DEWAN VS INSTITUTE OF ENGINEERING & TECHNOLOGY, MEERUT

COLLEGE CODE (311)

Department of CSE

COMPILER DESIGN LAB (BCS-652)

VI SEM CSE

LIST OF PROGRAMS

1. Write a program to check whether a given line is comment or not.
2. Write a program to test whether an identifier is valid or not.
3. Implement to simulate Lexical Analyzer for validating operators.
4. Write a program in C to calculate FIRST of a given grammar..
5. Write a program in C to calculate the FOLLOW of a given grammar.
6. Write a program in C for implementation of Symbol Table.
7. Implementation of Operator Precedence Parser.
8. Write a program in C to implement backend of compiler.
9. Write a program in C to implement three address codes.
10. Implementation of Recursive Descent Parsing.

(Mr Babloo Kumar)

PROGRAM NO. 1

Object: Write a program in C to check whether a given line is comment or not.

Theory: Comments are non-executable code used to provide documentation to programmer. Single Line Comment is used to comment out just Single Line in the Code. It is used to provide One Liner Description of line.

1. Single Line Comment Can be Placed Anywhere
2. Single Line Comment Starts with '//'
3. Any Symbols written after '//' are ignored by Compiler

Multi Line Comment

1. Multi line comment can be placed anywhere.
2. Multi line comment starts with /*.
3. Multi line comment ends with */.
4. Any symbols written between '/*' and '*/' are ignored by Compiler.
5. It can be split over multiple lines.

Coding:

```
#include<stdio.h>

#include<conio.h>

void main() {

char com[30];

int i=2,a=0;

clrscr();

printf("\n Enter comment:");

gets(com);

if(com[0]=='/') {

 if(com[1]=='/')
```

```
 printf("\n It is a comment");
 else if(com[1]=='*') {
 for(i=2;i<=30;i++)
 {
 if(com[i]=='*'&&com[i+1]=='/')
 {
 printf("\n It is a comment");
 a=1;
 break; }
 else
 continue; }
 if(a==0)
 printf("\n It is not a comment");
}
else
printf("\n It is not a comment");
 }
else
 printf("\n It is not a comment");
getch();
}
```

Output:


Enter comment://compiler Design
It is a comment_

PROGRAM NO. 2

Object: Write a program in C to test whether an identifier is valid or not.

Theory: In C programming, identifiers are names given to C entities, such as variables, functions, structures etc. Identifier are created to give unique name to C entities to identify it during the execution of program. For example:

```
int money;

int mango_tree;
```

Here, money is a identifier which denotes a variable of type integer. Similarly, mango_tree is another identifier, which denotes another variable of type integer.

Rules for writing identifier

1. An identifier can be composed of letters (both uppercase and lowercase letters), digits and underscore '_' only.
2. The first letter of identifier should be either a letter or an underscore. But, it is discouraged to start an identifier name with an underscore though it is legal. It is because, identifier that starts with underscore can conflict with system names. In such cases, compiler will complain about it. Some system names that start with underscore are _fileno, _iob, _wfopen etc.
3. There is no rule for the length of an identifier. However, the first 31 characters of an identifier are discriminated by the compiler. So, the first 31 letters of two identifiers in a program should be different.

Coding:

```
#include<stdio.h>

#include<conio.h>

#include<ctype.h>

void main()

{

char a[10];

int flag, i=1;

clrscr();

printf("\n Enter an identifier:");

gets(a);
```

```
if(isalpha(a[0]))
 flag=1;
else
 printf("\n Not a valid identifier");
while(a[i]!='\0')
{
 if(!isdigit(a[i])&&!isalpha(a[i]))
 {
 flag=0;
 break;
 }
 i++;
}
if(flag==1)
 printf("\n Valid identifier");
getch();
}
```

Output:

PROGRAM NO. 3

Object: Write a program in C to simulate Lexical Analyzer for validating operators.

