%s is a NUMBER",yytext);}
\)(\;)? {if(!COMMENT) printf("\n\t");ECHO;printf("\n");}
\( ECHO;
= {if(!COMMENT)printf("\n\t%s is an ASSIGNMENT OPERATOR",yytext);}
\<= |
\>= |
\< |
== |
\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}
%%
int main(int argc,char **argv)
{
if (argc > 1)
{
FILE *file;
file = fopen(argv[1],"r");
if(!file)
{
printf("could not open %s \n",argv[1]);
exit(0);
}
yyin = file;
}
yylex();
printf("\n\n Total No.Of comments are %d",cnt);
return 0;
}
int yywrap()
{
return 1;

}
/* program name is lexp.l */
%{
/* program to recognize a c program */
int COMMENT=0;
int cnt=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
```

```
#.* { printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);}
int |
float |
char |
double |
while |
for |
do |
if |
break |
continue |
void |
switch |
case |
long |
struct |
const |
typedef |
return |
else |
goto {printf("\n\t%s is a KEYWORD",yytext);}
"/*" {COMMENT = 1;}
"*/" {COMMENT = 0; cnt++;}
{identifier}\( {if(!COMMENT)printf("\n\nFUNCTION\n\t%s",yytext);}
\{ {if(!COMMENT) printf("\n BLOCK BEGINS");}
\} {if(!COMMENT) printf("\n BLOCK ENDS");}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
\".*\" {if(!COMMENT) printf("\n\t%s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n\t%s is a NUMBER",yytext);}
\)(\;)? {if(!COMMENT) printf("\n\t");ECHO;printf("\n");}
\( ECHO;
= {if(!COMMENT)printf("\n\t%s is an ASSIGNMENT OPERATOR",yytext);}
\<= |
\>= |
\< |
== |
\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}
%%
int main(int argc,char **argv)
{
if (argc > 1)
{
FILE *file;
file = fopen(argv[1],"r");
```

```
if(!file)
{
printf("could not open %s \n",argv[1]);
exit(0);
}
yyin = file;
}
yylex();
printf("\n\n Total No.Of comments are %d",cnt);
return 0;
}
int yywrap()
{
return 1;

}
```

Input:

```
 #include<stdio.h>
main()
{
int a,b;
}
```

Output:

```
#include<stdio.h> is a PREPROCESSOR DIRECTIVE
FUNCTION
main (
)
BLOCK BEGINS
int is a KEYWORD
a IDENTIFIER
b IDENTIFIER
BLOCK ENDS
```

PROGRAM-3:

AIM:Write a C program to simulate lexical analyzer to validating a given input string.

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char arithmetic[5]={'+','-','*','/','%'};
    char relational[4]={'<','>','!','='};
    char bitwise[5]={'&','^','~','|'};
    char str[2]={' ',' '};
    printf ("Enter value to be identified: ");
    scanf ("%s",&str);
    int i;
    if(((str[0]=='&' || str[0]=='|') && str[0]==str[1]) || (str[0]=='!' && str[1]=='\0'))
    {
            printf("\nIt is Logical operator");
    }
    for(i=0;i<4;i++)
    {
            if(str[0]==relational[i]&&(str[1]=='='||str[1]=='\0'))
            {
                    printf("\n It is releational Operator"); break;
            }
    }

for(i=0;i<4;i++)
    {
            if((str[0]==bitwise[i] && str[1]=='\0') || ((str[0]=='<' || str[0]=='>') &&
str[1]==str[0]))
            {
                    printf("\n It is Bitwise Operator"); break;
            }
    }
    if(str[0]=='?'    &&  str[1]==':')
    printf("\nIt  is  ternary operator");
    for(i=0;i<5;i++)
    {
            if((str[0]=='+' || str[0]=='-') && str[0]==str[1])
            {
                    printf("\nIt is unary operator"); break;
            }
            else if((str[0]==arithmetic[i] && str[1]=='=') || (str[0]=='=' && str[1]==' '))
            {
                    printf("\nIt is Assignment operator"); break;
```

```
            }
            else if(str[0]==arithmetic[i] && str[1]=='\0')
            {
                    printf("\nIt is arithmetic operator"); break
              }
    }


 return 0;
}
```

Output:

Enter value to be identified: =
It is relational Operator

PROGRAM-4:

Aim:Write a C program to implement the Brute Force Technique of Top down parsing.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class parse
{
int nt,t,m[20][20],i,s,n,p1,q,k,j;
char p[30][30],n1[20],t1[20],ch,b,c,f[30][30],fl[30][30];
public:
int scant(char);
int scannt(char);
void process();
void input();
};

int parse::scannt(char a)
{
int c=-1,i;
for(i=0;i<nt;i++)
{
if(n1[i]==a)
{
return i;
}
}
return c;
}

int parse::scant(char b)
{
int c1=-1,j;
for(j=0;j<t;j++)
```

```
{
if(t1[j]==b)
{
return j;
}
}
return c1;
}

void parse::input()
{
cout<<"Enter the number of productions:";
cin>>n;
cout<<"Enter the productions one by one"<<endl;
for(i=0;i<n;i++)
cin>>p[i];
nt=0;
t=0;
}

void parse::process()
{
for(i=0;i<n;i++)
{
if(scannt(p[i][0])==-1)
n1[nt++]=p[i][0];
}
for(i=0;i<n;i++)
{
for(j=3;j<strlen(p[i]);j++)
{
if(p[i][j]!='e')
{
if(scannt(p[i][j])==-1)
{
if((scant(p[i][j]))==-1)
t1[t++]=p[i][j];
}
}
}
}
t1[t++]='$';
for(i=0;i<nt;i++)
{
for(j=0;j<t;j++)
m[i][j]=-1;
}
for(i=0;i<nt;i++)
{
cout<<"Enter first["<<n1[i]<<"]:";
```

