DFA

Aim: Program to implement a DFA to accept strings ending with abc

Program:

```
#include <stdio.h>
#include <string.h>
void q0(char s[], int i)
{  if(i==strlen(s))
   {
      printf("Rejected and halted at state q0");
      return;
   }
   if(s[i]=='a')
   {  printf("(q0,a)-->q1");
      q1(s, i+1);
   }
   else if(s[i]=='b')
   {  printf("(q0,b)-->q0");
      q0(s, i+1);
   }
   else
   {  printf("(q0,c)->q0");
      q0(s,i+1);
   }
}
void q1(char s[], int i)
{
```

```
if (i==strlen(s))
{
   printf("Rejected and halted at state q1");
   return;
}
if(s[i]=='a')
{
   printf("(q1,a)-->q1");
   q1(s,i+1);
}
else if (s[i]=='b')
{
   printf("(q1,b)-->q2");
    q2(s,i+1);
}
else{
   printf("(q1,c)-->q0");
   q0(s,i+1);
}
}
void q2(char s[],int i)
{
  if(i==strlen(s))
  {
   printf("Rejected and halted at state q2");
   return;
  }
  if(s[i]=='a')
  {
   printf("(q2,a)-->q1");
    q1(s,i+1);
  }
```

```
else if(s[i]=='b'){
   printf("(q2, b) -> q0");
    q0(s,i+1);
 }
 else {
    printf("(q2,c) -> q3");
    q3(s,i+1);
 }
}
void q3(char s[],int i)
{
     if(i==strlen(s))
     { printf("Rejected Accepted");
       return;
     }
     if(s[i]=='a'){
       printf("(q3,a) -> q1");
       q1(s,i+1);
      }
     else if(s[i]=='b')
     { printf("(q3,b) -> q0");
       q0(s,i+1);
     }
     else{
       printf("(q3,c) -> q0");
        q0(s,i+1);
     }
}
```

```
int main()
{
    char s[100];
    int i=strlen(s);
    printf("Enter a string");
    scanf("%s",&s);
    q0(s,0);
    return 0;
}
```

Output:

1) Enter a string aabc

$(q_0, a) \rightarrow q_1$

$(q_1, a) \rightarrow q_1$

$(q_1, b) \rightarrow q_2$

$(q_2, c) \rightarrow q_3$

Accepted

2) Enter a string ababa

$(q_0, a) \rightarrow q_1$

$(q_1, b) \rightarrow q_2$

$(q_2, a) \rightarrow q_1$

$(q_1, b) \rightarrow q_2$

$(q_2, a) \rightarrow q_1$

Rejected and halted at state q_1

LEXICAL ANALYZER

Aim: Program to implement lexical Analyzer.

Source code:

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>
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);
}
bool isOperator(char ch)
{ if(ch=='+'||ch=='-'||ch=='*'||ch=='/'||ch=='>'||ch=='<'||ch=='=')
    return(true);
  return(false);
}
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);
}
```

```
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, "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);
}
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);
}
```

```
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);
}
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);
}
void parse(char* str)
{
    int left = 0, right = 0, count = 0, 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 REALNUMBER\n",substr);
        else if(validIdentifier(substr)==true && isDelimiter(str[right-1]==
                                                    false)
        {
            count++;
            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;
    }
}
printf("%d", count);
return;
}
int main()
{
    char str[100];
    scanf("%[^\n]s", str);
    parse(str);
    return 0;
}
```

Output:

1) "float x=a+1b;"

'u' IS A VALID IDENTIFIER

'float' IS A KEYWORD.

'x' IS A VALID IDENTIFIER

'=' IS AN OPERATOR

'a' IS A VALID IDENTIFIER

'+' IS AN OPERATOR

'1b' IS NOT A VALID IDENTIFIER

';' IS A VALID IDENTIFIER.

LEFT RECURSION

Aim: Program to eliminate left recursion using C program.

Source code:

