Aim: To design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines.

Algorithm:

Step 1: Start

Step 2: Declare the necessary variables.

Step 3: Declare an array and store the keywords in that array

Step 4: Open the input file in read open

Step 5: read the string from the file till the end of file.

Step 6: If the first character in the strings # then prints that string as header file.

Step 7: If the string matches with any of the keywords print that string is a keyword

Step 8: If the string matches with operator and special Symbols print the corresponding message

Step 9: If the string is not a keyword then prints that as an identifier.

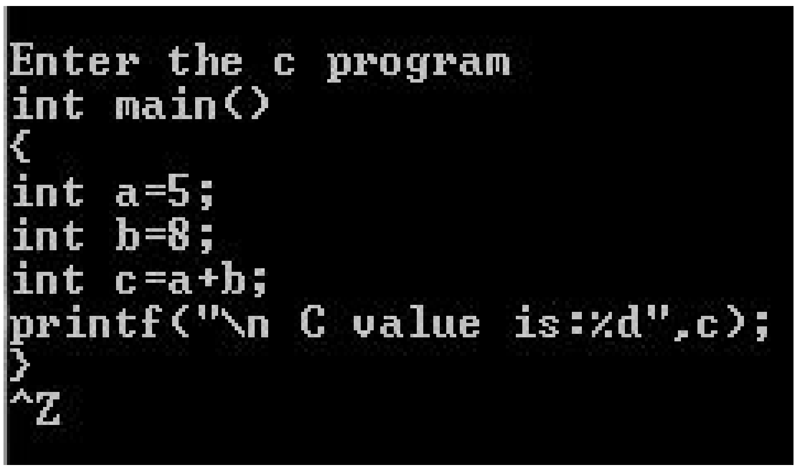
Step 10: End

Source Code:

```
#include<string.h>
#include<ctype.h>
#include<stdio.h>
void keyword(char str[10])
{
if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||
strcmp("int",str)==0||strcmp("float",str)==0||strcmp("char",str)==0||
strcmp("double",str)==0||strcmp("static",str)==0||strcmp("switch",str)==0||
strcmp("case",str)==0)
printf("\n%s is a keyword",str);
else
printf("\n%s is an identifier",str);
}
main()
{
FILE *f1,*f2,*f3;
char c,str[10],st1[10];
int num[100],lineno=0,tokenvalue=0,i=0,j=0,k=0;
clrscr();
printf("\nEnter the c program");/*gets(st1);*/
```

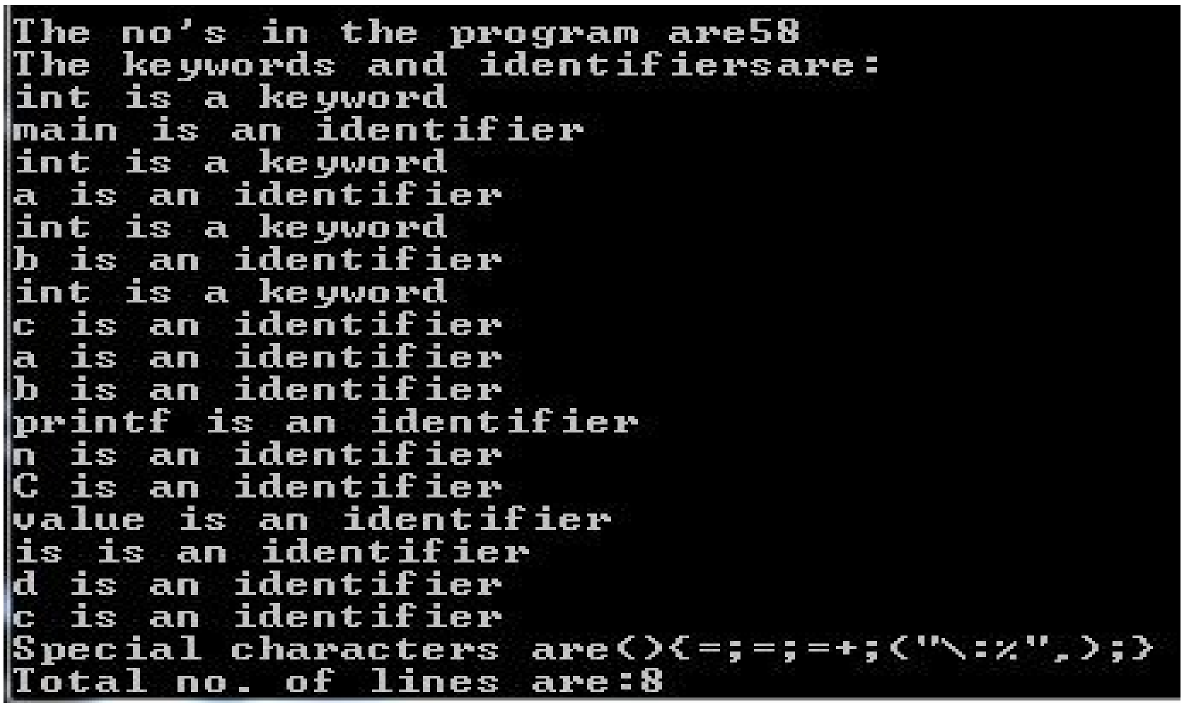
```
f1=fopen("input","w");
while((c=getchar())!=EOF)
putc(c,f1);
fclose(f1);
f1=fopen("input","r");
f2=fopen("identifier","w");
f3=fopen("specialchar","w");
while((c=getc(f1))!=EOF)
{
 if(isdigit(c))
{
tokenvalue=c-'0';
c=getc(f1);
while(isdigit(c))
{
tokenvalue*=10+c-'0';
c=getc(f1);
}
num[i++]=tokenvalue;
ungetc(c,f1);
}
else if(isalpha(c))
{
putc(c,f2);
c=getc(f1);
while(isdigit(c)||isalpha(c)||c=='_'||c=='$')
{
putc(c,f2);
c=getc(f1);
}
putc(' ',f2);
ungetc(c,f1);
}
else if(c==' '||c=='\t')
printf("");
else if(c=='\n')
lineno++;
else
putc(c,f3);
}
fclose(f2);
fclose(f3);
fclose(f1);
printf("\nThe no's in the program are");
for(j=0;j<i;j++)
printf("%d",num[j]);
printf("\n");
f2=fopen("identifier","r");
k=0;
printf("The keywords and identifiersare:");
while((c=getc(f2))!=EOF)
{
if(c!=' ')
str[k++]=c;
else
{
str[k]='\0';
keyword(str);
k=0;
}
}
fclose(f2);
f3=fopen("specialchar","r");
printf("\nSpecial characters are");
while((c=getc(f3))!=EOF)
printf("%c",c);
printf("\n");
fclose(f3);
printf("Total no. of lines are:%d",lineno);
}
```

Input:



```
Enter the c program
int main()
{
int a=5;
int b=8;
int c=a+b;
printf("\n C value is:%d",c);
}
^Z
```

Output:



```
The no's in the program are58
The keywords and identifiersare:
int is a keyword
main is an identifier
int is a keyword
a is an identifier
int is a keyword
b is an identifier
int is a keyword
c is an identifier
a is an identifier
b is an identifier
printf is an identifier
n is an identifier
C is an identifier
value is an identifier
is is an identifier
d is an identifier
c is an identifier
Special characters are(){=;=;=+;("\:%",);}
Total no. of lines are:8
```

Viva questions:

1. What is lexical analysis?

   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.

2. Type of tokens?

   C tokens are the basic buildings blocks in C language which are constructed together to write a C program. Each and every smallest individual unit in a C program is known as C token.

   C tokens are of six types. They are,

   ✓ Keywords (eg: int, while),
   ✓ Identifiers (eg: main, total),
   ✓ Constants (eg: 10, 20),
   ✓ Strings (eg: “total”, “hello”),
   ✓ Special symbols (eg: (), {}),

- ✓ Operators (eg: +, /,-,*)

3. What are the Identifiers in C language?

   An **identifier** is a string of alphanumeric characters that begins with an alphabetic character or an underscore character that are used to represent various programming elements such as variables, functions, arrays, structures, unions and so on. Actually, an **identifier** is a user-defined word.

4. Rules for constructing identifier name in C?
   - ✓ First character should be an alphabet or underscore.
   - ✓ Succeeding characters might be digits or letter.
   - ✓ Punctuation and special characters aren't allowed except underscore.
   - ✓ Identifiers should not be keywords.

5. Keywords in C language?

   Keywords are predefined; reserved words used in programming that have special meaning. **Keywords** are part of the syntax and they cannot be used as an identifier. For example: int money; Here, int is a **keyword** that indicates 'money' is a variable of type integer.There are **32 Keywords** in C

   Example: for, do, while, if, case and etc

Aim: To simulate First and Follow of a Grammar.

Algorithm for First:

Step 1: Start

Step 2: Declare FILE pointer

Step 3: open the file in write mode

Step 4: Read the Productions

Step 5: Compute First function

Step 6: stop the productions.

Function First:

Step 1: If X is a terminal then First(X) is just X!

Step 2: If there is a Production $X \rightarrow \varepsilon$ then add ε to First(X)

Step 3: If there is a Production $X \rightarrow Y_1Y_2\ldots Y_k$ then add First($Y_1Y_2\ldots Y_k$) to First(X), First($Y_1Y_2\ldots Y_k$) is either First(Y_1) (if First(Y_1) doesn't contain ε)OR (if First(Y_1) does contain ε) then First ($Y_1Y_2\ldots Y_k$) is everything in First(Y_1) <except for ε > as well as everything in First($Y_2\ldots Y_k$)

Step 4: If First(Y_1) First(Y_2)..First(Y_k) all contain ε then add ε to First($Y_1Y_2\ldots Y_k$) as well.