```
cin>>f[i];
}

for(i=0;i<nt;i++)
{
cout<<"Enter follow["<<n1[i]<<"]:";
cin>>fl[i];
}
for(i=0;i<n;i++)
{
p1=scannt(p[i][0]);
if((q=scant(p[i][3]))!=-1)
m[p1][q]=i;
if((q=scannt(p[i][3]))!=-1)
{
for(j=0;j<strlen(f[q]);j++)
m[p1][scant(f[q][j])]=i;
}
if(p[i][3]=='e')
{
for(j=0;j<strlen(fl[p1]);j++)
m[p1][scant(fl[p1][j])]=i;
}
}
for(i=0;i<t;i++)
cout<<"\t"<<t1[i];
cout<<endl;
for(j=0;j<nt;j++)
{
cout<<n1[j];
for(i=0;i<t;i++)
{
cout<<"\t"<<" ";
if(m[j][i]!=-1)
cout<<p[m[j][i]];
}
cout<<endl;
}
}

void main()
{
clrscr();
parse p;
p.input();
p.process();
getch();
}
```

Output:

```
Enter the number of productions:8
Enter the productions one by one
E->TA
A->+TA
A->e
T->FB
B->e
B->*FB
F->(E)
F->i

Enter first[E]: (i
Enter first[A]: +e
Enter first[T]: (i
Enter first[B]: *e
Enter first[F]: (i
Enter follow[E]: $)
Enter follow[A]: $)
Enter follow[T]: +)$
Enter follow[B]: +)$
Enter follow[F]: +*)$
+ ( ) i * $
E E->TA E->TA
A A->+TA A->e A->e
T T->FB T->FB
B B->e B->e B->*FB B->e
F F->(E) F->i
```

PROGRAM-5:

Aim: write a C program to implement a Recursive Descent parser

```
#include<stdio.h>
#include<string.h>
int E(),Edash(),T(),Tdash(),F();
char *ip;
char string[50];
int main()
{
printf("Enter the string\n");
scanf("%s",string);
ip=string;
printf("\n\nInput\tAction\n--------------------------------\n");

if(E() && ip=="\0"){
printf("\n--------------------------------\n");
```

```
printf("\n String is successfully parsed\n");
}
else{
printf("\n--------------------------------\n");
printf("Error in parsing String\n");
}
}
int E()
{
printf("%s\tE->TE' \n",ip);
if(T())
{
if(Edash())
{
return 1;
}
else
return 0;
}
else
return 0;
}
int Edash()
{
if(*ip=='+')
{
printf("%s\tE'->+TE' \n",ip);
ip++;
if(T())
{
if(Edash())
{
return 1;
}
else
return 0;
}
else
return 0;
}
else
{
printf("%s\tE'->^ \n",ip);
return 1;
```

```
}
}
int T()
{
printf("%s\tT->FT' \n",ip);
if(F())
{

if(Tdash())
{
return 1;
}
else
return 0;
}
else
return 0;
}
int Tdash()
{
if(*ip=='*')
{
printf("%s\tT'->*FT' \n",ip);
ip++;
if(F())
{
if(Tdash())
{
return 1;
}
else
return 0;
}
else
return 0;
}
else
{
printf("%s\tT'->^ \n",ip);
return 1;
}
}
int F()
{
if(*ip=='(')
```

```
{
printf("%s\tF->(E) \n",ip);
ip++;
if(E())
{
if(*ip==')')
{
ip++;
return 0;
}
else
return 0;
}
else
return 0;
}

else if(*ip=='i')
{
ip++;
printf("%s\tF->id \n",ip);
return 1;
}
else
return 0;
}
```

Output:

```
Enter the string
Input    Action

   E->TE'
   T->FT'