```
#include<stdio.h>
#include<string.h>
#define SIZE 10
int main()
{
    char nt,a,b;
    int num,i;
    char p[10][SIZE];
    int index=3;
    printf("Enter Number of Productions:");
    scanf("%d",&num);
    printf("Enter the grammer as E->EA:\n");
    for(i=0;i<num;i++)
        scanf("%s",p[i]);
    for(i=0;i<num;i++)
    {   printf("\n GRAMMER: %s",p[i]);
        nt=p[i][0];
        if(nt==p[i][index])
        {
            a=p[i][index+1];
            printf("is left recursive:\n");
            while(p[i][index]!=0 && p[i][index]!='|')
                index++;
```

```
            if (p[i][index]!=0)
            {
                b=p[i][index+1];
                printf("Grammer without left recursion\n");
                printf("%c->%c%c\'\n",nt,b,nt);
                printf("\n%c\'->%c%c\'/E\n",nt,a,nt);
            }
            else
                printf("cant be reduced");
        }
        else
            printf(" is not recursive);
        index=3;
    }
}
```

Output:

Enter Number of Productions: 4 Enter the grammer as E->E-A:

E->EA|A

A->AT|a

T->a

E->i

GRAMMER: E->EA|A is left recursive

Grammer without left recursion

E->AE'

E'->AE'|E

Grammer: A->AT|a is left recursive

Grammer without left recursion:

A->aA'

A'->TA'|E

GRAMMER: T->a is not left recursive

GRAMMER: E->i is not left recursive

LEFT FACTORING

Aim: Program to eliminate left factoring using C program.

Source code:

```
#include<stdio.h>
#include<string.h>
int main()
{
    char g[20], p1[20], p2[20], mg[20], ng[20], tg[20];
    int i, j=0, k=0, l=0, pos;
    printf("Enter Production: A->");
    fgets(g, M20, stdin);
    for(i=0; g[l]!='|'; i++, j++)
        p1[j] = g[i];
    p1[j] = '\0';
    for(j=++i, i=0; g[j]!='\0'; j++, i++)
        p2[i] = g[j];
    p2[i] = '\0';
    for(i=0; i<strlen(p1) || i<strlen(p2); i++)
    {
        if(p1[i]==p2[i])
        {
            mg[k] = p1[i];
            k++;
            pos = i+1;
        }
    }
    for(i=pos, j=0; p1[i]!='\0'; i++, j++)
        ng[j] = p1[i];
```

```
ng[j++] = '|';
for(i=pos; pa[i]!='\0'; i++, j++)
    ng[j] = pa[i];
mg[k] = 'X';
mg[++k] = '\0';
ng[j] = '\0';
printf("\n A->%s", mg);
printf("\n X->%s", ng);
}
```

Output:

1) Enter Production : A → aE+bCD|aE+eIT

A → aE+X
X → bCD|eIT

2) Enter Production: A → bCdT|bcBit

A → bcX
X → dT|Bit

3) Enter Production: A → CE+bCD|CE+cdT

A → CE+X
X → bCD|cdT

FIRST AND FOLLOW

Aim: Program to implement first and follow for a given grammer using C program.

Source code:

```
#include<stdio.h>
#include <math.h>
#include<string.h>
#include <ctype.h>
#include <stdlib.h>
int n,m=0,p,i=0,j=0;
char a[10][10],f[10];
void follow (char c)
{
    if(a[0][0]==c)
        f[m++]='$';
    for(i=0;i<n;i++)
    {
        for(j=2;j<strlen(a[i]);j++)
        {
            if(a[i][j]==c)
            {
                if(a[i][j+1]!='\0')
                    first(a[i][j+1]);
                if(a[i][j+1]=='\0' && c!=a[i][0])
                    follow(a[i][0]);
            }
        }
    }
}
```