Algorithm for Follow:

Step 1: Start

Step 2: Declare FILE pointer

Step 3: open the file in write mode

Step 4: Read the follow Productions

Step 5: Compute First function

Step 6: stop the productions.

Function Follow :

Step 1: First put $ (the end of input marker) in Follow(S) (S is the start symbol)

Step 2: If there is a production $A \rightarrow aBb$, (where a can be a whole string) then everything in

FIRST(b) except for ε is placed in FOLLOW(B).

Step 3: If there is a production A → aB, then everything in FOLLOW(A) is in FOLLOW(B)

Step 4: If there is a production A → aBb, where FIRST(b) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)

Source Code:

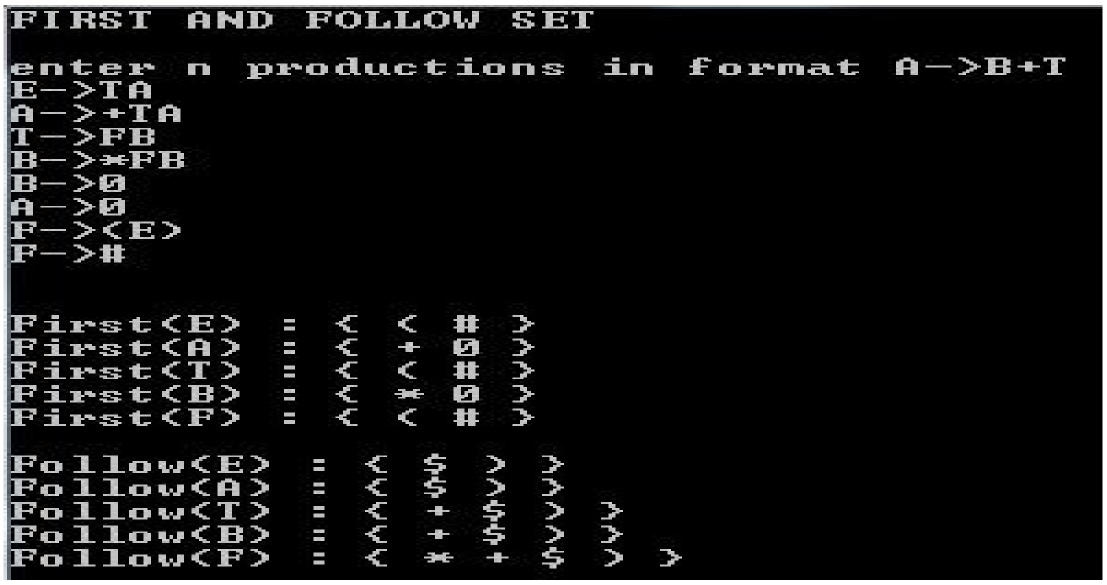
```
#include<stdio.h>
#include<ctype.h>
char a[8][8];

struct firTab
{
   int n;
   char firT[5];
};
struct folTab
{
   int n;
   char folT[5];
};
struct folTab follow[5];
struct firTab first[5];
int col;
void findFirst(char,char);
void findFollow(char,char);
void folTabOperation(char,char);
void firTabOperation(char,char);
void main()
{
   int i,j,c=0,cnt=0;
   char ip;
   char b[8];
   clrscr();
   printf("\nFIRST AND FOLLOW SET \n\nenter n productions in format A->B+T\n");
   for(i=0;i<8;i++)
   {
   scanf("%s",&a[i]);
   }
   for(i=0;i<8;i++)
```

```
 {   c=0;
 for(j=0;j<i+1;j++)
 {
    if(a[i][0] == b[j])
    {
        c=1;
        break;
    }
    }
 if(c !=1)
 {
   b[cnt] = a[i][0];
   cnt++;
 }

 }
  printf("\n");
 for(i=0;i<cnt;i++)
 {   col=1;
 first[i].firT[0] = b[i];
 first[i].n=0;
 findFirst(b[i],i);
 }
 for(i=0;i<cnt;i++)
 {
 col=1;
 follow[i].folT[0] = b[i];
 follow[i].n=0;
 findFollow(b[i],i);
  }
 printf("\n");
for(i=0;i<cnt;i++)
{
 for(j=0;j<=first[i].n;j++)
 {
        if(j==0)
        {
           printf("First(%c) : {",first[i].firT[j]);
        }
        else
```

```
            {
                printf(" %c",first[i].firT[j]);
            }
    }
    printf(" } ");
    printf("\n");
    }
     printf("\n");
   for(i=0;i<cnt;i++)
    {
    for(j=0;j<=follow[i].n;j++)
    {
            if(j==0)
            {
                printf("Follow(%c) : {",follow[i].folT[j]);
            }
            else
            {
                printf(" %c",follow[i].folT[j]);
            }
    }
    printf(" } ");

    printf("\n");
    }
}
void findFirst(char ip,char pos)
{
    int i;
    for(i=0;i<8;i++)
    {
        if(ip == a[i][0])
        {
            if(isupper(a[i][3]))
            {
                findFirst(a[i][3],pos);
            }
            else
        {
```

```
        first[pos].firT[col]=a[i][3];
        first[pos].n++;
        col++;
           }
        }
    }
}
void findFollow(char ip,char row)
{   int i,j;
    if(row==0 && col==1)
    {
        follow[row].folT[col]= '$';
        col++;
        follow[row].n++;
    }
    for(i=0;i<8;i++)
    {
        for(j=3;j<7;j++)
        {
            if(a[i][j] == ip)
            {
                if(a[i][j+1] == '\0')
                {
                    if(a[i][j] != a[i][0])
                    {
                        folTabOperation(a[i][0],row);
                    }
                }
                else if(isupper(a[i][j+1]))
                {   if(a[i][j+1] != a[i][0])
                    {
                        firTabOperation(a[i][j+1],row);
                }
                }
                else
                {
                    follow[row].folT[col] = a[i][j+1];
                    col++;
                    follow[row].n++;
                }
            }
        }
    }
}
void folTabOperation(char ip,char row)
{   int i,j;
    for(i=0;i<5;i++)
    {
        if(ip == follow[i].folT[0])
        {
            for(j=1;j<=follow[i].n;j++)
            {
                follow[row].folT[col] = follow[i].folT[j];
                col++;
                follow[row].n++;
            }
        }
    }
}
void firTabOperation(char ip,char row)
{
        int i,j;
    for(i=0;i<5;i++)
    {
        if(ip == first[i].firT[0])
        {
            for(j=1;j<=first[i].n;j++)
            {
                if(first[i].firT[j] != '0')
                {
                    follow[row].folT[col] = first[i].firT[j];
                    follow[row].n++;
                    col++;
                }
                else
                {
                    folTabOperation(ip,row);
                }
            }
        }
    }

}
```

Output:



```
FIRST AND FOLLOW SET

enter n productions in format A->B+T
E->TA
A->+TA
T->FB
B->*FB
B->0
A->0
F->(E)
F->#

First(E)  =  {  (  #  }
First(A)  =  {  +  0  }
First(T)  =  {  (  #  }
First(B)  =  {  *  0  }
First(F)  =  {  (  #  }

Follow(E)  =  {  $  )  }
Follow(A)  =  {  $  )  }
Follow(T)  =  {  +  $  )  }
Follow(B)  =  {  +  $  )  }
Follow(F)  =  {  *  +  $  )  }
```

Viva questions:

1. Rules for First Sets?
   - ✓ If X is a terminal then First(X) is just X!
   - ✓ If there is a Production X → ε then add ε to First(X)
   - ✓ If there is a Production X → $Y_1Y_2…Y_k$ then add First($Y_1Y_2…Y_k$) to First(X),
   - ✓ First($Y_1Y_2…Y_k$) is either First(Y_1) (if First(Y_1) doesn't contain ε)OR (if First(Y_1) does contain ε) then First ($Y_1Y_2…Y_k$) is everything in First(Y_1) <except for ε > as well as everything in First($Y_2…Y_k$)
   - ✓ If First(Y_1) First(Y_2)..First(Y_k) all contain ε then add ε to First($Y_1Y_2…Y_k$) as well.

2. Rules for Follow Sets?
   - ✓ First put $ (the end of input marker) in Follow(S) (S is the start symbol)
   - ✓ If there is a production A → aBb, (where a can be a whole string) then everything in FIRST(b) except for ε is placed in FOLLOW(B).
   - ✓ If there is a production A → aB, then everything in FOLLOW(A) is in FOLLOW(B)
   - ✓ If there is a production A → aBb, where FIRST(b) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)

Aim: To develop an operator precedence parser for a given language.

Algorithm:

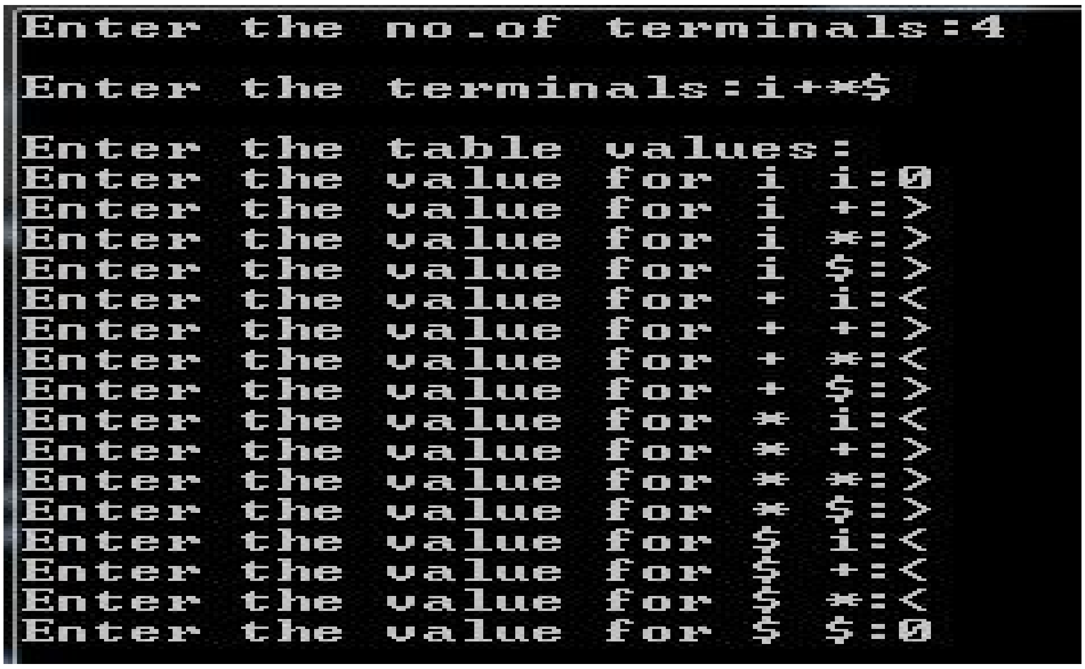
```
Step 1: Start
Step 2: Initially set ip to point to the first symbol of the input string w$
Step 3: Let b be the top stack symbol, a the input symbol pointed to by ip
Step 4: if a is $ and b is $ then return
Step 5: else if  b <· a or b =· a then push a onto the stack
Step 6: advance ip to the next input symbol
Step 7: else if b ·> a then pop b from the stack.
Step 8: repeat the Step 2  to Step 6 until the string is accepted.
Step 9: else error
Step 10: End
```

Source Code:

```
#include<stdio.h>
#include<conio.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");
```

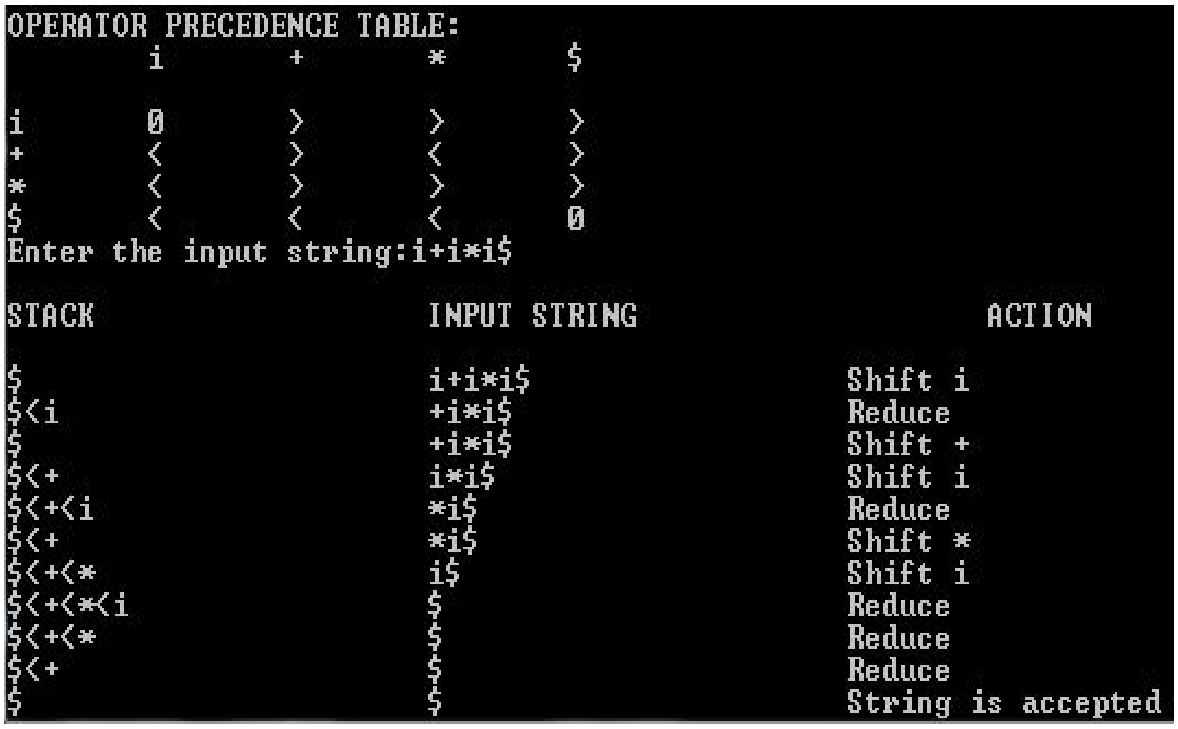
```
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
printf("Enter the value for %c %c:",ter[i],ter[j]);
scanf("%s",opt[i][j]);
}
}
printf("\nOPERATOR PRECEDENCE TABLE:\n");
for(i=0;i<n;i++)
{
printf("\t%c",ter[i]);
}
printf("\n");
for(i=0;i<n;i++)
{
printf("\n%c",ter[i]);
for(j=0;j<n;j++)
{
printf("\t%c",opt[i][j][0]);
}
}
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();
}
```

Input:



```
Enter the no.of terminals:4

Enter the terminals:i+*$

Enter the table values:
Enter the value for i i:0
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 $ $:0
```

Output:



```
OPERATOR PRECEDENCE TABLE:
        i       +       *       $

i       0       >       >       >
+       <       >       <       >
*       <       >       >       >
$       <       <       <       0
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
```

<u>Viva questions:</u>

1. What is operator grammar?

   Operator Grammar is a context-free **grammar** that has the property (among others) that no production has either an empty right-hand side or two adjacent non-terminals in its right-hand side. These properties allow precedence relations to be **defined** between the terminals of the **grammar**.

2. Represent the precedence relation and its meaning table with example?

Relation	Meaning
a <· *b*	*a* yields precedence to *b*
a =· *b*	*a* has the same precedence as *b*
a ·>*b*	*a* takes precedence over *b*

   For example, the following operator precedence relations canbe introduced for simple expressions:

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

3. What type of parsing it is?

   It is a kind of bottom-up parser.

Aim: To construct a recursive descent parser for an expression.

Algorithm:

Step 1: Start.

Step 2: Declare the terminals as variables

Step 3: Implement the non-terminals as [recursive] procedures depends upon the productions

Step 4: Read the input string to be parsed.

Step 5: Parser parse each input symbol depending upon procedures

Step 6: if the input string parsing is completed, parser return

Step 7: else error

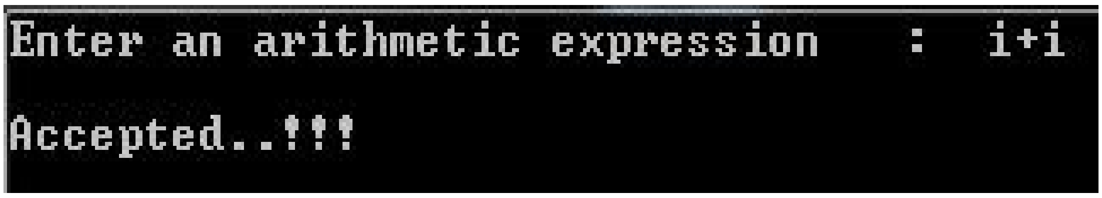
Step 8: Stop

Source Code:

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

char input[10];
int i,error;
void E();
void T();
void Eprime();
void Tprime();
void F();
main()
{
i=0;
error=0;
clrscr();
printf("Enter an arithmetic expression   :  ");
gets(input);
E();
if(strlen(input)==i&&error==0)
printf("\nAccepted..!!!\n");
else printf("\nRejected..!!!\n");
```

```
}

void E()
{
    T();
    Eprime();
}
void Eprime()
{
    if(input[i]=='+')
    {
    i++;
    T();
    Eprime();
    }
}
void T()
{
    F();
    Tprime();
}
void Tprime()
{
    if(input[i]=='*')
    {
                 i++;
                 F();
                 Tprime();
                        }
     }

void F()
    {
           if(isalnum(input[i]))
           i++;
        else if(input[i]=='(')
        {
        i++;
        E();
        if(input[i]==')')
        i++;
          else
          error=1;
          }
          else
          error=1;
    }
```

Output:



```
Enter an arithmetic expression   :  i+i

Accepted..!!!
```

Viva questions:

1. What type of parsing it is ?
   it is a kind of top-down parsing.

2. What is Recursive descent parser?
   Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.

3. What is back-tracking?
   A predictive parser runs in linear time. Recursive descent with backtracking is a technique that determines which production to use by trying each production in turn. Recursive descent with backtracking is not limited to LL(k) grammars, but is not guaranteed to terminate unless the grammar is LL(k).

Aim: To construct a LL(1) parser for an expression.

Algorithm for LL(1)parser:

Step 1: If A → α is a production choice, and there is a derivation α ⇒∗ aβ,where a is a token, then we add A → α to the table entry M[A, a].

Step 2: If A → α is a production choice, and there are derivations α ⇒∗ε andS$ ⇒∗ βAaγ, where S is the start symbol and a is a token (or $), then we add A → α to the table entry M[A, a]

Source Code:

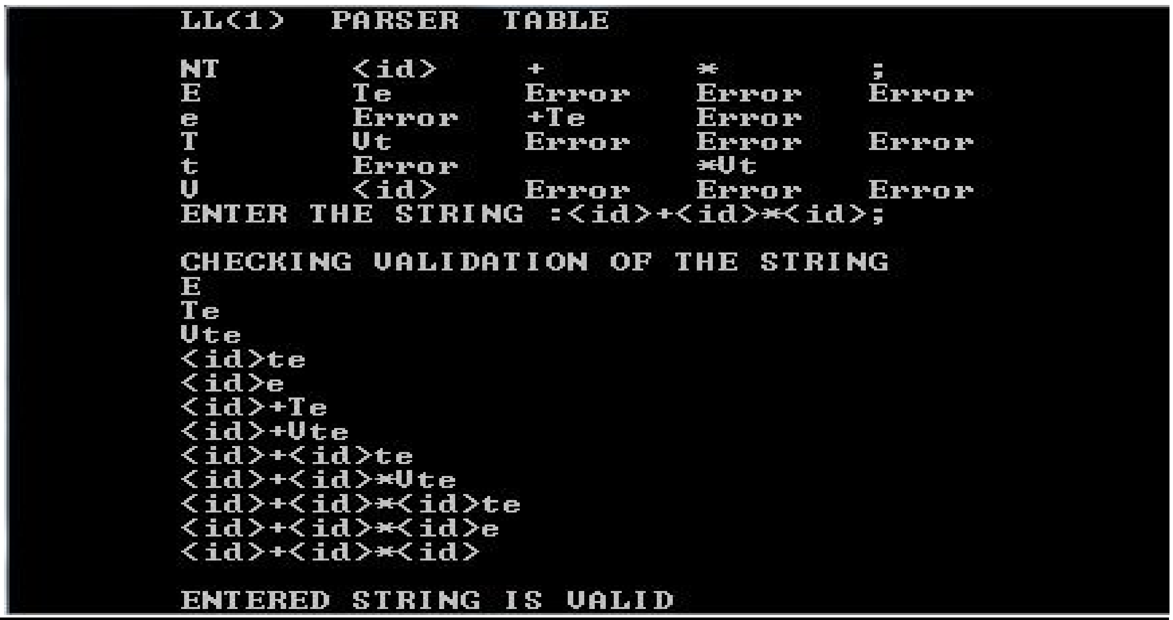
```
#include <conio.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void main()
{
   int i=0,j=0,k=0,m=0,n=0,o=0,o1=0,var=0,l,f,c=0,f1=0,len;
   char str[30],str1[40]="E",temp[20],temp1[20],temp2[20],tt[20],t3[20];
   char t[10];
   char array[6][5][10] = {
"NT", "<id>","+","*",";",
"E", "Te","Error","Error","Error",
"e", "Error","+Te","Error","\0",
"T", "Vt","Error","Error","Error",
"t", "Error","\0","*Vt","\0",
"V", "<id>","Error","Error","Error"
              };
   clrscr();
   strcpy(temp1,'\0');
   strcpy(temp2,'\0');
   printf("\n\tLL(1)  PARSER  TABLE \n");
   for(i=0;i<6;i++)
   {
        printf("\n");
      for(j=0;j<5;j++)
      {
          printf("\t");
        printf("%s",array[i][j]);
      }
   }
```

```
printf( "\n\tENTER THE STRING :");
gets(str);
if(str[strlen(str)-1]!=';')
{
     printf( "END OF STRING MARKER SHOULD BE ';'");
     getch();
     exit(1);
}
printf("\n\tCHECKING VALIDATION OF THE STRING");
printf("\n\t%s",str1);
i=0;

while(i<strlen(str))
{
 again:
     if(str[i] == ' ' && i<strlen(str))
     {
         printf( "\n\tSPACES IS NOT ALLOWED IN SOURCE STRING ");
         getch();
         exit(1);
     }
     temp[k]=str[i];
     temp[k+1]='\0';
     f1=0;
 again1:
     if(i>=strlen(str))
     {
         getch();
         exit(1);
     }
     for(l=1;l<=4;l++)
     {
       if(strcmp(temp,array[0][l])==0)
       {
          f1=1;
          m=0,o=0,var=0,o1=0;
          strcpy(temp1,'\0');
          strcpy(temp2,'\0');
          len=strlen(str1);
          while(m<strlen(str1)&& m<strlen(str))
```

```
{
      if(str1[m]==str[m])
      {
           var=m+1;
           temp2[o1]=str1[m];
           m++;
           o1++;
      }
      else
      {
           if((m+1)<strlen(str1))
           {
               m++;
               temp1[o]=str1[m];
               o++;
           }
           else
               m++;
      }

}
temp2[o1] = '\0';
temp1[o] = '\0';
t[0] = str1[var];
t[1] = '\0';
for(n=1;n<=5;n++)
{
    if(strcmp(array[n][0],t)==0)
        break;
}
strcpy(str1,temp2);
strcat(str1,array[n][l]);
strcat(str1,temp1);
printf( "\n\t%s",str1);
getch();

if(strcmp(array[n][l],'\0')==0)
{
    if(i==(strlen(str)-1))
    {
          len=strlen(str1);
          str1[len-1]='\0';
          printf( "\n\t%s",str1);
          printf( "\n\n\tENTERED STRING IS VALID");
          getch();
          exit(1);
      }
      strcpy(temp1,'\0');
      strcpy(temp2,'\0');
      strcpy(t,'\0');
      goto again1;
  }
  if(strcmp(array[n][l],"Error")==0)
  {
      printf( "\n\tERROR IN YOUR SOURCE STRING");
      getch();
      exit(1);
  }
  strcpy(tt,'\0');
  strcpy(tt,array[n][l]);
  strcpy(t3,'\0');
  f=0;
  for(c=0;c<strlen(tt);c++)
  {
     t3[c]=tt[c];
     t3[c+1]='\0';
     if(strcmp(t3,temp)==0)
     {
          f=0;
          break;
     }
     else
          f=1;
   }

   if(f==0)
   {
     strcpy(temp,'\0');
     strcpy(temp1,'\0');
     strcpy(temp2,'\0');
```

```
                strcpy(t,'\0');
                i++;
                k=0;
                goto again;
              }
              else
              {
                strcpy(temp1,'\0');
                strcpy(temp2,'\0');
                strcpy(t,'\0');
                goto again1;
              }
           }
         }
         i++;
         k++;
   }
   if(f1==0)
          printf( "\nENTERED STRING IS INVALID");
   else
         printf( "\n\n\tENTERED STRING IS VALID");
   getch();
}
```

Output:



```
LL(1)  PARSER  TABLE

NT      <id>     +        *        ;
E       Te       Error    Error    Error
e       Error    +Te      Error
T       Vt       Error    Error    Error
t       Error             *Vt
V       <id>     Error    Error    Error
ENTER THE STRING :<id>+<id>*<id>;

CHECKING VALIDATION OF THE STRING
E
Te
Vte
<id>te
<id>e
<id>+Te
<id>+Vte
<id>+<id>te
<id>+<id>*Vte
<id>+<id>*<id>te
<id>+<id>*<id>e
<id>+<id>*<id>

ENTERED STRING IS VALID
```

Viva questions:

1. What is LL(1)?

   LL(1) is a kind of top-down parser, in the name LL(1), the first L stands for scanning the input from left to right, the second L stands for producing a leftmost derivation, and the **1** stands for using one input symbol of look-ahead at each step to make parsing action decision.

2. What are the rules for FOLLOW?
   - ✓ If S is a start symbol, then FOLLOW(S) contains $.
   - ✓ If there is a production $A \rightarrow \alpha B\beta$, then everything in FIRST(β) except ε is placed in FOLLOW(B).
   - ✓ If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)

3. What are the rules for FIRST?
   - ✓ If X is terminal, then FIRST(X) is {X}.
   - ✓ If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).
   - ✓ If X is non-terminal and $X \rightarrow a\alpha$ is a production then add a to FIRST(X).

4. Write an algorithm for LL(1) table?

   For every production $A \rightarrow \alpha$ in the grammar:

   Step 1: If α can derive a string starting with *a* (i.e., for all *a* in FIRST(α) ,Table [A, *a*] = $A \rightarrow \alpha$

   Step 2: If α can derive the empty string, ε , then, for all *b* that can follow a string derived from A (i.e., for all *b* in FOLLOW (A) , Table [A,*b*] = $A \rightarrow \alpha$

Aim: To design predictive parser for the given language.

Algorithm:

Step 1: Start

Step 2: Read the input string.

Step 3: X symbol on top of stack, a current input symbol

Step 4: Stack contents and remaining input called parser configuration (initially $S on stack and complete input string)

Step 5: If X=a=$ halt and announce success

Step 6: If X=a≠$ pop X off stack advance input to next symbol

Step 7: If X is a non-terminal use M[X ,a] which contains production

Step 8: X->RHS or error replace X on stack with rhs or call error routine, respectively.

EX: X->UVW replace X with WVU(U on top) output the production (or augment parse tree)

Step 9: Stop

Source Code:

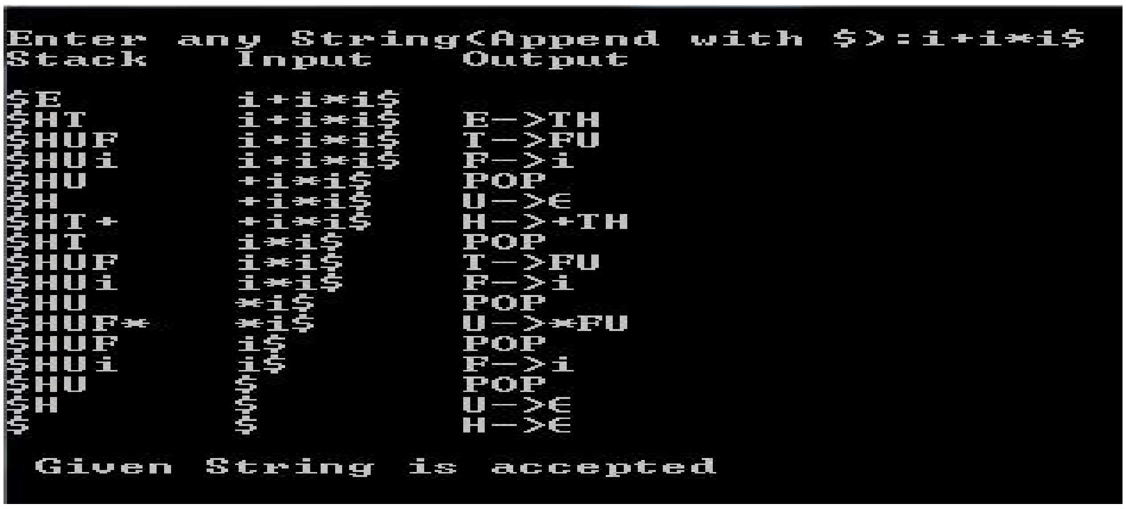
```
/*To Implement Predictive Parsing*/
#include<string.h>
#include<conio.h>
char a[10];
int top=-1,i;
void error(){
printf("Syntax Error");
}
void push(char k[]) /*Pushes The Set Of Characters on to the Stack*/
{
 for(i=0;k[i]!='\0';i++)
 {
  if(top<9)
  a[++top]=k[i];
 }
}
char TOS()       /*Returns TOP of the Stack*/
```