Error in parsing String
```

PROGRAM-6:

AIM: Write a C program to follow First and Follow sets for given Grammar.

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
 // Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);
 // Function to calculate First
void findfirst(char, int, int);
int count, n = 0;

// Stores the final result
// of the First Sets
char calc_first[10][100];
 // Stores the final result
// of the Follow Sets
char calc_follow[10][100];
int m = 0;

// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;
int main(int argc, char **argv)
{
   int jm = 0;
   int km = 0;
   int i, choice;
   char c, ch;
   count = 8;

   // The Input grammar
   strcpy(production[0], "E=TR");
   strcpy(production[1], "R=+TR");
   strcpy(production[2], "R=#");
   strcpy(production[3], "T=FY");
   strcpy(production[4], "Y=*FY");
   strcpy(production[5], "Y=#");
   strcpy(production[6], "F=(E)");
   strcpy(production[7], "F=i");
     int kay;
   char done[count];
   int ptr = -1;

   // Initializing the calc_first array
   for(k = 0; k < count; k++) {
      for(kay = 0; kay < 100; kay++) {
```

```
            calc_first[k][kay] = '!';
        }
    }
    int point1 = 0, point2, xxx;
for(k = 0; k < count; k++)
    {
        c = production[k][0];
        point2 = 0;
        xxx = 0;

        // Checking if First of c has
        // already been calculated
        for(kay = 0; kay <= ptr; kay++)
            if(c == done[kay])
                xxx = 1;

        if (xxx == 1)
            continue;

        // Function call
        findfirst(c, 0, 0);
        ptr += 1;

        // Adding c to the calculated list
        done[ptr] = c;
        printf("\n First(%c) = { ", c);
        calc_first[point1][point2++] = c;

        // Printing the First Sets of the grammar
        for(i = 0 + jm; i < n; i++) {
            int lark = 0, chk = 0;

            for(lark = 0; lark < point2; lark++) {

                if (first[i] == calc_first[point1][lark])
                {
                    chk = 1;
                    break;
                }
            }
            if(chk == 0)
            {
                printf("%c, ", first[i]);
                calc_first[point1][point2++] = first[i];
            }
        }
        printf("}\n");
        jm = n;
        point1++;
    }
```

```
    printf("\n");
    printf("-----------------------------------------------\n\n");
char donee[count];
    ptr = -1;

    // Initializing the calc_follow array
    for(k = 0; k < count; k++) {
        for(kay = 0; kay < 100; kay++) {
            calc_follow[k][kay] = '!';
        }
    }
    point1 = 0;
    int land = 0;
    for(e = 0; e < count; e++)
    {
        ck = production[e][0];
        point2 = 0;
        xxx = 0;

        // Checking if Follow of ck
        // has already been calculated
        for(kay = 0; kay <= ptr; kay++)
            if(ck == donee[kay])
                xxx = 1;

        if (xxx == 1)
            continue;
        land += 1;
        // Function call
        follow(ck);
        ptr += 1;

        // Adding ck to the calculated list
        donee[ptr] = ck;
        printf(" Follow(%c) = { ", ck);
        calc_follow[point1][point2++] = ck;

        // Printing the Follow Sets of the grammar
        for(i = 0 + km; i < m; i++) {
            int lark = 0, chk = 0;
            for(lark = 0; lark < point2; lark++)
            {
                if (f[i] == calc_follow[point1][lark])
                {
                    chk = 1;
                    break;
                }
            }
            if(chk == 0)
{
```

```
                printf("%c, ", f[i]);
                calc_follow[point1][point2++] = f[i];
            }
        }
        printf(" }\n\n");
        km = m;
        point1++;
    }
}
void follow(char c)
{
    int i, j;

    // Adding "$" to the follow
    // set of the start symbol
    if(production[0][0] == c) {
        f[m++] = '$';
    }
    for(i = 0; i < 10; i++)
    {
        for(j = 2;j < 10; j++)
        {
            if(production[i][j] == c)
            {
                if(production[i][j+1] != '\0')
                {
                    // Calculate the first of the next
                    // Non-Terminal in the production
                    followfirst(production[i][j+1], i, (j+2));
                }

                if(production[i][j+1]=='\0' && c!=production[i][0])
                {
                    // Calculate the follow of the Non-Terminal
                    // in the L.H.S. of the production
                    follow(production[i][0]);
                }
            }
        }
    }
}
void findfirst(char c, int q1, int q2)
{
    int j;

    // The case where we
    // encounter a Terminal
    if(!(isupper(c))) {
        first[n++] = c;
    }
```

```
    for(j = 0; j < count; j++)
    {
        if(production[j][0] == c)
        {
            if(production[j][2] == '#')
            {
if(production[q1][q2] == '\0')
                    first[n++] = '#';
                else if(production[q1][q2] != '\0'
                        && (q1 != 0 || q2 != 0))
                {
                    // Recursion to calculate First of New
                    // Non-Terminal we encounter after epsilon
                    findfirst(production[q1][q2], q1, (q2+1));
                }
                else
                    first[n++] = '#';
            }
            else if(!isupper(production[j][2]))
            {
                first[n++] = production[j][2];
            }
            else
            {
                // Recursion to calculate First of
                // New Non-Terminal we encounter
                // at the beginning
                findfirst(production[j][2], j, 3);
            }
        }
}
}
void followfirst(char c, int c1, int c2)
{
    int k;

    // The case where we encounter
    // a Terminal
    if(!(isupper(c)))
        f[m++] = c;
    else
    {
        int i = 0, j = 1;
        for(i = 0; i < count; i++)
        {
            if(calc_first[i][0] == c)
                break;
        }
        //Including the First set of the
        // Non-Terminal in the Follow of
```

```
        // the original query
    while(calc_first[i][j] != '!')
        {
            if(calc_first[i][j] != '#')
            {
                f[m++] = calc_first[i][j];
            }
            else
            {
                if(production[c1][c2] == '\0')
                {
                    // Case where we reach the
                    // end of a production
                    follow(production[c1][0]);
                }
                else
                {
                    // Recursion to the next symbol
                    // in case we encounter a "#"
                    followfirst(production[c1][c2], c1, c2+1);
                }
            }
            j++;
        }
    }
}
```