```
void first(char c)
{
    int k;
    if(!issuper(c))
      f[m++]=c;
    for(k=0;k<n;k++)
    {
        if(a[k][0]==c)
        {
            if(a[k][2]=='$')
                follow(a[k][0]);
            else if(islower(a[k][2]))
                f[m++]=a[k][2];
            else
                first(a[k][2]);
        }
    }
}
int main()
{
    int i,z;
    char c,ch;
    printf("Enter the no of productions:\n");
    scanf("%d",&n);
    printf("Enter the productions:\n");
    for(i=0;i<n;i++)
      scanf("%s%c",a[i],&ch);
    do
    {
        m=0;
        printf("Enter the elements whose first and follow is to be
                                                found:");
        scanf("%c",&c);
```

```
    first(c);
    printf("First(%c)={",c);
    for(i=0;i<m;i++)
        printf("%c",f[i]);
    printf("}\n");
    strcpy(f," ");
    m=0;
    follow(c);
    printf("Follow(%c)={",c);
    for(i=0;i<m;i++)
        printf("%c",f[i]);
    printf("}\n");
    printf("continue(0/1)?");
    scanf("%d%c",&z,&ch);
  }while(z==1);
  return 0;
}
```

Output:

```
Enter the no.of productions:
5
Enter the productions:
S=AbCd
A=Cf
A=a
C=gE
E=h
```

Enter the elements whose first & follow is to be found: S

First (S) = {g, a}

Follow (S) = {$}

continue (0/1) ? 1

Enter the elements whose first & follow is to be found: A

First (A) = {g, a}

Follow (A) = {b}

continue (0/1) ? 1

Enter the elements whose first & follow is to be found: C

First (C) = {g}

Follow (C) = {d}

continue (0/1) ? 0

NON-RECURSIVE PREDICTIVE PARSER

Aim: Implementation of Non-Recursive Predictive Parser using C programming

Source code:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
char s[30],st[30];
void main()
{
    char tab[5][6][5]={"ta","@","ta","@","@","@","+ta","@","@",
    "!","!","fb","@","@","fb","@","@","@","!","*fb","@","!",
    "!",";","@","@","(e)","@","@","@"};
    printf("Enter the string:\n");
    scanf("%s",s);
    strcat(s,"$");
    st[0]='$';
    st[1]='e';
    int st_i,s_i,i,j,k,n,s1,s2;
    st_i=1;
    s_i=0;
    char temp[20];
    printf("\n stack Input\n");
    while(st[st_i]!='$'||s[s_i]!='$')
    {
        switch(st[st_i])
        {
            case 'e':s1=0;break;
```

```
        case 'a': s1=1; break;
        case 't': s1=2; break;
        case 'b': s1=3; break;
        case 'f': s1=4; break;
        default : s1=-1;
    }
switch(s[s_i])
{
        case 'i': s2=0; break;
        case 't': s2=1; break;
        case '*': s2=2; break;
        case '(': s2=3; break;
        case ')': s2=4; break;
        case '$': s2=5; break;
        default: s2=-1;
    }
if(s1==-1 || s2==-1)
{
        printf("Failure");
        exit(0);
    }
if(tab[s1][s2]=="@")
{
        printf("Failure");
        exit(0);
    }
if(tab[s1][s2]=="!")
{
        st[st_i]='\0';
        st_i--;
    }
```

```
else {
    j=strlen(tab[s1][s2]);
    for(k=0; tab[s1][s2][k]!='\0'; k++)
    {
        temp[j-k-1]=tab[s1][s2][k];
    }
    temp[j]='\0';
    st[st_i]='\0';
    strcat(st,temp);
}
printf("%s",st);
for(n=s_i; s[n]!='\0'; n++)
    printf("%c",s[n]);
printf("\n");
if(st[st_i]==s[s_i] && s[s_i]!='$')
{
    st[st_i]='\0';
    s_i++;
    st_i--;
}
}
printf("Success");
}
```

Output:

Enter the string:

i*i+i*i

Stack	Input
$at	i*i+i*i$
$abf	i*i+i*i$
$abi	i*i+i*i$
$abf*	*i+i*i$
$abi	i+i*i$
$a	+i*i$
$at+	+i*i$
$abf	i*i$
$abi	i*i$
$abf*	*i$
$abi	i$
$a	$
$	$

Success.