```
{
  return a[top];
}
void pop()       /*Pops 1 element from the Stack*/
{
  if(top>=0)
    a[top--]='\0';
}
void display()  /*Displays Elements Of Stack*/
{
  for(i=0;i<=top;i++)
    printf("%c",a[i]);
}
void display1(char p[],int m) /*Displays The Present Input String*/
{
  int l;
  printf("\t");
  for(l=m;p[l]!='\0';l++)
    printf("%c",p[l]);
}
char* stack(){
return a;
}
int main()
{
  char ip[20],r[20],st,an;
  int ir,ic,j=0,k;
  char t[5][6][10]={"$","$","TH","$","TH","$",
"+TH","$","e","e","$","e",
"$","$","FU","$","FU","$",
"e","*FU","e","e","$","e",
"$","$","(E)","$","i","$"};
  clrscr();
  printf("\nEnter any String(Append with $):");
  gets(ip);
  printf("Stack\tInput\tOutput\n\n");
  push("$E");
  display();
  printf("\t%s\n",ip);
  for(j=0;ip[j]!='\0';)
```

```
{
if(TOS()==an)
    {
  pop();
  display();
  display1(ip,j+1);
  printf("\tPOP\n");
  j++;
    }
  an=ip[j];
  st=TOS();
    if(st=='E')ir=0;
    else if(st=='H')ir=1;
    else if(st=='T')ir=2;
    else if(st=='U')ir=3;
    else if(st=='F')ir=4;
    else {
      error();
      break;
      }
    if(an=='+')ic=0;
    else if(an=='*')ic=1;
    else if(an=='(')ic=2;
    else if(an==')')ic=3;
    else if((an>='a'&&an<='z')||(an>='A'&&an<='Z')){ic=4;an='i';}
    else if(an=='$')ic=5;
    strcpy(r,strrev(t[ir][ic]));
    strrev(t[ir][ic]);
    pop();
    push(r);
    if(TOS()=='e')
    {
  pop();
  display();
  display1(ip,j);
  printf("\t%c->%c\n",st,238);
    }
    else{
    display();
    display1(ip,j);
```

```
      printf("\t%c->%s\n",st,t[ir][ic]);
      }
      if(TOS()=='$'&&an=='$')
      break;
      if(TOS()=='$'){
    error();
    break;
    }
      }
      k=strcmp(stack(),"$");
      if(k==0)
    printf("\n Given String is accepted");
    else
    printf("\n Given String is not accepted");
  return 0;
}
```

Output:



```
Enter any String(Append with $):i+i*i$
Stack     Input     Output

$E        i+i*i$
$HT       i+i*i$    E->TH
$HUF      i+i*i$    T->FU
$HUi      i+i*i$    F->i
$HU       +i*i$     POP
$H        +i*i$     U->∈
$HT+      +i*i$     H->+TH
$HT       i*i$      POP
$HUF      i*i$      T->FU
$HUi      i*i$      F->i
$HU       *i$       POP
$HUF*     *i$       U->*FU
$HUF      i$        POP
$HUi      i$        F->i
$HU       $         POP
$H        $         U->∈
$         $         H->∈

 Given String is accepted
```

Viva questions:

1. What are steps require to constructing predictive parser?
    - ✓ Elimination of left recursion, left factoring and ambiguous grammar.
    - ✓ Construct FIRST() and FOLLOW() for all non-terminals.
    - ✓ Construct predictive parsing table.
    - ✓ Parse the given input string using stack and parsing table.

2. What are the steps for predictive parser table?

   For each production A → α of the grammar, do steps 2 and 3.

   Step 1: For each terminal a in FIRST(α), add A → α to M[A, a].

   Step 2: If ε is in FIRST(α), add A → α to M[A, b] for each terminal b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A) , add A → α to M[A, $].

   Step 3: Make each undefined entry of M be error.

3. What is left recursion and left factoring?

   Left factoring is removing the common left factor that appears in two productions of the same non-terminal. It is done to avoid back-tracing by the parser. Suppose the parser has a look-ahead, consider this example-

   A->qB|qC

   where A,B,C are non-terminals and q is a sentence. In this case, the parser will be confused as to which of the two productions to choose and it might have to back-trace. After left factoring, the grammar is converted to-

   A -> qD
   D -> B | C

   In this case, a parser with a look-ahead will always choose the right production.

   Left recursion is a case when the left-most non-terminal in a production of a non-terminal is the non-terminal itself (direct left recursion) or through some other non-terminal definitions, rewrites to the non-terminal again(indirect left recursion). Consider these examples:

   (1) A -> A**α**|**β** (direct)
   (2) A -> Bq
   B -> Ar (indirect)

   While designing a top down-parser, if the left recursion exist in the grammar then the parser falls in an infinite loop, here because A is trying to match A itself, which is not possible. We can eliminate the above left recursion by rewriting the offending production. As:

   A -> βA'
   A' -> αA' | epsilon

Aim: To implementation of shift reduce parsing algorithm.

Algorithm:

Step 1: Start

Step2: Initial State: the stack consists of the single state, s0; ip points to the first character in w.

Step 3: For top-of-stack symbol, s, and next input symbol, a case action of T[s,a]

Step 4: shift x: (x is a STATE number) push a, then x on the top of the stack and advance ip to point to the next input symbol.

Step 5: reduce y: (y is a PRODUCTION number) Assume that the production is of the form A ==> beta pop 2 * |beta| symbols of the stack.

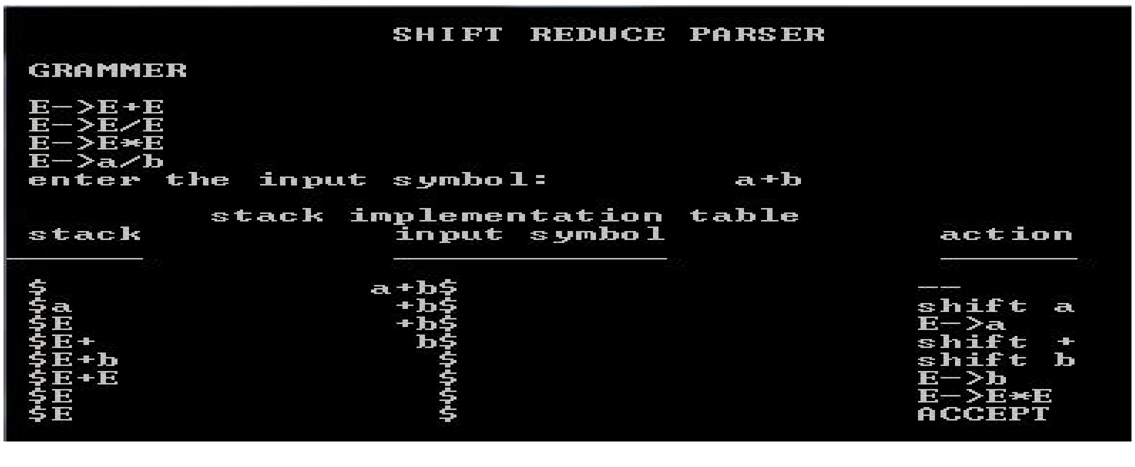
Step 6: At this point the top of the stack should be a state number, say s'. push A, then goto of T[s',A] (a state number) on the top of the stack.

Step 7: Stop

Source Code:

```
#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
```

Viva questions:

1. What type parser it is?
   It is Bottom-Up Parser.

2. What is Bottom-Up parser?
   The parser which is used to constructing a parse tree from an input symbols and going towards the root symbol or starting symbol of a grammar is called bottom-up parser. It is quite opposite to top-down parser

3. What is SHIFT?
   The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.

4. What is REDUCE?
   When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

5. What are the two types of conflicts?
   - ✓ shift reduce conflict
   - ✓ reduce reduce conflict

Aim: To design a LALR bottom up parser for the given language.

Algorithm :

STEP 1: Represent Ii by its CLOSURE, those items that are either the initial
Item [S’← .S; EOF] or do not have the . at the left end of the RHS.

STEP 2: Compute shift, reduce, and go to actions for the state derived from Ii directly from
CLOSURE (Ii)

Source Code:

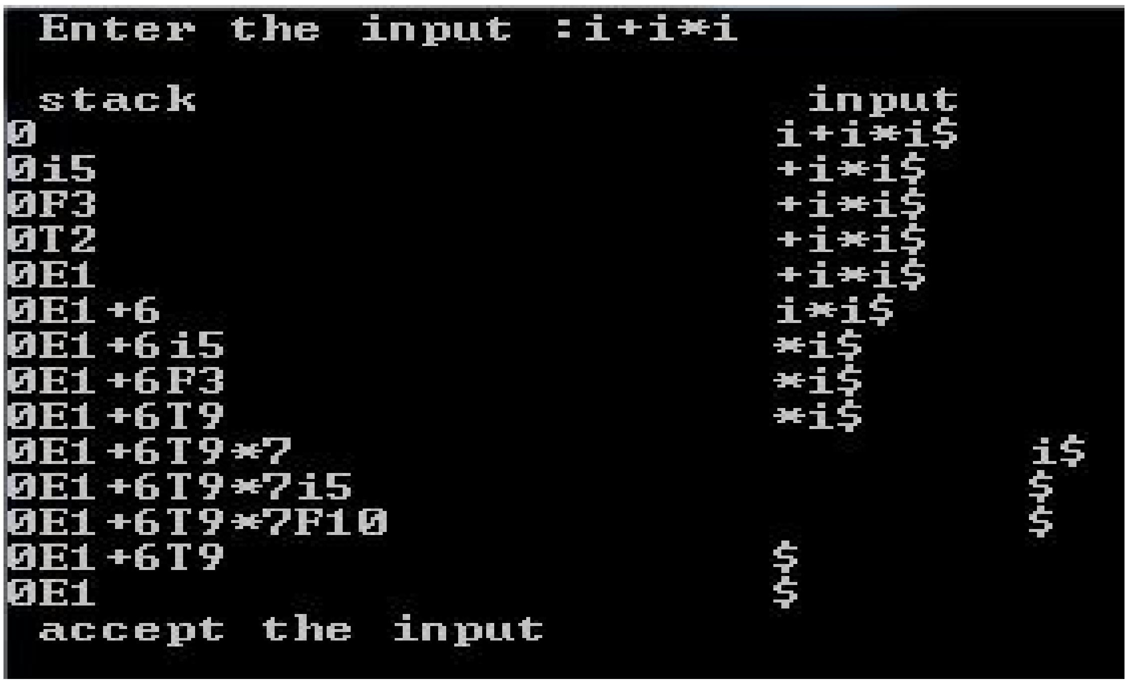
```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>

void push(char *,int *,char);
char stacktop(char *);
void isproduct(char,char);
int ister(char);
int isnter(char);
int isstate(char);
void error();
void isreduce(char,char);
char pop(char *,int *);
void printt(char *,int *,char [],int);
void rep(char [],int);
struct action
{
char row[6][5];
};
const struct action A[12]={{"sf","emp","emp","se","emp","emp"},
                          {"emp","sg","emp","emp","emp","acc"},
                          {"emp","rc","sh","emp","rc","rc"},
                          {"emp","re","re","emp","re","re"},
                          {"sf","emp","emp","se","emp","emp"},
                          {"emp","rg","rg","emp","rg","rg"},
                          {"sf","emp","emp","se","emp","emp"},
                          {"sf","emp","emp","se","emp","emp"},
                          {"emp","sg","emp","emp","sl","emp"},
                          {"emp","rb","sh","emp","rb","rb"},
                                    {"emp","rb","rd","emp","rd","rd"},
                                    {"emp","rf","rf","emp","rf","rf"}
                                   };
struct gotol
{
char r[3][4];
};
const struct gotol G[12]={{"b","c","d"},
                                   {"emp","emp","emp"},
                                   {"emp","emp","emp"},
                                   {"emp","emp","emp"},
                                   {"i","c","d"},
                                   {"emp","emp","emp"},
                                   {"emp","j","d"},
                                   {"emp","emp","k"},
                                   {"emp","emp","emp"},
                                   {"emp","emp","emp"},
                                  };
char ter[6]={'i','+','*',')','(','$'};
char nter[3]={'E','T','F'};
char states[12]={'a','b','c','d','e','f','g','h','m','j','k','l'};
char stack[100];
int top=-1;
char temp[10];
struct grammar
{
char left;
char right[5];
};
const struct grammar rl[6]={{'E',"e+T"},
                                     {'E',"T"},
                                     {'T',"T*F"},
                                     {'T',"F"},
                                     {'F',"(E)"},
                                     {'F',"i"},
                                    };

void main()
{
char inp[80],x,p,dl[80],y,bl='a';
```

```
int i=0,j,k,l,n,m,c,len;
clrscr();
printf(" Enter the input :");
scanf("%s",inp);
len=strlen(inp);
inp[len]='$';
inp[len+1]='\0';
push(stack,&top,bl);
printf("\n stack\t\t\t input");
printt(stack,&top,inp,i);
do
{
x=inp[i];
p=stacktop(stack);
isproduct(x,p);
if(strcmp(temp,"emp")==0)
error();
else if(strcmp(temp,"acc")==0)
break;
else
{
if(temp[0]=='s')
{
push(stack,&top,inp[i]);
push(stack,&top,temp[1]);
i++;
}
else
{
if(temp[0]=='r')
{
j=isstate(temp[1]);
strcpy(temp,rl[j-2].right);
dl[0]=rl[j-2].left;
dl[1]='\0';
n=strlen(temp);
for(k=0;k<2*n;k++)
pop(stack,&top);
for(m=0;dl[m]!='\0';m++)
push(stack,&top,dl[m]);
l=top;
y=stack[l-1];
isreduce(y,dl[0]);
for(m=0;temp[m]!='\0';m++)
push(stack,&top,temp[m]);
}
}
}
printt(stack,&top,inp,i);
}while(inp[i]!='\0');
if(strcmp(temp,"acc")==0)
printf("\n accept the input ");
else
printf("\n do not accept the input ");
getch();
}
void push(char *s,int *sp,char item)
{
if(*sp==100)
printf(" stack is full "); else
{
*sp=*sp+1;
s[*sp]=item;
}
}
char stacktop(char *s)
{
char i;
i=s[top];
return i;
}
void isproduct(char x,char p)
{
int k,l;
k=ister(x);
l=isstate(p);
strcpy(temp,A[l-1].row[k-1]);
}
int ister(char x)
{
```

```
int i;
for(i=0;i<6;i++)
if(x==ter[i])
return i+1;
return 0;
}
int isnter(char x)
{
int i;
for(i=0;i<3;i++)
if(x==nter[i])
return i+1;
return 0;
}
int isstate(char p)
{
int i;
for(i=0;i<12;i++)
if(p==states[i])
return i+1;
return 0;
}
void error()
{
printf(" error in the input ");
exit(0);
}
void isreduce(char x,char p)
{
int k,l;
k=isstate(x);
l=isnter(p);
strcpy(temp,G[k-1].r[l-1]);
}
char pop(char *s,int *sp)
{
char item;
if(*sp==-1)
printf(" stack is empty ");
else
```

```
{
item=s[*sp];
*sp=*sp-1;
}
return item;
}
void printt(char *t,int *p,char inp[],int i)
{
int r;
printf("\n");
for(r=0;r<=*p;r++)
rep(t,r);
printf("\t\t\t");
for(r=i;inp[r]!='\0';r++)
printf("%c",inp[r]);
}
void rep(char t[],int r)
{
char c;
c=t[r];
switch(c)
{
case 'a':printf("0");
            break;
case 'b':printf("1");
            break;
case 'c':printf("2");
            break;
case 'd':printf("3");
            break;
case 'e':printf("4");
            break;
case 'f':printf("5");
            break;
case 'g':printf("6");
            break;
case 'h':printf("7");
            break;
case 'm':printf("8");
            break;
case 'j':printf("9");
            break;
case 'k':printf("10");
            break;
case 'l':printf("11");
            break;
default :printf("%c",t[r]);
            break;
}
}
```

Output:



```
 Enter the input :i+i*i

 stack                          input
0                              i+i*i$
0i5                            +i*i$
0F3                            +i*i$
0T2                            +i*i$
0E1                            +i*i$
0E1+6                          i*i$
0E1+6i5                        *i$
0E1+6F3                        *i$
0E1+6T9                        *i$
0E1+6T9*7                              i$
0E1+6T9*7i5                            $
0E1+6T9*7F10                           $
0E1+6T9                        $
0E1                            $
 accept the input
```

Viva questions:

1. What is full form of LALR?
   look ahead LR parser

2. What is CLOSURE?

   Say I is a set of items and one of these items is $A \rightarrow \alpha \cdot B\beta$. This item represents the parser having seen α and records that the parser might soon see the remainder of the RHS. For that to happend the parser must first see a string derivable from B. Now consider any production starting with B, say $B \rightarrow \gamma$. If the parser is to make progress on $A \rightarrow \alpha \cdot B\beta$, it will need to be making progress on one such $B \rightarrow \cdot\gamma$. Hence we want to add all the latter productions to any state that contains the former. We formalize this into the notion of closure.

   For any set of items I, *CLOSURE(I)* is formed as follows.

   - ✓ Initialize CLOSURE(I) = I
   - ✓ If $A \rightarrow \alpha \cdot B\ \beta$ is in CLOSURE(I) and $B \rightarrow \gamma$ is a production, then add $B \rightarrow \cdot\ \gamma$ to the closure and repeat.

3. What are Kernel and non-kernel Items?

   Kernel Items: Includes initial item S' --> .S and all items in which dot does not appear at the left most position.

   Non-kernel Items: All other items which have dots at the leftmost position. Except initial item S' --> .S

4. What is an augmented grammar?

   If G is a grammar with start symbol S then G' is the augmented grammar for G, G with a new start symbol S' and with production S'->S

Aim: To implement the lexical analyzer using JLex, flex or lex or other lexical analyzer generating tools.

Algorithm:

```
Step 1: WHILE there is more input
            InputChar := GetChar
            State := Table[0, InputChar]
Step 2: WHILE State != Blank
            InputChar := GetChar
            State := Table[State, InputChar]
Step 3: ENDWHILE
Step 4: Retract
Step 5: Accept
Step 6: Return token = (Class, Value)
Step 7: ENDWHILE
```

Source Code:

lexpr.l

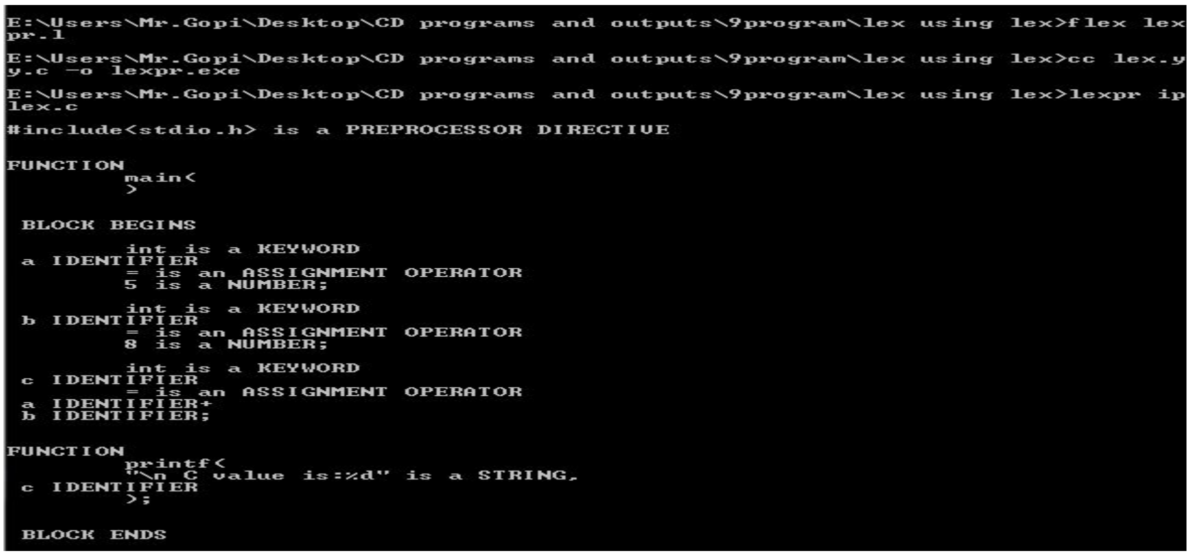
```
%{
/* program to recognize a c program */
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* { printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);}
int |
float |
char |
double |
while |
for |
do |
if  |
break |
continue |
void |
switch |
case |
long |
struct |
const |
typedef |
return |
else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

Input:

iplex.c

```
#include<stdio.h>
main()
{
int a=5;
int b=8;
int c=a+b;
printf("\n C value is:%d",c);
}
```

Output:



```
E:\Users\Mr.Gopi\Desktop\CD programs and outputs\9program\lex using lex>flex lex
pr.l

E:\Users\Mr.Gopi\Desktop\CD programs and outputs\9program\lex using lex>cc lex.y
y.c -o lexpr.exe

E:\Users\Mr.Gopi\Desktop\CD programs and outputs\9program\lex using lex>lexpr ip
lex.c

#include<stdio.h> is a PREPROCESSOR DIRECTIVE

FUNCTION
        main(
        )

 BLOCK BEGINS

        int is a KEYWORD
 a IDENTIFIER
        = is an ASSIGNMENT OPERATOR
        5 is a NUMBER;

        int is a KEYWORD
 b IDENTIFIER
        = is an ASSIGNMENT OPERATOR
        8 is a NUMBER;

        int is a KEYWORD
 c IDENTIFIER
        = is an ASSIGNMENT OPERATOR
 a IDENTIFIER+
 b IDENTIFIER;

FUNCTION
        printf(
        "\n C value is:%d" is a STRING,
 c IDENTIFIER
        );

 BLOCK ENDS
```

Viva questions:

1. What is A lexical analyzer?

   A lexical analyzer breaks an input stream of characters into tokens. Writing lexical analyzers by hand can be a tedious process, so software tools have been developed to easy this task.

2. Lex is a lexical analyzer generator for what?

   The UNIX operating system, targeted to the C programming language.

3. How many sections a Lex file is organized?

   A Lex file is organized into three sections

4. What are the sections in lex?

   The structure of a Lex file is intentionally similar to that of a yacc file; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

   Definition section
   %%
   Rules section
   %%
   C code section

   - ✓ The **definition** section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
   - ✓ The **rules** section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.
   - ✓ The **C code** section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

Aim: To write a program to perform loops unrolling.

Description:

Loop unrolling transforms a loop into a sequence of statements. It is a parallelizing and optimizing compiler technique where loop unrolling is used to eliminate loop overhead to test loop control flow such as loop index values and termination conditions technique was also used to expose instruction-level parallelism.

Algorithm :

Step1: start

Step2: declare n

Step3: enter n value

Step4: loop rolling display countbit1 or move to next step 5

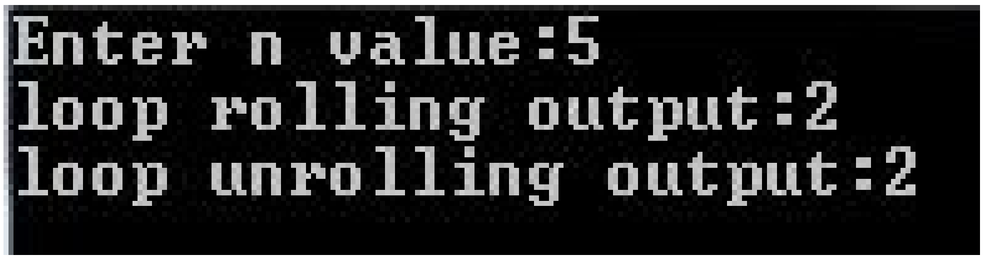
Step5: loop unrolling display countbit2

Step6: stop

Source Code:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n;
clrscr();
printf("Enter n value:");
scanf("%d",&n);
printf("loop rolling output:%d\n",countbit1(n));
printf("loop unrolling output:%d\n",countbit2(n));
getch();
}
int countbit1(unsigned int n)
{
int bits=0;
while(n!=0)
{
if(n&1)
bits++;
n>>=1;
}
return bits;
}
int countbit2(unsigned int n)
{
int bits=0;
while(n!=0)
{
if(n&1)
bits++;
if(n&2)
bits++;
if(n&4)
bits++;
if(n&8)
bits++;
n>>=4;
}
return bits;
}
```

Output:



```
Enter n value:5
loop rolling output:2
loop unrolling output:2
```

Viva questions:

1. Define loop unrolling?

   Loop unrolling, also known as loop unwinding, is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size, which is an approach known as the space-time tradeoff. The transformation can be undertaken manually by the programmer or by an optimizing compiler.

2. Example of loop unrolling?
   - ✓ an array sum loop
   - ✓ a dot product loop
   - ✓ a row operation loop

3. Write loop unrolling code in c for array sum?

   The following C code will compute the sum of the array entries

```
int length = ARRAY_SIZE;
int limit = length-4;
for (j=0; j < limit; j+=5)
{
sum += array[j] + array[j+1] + array[j+2] + array[j+3] + array[j+4];
}
for(; j < length; j++)
{
sum += array[j];
}
```

Aim: To convert the BNF rules into YACC form and write code to generate abstract Syntax tree.

Algorithm :

Step 1: To specify the syntax of a language: CFG and BNF

Step 2: ex: if-else statement in C has the form of statement →if (expression) statement else statement

Step 3: An alphabet of a language is a set of symbols.

Step 4: ex:{0,1} for a binary number system(language)={0,1,100,101,...}

Step 5:{a,b,c} for language={a,b,c, ac,abcc..}

Step 6: {if,(,),else ...} for a if statements={if(a==1)goto10, if--}

Step7: A string over an alphabet
is a sequence of zero or more symbols from the alphabet.

Step 8: Ex: 0,1,10,00,11,111,0202 ... strings for a alphabet {0,1}

Step 9: Null string is a string which does not have any symbol of alphabet Language.
a subset of all the strings over a given alphabet.

Step10: Alphabets Ai Languages Li for Ai
A0={0,1} L0={0,1,100,101,...} A1={a,b,c} L1={a,b,c, ac, abcc..}
A2={all of C tokens} L2= {all sentences of C program }

Source Code:

yacpr.l

```
%option noyywrap
%{
#include"y.tab.h"
#include<stdio.h>
#include<string.h>
int LineNo=1;
%}
identifier [a-zA-Z][_a-zA-Z0-9]*
number [0-9]+|([0-9]*\.[0-9]+)
%%
main\(\) return MAIN;
```

```
if return IF;
else return ELSE;
while return WHILE;
int |
char |
float return TYPE;
{identifier} {strcpy(yylval.var,yytext);
return VAR;}
{number} {strcpy(yylval.var,yytext);
return NUM;}
\< |
\> |
\>= |
\<= |
== {strcpy(yylval.var,yytext);
return RELOP;}
[ \t] ;
\n LineNo++;
. return yytext[0];
%%
```

yacpr.y

```
%{
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
struct quad
{
        char op[5];
        char arg1[10];
        char arg2[10];
        char result[10];
}QUAD[30];
struct stack
{
        int items[100];
        int top;
}stk;
int Index=0,tIndex=0,StNo,Ind,tInd;
extern int LineNo;
%}
%union
{
        char var[10];
}
%token <var> NUM VAR RELOP
%token MAIN IF ELSE WHILE TYPE
%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP
%left '-' '+'
%left '*' '/'
%%
PROGRAM : MAIN BLOCK
;
BLOCK: '{' CODE '}'
;
CODE: BLOCK
| STATEMENT CODE
| STATEMENT
;
STATEMENT: DESCT ';'
| ASSIGNMENT ';'
| CONDST
| WHILEST
;
DESCT: TYPE VARLIST
;
VARLIST: VAR ',' VARLIST
| VAR
;
ASSIGNMENT: VAR '=' EXPR{
strcpy(QUAD[Index].op,"=");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,$1);
strcpy($$,QUAD[Index++].result);
}
;
EXPR: EXPR '+' EXPR {AddQuadruple("+",$1,$3,$$);}
| EXPR '-' EXPR {AddQuadruple("-",$1,$3,$$);}
| EXPR '*' EXPR {AddQuadruple("*",$1,$3,$$);}
| EXPR '/' EXPR {AddQuadruple("/",$1,$3,$$);}
| '-' EXPR {AddQuadruple("UMIN",$2,"",$$);}
| '(' EXPR ')' {strcpy($$,$2);}
| VAR
| NUM
;
CONDST: IFST{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
| IFST ELSEST
;
IFST: IF '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
};
ELSEST: ELSE{
tInd=pop();
Ind=pop();
push(tInd);
sprintf(QUAD[Ind].result,"%d",Index);
}
BLOCK{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
};
```