Output :

```
First(E)= { (, i, }
First(R)= { +, #, }
First(T)= { (, i, }
First(Y)= { *, #, }
First(F)= { (, i, }

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

Follow(E) = { $, ),  }
Follow(R) = { $, ),  }
Follow(T) = { +, $, ),  }
Follow(Y) = { +, $, ),  }
Follow(F) = { *, +, $, ),  }
```

PROGRAM:7

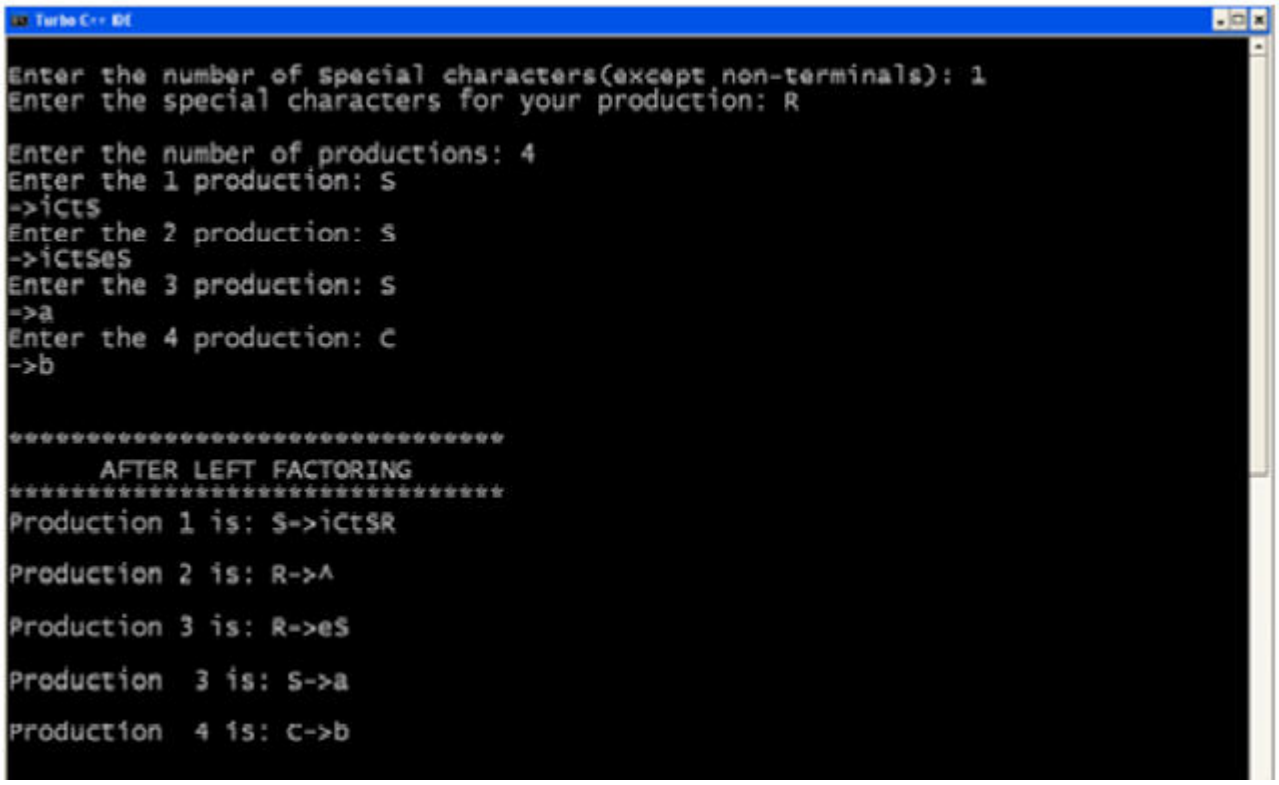
AIM:Write a C program for eliminating the left recursion and left factoring of a given grammar.

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<string.h>

//Structure Declaration

struct production
{
        char lf;
        char rt[10];
        int prod_rear;
        int fl;
};
struct production prodn[20],prodn_new[20];     //Creation of object

//Variables Declaration


int b=-1,d,f,q,n,m=0,c=0;
char terminal[20],nonterm[20],alpha[10],extra[10];
char epsilon='^';

//Beginning of Main Program


void main()
{
  clrscr();


  //Input of Special characters
  cout<<"\nEnter the number of Special characters(except non-terminals): ";
  cin>>q;
  cout<<"Enter the special characters for your production: ";
  for(int cnt=0;cnt<q;cnt++)
  {
    cin>>alpha[cnt];
  }

  //Input of Productions


  cout<<"\nEnter the number of productions: ";
  cin>>n;
```

```
for(cnt=0;cnt<=n-1;cnt++)
{
 cout<<"Enter the "<< cnt+1<<" production: ";
 cin>>prodn[cnt].lf;
 cout<<"->";
 cin>>prodn[cnt].rt;
 prodn[cnt].prod_rear=strlen(prodn[cnt].rt);
 prodn[cnt].fl=0;
}