SHIFT REDUCE PARSING

Aim: Implementation of Shift Reduce Parsing using C programming

Source code:

```
#include<stdio.h>
#include<string.h>
int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check()
{
    strcpy(ac,"REDUCE TO E");
    for(z=0;z<c;z++)
      if(stk[z]=='i' && stk[z+1]=='d')
      {
          stk[z]='E';
          stk[z+1]='\0';
          printf("\n$%s\t%s$\t%s",stk,a,ac);
          j++;
      }
    for(z=0;z<c;z++)
      if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='E')
      {
          stk[z]='E';
          stk[z+1]='\0';
          stk[z+2]='\0';
          printf("\n$%s\t%s$\t%s",stk,a,ac);
          i=i-2;
      }
    for(z=0;z<c;z++)
      if(stk[z]=='E' && stk[z+1]=='*' && stk[z+2]=='E')
      {
          stk[z]='E';
```

```
            stk[z+1] = '\0';
            stk[z+1] = '\0';
            printf("\n$%s\t%s$\t%s", stk, a, ac);
            i = i-2;
        }
    for(z=0; z<c; z++)
        if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]==')')
        {
            stk[z] = 'E';
            stk[z+1] = '\0';
            stk[z+2] = '\0';
            printf("\n$%s\t%s$\t%s", stk, a, ac);
            i = i-2;
        }
}
int main()
{
    puts("GRAMMER is E->E+E \n E->E*E \n E->(E) \n E->id");
    puts("Enter input string");
    gets(a);
    c = strlen(a);
    strcpy(act, "SHIFT->");
    puts("Stack \t Input \t Action");
    for(k=0, i=0; j<c; k++, i++, j++)
    {
        if(a[j]=='i' && a[j+1]=='d')
        {
            stk[i] = a[j];
            stk[i+1] = a[j+1];
```

```
            stk[i+2]='\0';
            a[j]=' ';
            a[j+1]=' ';
            printf("\n$%s\t%s$\t%s id",stk,a,act);
            check();
        }
        else
        {
            stk[i+1]=a[j];
            stk[i+1]='\0';
            a[j]=' ';
            printf("\n$%s\t%s$\t%s symbols",stk,a,act);
            check();
        }
    }
    if(stk[0]=='E' && stk[1]=='\0')
        printf("\n Accept");
    else
        printf("\n Reject");
}
```

Output:

GRAMMER is E→E+E

E→E*E

E→(E)

E→id

Enter input string

id * id + id

Stack	Input	Action
$id	*id+id$	Shift → id
$E	*id+id$	REDUCE TO E
$E*	id+id$	SHIFT → symbols
$E*id	+id$	SHIFT → id
$E*E	+id$	REDUCE to E
$E	+id$	REDUCE to E
$E+	id$	SHIFT → symbols
$E+id	$	SHIFT → id
$E+E	$	REDUCE to E
$E	$	REDUCE to E

Accept.

OPERATOR PRECEDENCE PARSING

Aim: Implementation of Operator Precedence Parsing using C programming.