Theory: Lexical analysis is the first phase of a compiler. It takes the modified source code from language pre-processors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

Coding:

```
#include<stdio.h>

#include<conio.h>

void main()

{

char s[5];

clrscr();

printf("\n Enter any operator:");

gets(s);

switch(s[0])

{

case'>': if(s[1]=='=')

printf("\n Greater than or equal");

else

printf("\n Greater than");

break;

case'<': if(s[1]=='=')

printf("\n Less than or equal");
```

```
else
printf("\nLess than");
break;
case'=': if(s[1]=='=')
printf("\nEqual to");
else
printf("\nAssignment");
break;
case'!': if(s[1]=='=')
printf("\nNot Equal");
else
printf("\n Bit Not");
break;
case'&': if(s[1]=='&')
printf("\nLogical AND");
else
printf("\n Bitwise AND");
break;
case'|': if(s[1]=='|')
printf("\nLogical OR");
else
printf("\nBitwise OR");
break;
case'+': printf("\n Addition");
break;
case'-': printf("\nSubstraction");
break;
```

```
case'*': printf("\nMultiplication");
break;
case'/': printf("\nDivision");
break;
case'%': printf("Modulus");
break;
default: printf("\n Not a operator");
}
getch();
}
```

Output:

PROGRAM NO. 4

Object: Write a program in C for the implementation of Lexical analyser.

Theory: Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

Coding:

```
#include<stdio.h>

#include<string.h>

#include<stdlib.h>

void removeduplicate();

void final();

int Isiden(char ch);

int Isop(char ch);

int Isdel(char ch);

int Iskey(char * str);

void removeduplicate();

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];
```

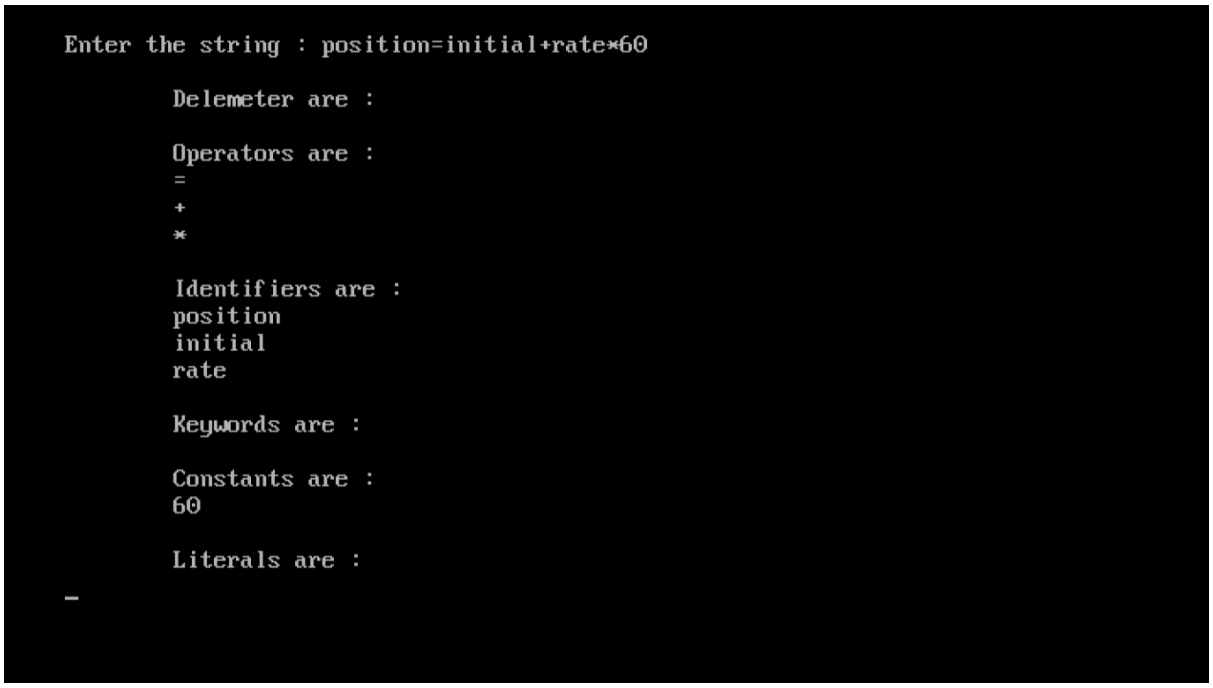
```
      for(j=i;j<deli;j++)
     {
         if(delem[i-1]==delem[j])
          uqdeli[j]=1;
     }
  }
 }
for(i=1;i<idi+1;i++)  //removing duplicate  identifiers
   {
  if(uqideni[i-1]==0)
     {
    strcpy(uqiden[uqidi++],iden[i-1]);
    for(j=i;j<idi;j++)
     {
       if(!strcmp(iden[i-1],iden[j]))
          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]);
   }
printf("\n\tDelemeter 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();
}
```