//Condition for left factoring


for(int cnt1=0;cnt1<n;cnt1++)
{
 for(int cnt2=cnt1+1;cnt2<n;cnt2++)
 {
       if(prodn[cnt1].lf==prodn[cnt2].lf)
       {
         cnt=0;
         int p=-1;
         while((prodn[cnt1].rt[cnt]!='\0')&&(prodn[cnt2].rt[cnt]!='\0'))
         {
           if(prodn[cnt1].rt[cnt]==prodn[cnt2].rt[cnt])
           {
             extra[++p]=prodn[cnt1].rt[cnt];
             prodn[cnt1].fl=1;
             prodn[cnt2].fl=1;
           }
           else
           {
             if(p==-1)
                   break;
             else
             {
                   int h=0,u=0;
                   prodn_new[++b].lf=prodn[cnt1].lf;
                   strcpy(prodn_new[b].rt,extra);
                   prodn_new[b].rt[p+1]=alpha[c];
                   prodn_new[++b].lf=alpha[c];
                   for(int g=cnt;g<prodn[cnt2].prod_rear;g++)
                    prodn_new[b].rt[h++]=prodn[cnt2].rt[g];
                    prodn_new[++b].lf=alpha[c];
                   for(g=cnt;g<=prodn[cnt1].prod_rear;g++)
                    prodn_new[b].rt[u++]=prodn[cnt1].rt[g];
                    m=1;
                    break;
             }
           }
```

```
            cnt++;
           }
           if((prodn[cnt1].rt[cnt]==0)&&(m==0))
           {
                  int h=0;
                  prodn_new[++b].lf=prodn[cnt1].lf;
                  strcpy(prodn_new[b].rt,extra);
                  prodn_new[b].rt[p+1]=alpha[c];
                  prodn_new[++b].lf=alpha[c];
                  prodn_new[b].rt[0]=epsilon;
                  prodn_new[++b].lf=alpha[c];
                  for(int g=cnt;g<prodn[cnt2].prod_rear;g++)
                  prodn_new[b].rt[h++]=prodn[cnt2].rt[g];
           }
           if((prodn[cnt2].rt[cnt]==0)&&(m==0))
           {
            int h=0;
            prodn_new[++b].lf=prodn[cnt1].lf;
            strcpy(prodn_new[b].rt,extra);
            prodn_new[b].rt[p+1]=alpha[c];
            prodn_new[++b].lf=alpha[c];
            prodn_new[b].rt[0]=epsilon;
            prodn_new[++b].lf=alpha[c];
            for(int g=cnt;g<prodn[cnt1].prod_rear;g++)
             prodn_new[b].rt[h++]=prodn[cnt1].rt[g];
           }
           c++;
           m=0;
          }
     }
  }

 //Display of Output


  cout<<"\n\n*********************************";
  cout<<"\n      AFTER LEFT FACTORING      ";
  cout<<"\n*********************************";
  cout<<endl;
  for(int cnt3=0;cnt3<=b;cnt3++)
        {
                cout<<"Production "<<cnt3+1<<" is: ";
                cout<<prodn_new[cnt3].lf;
                cout<<"->";
                cout<<prodn_new[cnt3].rt;
                cout<<endl<<endl;
        }


  for(int cnt4=0;cnt4<n;cnt4++)
```

```
    {
    if(prodn[cnt4].fl==0)
    {
    cout<<"Production  "<<cnt3++<<" is: ";
    cout<<prodn[cnt4].lf;
    cout<<"->";
    cout<<prodn[cnt4].rt;
    cout<<endl<<endl;
    }
  }
 getche();
}    //end of main program
```



PROGRAM-8
AIM:write a C program to check the validity of input string using Predictive Parser.

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

char prol[7][10] = { "S", "A", "A", "B", "B", "C", "C" };
char pror[7][10] = { "A", "Bb", "Cd", "aB", "@", "Cc", "@" };

char prod[7][10] = { "S->A", "A->Bb", "A->Cd", "B->aB", "B->@", "C->Cc", "C->@" };
char first[7][10] = { "abcd", "ab", "cd", "a@", "@", "c@", "@" };
char follow[7][10] = { "$", "$", "$", "a$", "b$", "c$", "d$" };
char table[5][6][10];

int numr(char c)
{
  switch (c)
  {
    case 'S':
      return 0;

    case 'A':
      return 1;

    case 'B':
      return 2;

    case 'C':
      return 3;

    case 'a':
      return 0;

    case 'b':
      return 1;

    case 'c':
      return 2;

    case 'd':
      return 3;

    case '$':
      return 4;
  }

  return (2);
}

int main()
```

```
{
  int i, j, k;

  for (i = 0; i < 5; i++)
    for (j = 0; j < 6; j++)
      strcpy(table[i][j], " ");

  printf("The following grammar is used for Parsing Table:\n");

  for (i = 0; i < 7; i++)
    printf("%s\n", prod[i]);

  printf("\nPredictive parsing table:\n");

  fflush(stdin);

  for (i = 0; i < 7; i++)
  {
    k = strlen(first[i]);
    for (j = 0; j < 10; j++)
      if (first[i][j] != '@')
        strcpy(table[numr(prol[i][0]) + 1][numr(first[i][j]) + 1], prod[i]);
  }

  for (i = 0; i < 7; i++)
  {
    if (strlen(pror[i]) == 1)
    {
      if (pror[i][0] == '@')
      {
        k = strlen(follow[i]);
        for (j = 0; j < k; j++)
          strcpy(table[numr(prol[i][0]) + 1][numr(follow[i][j]) + 1], prod[i]);
      }
    }
  }

  strcpy(table[0][0], " ");

  strcpy(table[0][1], "a");

  strcpy(table[0][2], "b");

  strcpy(table[0][3], "c");

  strcpy(table[0][4], "d");

  strcpy(table[0][5], "$");

  strcpy(table[1][0], "S");
```

```
  strcpy(table[2][0], "A");

  strcpy(table[3][0], "B");

  strcpy(table[4][0], "C");

  printf("\n----------------------------------------------------------\n");

  for (i = 0; i < 5; i++)
    for (j = 0; j < 6; j++)
    {
      printf("%-10s", table[i][j]);
      if (j == 5)
        printf("\n----------------------------------------------------------\n");
    }
}
```