Source code:

```
#include<stdio.h>
#include<string.h>
void main()
{
    char stk[20], ip[20], opt[10][10][1], ter[10];
    int i,j,k,n, top=0, col, row;
    for(i=0;i<10;i++)
    {
        stk[i]=0;
        ip[i]=0;
        for(j=0;j<10;j++)
            opt[i][j][1]=0;
    }
    printf("Enter the no.of terminals:");
    scanf("%d",&n);
    printf("\n Enter the terminals:");
    scanf("%s",ter);
    printf("\n Enter the table values:");
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("\n 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]);
}
stk[top]='$';
printf("\n Enter the input string:");
scanf("%s", ip);
i=0;
printf("\n Stack \t\t\t INPUT STRING \t\t\t ACTION\n");
printf("\n%s\t\t\t\t%s\t\t\t", stk, ip);
while(i<=strlen(ip))
{
    for(k=0;k<n;k++)
    {   if(stk[top]==ter[k])
            col=k;
        if(ip[i]==ter[k])
            row=k;
    }
    if((stack[top]=='$')&&(ip[i]=='$'))
    {
        printf("String is accepted");
        break;
    }
```

```
else if ((opt[col][row][0] == '<') || (opt[col][row][0] == '='))
{
        stk[++top] = opt[col][row][0];
        stk[++top] = ip[i];
        printf("Shift %c", ip[i]);
        i++;
    }
    else
    {
        if (opt[col][row][0] == '>')
        {
            while (stk[top] != '<')
                --top;
            top = top - 1;
            printf("Reduce");
        }
        else
        {
            printf("\n string is not accepted");
            break;
        }
    }
    printf("\n");
    for (k=0; k<=top; k++)
        printf("%c", stk[k]);
    printf("\t\t\t");
    for (k=i; k<strlen(ip); k++)
        printf("%c", ip[k]);
    printf("\t\t\t");
  }
}
```

Output:

```
Enter the no.of terminals: 4
Enter the terminals: i+*$
Enter the table values:
Enter the value for i i: =
Enter the value for i+: >
Enter the value for i*: >
Enter the value for i$: >
Enter the value for +i: <
Enter the value for ++: >
Enter the value for +*: <
Enter the value for +$: >
Enter the value for *i: <
Enter the value for *+: >
Enter the value for **: >
Enter the value for *$: >
Enter the value for $i: <
Enter the value for $+: <
Enter the value for $*: <
Enter the value for $$: =
```

OPERATOR PRECEDENCE TABLE:

	i	+	*	$
i	=	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	=

Enter the input string: i*i+i$

STACK	INPUT STRING	ACTION
$	i*i+i$	Shift i
$<i	*i+i$	Reduce
$	*i+i$	Shift *
$<*	i+i$	Shift i
$<*<i	+i$	Reduce
$<*	+i$	Reduce
$	+i$	Shift +
$<+	i$	Shift i
$<+<i	$	Reduce
$<+	$	Reduce
$	$	String is accepted

STACK ALLOCATION

Aim: Implementation of Stack Allocation Strategy using C programming.

Source code:

```
#include<stdio.h>
int stack[100],ch,n,top,x,i;
void push()
{
   if (top>=n-1)
      printf("Stack overflow");
   else{
      printf("Enter value to be pushed");
      scanf("%d",&x);
      top++;
      stack[top]=x;
    }
}
void pop()
{
  if (top==-1)
     printf("Stack underflow");
  else
  { printf("Popped element is %d",stack[top]);
     top--;
  }
}
void display()
{
   if(top>=0)
   {
```

```
    printf("The elements in stack");
    for(i=top; i>=0; i--)
        printf("%d",stack[i]);
    printf("Select next choice");
  }
  else
     printf("Stack is empty");
}
int main()
{
    top=-1;
    printf("Enter the size of stack");
    scanf("%d",&n);
    printf("Stack Operations:");
    printf("----------");
    printf("1.Push\t 2.Pop\t 3.Display\t 4.Exit");
    do{
      printf("\n Enter the choice:");
      scanf("%d",&ch);
      switch(ch)
      {
          case 1: push(); break;
          case 2: pop(); break;
          case 3: display(); break;
          case 4: printf("Exit"); break;
          default: printf("Please enter a valid choice:");
      }
```

```
    }while(ch!=4);
    return 0;
}
```

Output:

```
Enter the size of stack:10
STACK Operations:
1.Push 2.Pop 3.Display 4.Exit
Enter the choice:2
Enter a value to be pushed:52
Enter Stack underflow.
Enter the choice:1
Enter a value to be pushed:10
Enter the choice:1
Enter a value to be pushed:20
Enter the choice:1
Enter a value to be pushed:30
Enter the choice:3
The elements in stack:
  30
  20
  10
Select next choice:
Enter the choice:2
The popped element is 30
Enter the choice:1
```

```
Enter a value to be pushed:40
Enter a choice:1
Enter a value to be pushed:50
Enter a choice:3
The elements in stack
  50
  40
  20
  10
Select next choice
Enter a choice:4
  EXIT
```

INTERMEDIATE CODE GENERATION

Aim: Implementation of Intermediate code generation using C programming.

Source code:

```
#include<stdio.h>
#include <string.h>
int i=1,j=0, no=0, tmpch=90;
char str[100], left[15], right[15];
void findopr()
{
    for(i=0;str[i]!='\0';i++)
        if(str[i]==':')
        {
            k[j].pos=i;
            k[j++].op=':';
        }
    for(i=0;str[i]!='\0';i++)
        if(str[i]=='/')
        {
            k[j].pos=i;
            k[j++].op='/';
        }
    for(i=0;str[i]!='\0';i++)
        if(str[i]=='*')
        {
            k[j].pos=i;
            k[j++].op='*';
        }
    for(i=0;str[i]!='\0';i++)
        if(str[i]=='+'){
```

```
            k[j].pos=i;
            k[j++].op='+';
        }
        for(i=0;str[i]!='\0';i++)
            if(str[i]=='-')
            {
                k[j].pos=i;
                k[j++].op='-';
            }
}
void explore()
{
    i=1;
    while(k[i].op!='\0')
    {
        fleft(k[i].pos);
        fright(k[i].pos);
        str[k[i].pos]=tmpch--;
        printf("\t%c:=%s%c%s\t\t",str[k[i].pos],left,k[i].op,right);
        for(j=0;j<strlen(str);j++)
            if(str[j]!='$')
                printf("%c",str[j]);
        printf("\n");
        i++;
    }
    fright(-1);
    if(no==0){
        fleft(strlen(str));
        printf("\t%s:=%s",right,left);}
```

```
    printf("\t%s:=%c",right,str[k[--i].pos]);
}
void fleft(int x)
{
    int w=0,flag=0;
    x--;
    while(x!=-1 && str[x]!='#' && str[x]!='*' && str[x]!='±' && str[x]!=
    '\0' && str[x]!='-' && str[x]!='/' && str[x]!=':')
    {
        if(str[x]!='$' && flag==0)
        {
            left[w++]=str[x];
            left[w]='\0';
            str[x]='$';
            flag=1;
        }
        x--;
    }
}
void fright(int x)
{
    int w=0,flag=0;
    x++;
    while(x!=-1 && str[x]!='+' && str[x]!='*' && str[x]!='=' &&
    str[x]!='\0' && str[x]!='=' && str[x]!='/' && str[x]!=':')
    {
        if(str[x]!='$' && flag==0)
        {
            right[w++]=str[x];
            right[w]='\0';
            str[x]='$';
            flag=1;
        }
        x++;
    }
}
```

```
struct exp
{
    int pos;
    char op;
}k[15];
void main()
{
    printf("Intermediate Code Generation");
    printf("Enter the expression");
    scanf("%s",str);
    printf("The intermediate code:\t\t Expression\n");
    findopr();
    explore();
}
```

Output:

Intermediate Code Generation

Enter the expression: w:=a/b*c+d/e-g*h

The intermediate code	Expression
z:=a/b	w:=z*c+d/e-g*h
y:=d/e	w:=z*c+y-g*h
x:=z*c	w:=x+y-g*h
w:=g*h	w:=x+y-w
v:=x+y	w:=v-w
u:=v-w	w:=u
w:u w:=$	