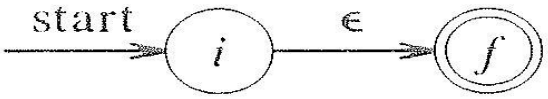
OUTPUT:



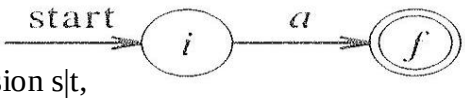
PROGRAM NO. 5

Object: Write a program in C to implement NFA from a regular expression.

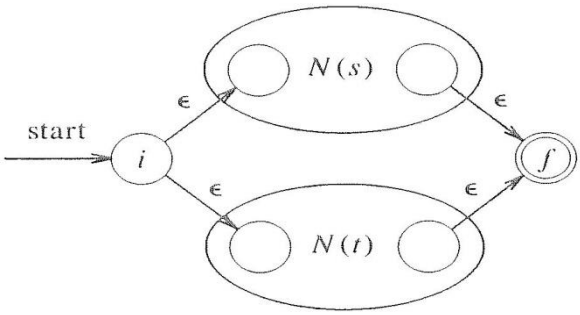
Theory: The rules for constructing an NFA consist of basis rules for handling subexpressions with no operators, and inductive rules for constructing larger NFA's from the NFA's for the immediate subexpressions of a given expression.
For the Regular expression ε the NFA



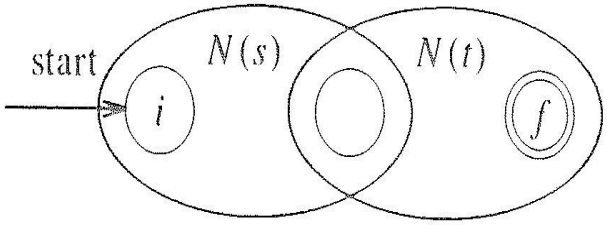
For any subexpression a in C, construct the NFA



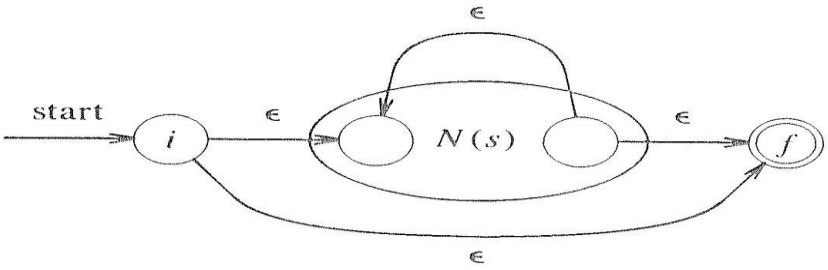
For the regular expression s|t,



For the regular expression st,



For the regular expression S*,



Coding:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
struct current{int first,last;}stat[15];
int l,j,change,n=0,i=0,state=1,x,y,start,final;
char store,*input1,input[15];
clrscr();
printf("\n\n****ENTER THE REGULAR EXPRESSION****\n\n");
scanf("%s",input1);/*ex inputs:1.(a*) 2.(a|b) 3.(a.b) 4.((a|b).(a*))*/
for(i=0;i<10;i++)
input[i]=NULL;
l=strlen(input1);
a:
for(i=0;input1[i]!=')';i++);
for(j=i;input1[j]!='(';j--);
for(x=j+1;x<i;x++)
if(isalpha(input1[x]))
input[n++]=input1[x];
else if(input1[x]!='0')
store=input1[x];
input[n++]=store;
for(x=j;x<=i;x++)
input1[x]='0';
```