OUTPUT:

The following grammar is used for Parsing Table:

S->A
A->Bb
A->Cd
B->aB
B->@
C->Cc
C->@

Predictive parsing table:

	a	b	c	d	$
S	S->A	S->A	S->A	S->A	
A	A->Bb	A->Bb	A->Cd	A->Cd	
B	B->aB	B->@	B->@		B->@
C			C->@	C->@	C->@

PROGRAM-9

AIM:Write a C Program for implementation of LR Parsing algorithmto accept a given input string.

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

//Global Variables
int z = 0, i = 0, j = 0, c = 0;

// Modify array size to increase
// length of string to be parsed
char a[16], ac[20], stk[15], act[10];

// This Function will check whether
// the stack contain a production rule
// which is to be Reduce.
// Rules can be E->2E2 , E->3E3 , E->4
void check()
{
   // Copying string to be printed as action
   strcpy(ac,"REDUCE TO E -> ");

   // c=length of input string
   for(z = 0; z < c; z++)
   {
      //checking for producing rule E->4
      if(stk[z] == '4')
      {
         printf("%s4", ac);
         stk[z] = 'E';
         stk[z + 1] = '\0';

         //printing action
         printf("\n$%s\t%s$\t", stk, a);
      }
   }

   for(z = 0; z < c - 2; z++)
   {
      //checking for another production
      if(stk[z] == '2' && stk[z + 1] == 'E' &&
                        stk[z + 2] == '2')
      {
         printf("%s2E2", ac);
         stk[z] = 'E';
         stk[z + 1] = '\0';
         stk[z + 2] = '\0';
         printf("\n$%s\t%s$\t", stk, a);
         i = i - 2;
```

```
        }

    }

    for(z=0; z<c-2; z++)
    {
        //checking for E->3E3
        if(stk[z] == '3' && stk[z + 1] == 'E' &&
                            stk[z + 2] == '3')
        {
            printf("%s3E3", ac);
            stk[z]='E';
            stk[z + 1]='\0';
            stk[z + 1]='\0';
            printf("\n$%s\t%s$\t", stk, a);
            i = i - 2;
        }
    }
    return ; //return to main
}

//Driver Function
int main()
{
    printf("GRAMMAR is -\nE->2E2 \nE->3E3 \nE->4\n");

    // a is input string
    strcpy(a,"32423");

    // strlen(a) will return the length of a to c
    c=strlen(a);

    // "SHIFT" is copied to act to be printed
    strcpy(act,"SHIFT");

    // This will print Labels (column name)
    printf("\nstack \t input \t action");

    // This will print the initial
    // values of stack and input
    printf("\n$\t%s$\t", a);

    // This will Run upto length of input string
    for(i = 0; j < c; i++, j++)
    {
        // Printing action
        printf("%s", act);

        // Pushing into stack
        stk[i] = a[j];
```

```
        stk[i + 1] = '\0';

        // Moving the pointer
        a[j]=' ';

        // Printing action
        printf("\n$%s\t%s$\t", stk, a);

        // Call check function ..which will
        // check the stack whether its contain
        // any production or not
        check();
    }

    // Rechecking last time if contain
    // any valid production then it will
    // replace otherwise invalid
    check();

    // if top of the stack is E(starting symbol)
    // then it will accept the input
    if(stk[0] == 'E' && stk[1] == '\0')
        printf("Accept\n");
    else //else reject
        printf("Reject\n");
}
```

Output