```
CONDITION: VAR RELOP VAR {AddQuadruple($2,$1,$3,$$);
StNo=Index-1;
}
| VAR
| NUM
;
WHILEST: WHILELOOP{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",StNo);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
;
WHILELOOP: WHILE '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
;
%%
extern FILE *yyin;
int main(int argc,char *argv[])
{
FILE *fp;
int i;
if(argc>1)
{
fp=fopen(argv[1],"r");
if(!fp)
{
printf("\n File not found");
exit(0);
}
yyin=fp;
}
yyparse();
printf("\n\n\t\t ----------------------------------""\n\t\t Pos  Operator  Arg1  Arg2  Result""\n\t\t---------
--------------------------");
for(i=0;i<Index;i++)
{
printf("\n\t\t%d\t%s\t%s\t%s\t%s",i,QUAD[i].op,QUAD[i].arg1,QUAD[i].arg2,QUAD[i].result);
}
printf("\n\t\t ----------------------------------");
printf("\n\n");
return 0;
}
push(int data)
{
stk.top++;
if(stk.top==100)
{
printf("\n Stack overflow\n");
exit(0);
}
stk.items[stk.top]=data;
}
int pop()
{
int data;
if(stk.top==-1)
{
printf("\n Stack underflow\n");
exit(0);
}
data=stk.items[stk.top--];
return data;
}
AddQuadruple(char op[5],char arg1[10],char arg2[10],char result[10])
{
```

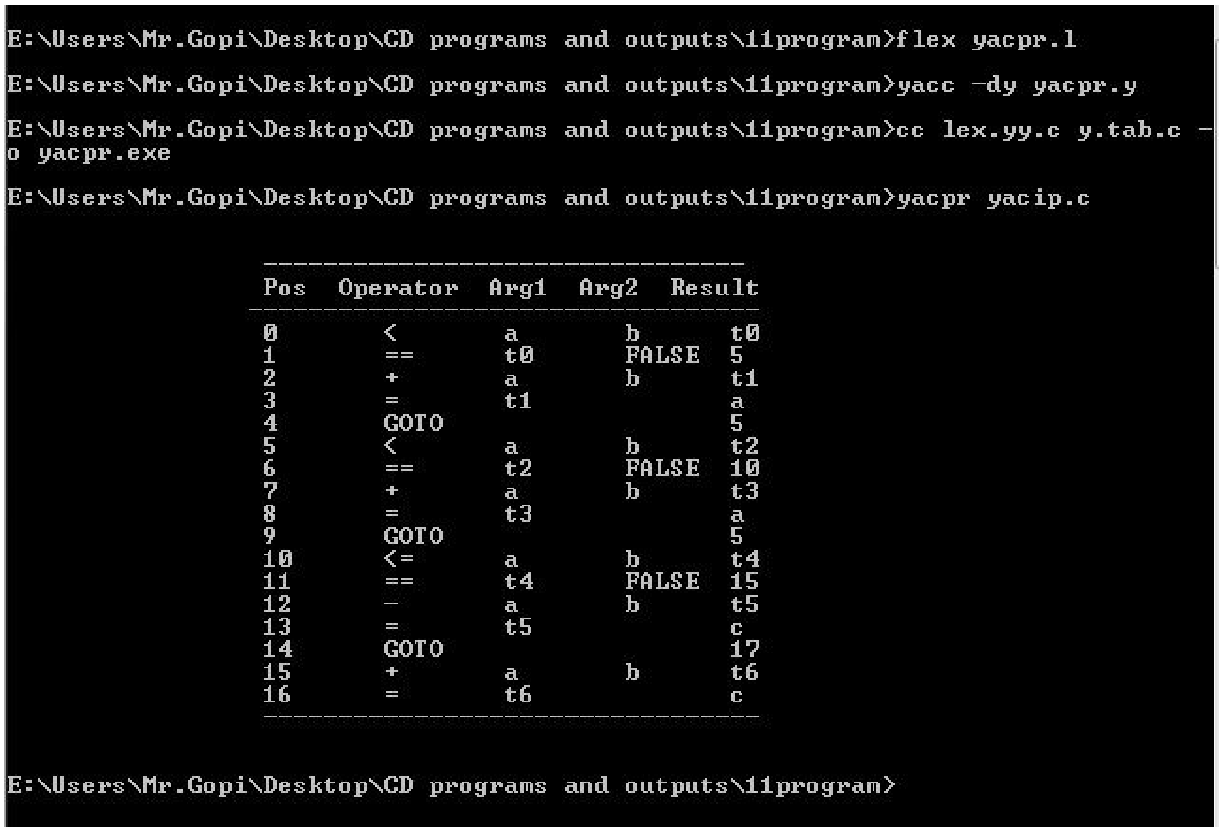
```
strcpy(QUAD[Index].op,op);
strcpy(QUAD[Index].arg1,arg1);
strcpy(QUAD[Index].arg2,arg2);
sprintf(QUAD[Index].result,"t%d",tIndex++);
strcpy(result,QUAD[Index++].result);
}
yyerror()
{
printf("\n Error on line no:%d",LineNo);
}
```

Input:

yacip.c

```
main()
{
int a,b,c;
if(a<b)
{
a=a+b;
}
while(a<b)
{
a=a+b;
}
if(a<=b)
{
c=a-b;
}
else
{
c=a+b;
}
}
```

Output:



```
E:\Users\Mr.Gopi\Desktop\CD programs and outputs\11program>flex yacpr.l

E:\Users\Mr.Gopi\Desktop\CD programs and outputs\11program>yacc -dy yacpr.y

E:\Users\Mr.Gopi\Desktop\CD programs and outputs\11program>cc lex.yy.c y.tab.c -
o yacpr.exe

E:\Users\Mr.Gopi\Desktop\CD programs and outputs\11program>yacpr yacip.c


                 --------------------------------
                 Pos   Operator  Arg1   Arg2   Result
                -----------------------------------
                 0       <        a      b      t0
                 1       ==       t0     FALSE  5
                 2       +        a      b      t1
                 3       =        t1            a
                 4       GOTO                   5
                 5       <        a      b      t2
                 6       ==       t2     FALSE  10
                 7       +        a      b      t3
                 8       =        t3            a
                 9       GOTO                   5
                 10      <=       a      b      t4
                 11      ==       t4     FALSE  15
                 12      -        a      b      t5
                 13      =        t5            c
                 14      GOTO                   17
                 15      +        a      b      t6
                 16      =        t6            c
                 -----------------------------------


E:\Users\Mr.Gopi\Desktop\CD programs and outputs\11program>
```

Viva questions:

1. What is the full form of BNF?
   BNF stands for either Backus-Naur Form or Backus Normal Form

2. What is structure for BNF?
   BNF is represented using the following symbols:
   - ✓ ::= 'is defined as'
   - ✓ | 'or'
   - ✓ <> ‘category names’

   The way that these symbols are laid out are as such:
   <Parent Expression> ::= <Child Expression 1> | <Child Expression 2>

3. What is the BNF Converter?
   The BNF Converter is a compiler construction tool generating a compiler front-end from a Labelle BNF grammar

4. What is full form of YACC?
   Yacc: Yet Another Compiler-Compiler

Aim: To write a program for constant propagation.

Description:

The algorithm we shall present basically tries to find for every statement in the program a mapping between variables, and values of N T ∪⊥ { , } . If a variable is mapped to a constant number, that number is the variables value in that statement on every execution. If a variable is mapped to T (top), its value in the statement is not known to be constant, and in the variable is mapped to ⊥ (bottom), its value is not initialized on every execution, or the statement is unreachable. The algorithm for assigning the mappings to the statements is an iterative algorithm, that traverses the control flow graph of the algorithm, and updates each mapping according to the mapping of the previous statement, and the functionality of the statement. The traversal is iterative, because non-trivial programs have circles in their control flow graphs, and it ends when a "fixed-point" is reached – i.e., further iterations don't change the mappings.

Algorithm:

Step 1: Start

Step 2: Declare a, b, c

Step 3: Initialize a←value and b←value

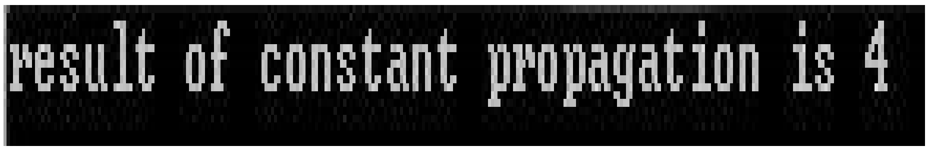
Step 4: Display result of propagation

Step 5: Stop

Source Code:

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
printf("result of constant propagation is %d",constantpropagation()); getch();
}
int constantpropagation()
{
int a=30;
int b=3;
int c;
c=b*4;
if(c>10)
{
c=c-10;
}
return c*2;
}
```

Output:



```
result of constant propagation is 4
```

Viva questions:

1. What is a compiler?

   Compiler is a translator which scans the entire source code and translates it as a whole into machine code [Target code]. It generates object code, hence it requires more memory. Here the code execution fast

2. What is an interpreter?

   Interpreter is also a translator which is also used to translate a source code at a time only single instruction. It doesn't generate any intermediate object code, so memory is saved. Here the execution speed is less