```
if(input1[0]!='0')
goto a;
printf("\n\n\tFROM\tTO\tINPUT\n\n");
i=0;
while(input[i]!='\0')
{
if(isalpha(input[i]))
{
stat[i].first=state++;
stat[i].last=state++;
printf("\n\t%d\t%d\t%c",stat[i].first,stat[i].last,input[i]);
}
else
{
change=0;
switch(input[i])
{
case'|':
stat[i].first=state++;
stat[i].last=state++;
x=i-2;
y=i-1;
if(!isalpha(input[y]))
{
b:
switch(input[y])
{
```

```
case'*':if(!isalpha(input[y-1]))
{
y=y-1;
goto b;
}
else
x=y-2;
break;
case'|':x=y-3;
break;
case '.':x=y-3;break;
}
change=1;
}
if(!isalpha(input[y]&&change==0))
c:switch(input[x])
{
case '*':
if(!isalpha(input[x-1]))
{x=x-1;goto c;
}
else x=x-2;
break;
case'|':x=x-2;
break;
case '.':x=x-3;
break;
```

```
}
printf("\n\t%d\t%d\tE",stat[i].first,stat[x].first);
printf("\n\t%d\t%d\tE",stat[x].last,stat[i].last);
printf("\n\t%d\t%d\tE",stat[i].first,stat[i-1].first);
printf("\n\t%d\t%d\tE",stat[i-1].last,stat[i].last);
start=stat[i].first;
final=stat[i].last;
break;
case'.':
x=i-2;
y=i-1;
if(!isalpha(input[y]))
{
d:
switch(input[y])
{
case'*':if(!isalpha(input[y-1]))
{
y=y-1;
goto d;
}
else
x=y-2;
break;
case'|':x=y-3;
break;
case '.':x=y-3;
```

```
break;
}
change=1;
}
if(!isalpha(input[y]&&change==0))
e:switch(input[x])
{
case'*':
if(!isalpha(input[x-1]))
{
x=x-1;
goto e;
}
else x=x-2;
break;
case'|':x=x-3;
break;
case'.':x=x-3;
break;
}
stat[i].last=stat[y].last;
stat[i].first=stat[x].first;
printf("\n\t%d\t%d\tE",stat[x].last,stat[i-1].first);
start=stat[x].first;
final=stat[i-1].last;
break;
case'*':
```

```
stat[i].first=state++;
stat[i].last=state++;
printf("\n\t%d\t%d\tE",stat[i].first,stat[i-1].first);
printf("\n\t%d\t%d\tE",stat[i].first,stat[i].last);
printf("\n\t%d\t%d\tE",stat[i-1].last,stat[i-1].first);
printf("\n\t%d\t%d\tE",stat[i-1].last,stat[i].last);
start=stat[i].first;
final=stat[i].last;
break;
}
}
i++;
}
printf("\n the starting state is %d",start);
printf("\n the final state is %d",final);
getch();
}
```

OUTPUT:

```
****ENTER THE REGULAR EXPRESSION****

((a¦b)*)


        FROM    TO      INPUT


        1       2       a
        3       4       b
        5       1       E
        2       6       E
        5       3       E
        4       6       E
        7       5       E
        7       8       E
        6       5       E
        6       8       E
 the starting state is 7
 the final state is 8
```

Program No. 6

Object: Implementation of Shift Reduce Parsing

Coding:

```
#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);
```

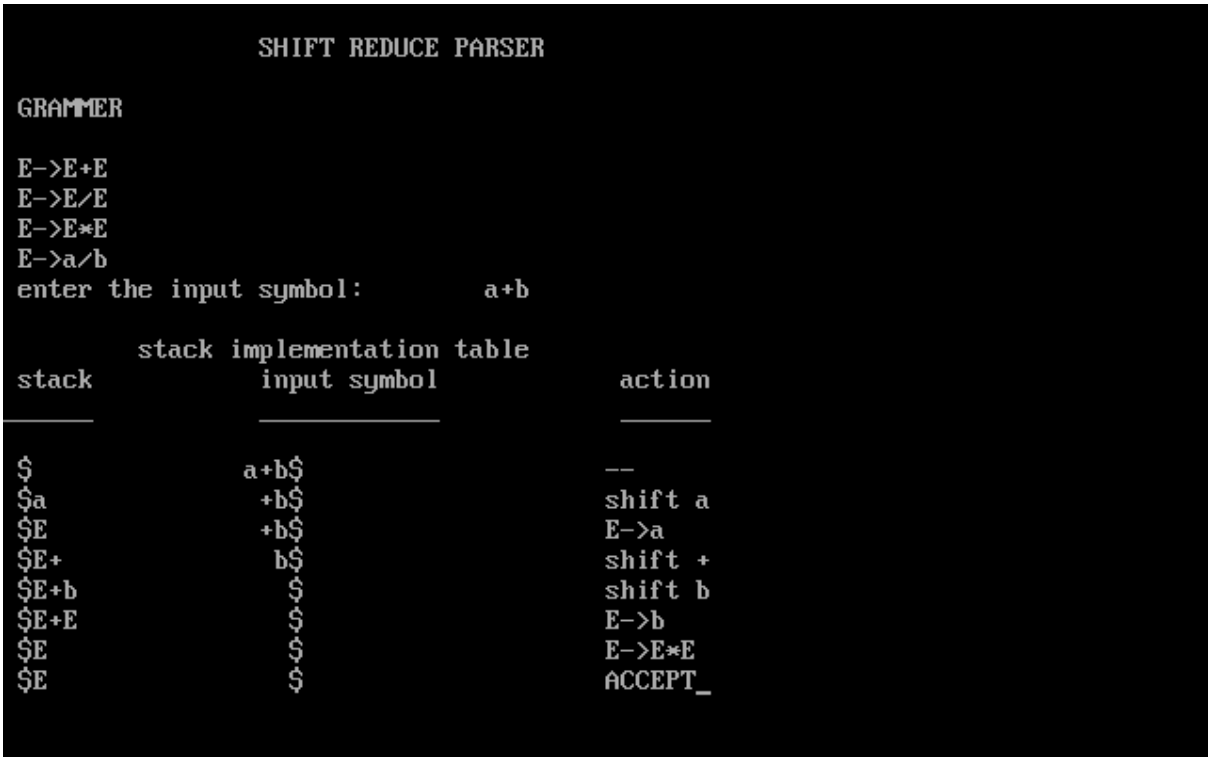
```
printf("\n\t stack implementation

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;
}
```

Output:


SHIFT REDUCE PARSER
GRAMMER
E->E+E
E->E/E
E->E*E
E->a/b
enter the input symbol: a+b
stack implementation table
stack input symbol action
$ a+b$ --
$a +b$ shift a
$E +b$ E->a
$E+ b$ shift +
$E+b $ shift b
$E+E $ E->b
$E $ E->E*E
$E $ ACCEPT_

\

PROGRAM No. 7

Program: Implementation of Operator Precedence Parser.

Coding:

```
/*OPERATOR PRECEDENCE PARSER*/


#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char stack[20],ip[20],opt[10][10][1],ter[10];
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:");
scanf("%d",&n);
printf("\nEnter the terminals:");
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("\nOPERATOR 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");
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();
}
```

Output:

```
Enter the no.of terminals:4

Enter the terminals:+*i$

Enter the table values:
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 i +:>
Enter the value for i *:>
Enter the value for i i:=
Enter the value for i $:>
Enter the value for $ +:<
Enter the value for $ *:<
Enter the value for $ i:<
Enter the value for $ $:A
```

OPERATOR PRECEDENCE TABLE:
+ * i $

+ > < < >
* > > < >
i > > = >
$ < < < A
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
$<+ *i$ Shift *
$<+<* i$ Shift i
$<+<*<i $ Reduce
$<+<* $ Reduce
$<+ $ Reduce
$ $ String is accepted
Press Enter to return to Quincy...

PROGRAM NO. 8

Object: Write a program in C to calculate the FIRST of a given grammar.

Theory: For a terminal symbol a, let FIRST(a) = {a}.

1. For a production rule of the form A → aα, add the terminal symbol a to FIRST(A).
2. For a production rule of the form A → ϵ, add the empty string ϵ to FIRST(A).
3. For a production rule of the form A → X1⋯Xk, add to FIRST(A)
    - The empty string ϵ, if ϵ is in FIRST(X1),…, FIRST(Xk)
    - The terminal symbol a, if ϵ is in FIRST(X1),…, FIRST(Xi−1) and a is in FIRST(Xi)

Coding:

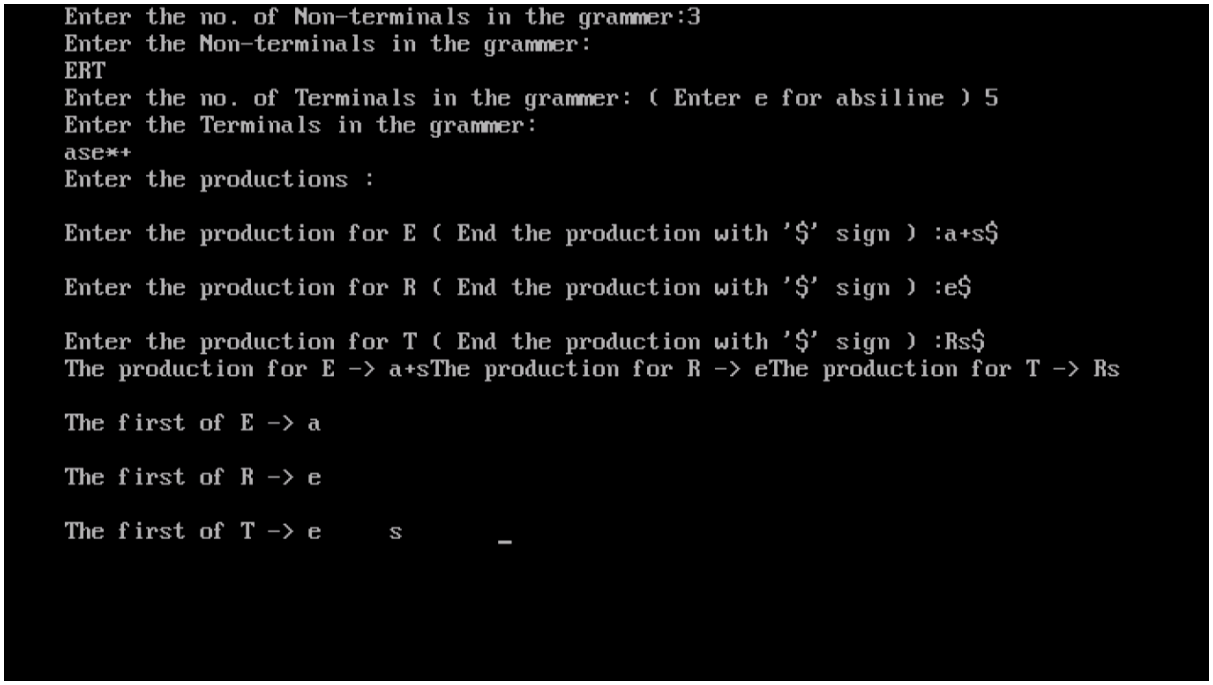
```
#include<stdio.h>

#include<conio.h>

#include<string.h>

void main()

{

char t[5],nt[10],p[5][5],first[5][5],temp; int i,j,not,nont,k=0,f=0;

clrscr();

printf("\nEnter the no. of Non-terminals in the grammer:"); scanf("%d",&nont);

printf("\nEnter the Non-terminals in the grammer:\n"); for(i=0;i<nont;i++)

{

scanf("\n%c",&nt[i]);

}

printf("\nEnter the no. of Terminals in the grammer: ( Enter e for absiline ) ");
scanf("%d",&not);

printf("\nEnter the Terminals in the grammer:\n"); for(i=0;i<not||t[i]=='$';i++)