```
GRAMMAR is -
E->2E2
E->3E3
E->4

stack      input      action
$    32423$    SHIFT
$3     2423$    SHIFT
$32      423$    SHIFT
$324       23$    REDUCE TO E -> 4
$32E       23$    SHIFT
$32E2        3$    REDUCE TO E -> 2E2
$3E        3$    SHIFT
$3E3         $    REDUCE TO E -> 3E3
$E         $    Accept
```

PROGRAM-11
AIM:Simulate the calculator using LEX and YACC tool.

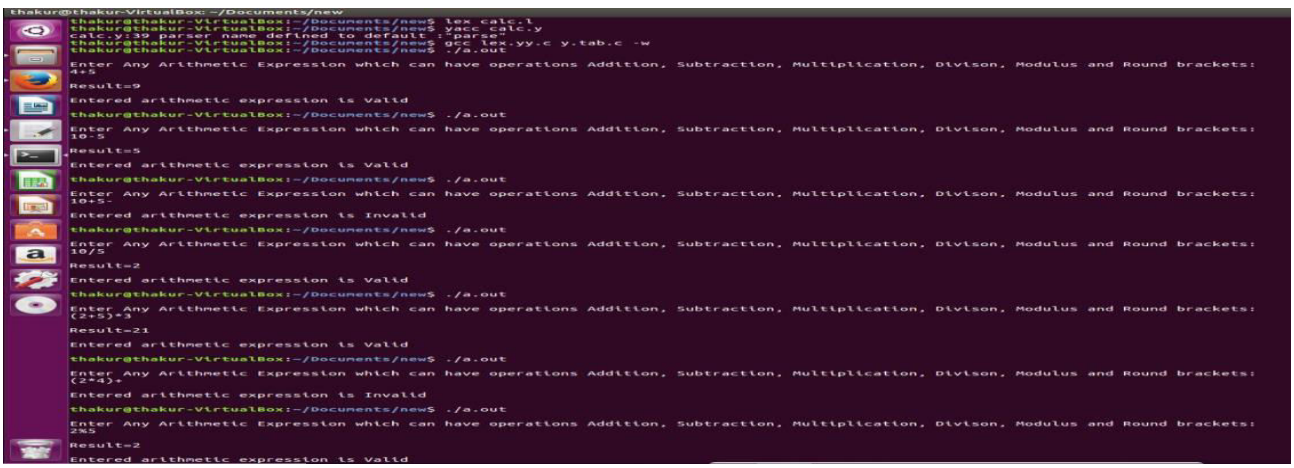
```
%{
   /* Definition section */
  #include<stdio.h>
  #include "y.tab.h"
  extern int yylval;
%}

/* Rule Section */
%%
[0-9]+ {
          yylval=atoi(yytext);
          return NUMBER;

        }
[\t] ;

[\n] return 0;

. return yytext[0];
  %%

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

PARSER SOURCE CODE

```
%{
   /* Definition section */
  #include<stdio.h>
  int flag=0;
%}

%token NUMBER

%left '+' '-'

%left '*' '/' '%'

%left '(' ')'
  /* Rule Section */
%%

ArithmeticExpression: E{

        printf("\nResult=%d\n", $$);

        return 0;
```

```
        };
 E:E'+'E {$$=$1+$3;}

 |E'-'E {$$=$1-$3;}

 |E'*'E {$$=$1*$3;}

 |E'/'E {$$=$1/$3;}

 |E'%'E {$$=$1%$3;}

 |'('E')' {$$=$2;}
 | NUMBER {$$=$1;}

 ;

%%
//driver code
void main()
{
  printf("\nEnter Any Arithmetic Expression which
              can have operations Addition,
              Subtraction, Multiplication, Division,
                    Modulus and Round brackets:\n");

  yyparse();
  if(flag==0)
  printf("\nEntered arithmetic expression is Valid\n\n");
}

void yyerror()
{
  printf("\nEntered arithmetic expression is Invalid\n\n");
   flag=1;
}
```

Output:



PROGRAM-12
AIM:Generate YACC specification for a syntactic categories.

```
%{
/* This LEX program returns the tokens for the expression */
#include "y.tab.h"
%}

%%
"=" {printf("\n Operator is EQUAL");}
"+" {printf("\n Operator is PLUS");}
"-" {printf("\n Operator is MINUS");}
"/" {printf("\n Operator is DIVISION");}
"*" {printf("\n Operator is MULTIPLICATION");}

[a-z A-Z]*[0-9]* {
printf("\n Identifier is %s",yytext);
return ID;
}
return yytext[0];
\n return 0;
%%

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

Program Name : arith_id.y

```
%{
#include
/* This YYAC program is for recognizing the Expression */
%}
%%
statement: A'='E
| E {
printf("\n Valid arithmetic expression");
$$ = $1;
};

E: E'+'ID
| E'-'ID
| E'*'ID
| E'/'ID
| ID
;
%%
extern FILE *yyin;
```

```
main()
{
do
{
yyparse();
}while(!feof(yyin));
}

yyerror(char*s)
{
}

/* This YYAC program is for recognizing the Expression */
%}
%%
statement: A'='E
| E {
printf("\n Valid arithmetic expression");
$$ = $1;
};

E: E'+'ID
| E'-'ID
| E'*'ID
| E'/'ID
| ID
;
%%
extern FILE *yyin;
main()
{
do
{
yyparse();
}while(!feof(yyin));
}

yyerror(char*s)
{
}


/* This YYAC program is for recognizing the Expression */
%}
%%
statement: A'='E
| E {
printf("\n Valid arithmetic expression");
$$ = $1;
};
```

```
E: E'+'ID
| E'-'ID
| E'*'ID
| E'/'ID
| ID
;
%%
extern FILE *yyin;
main()
{
do
{
yyparse();
}while(!feof(yyin));
}

yyerror(char*s)
{
}
```

Output:

```
[root@localhost]# lex arith_id.1
[root@localhost]# yacc –d arith_id.y
[root@localhost]# gcc lex.yy.c y.tab.c
[root@localhost]# ./a.out
x=a+b;

Identifier is x
Operator is EQUAL
Identifier is a
Operator is PLUS
Identifier is b
```

PROGRAM-13

AIM:Write a C program for generating the three address code of a given expression/statement.

```
#include<stdio.h>
#include<string.h>
void pm();
void plus();
void div();
int i,ch,j,l,addr=100;
char ex[10], exp[10] ,exp1[10],exp2[10],id1[5],op[5],id2[5];
void main()
{
clrscr();
while(1)
{
```

```
printf("\n1.assignment\n2.arithmetic\n3.relational\n4.Exit\nEnter the choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("\nEnter the expression with assignment operator:");
scanf("%s",exp);
l=strlen(exp);
exp2[0]='\0';
i=0;
while(exp[i]!='=')
{
i++;
}
strncat(exp2,exp,i);
strrev(exp);
exp1[0]='\0';
strncat(exp1,exp,l-(i+1));
strrev(exp1);
printf("Three address code:\ntemp=%s\n%s=temp\n",exp1,exp2);
break;

case 2:
printf("\nEnter the expression with arithmetic operator:");
scanf("%s",ex);
strcpy(exp,ex);
l=strlen(exp);
exp1[0]='\0';

for(i=0;i<l;i++)
{
if(exp[i]=='+'||exp[i]=='-')
{
if(exp[i+2]=='/'||exp[i+2]=='*')
{
pm();
break;
}
else
{
plus();
break;
}
}
else if(exp[i]=='/'||exp[i]=='*')
{
div();
break;
}
}
```

```
break;

case 3:
printf("Enter the expression with relational operator");
scanf("%s%s%s",&id1,&op,&id2);
if(((strcmp(op,"<")==0)||(strcmp(op,">")==0)||(strcmp(op,"<=")==0)||(strcmp(op,">=")==0)||(
strcmp(op,"==")==0)||(strcmp(op,"!=")==0))==0)
printf("Expression is error");
else
{
printf("\n%d\tif %s%s%s goto %d",addr,id1,op,id2,addr+3);
addr++;
printf("\n%d\t T:=0",addr);
addr++;
printf("\n%d\t goto %d",addr,addr+2);
addr++;
printf("\n%d\t T:=1",addr);
}
break;
case 4:
exit(0);
}
}
}
void pm()
{
strrev(exp);
j=l-i-1;
strncat(exp1,exp,j);
strrev(exp1);
printf("Three address code:\ntemp=%s\ntemp1=%c%ctemp\n",exp1,exp[j+1],exp[j]);
}
void div()
{
strncat(exp1,exp,i+2);
printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);
}
void plus()
{
strncat(exp1,exp,i+2);
printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);
}
```

Output:

```
1. assignment
2. arithmetic
3. relational
4. Exit
Enter the choice:1
Enter the expression with assignment operator:
a=b
Three address code:
temp=b
a=temp

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:
a+b-c
Three address code:
temp=a+b
temp1=temp-c

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:
a-b/c
Three address code:
temp=b/c
temp1=a-temp

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:
a*b-c
Three address code:
temp=a*b
temp1=temp-c

1.assignment
2.arithmetic
3.relational
4.Exit
```

```
Enter the choice:2
Enter the expression with arithmetic operator:a/b*c
Three address code:
temp=a/b
temp1=temp*c
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:3
Enter the expression with relational operator
a
<=
b

100 if a<=b goto 103
101 T:=0
102 goto 104
103 T:=1

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:4
```

PROGRAM-14

AIM:Write a C program for implementation of Code Generation Algorithm of a given expression/statement.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char op[2],arg1[5],arg2[5],result[5];
void main()
{
 FILE *fp1,*fp2;
 fp1=fopen("input.txt","r");
 fp2=fopen("output.txt","w");
 while(!feof(fp1))
 {

  fscanf(fp1,"%s%s%s%s",op,arg1,arg2,result);
  if(strcmp(op,"+")==0)
  {
   fprintf(fp2,"\nMOV R0,%s",arg1);
   fprintf(fp2,"\nADD R0,%s",arg2);
   fprintf(fp2,"\nMOV %s,R0",result);
  }
```

```
     if(strcmp(op,"*")==0)
    {
      fprintf(fp2,"\nMOV R0,%s",arg1);
      fprintf(fp2,"\nMUL R0,%s",arg2);
      fprintf(fp2,"\nMOV %s,R0",result);
    }
    if(strcmp(op,"-")==0)
    {
      fprintf(fp2,"\nMOV R0,%s",arg1);
      fprintf(fp2,"\nSUB R0,%s",arg2);
      fprintf(fp2,"\nMOV %s,R0",result);
    }
       if(strcmp(op,"/")==0)
    {
      fprintf(fp2,"\nMOV R0,%s",arg1);
      fprintf(fp2,"\nDIV R0,%s",arg2);
      fprintf(fp2,"\nMOV %s,R0",result);
    }
if(strcmp(op,"=")==0)
    {
      fprintf(fp2,"\nMOV R0,%s",arg1);
      fprintf(fp2,"\nMOV %s,R0",result);
    }
    }
    fclose(fp1);
    fclose(fp2);
    getch();
  }
}
```

input:

```
+ a b t1
* c d t2
- t1 t2 t
= t ? x
```

Output:

```
MOV R0,a
ADD R0,b
MOV t1,R0
MOV R0,c
MUL R0,d
MOV t2,R0
MOV R0,t1
SUB R0,t2
MOV t,R0
MOV R0,t
MOV x,R0
```