{

scanf("\n%c",&t[i]);
```

```
}
for(i=0;i<nont;i++)
{
p[i][0]=nt[i];
first[i][0]=nt[i];
}
printf("\nEnter the productions :\n"); for(i=0;i<nont;i++)
{
scanf("%c",&temp);
printf("\nEnter the production for %c ( End the production with '$' sign ) :",p[i][0]);
for(j=0;p[i][j]!='$';)
{
j+=1;
scanf("%c",&p[i][j]);
}
}
for(i=0;i<nont;i++)
{
printf("\nThe production for %c -> ",p[i][0]);
for(j=1;p[i][j]!='$';j++)
{
printf("%c",p[i][j]);
}
}
for(i=0;i<nont;i++)
{
f=0; for(j=1;p[i][j]!='$';j++)
```

```
{
for(k=0;k<not;k++)
{
if(f==1)
break;
if(p[i][j]==t[k])
{
first[i][j]=t[k]; first[i][j+1]='$'; f=1;
break;
}
else if(p[i][j]==nt[k])
{
first[i][j]=first[k][j];
if(first[i][j]=='e')
continue; first[i][j+1]='$'; f=1;
break;
}
}
}
}
for(i=0;i<nont;i++)
{
printf("\n\nThe first of %c -> ",first[i][0]); for(j=1;first[i][j]!='$';j++)
{
printf("%c\t",first[i][j]);
}
}
```

```
getch();

}
```

OUTPUT:



PROGRAM NO. 9

Object: Write a program in C to calculate the FOLLOW of the given grammar.

Theory:

1. For the start symbol S, add the end-of-input marker $ into FOLLOW(S)
2. For a production rule A → αB, add FOLLOW(A)-{ϵ} to FOLLOW(B)
3. For a production rule A → αBβ where β≠ϵ, add FIRST(β)-{ϵ} to FOLLOW(B)
4. For a production rule A → αBβ where ϵ is in FIRST(β), add FOLLOW(A) to FOLLOW(B)

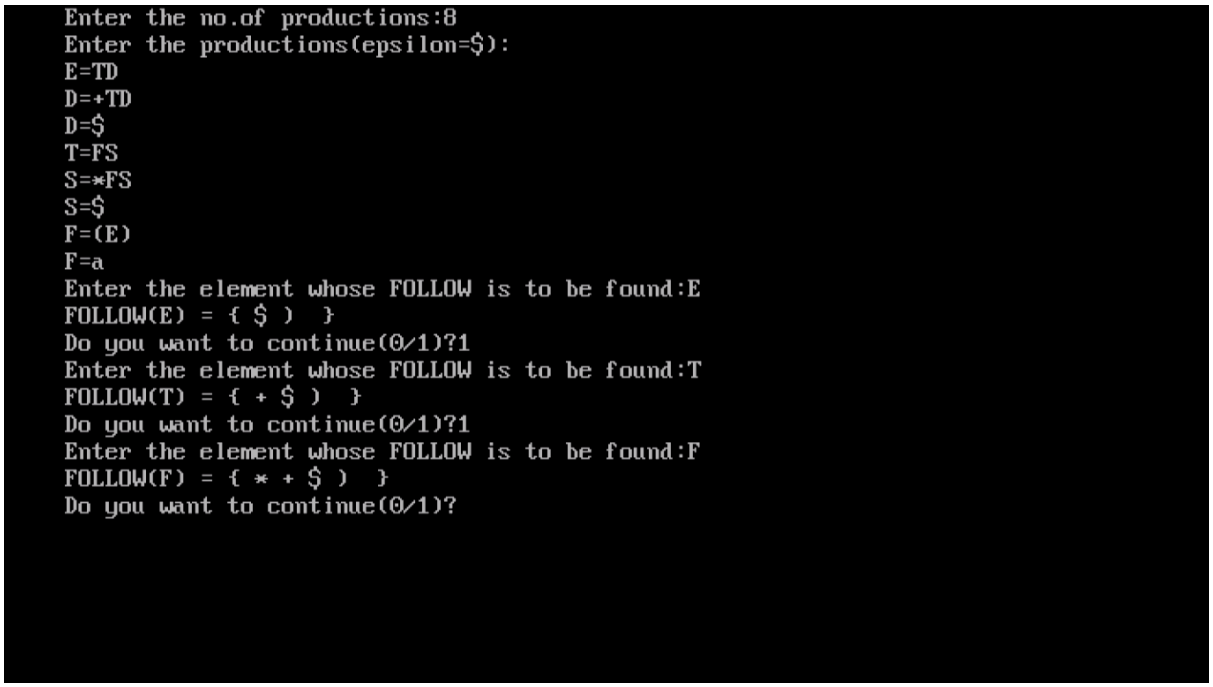
Coding:

```
#include<stdio.h>

#include<string.h>

#include<ctype.h>

int n,m=0,p,i=0,j=0;

char a[10][10],f[10];

void follow(char c);

void first(char c);

int main()

{

 int i,z;

 char c,ch;

printf("Enter the no.of productions:");

 scanf("%d",&n);

 printf("Enter the productions(epsilon=$):\n");

 for(i=0;i<n;i++)

  scanf("%s%c",a[i],&ch);

 do
```

```
 {
  m=0;
  printf("Enter the element whose FOLLOW is to be found:");
  scanf("%c",&c);
  follow(c);
  printf("FOLLOW(%c) = { ",c);
  for(i=0;i<m;i++)
   printf("%c ",f[i]);
  printf(" }\n");
  printf("Do you want to continue(0/1)?");
  scanf("%d%c",&z,&ch);
 }
 while(z==1);
}
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(!(isupper(c)))f[m++]=c;
               for(k=0;k<n;k++)
               {
               if(a[k][0]==c)
               {
               if(a[k][2]=='$') follow(a[i][0]);
               else if(islower(a[k][2]))f[m++]=a[k][2];
               else first(a[k][2]);
               }
               }
}
```

OUTPUT:



PROGRAM NO. 10

Object: Write a program in C to implement Recrusive Descent Parsing.

Theory: One of the most straightforward forms of parsing is recursive descent parsing. This is a top-down process in which the parser attempts to verify that the syntax of the input stream is correct as it is read from left to right. A basic operation necessary for this involves reading characters from the input stream and matching then with terminals from the grammar that describes the syntax of the input. Our recursive descent parsers will look ahead one character and advance the input stream reading pointer when proper matches occur.

Coding:

```
#include<stdio.h>

#include<conio.h>

#include<string.h>

char input[100];

int i,l;

void main()

{

clrscr();

printf("\nRecursive descent parsing for the following grammar\n");

 printf("\nE->TE'\nE'->+TE'/@\nT->FT'\nT'->*FT'/@\nF->(E)/ID\n");

 printf("\nEnter the string to be checked:");

 gets(input);

 if(E())

 {

 if(input[i+1]=='\0')

 printf("\nString is accepted");

 else

 printf("\nString is not accepted");

 }
```

```
else
 printf("\nString not accepted");
 getch();
}
E()
{
 if(T())
 {
 if(EP())
 return(1);
 else
 return(0);
 }
else
 return(0);
}
EP()
{
 if(input[i]=='+')
 {
 i++;
 if(T())
 {
 if(EP())
return(1);
 else
return(0);
```

```
 }
else
 return(0);
 }
 else
 return(1);
}
T()
{
if(F())
 {
 if(TP())
 return(1);
 else
 return(0);
 }
else
 return(0);
}
TP()
{
 if(input[i]=='*')
 {
 i++;
 if(F())
 {
 if(TP())
```

```
 return(1);
 else
 return(0);
 }
else
 return(0);
 }
 else
 return(1);
}
F()
{
if(input[i]=='(')
 {
 i++;
 if(E())
 {
 if(input[i]==')')
 {
i++;
return(1);
 }
else
 return(0);
 }
else
 return(0);
```

```
 }
 else if(input[i]>='a'&&input[i]<='z'||input[i]>='A'&&input[i]<='Z')
{
i++;
return(1);
}
 else
return(0);
}
```

Output: