# Program-1

**Aim-** Write a program to identify whether a given line is a comment or not. If it is not a comment then recognize the strings under ‘a*’, ‘a*b+’, ‘abb’.

**Compiler used-** Online GDB compiler (C++)

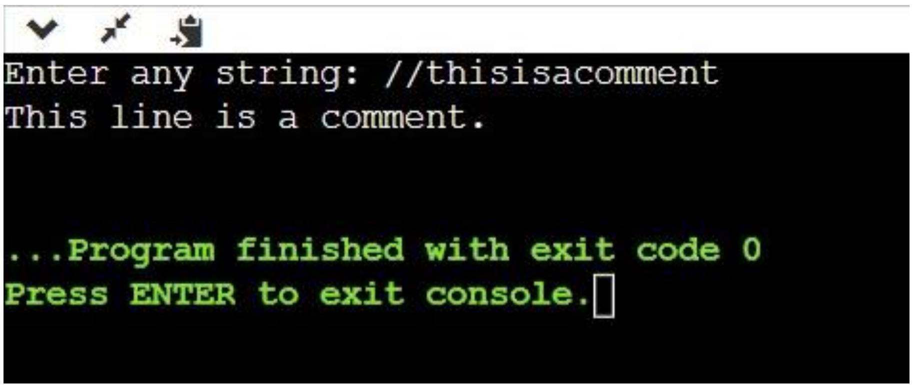
**Code-**

```
#include<iostream>
#include<string>
using namespace std;
int comment(string txt)
{
   int n=txt.size();
   int flag=0;
   if (txt[0]=='/' && txt[1]=='/')
      flag=1;
   if (txt[0]=='/' && txt[1]=='*' && txt[n-2]=='*' && txt[n-1]=='/')
      flag=1;
   return flag;
}
void check(string txt)
{
   int i=0,state=1;
   char ch;
   while(txt[i]!='\0')
   {
      switch(state)
      {
         case 1: ch=txt[i];
               if(ch=='a')
                  state=2;
               else if(ch=='b')
```

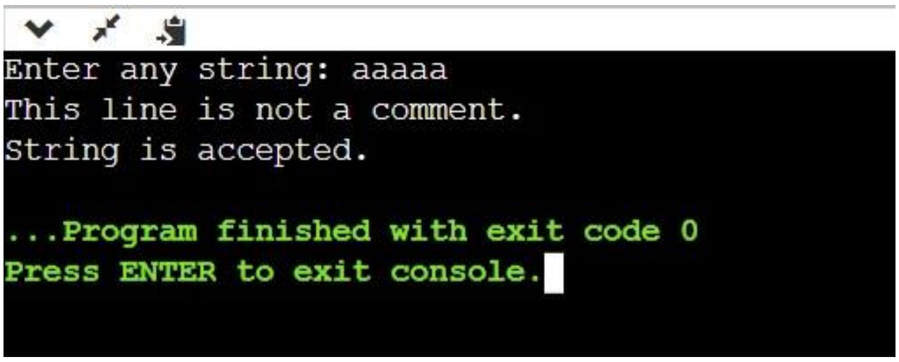
```
                state=3;
              else
                state=4;
              break;
          case 2: ch=txt[i];
              if(ch=='a')
                state=2;
              else if(ch=='b')
                state=3;
              else
                state=4;
              break;
          case 3: ch=txt[i];
              if(ch=='a')
                state=4;
              else if(ch=='b')
                state=3;
              else
                state=4;
              break;
          case 4: state=4;
              break;
        }
        i++;
    }
    if(state==2||state==3)
    cout<<"String is accepted.";
    if(state==4)
    cout<<"String is not accepted.";
}
```

```
int main()
{
    string txt;
    cout<<"Enter any string: ";
    cin>>txt;
    int flag=comment(txt);
    if(flag==1)
    {
        cout<<"This line is a comment."<<endl;
    }
    else
    {
        cout<<"This line is not a comment."<<endl;
        check(txt);
    }
    return 0;
}
```

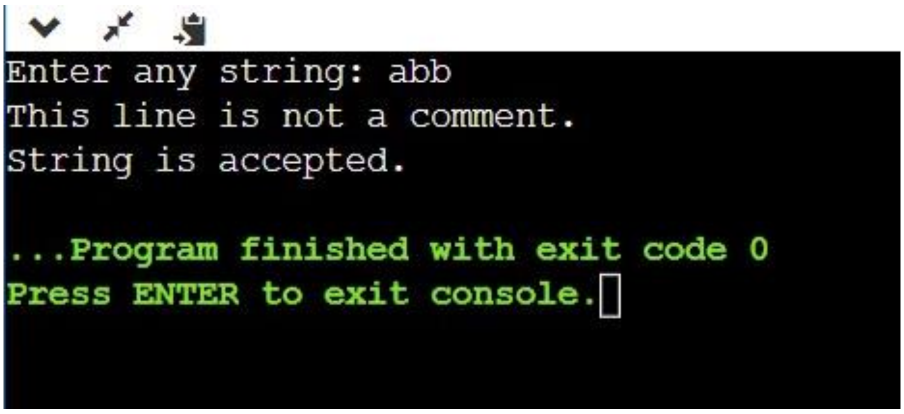
**Output-**



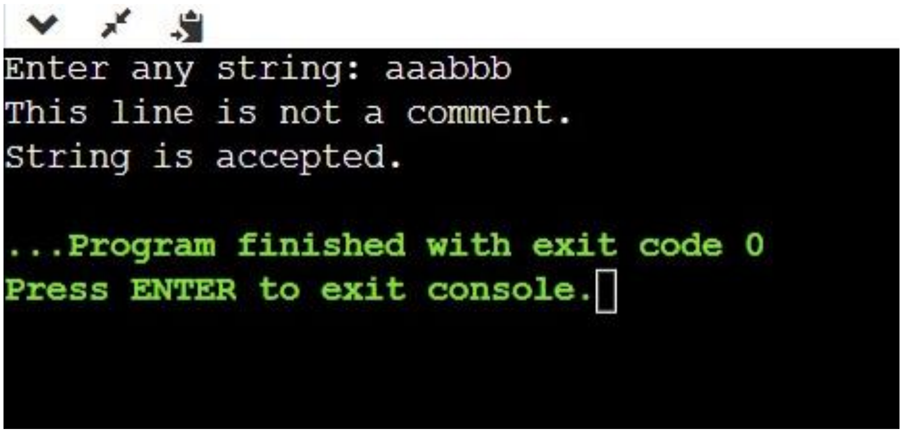
```
Enter any string: //thisisacomment
This line is a comment.


...Program finished with exit code 0
Press ENTER to exit console.
```



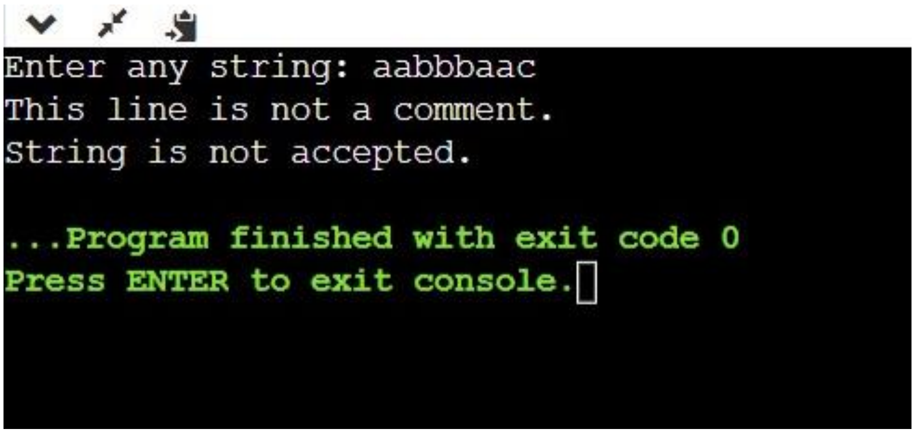
```
Enter any string: aaaaa
This line is not a comment.
String is accepted.

...Program finished with exit code 0
Press ENTER to exit console.
```



```
Enter any string: abb
This line is not a comment.
String is accepted.

...Program finished with exit code 0
Press ENTER to exit console.
```



```
Enter any string: aaabbb
This line is not a comment.
String is accepted.

...Program finished with exit code 0
Press ENTER to exit console.
```



```
Enter any string: aabbbaac
This line is not a comment.
String is not accepted.

...Program finished with exit code 0
Press ENTER to exit console.
```

## Program-2(a)

**Aim-** Write a program to convert the given infix expression to postfix expression.

**Compiler Used-** Online GDB compiler

**Code-**

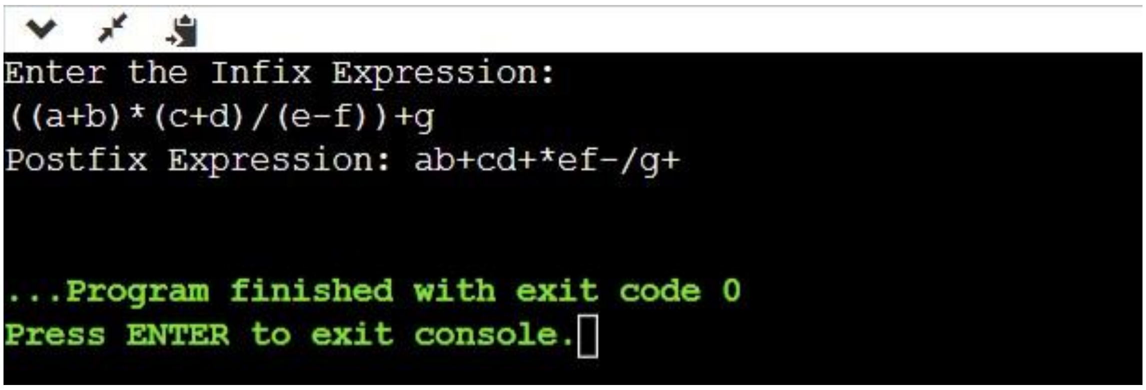
```
#include<iostream>
#include<stack>
#include<algorithm>
using namespace std;

bool isOperator(char c)
{
   if(c=='+'||c=='-'||c=='*'||c=='/'){
      return true;
   }
   else{
      return false;
   }
}

int precedence(char c)
{
   if(c=='*'||c=='/')
      return 2;
   else if(c=='+'||c=='-')
      return 1;
   else
      return -1;
}

string InfixToPostfix(stack<char> s,string infix)
```

```
{
   string postfix;
   for(int i=0;i<infix.length();i++)
     {
               if((infix[i]>='a'&&infix[i]<='z')||(infix[i]>='A'&&infix[i]<='Z')||(infix[i]>=0 &&
infix[i]<=9))
               {
                       postfix+=infix[i];
               }
               else if(infix[i] == '(')
               {
                       s.push(infix[i]);
               }
               else if(infix[i] == ')')
               {
                       while((s.top()!='(') && (!s.empty()))
                       {
                               postfix+=s.top();
                               s.pop();
                       }
                       if(s.top()=='(')
                       {
                               s.pop();
                       }
               }
               else if(isOperator(infix[i]))
               {
                       if(s.empty())
                       {
                               s.push(infix[i]);
                       }
```

```
                        else
                        {
                                    if(precedence(infix[i])>precedence(s.top()))
                                    {
                                                s.push(infix[i]);
                                    }
                                    else
                                    {
            while((!s.empty())&&(precedence(infix[i])<=precedence(s.top())))
                                                {
                                                            postfix+=s.top();
                                                            s.pop();
                                                }
                                                s.push(infix[i]);
                                    }
                        }
            }
    }
    while(!s.empty())
    {
            postfix+=s.top();
            s.pop();
    }
    return postfix;
}

int main()
{
    string infix,postfix;
    cout<<"Enter the Infix Expression: "<<endl;
```

```
        cin>>infix;

        stack<char> stack;

        postfix=InfixToPostfix(stack,infix);

        cout<<"Postfix Expression: "<<postfix<<endl;

        return 0;

    }
```

**Output-**



```
Enter the Infix Expression:
((a+b)*(c+d)/(e-f))+g
Postfix Expression: ab+cd+*ef-/g+


...Program finished with exit code 0
Press ENTER to exit console.
```

## Program-2(b)

**Aim-** Write a program to convert the given infix expression to prefix expression.

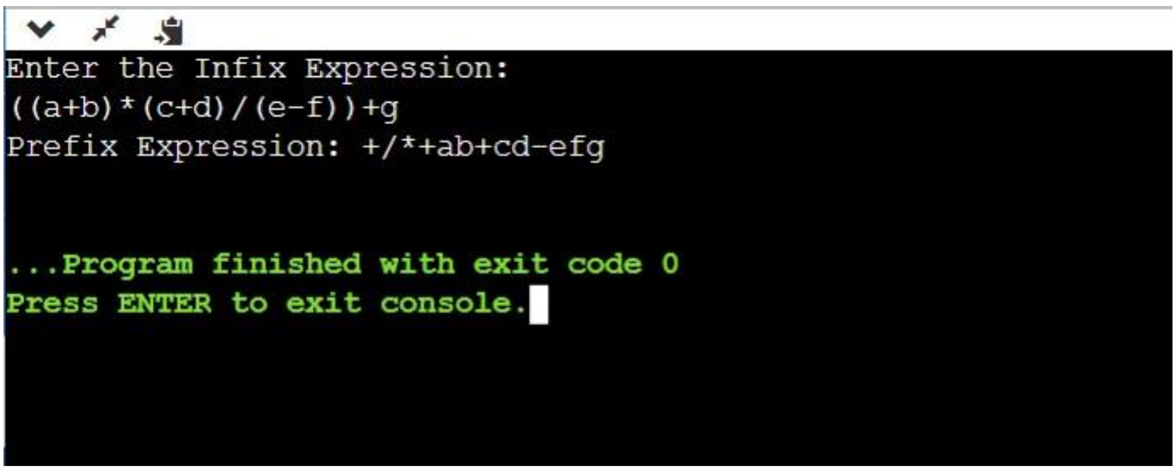
**Compiler Used-** Online GDB compiler

**Code-**

```
#include<iostream>

#include<stack>

#include<algorithm>

using namespace std;


bool isOperator(char c)

{

  if(c=='+'||c=='-'||c=='*'||c=='/'){

    return true;
    }
    else{
        return false;
    }
}

int precedence(char c)
{
    if(c=='*'||c=='/')
        return 2;
    else if(c=='+'||c=='-')
        return 1;
    else
        return -1;
}

string InfixToPrefix(stack<char> s, string infix)
{
    string prefix;
    reverse(infix.begin(), infix.end());
    for (int i=0;i<infix.length();i++){
        if (infix[i] == '(') {
            infix[i] = ')';
        }
        else if (infix[i] == ')') {
            infix[i] = '(';
        }
    }
    for (int i=0;i<infix.length();i++) {
        if((infix[i]>='a'&&infix[i]<='z')||(infix[i]>='A'&&infix[i]<='Z')||(infix[i]>=0 &&
infix[i]<=9)){
    prefix += infix[i];
}
else if (infix[i] == '(') {
    s.push(infix[i]);
}
else if (infix[i] == ')') {
    while ((s.top() != '(') && (!s.empty())) {
        prefix += s.top();
        s.pop();
    }
    if (s.top() == '(') {
        s.pop();
    }
}
else if (isOperator(infix[i])) {
    if (s.empty()) {
        s.push(infix[i]);
    }
    else {
        if (precedence(infix[i])>=precedence(s.top())) {
            s.push(infix[i]);
        }
        else {
            while ((!s.empty()) && (precedence(infix[i]) < precedence(s.top()))) {
                prefix += s.top();
                s.pop();
            }
            s.push(infix[i]);
        }
    }
```

```
        }
    }
    while(!s.empty()){
        prefix+=s.top();
        s.pop();
    }
    reverse(prefix.begin(),prefix.end());
    return prefix;
}

int main()
{
    string infix,prefix;
    cout<<"Enter the Infix Expression: "<<endl;
    cin>>infix;
    stack<char> stack;
    prefix=InfixToPrefix(stack,infix);
    cout<<"Prefix Expression: "<<prefix<<endl;
    return 0;
}
```

**Output-**



```
Enter the Infix Expression:
((a+b)*(c+d)/(e-f))+g
Prefix Expression: +/*+ab+cd-efg


...Program finished with exit code 0
Press ENTER to exit console.
```

## Program-3

**Aim-** Write a program to find number of tokens in the expression and to print identified tokens.

**Compiler Used-** Online GDB compiler

**Code-**

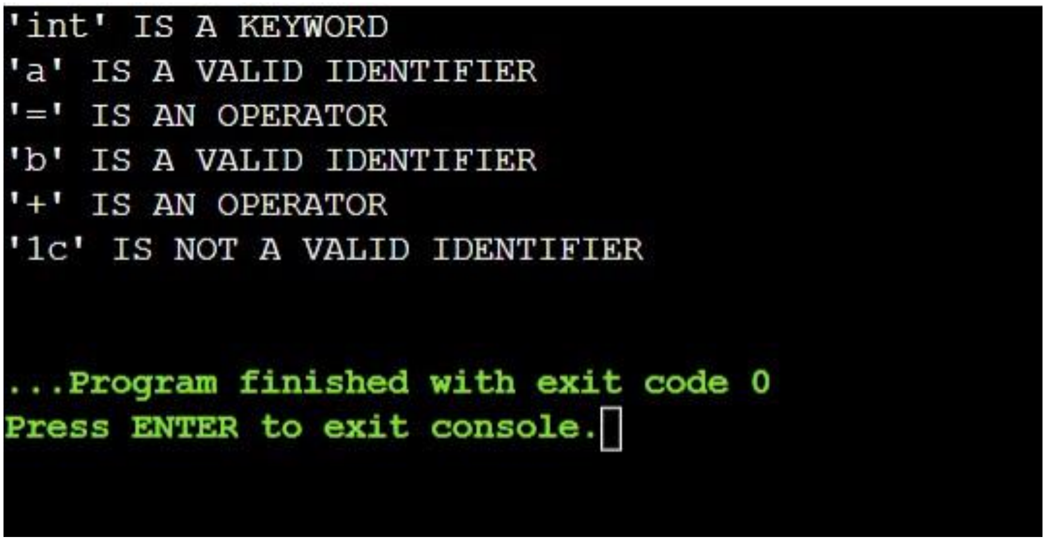
```
#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>


// Returns 'true' if the character is a DELIMITER.

bool isDelimiter(char ch)

{

    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||

        ch == '/' || ch == ',' || ch == ';' || ch == '>' ||

        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||

        ch == '[' || ch == ']' || ch == '{' || ch == '}')

        return (true);

    return (false);

}


// Returns 'true' if the character is an OPERATOR.

bool isOperator(char ch)

{

    if (ch == '+' || ch == '-' || ch == '*' ||

        ch == '/' || ch == '>' || ch == '<' ||

        ch == '=')

        return (true);

    return (false);

}
```

```
// Returns 'true' if the string is a VALID IDENTIFIER.
bool validIdentifier(char* str)
{
   if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
      str[0] == '3' || str[0] == '4' || str[0] == '5' ||
      str[0] == '6' || str[0] == '7' || str[0] == '8' ||
      str[0] == '9' || isDelimiter(str[0]) == true)
      return (false);
   return (true);
}

// Returns 'true' if the string is a KEYWORD.
bool isKeyword(char* str)
{
   if (!strcmp(str, "if") || !strcmp(str, "else") ||
      !strcmp(str, "while") || !strcmp(str, "do") ||
      !strcmp(str, "break") ||
       !strcmp(str, "continue") || !strcmp(str, "int")
      || !strcmp(str, "double") || !strcmp(str, "float")
      || !strcmp(str, "return") || !strcmp(str, "char")
      || !strcmp(str, "case") || !strcmp(str, "char")
      || !strcmp(str, "sizeof") || !strcmp(str, "long")
      || !strcmp(str, "short") || !strcmp(str, "typedef")
      || !strcmp(str, "switch") || !strcmp(str, "unsigned")
      || !strcmp(str, "void") || !strcmp(str, "static")
      || !strcmp(str, "struct") || !strcmp(str, "goto"))
      return (true);
   return (false);
}
```

```
// Returns 'true' if the string is an INTEGER.
bool isInteger(char* str)
{
    int i, len = strlen(str);

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

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

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

// Extracts the SUBSTRING.
char* subString(char* str, int left, int right)
{
    int i;
    char* subStr = (char*)malloc(
                sizeof(char) * (right - left + 2));

    for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
    return (subStr);
}

// Parsing the input STRING.
void parse(char* str)
{
    int left = 0, right = 0;
    int len = strlen(str);

    while (right <= len && left <= right) {
        if (isDelimiter(str[right]) == false)
            right++;


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


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


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


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


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


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


            else if (validIdentifier(subStr) == false
                    && isDelimiter(str[right - 1]) == false)
                printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
            left = right;
        }
```

```
    }
    return;
}

// DRIVER FUNCTION
int main()
{
     // maximum length of string is 100 here
    char str[100] = "int a = b + 1c; ";
    parse(str); // calling the parse function
    return (0);
}
```

**Output-**



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


...Program finished with exit code 0
Press ENTER to exit console.
```

# Program-4

**Aim-** Brief study about Regular Expressions

**Theory-**

**Regular Expressions:** Regular Expressions are used to denote regular languages. An expression is regular if: ϕ is a regular expression for regular language ϕ. ε is a regular expression for regular language {ε}. If a ∈ Σ (Σ represents the input alphabet), a is regular expression with language {a}. If a and b are regular expression, a + b is also a regular expression with language {a,b}. If a and b are regular expression, ab (concatenation of a and b) is also regular.
If a is regular expression, a* (0 or more times a) is also regular.

**Closure Properties of Regular Languages:**

**Union :** If L1 and If L2 are two regular languages, their union L1 ∪ L2 will also be regular. For example, L1 = {an | n ≥ 0} and L2 = {bn | n ≥ 0}

L3 = L1 ∪ L2 = {an ∪ bn | n ≥ 0} is also regular.

**Intersection :** If L1 and If L2 are two regular languages, their intersection L1 ∩ L2 will also be regular. For example,

L1= {am bn | n ≥ 0 and m ≥ 0} and L2= {am bn ∪ bn am | n ≥ 0 and m ≥ 0} L3 = L1 ∩ L2 = {am bn

| n ≥ 0 and m ≥ 0} is also regular.

**Concatenation :** If L1 and If L2 are two regular languages, their concatenation L1.L2 will also be regular. For example,

L1 = {an | n ≥ 0} and L2 = {bn | n ≥ 0}

L3 = L1.L2 = {am . bn | m ≥ 0 and n ≥ 0} is also regular.

**Kleene Closure :** If L1 is a regular language, its Kleene closure L1* will also be regular. For example,

L1 = (a ∪ b) L1* = (a ∪ b)*

**Complement :** If L(G) is regular language, its complement L'(G) will also be regular. Complement of a language can be found by subtracting strings which are in L(G) from all possible strings. For example,
L(G) = {an | n > 3}

L'(G) = {an | n <= 3}

**Constructing an Equivalent Regular Grammar from a Regular Expression:**

**Algorithm:** This algorithm is not very straightforward, and may take a while to understand. We will show how to construct a regular grammar from a regular expression, and it is suggested that you try a few simple exercises using RELIC to confirm your results. The method shown here is simply an example on how you can convert a simple regular expression to a regular grammar.

First of all you must remember the definition of a regular expression and the theorem: for any regular expression e, there exists a regular grammar G recognizing exactly the language described by e. Refer to the relevant section in the course notes for the proof upon which the algorithm is based.

**Example:** Consider the regular expression (a + b)*a. We will now construct a regular grammar for this regular expression. For every terminal symbol a, we create a regular grammar with the rule S \arrow a, start symbol S. We then apply the transformations to these regular grammars, progressively constructing the regular grammar.

First consider the expression a + b. We create two regular grammars:

| | | |
|---|---|---|
| S1 | a | and |
| S2 | b | |

where S1 and S2 are the start symbols. Clearly, these grammars recognize the regular expressions a and b respectively.

Now, we apply the union transformation for regular grammars to get:

| | |
|---|---|
| S3 | a \| b |
| S1 | a |
| S2 | b |

where S3 is the start symbol. This grammar obviously recognizes a + b.

Next, we consider the expression (a + b)*. We already have a regular grammar for (a + b), so now we apply the Kleene star transformation on the regular grammar:

| | |
|---|---|
| S4 | a \| b \| |
| S3 | a \| b |
| S1 | aS3 |
| S2 | bS3 |

where S4 is the start symbol.

Recall that we need a regular grammar that recognizes (a + b)*a. We thus consider again the regular expression a. Again, we create a regular grammar that describes the language:

| | |
|---|---|
| S5 | a |

where S5 is the start symbol.

We now construct the catenation of the regular grammar describing (a + b) * together with this one. We simply apply the transformation that catenates two regular grammars, to get:

| | |
|---|---|
| S4 | a \| b \| |
| S3 | aS5 \| bS5 |
| S1 | aS3 |
| S2 | bS3 |
| S5 | a |

Where, S4 is the start symbol.

This regular grammar is equivalent to the regular expression (a + b) *a

<u>**Program-5**</u>

**<u>Aim-</u>** Write a program which accepts a regular expression from the user and generates a regular grammar which is equivalent to the R.E. entered by user. The grammar will be printed to a text file, with only one production rule in each line. Also, make sure that all production rules are displayed in compact forms e.g. the production rules: S--> aB, S--> cd, S--> PQ

Should be written as S--> aB | cd | PQ  And not as three different production rules. Also, there should not be any repetition of production rules.

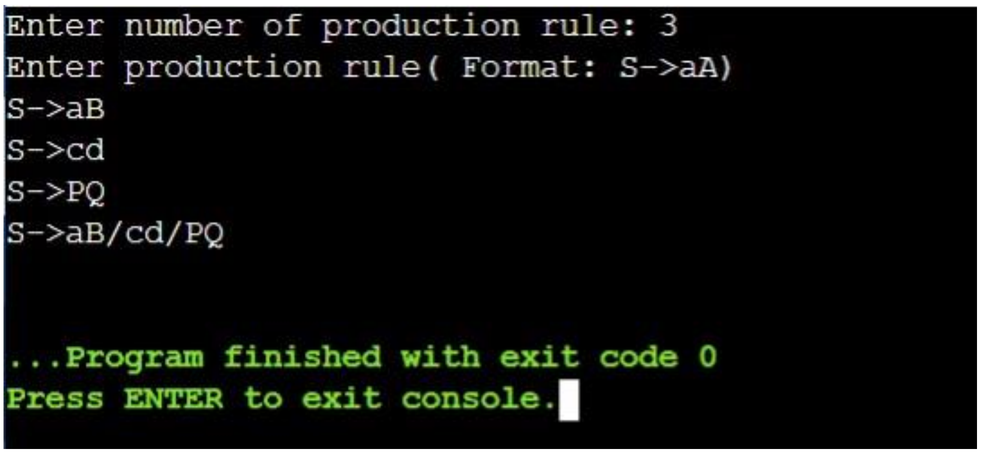
**<u>Compiler Used-</u>** Online GDB compiler

**<u>Code-</u>**

```
import itertools
def main():
    flatten = itertools.chain.from_iterable
    try:
        num = int(input("Enter number of production rule: "))
    except:
        print("Only numbers allowed")
        return
    print("Enter production rule( Format: S->aA)")

    pr = {}
    for i in range(num):
        inp = input().replace(" ","").split('->')
        if inp[0] not in pr:
            pr[inp[0]] = inp[1]
        elif inp[1] not in pr[inp[0]]:
            temp = []
            for z in pr[inp[0]]:
                if len(z) <= 1:
                    temp.append(pr[inp[0]])
                    break
                else:
                    temp.append(z)
            temp.append(inp[1])
            pr[inp[0]] = temp
    s = ""
    for key,values in pr.items():
        s = str(key)+"->"
        for j in values:
            if len(j) == 1:
                s = s + str(j)
            else:
                s = s + str(j) +"/"
        print(s.rstrip("/"))
```

```
main()
```

**Output-**



```
Enter number of production rule: 3
Enter production rule( Format: S->aA)
S->aB
S->cd
S->PQ
S->aB/cd/PQ


...Program finished with exit code 0
Press ENTER to exit console.
```

## Program-6

**Aim-** Write a program to eliminate left recursion

**Compiler Used-** Online GDB compiler

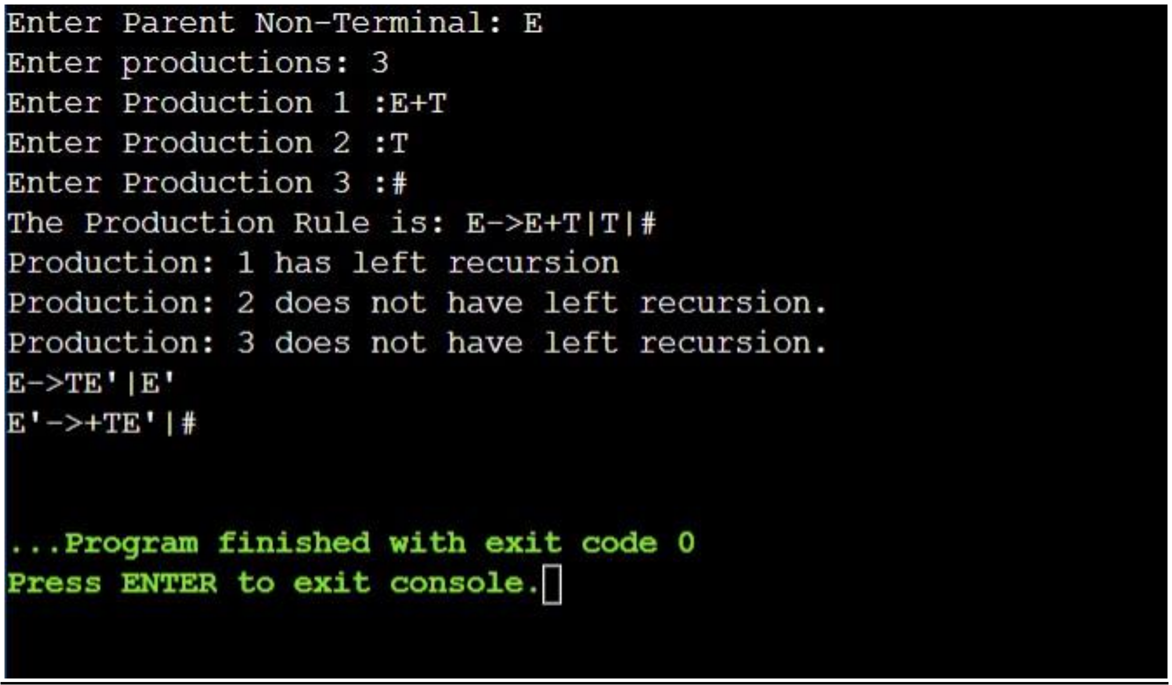
**Code-**

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   int n,i,j,k,l;
   int length[10] = {};
   string d, a, b, flag;
   char c;
   cout<<"Enter Parent Non-Terminal: ";
   cin >> c;
   d.push_back(c);
   a += d + "\'->";
   d += "->";
   b += d;
   cout<<"Enter productions: ";
   cin >> n;
   for (int i = 0; i < n; i++)
   {
      cout<<"Enter Production ";
      cout<<i + 1<<" :";
      cin >> flag;
```

```
    length[i] = flag.size();
    d += flag;
    if (i != n - 1)
    {
        d += "|";
    }
}
cout<<"The Production Rule is: ";
cout<<d<<endl;
for (i = 0, k = 3; i < n; i++)
{
    if (d[0] != d[k])
    {
        cout<<"Production: "<< i + 1;
        cout<<" does not have left recursion.";
        cout<<endl;
        if (d[k] == '#')
        {
            b.push_back(d[0]);
            b += "\'";
        }
        else
        {
            for (j = k; j < k + length[i]; j++)
            {
                b.push_back(d[j]);
            }
            k = j + 1;
```

```
                b.push_back(d[0]);
                b += "\'|";
            }
        }
        else
        {
            cout<<"Production: "<< i + 1 ;
            cout<< " has left recursion";
            cout<< endl;
            if (d[k] != '#')
            {
                for (l = k + 1; l < k + length[i]; l++)
                {
                    a.push_back(d[l]);
                }
                k = l + 1;
                a.push_back(d[0]);
                a += "\'|";
            }
        }
    }
    a += "#";
    cout << b << endl;
    cout << a << endl;
    return 0;
}
```

**Output-**



```
Enter Parent Non-Terminal: E
Enter productions: 3
Enter Production 1 :E+T
Enter Production 2 :T
Enter Production 3 :#
The Production Rule is: E->E+T|T|#
Production: 1 has left recursion
Production: 2 does not have left recursion.
Production: 3 does not have left recursion.
E->TE'|E'
E'->+TE'|#


...Program finished with exit code 0
Press ENTER to exit console.
```

## Program-7

**Aim-** Write a program for Recursive Descent Parser

**Compiler Used-** Online GDB compiler

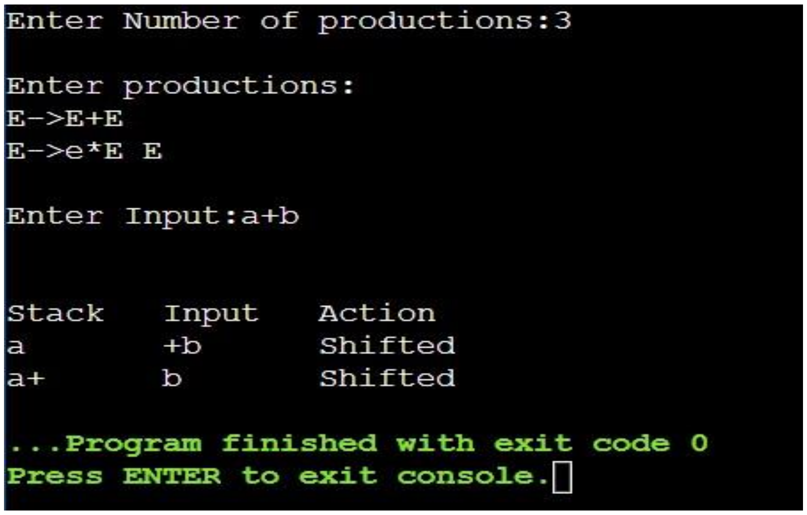
**Code-**

```
#include<iostream>

#include<stack>

#include<cstring>


using namespace std;

struct grammer {

    char p[20];

    char production_symbol[20];

}

grammar[10];


int main() {

    int stpos, j, k, l, m, o, p, f, r;

    int numberofp, tspos, cr;

    cout << "\nEnter Number of productions:";

    cin >> numberofp;

    char sc, terminalsymbol[10];

    cout << "\nEnter productions:\n";

    for (int i = 0; i < numberofp; i++) {

        cin >> terminalsymbol;

        strncpy(grammar[i].p, terminalsymbol, 1);
    strcpy(grammar[i].production_symbol, & terminalsymbol[3]);

}
char inputstring[10];
cout << "\nEnter Input:";
cin >> inputstring;
int lengthinput = strlen(inputstring);
char stack[10];
stpos = 0;
int i = 0;
sc = inputstring[i];
stack[stpos] = sc;
i++;
stpos++;
cout << "\n\nStack\tInput\tAction";
do {
    r = 1;
    while (r != 0) {
        cout << "\n";
        for (p = 0; p < stpos; p++) {
            cout << stack[p];
        }
        cout << "\t";
        for (p = i; p < lengthinput; p++) {
            cout << inputstring[p];
```

```
}
if (r == 2) {
    cout << "\tReduced";
} else {
    cout << "\tShifted";
}
r = 0;
for (k = 0; k < stpos; k++) {
    f = 0;
    for (l = 0; l < 10; l++) {
        terminalsymbol[l] = '\0';
    }
    tspos = 0;
    for (l = k; l < stpos; l++) {
        terminalsymbol[tspos] = stack[l];
        tspos++;
    }
    for (m = 0; m < numberofp; m++) {
        cr = strcmp(terminalsymbol, grammar[m].production_symbol);
        if (cr == 0) {
            for (l = k; l < 10; l++) {
                stack[l] = '\0';
                stpos--;
            }
            stpos = k;
```

```
                    strcat(stack, grammar[m].p);
                    stpos++;
                    r = 2;
                  }
                }
              }
            }
            sc = inputstring[i];
            stack[stpos] = sc;
            i++;
            stpos++;
          } while (strlen(stack) != 1 && stpos != lengthinput);
          if (strlen(stack) == 1) {
            cout << "\n String Accepted";
          }
          return 0;
        }
```

**Output:**



```
Enter Number of productions:3

Enter productions:
E->E+E
E->e*E E

Enter Input:a+b


Stack   Input   Action
a       +b      Shifted
a+      b       Shifted

...Program finished with exit code 0
Press ENTER to exit console.
```

## Program-8

**Aim-** Write a program for Shift Reduce Parser

**Compiler Used-** Online GDB compiler

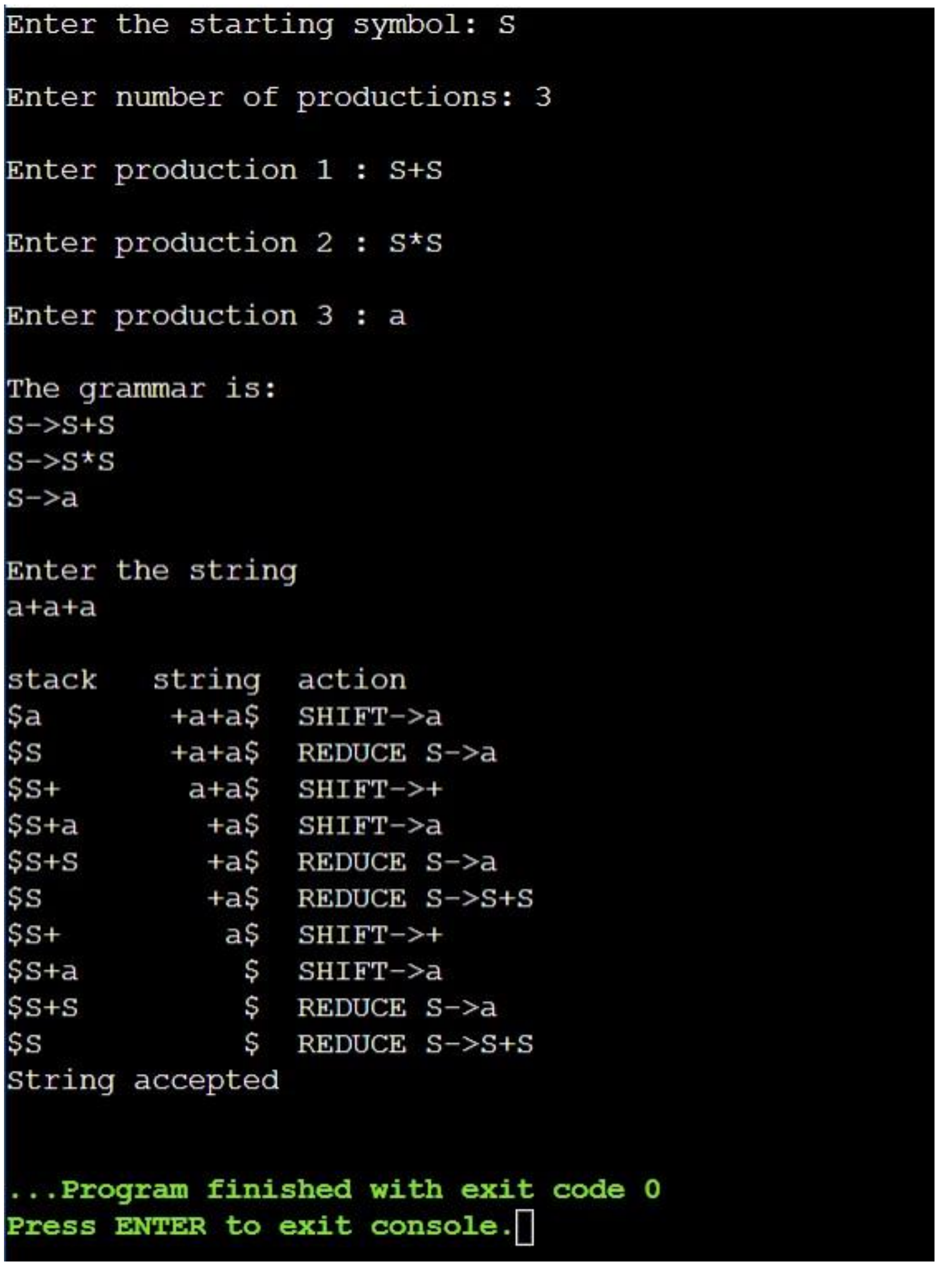
**Code-**

```
#include<iostream>
#include<vector>
using namespace std;

vector<char> a;
vector<char> st;
char start;
string s;
int num,n,i;

void Printout(){
  cout<<"$";
  for(auto x: st)
      cout<<x;
  cout<<"\t";
  for(auto x : a)
      cout<<x;
}

void check(string prod[]){
    string str;
    string pr;
    bool val=false;
    for(int i=0;i<st.size();i++){
        for(int j=i;j<st.size();j++){
            str=str+st[j];
            for(int k=0;k<num;k++){
              if(str==prod[k]){
                  pr=prod[k];
                val=true;
                break;
              }
            }
            if(val==true){
                for(int k=0;k<str.length();k++){
                    st.pop_back();
                }
                st.push_back(start);
                val=false;
                Printout();
                cout<<"$\t"<<"REDUCE "<<start<<"->"<<pr<<endl;
                check(prod);
            }
        }
        str.clear();
    }
}

int main(){
    cout<<"Enter the starting symbol: ";
    cin>>start;
```

```
cout<<"\nEnter number of productions: ";
cin>>num;
string prod[num];
for(i=0;i<num;i++){
   cout<<"\nEnter production "<<(i+1)<<" : ";
   cin>>prod[i];
}
cout<<"\nThe grammar is: "<<endl;
for(i=0;i<num;i++){
   cout<<start<<"->"<<prod[i]<<endl;;
}
cout<<"\nEnter the string"<<endl;
cin>>s;
n=s.length();
for(i=n-1;i>=0;i--){
   a.push_back(s[i]);
}

cout<<"\nstack\tstring\taction"<<endl;
for(i=0;i<n;i++){
   st.push_back(a[i]);
   a[i]=' ';
   Printout();
   cout<<"$\t"<<"SHIFT->"<<st.back()<<endl;;
   check(prod);
}
if(st.size()==1 && st[0]==start)
   cout<<"String accepted"<<endl;
```

```
        else
           cout<<"String rejected"<<endl;
    }
```

**Output:**



```
Enter the starting symbol: S

Enter number of productions: 3

Enter production 1 : S+S

Enter production 2 : S*S

Enter production 3 : a

The grammar is:
S->S+S
S->S*S
S->a

Enter the string
a+a+a

stack   string  action
$a       +a+a$  SHIFT->a
$S       +a+a$  REDUCE S->a
$S+       a+a$  SHIFT->+
$S+a       +a$  SHIFT->a
$S+S       +a$  REDUCE S->a
$S         +a$  REDUCE S->S+S
$S+         a$  SHIFT->+
$S+a         $  SHIFT->a
$S+S         $  REDUCE S->a
$S           $  REDUCE S->S+S
String accepted


...Program finished with exit code 0
Press ENTER to exit console.
```

## Program-9

**Aim-** Write a program to implement LL(1) parser on production

$$S\text{–>}ABC$$

$$A\text{–>}abA|ab$$

$$B\text{–>}b|BC$$

$$C\text{–>}c|cC$$

**Compiler Used-** Online GDB compiler

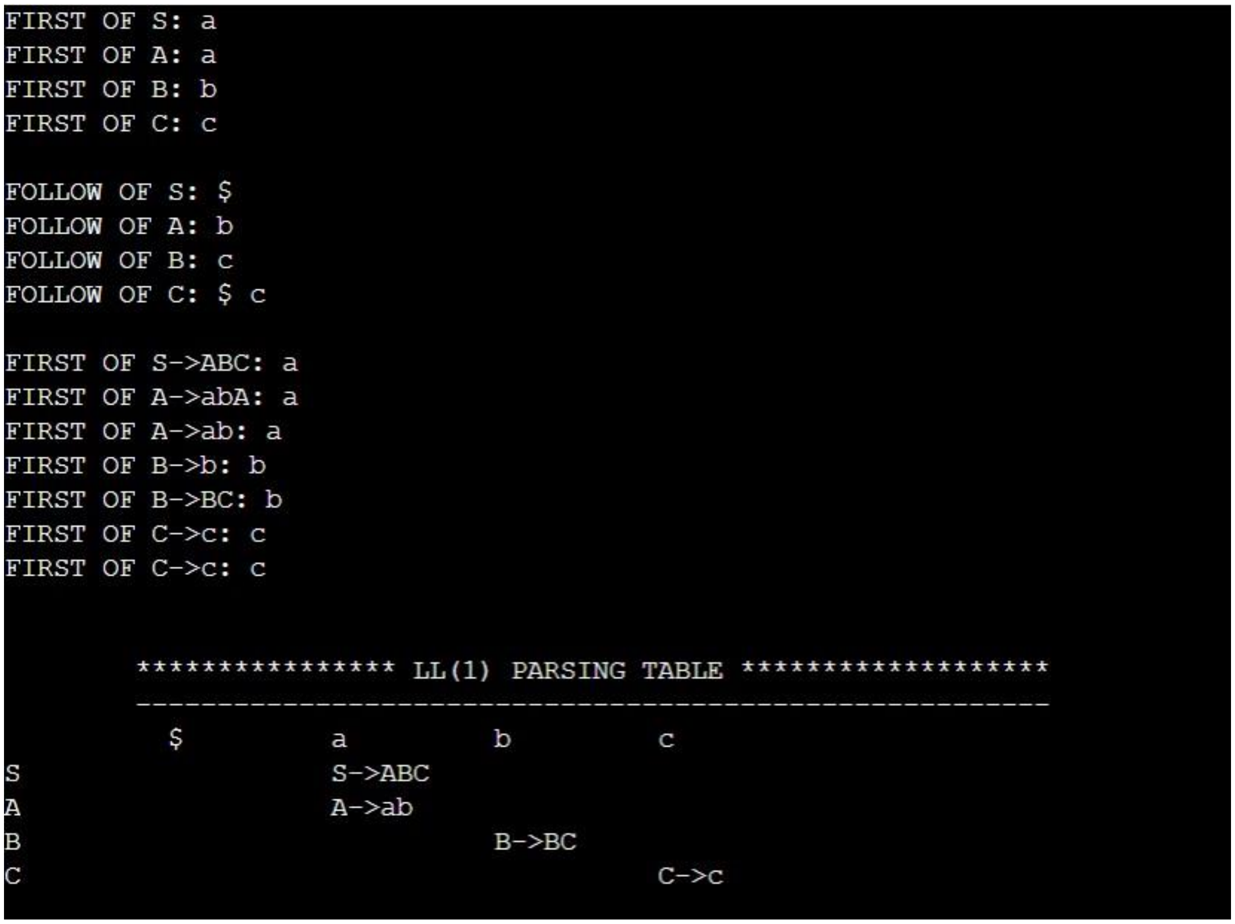
**Code-**

```
#include<stdio.h>

#include<string.h>

#define TSIZE 128

int table[100][TSIZE];

char terminal[TSIZE];

char nonterminal[26];


struct product {

   char str[100];

   int len;

}pro[20];


int no_pro;

char first[26][TSIZE];

char follow[26][TSIZE];

char first_rhs[100][TSIZE];


int isNT(char c) {
    return c >= 'A' && c <= 'Z';
}


void readFromFile() {
    FILE* fptr;
    fptr = fopen("production.txt", "r");
    char buffer[255];
    int i;
    int j;
    while (fgets(buffer, sizeof(buffer), fptr)) {
        printf("%s", buffer);
        j = 0;
        nonterminal[buffer[0] - 'A'] = 1;
        for (i = 0; i < strlen(buffer) - 1; ++i) {
            if (buffer[i] == '|') {
                ++no_pro;
                pro[no_pro - 1].str[j] = '\0';
                pro[no_pro - 1].len = j;
                pro[no_pro].str[0] = pro[no_pro - 1].str[0];
                pro[no_pro].str[1] = pro[no_pro - 1].str[1];
                pro[no_pro].str[2] = pro[no_pro - 1].str[2];
                j = 3;
            }
            else {
                pro[no_pro].str[j] = buffer[i];
                ++j;
                if (!isNT(buffer[i]) && buffer[i] != '-' && buffer[i] != '>') {
                    terminal[buffer[i]] = 1;
                }
            }
        }
        pro[no_pro].len = j;
        ++no_pro;
    }
}
void add_FIRST_A_to_FOLLOW_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
        follow[B - 'A'][i] = follow[B - 'A'][i] || first[A - 'A'][i];
    }
}
void add_FOLLOW_A_to_FOLLOW_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
        follow[B - 'A'][i] = follow[B - 'A'][i] || follow[A - 'A'][i];
    }
}
void FOLLOW() {
    int t = 0;
    int i, j, k, x;
    while (t++ < no_pro) {
        for (k = 0; k < 26; ++k) {
            if (!nonterminal[k]) continue;
            char nt = k + 'A';
            for (i = 0; i < no_pro; ++i) {
                for (j = 3; j < pro[i].len; ++j) {
                    if (nt == pro[i].str[j]) {
                        for (x = j + 1; x < pro[i].len; ++x) {
                            char sc = pro[i].str[x];
                            if (isNT(sc)) {
                                add_FIRST_A_to_FOLLOW_B(sc, nt);
                                if (first[sc - 'A']['^'])
                                continue;
                            }
                            else {
                                follow[nt - 'A'][sc] = 1;
                            }
                            break;
                        }
                        if (x == pro[i].len)
                        add_FOLLOW_A_to_FOLLOW_B(pro[i].str[0], nt);
                    }
                }
            }
        }
    }
}
void add_FIRST_A_to_FIRST_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^') {
            first[B - 'A'][i] = first[A - 'A'][i] || first[B - 'A'][i];
        }
    }
}
void FIRST() {
    int i, j;
    int t = 0;
    while (t < no_pro) {
        for (i = 0; i < no_pro; ++i) {
            for (j = 3; j < pro[i].len; ++j) {
                char sc = pro[i].str[j];
                if (isNT(sc)) {
                    add_FIRST_A_to_FIRST_B(sc, pro[i].str[0]);
                    if (first[sc - 'A']['^'])
                    continue;
                }
                else {
                    first[pro[i].str[0] - 'A'][sc] = 1;
                }
                break;
            }
            if (j == pro[i].len)
            first[pro[i].str[0] - 'A']['^'] = 1;
        }
        ++t;
    }
}
void add_FIRST_A_to_FIRST_RHS__B(char A, int B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
        first_rhs[B][i] = first[A - 'A'][i] || first_rhs[B][i];
    }
}
void FIRST_RHS() {
    int i, j;
    int t = 0;
    while (t < no_pro) {
        for (i = 0; i < no_pro; ++i) {
            for (j = 3; j < pro[i].len; ++j) {
                char sc = pro[i].str[j];
                if (isNT(sc)) {
                    add_FIRST_A_to_FIRST_RHS__B(sc, i);
                    if (first[sc - 'A']['^'])
                    continue;
                }
                else {
                    first_rhs[i][sc] = 1;
                }
                break;
            }
            if (j == pro[i].len)
            first_rhs[i]['^'] = 1;
        }
        ++t;
    }
}
int main() {
    readFromFile();
    follow[pro[0].str[0] - 'A']['$'] = 1;
    FIRST();
    FOLLOW();
    FIRST_RHS();
    int i, j, k;
    printf("\n");
    for (i = 0; i < no_pro; ++i) {
        if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
            char c = pro[i].str[0];
            printf("FIRST OF %c: ", c);
            for (j = 0; j < TSIZE; ++j) {
                if (first[c - 'A'][j]) {
                    printf("%c ", j);
                }
            }
            printf("\n");
        }
    }
    printf("\n");
    for (i = 0; i < no_pro; ++i) {
```

```
    if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
        char c = pro[i].str[0];
        printf("FOLLOW OF %c: ", c);
        for (j = 0; j < TSIZE; ++j) {
            if (follow[c - 'A'][j]) {
                printf("%c ", j);
            }
        }
        printf("\n");
    }
}
printf("\n");
for (i = 0; i < no_pro; ++i) {
    printf("FIRST OF %s: ", pro[i].str);
    for (j = 0; j < TSIZE; ++j) {
        if (first_rhs[i][j]) {
            printf("%c ", j);
        }
    }
    printf("\n");
}
terminal['$'] = 1;
terminal['^'] = 0;
printf("\n");
printf("\n\t**************** LL(1) PARSING TABLE *******************\n");
printf("\t--------------------------------------------------------\n");
printf("%-10s", "");
```

```
for (i = 0; i < TSIZE; ++i) {
    if (terminal[i]) printf("%-10c", i);
}
printf("\n");
int p = 0;
for (i = 0; i < no_pro; ++i) {
    if (i != 0 && (pro[i].str[0] != pro[i - 1].str[0]))
    p = p + 1;
    for (j = 0; j < TSIZE; ++j) {
        if (first_rhs[i][j] && j != '^') {
            table[p][j] = i + 1;
        }
        else if (first_rhs[i]['^']) {
            for (k = 0; k < TSIZE; ++k) {
                if (follow[pro[i].str[0] - 'A'][k]) {
                    table[p][k] = i + 1;
                }
            }
        }
    }
}
k = 0;
for (i = 0; i < no_pro; ++i) {
    if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
        printf("%-10c", pro[i].str[0]);
        for (j = 0; j < TSIZE; ++j) {
            if (table[k][j]) {
                    printf("%-10s", pro[table[k][j] - 1].str);
                }
                else if (terminal[j]) {
                    printf("%-10s", "");
                }
            }
            ++k;
            printf("\n");
        }
    }
}
```

**Output-**



```
FIRST OF S: a
FIRST OF A: a
FIRST OF B: b
FIRST OF C: c

FOLLOW OF S: $
FOLLOW OF A: b
FOLLOW OF B: c
FOLLOW OF C: $ c

FIRST OF S->ABC: a
FIRST OF A->abA: a
FIRST OF A->ab: a
FIRST OF B->b: b
FIRST OF B->BC: b
FIRST OF C->c: c
FIRST OF C->c: c


        **************** LL(1) PARSING TABLE *******************
        --------------------------------------------------------
          $         a         b         c
S                   S->ABC
A                   A->ab
B                             B->BC
C                                       C->c
```

## Program-10

**Aim-** Write a program which accepts a regular grammar with no left recursion, and no null production rules, and then it accepts a string and reports whether the string is accepted by the grammar or not.

**Compiler Used-** Online GDB compiler

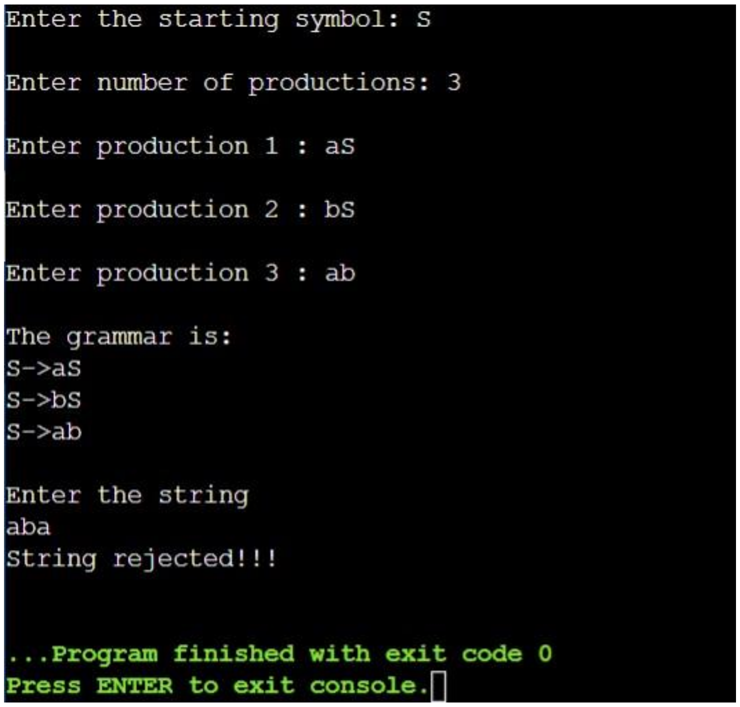
**Code-**

```
#include<iostream>

#include<vector>

using namespace std;

vector<char> a;

vector<char> st;

char start;

string s;

int num,n,i;

void check(string prod[]){

   string str;

   string pr;

   bool val=false;

   for(int i=0;i<st.size();i++){

      for(int j=i;j<st.size();j++){

         str=str+st[j];

         for(int k=0;k<num;k++){

           if(str==prod[k]){

              pr=prod[k];

             val=true;

             break; }}
```

```
            if(val==true){
                for(int k=0;k<str.length();k++){
                    st.pop_back();
                }
                st.push_back(start);
                val=false;
                check(prod);
            } }
        str.clear();
    } }
int main(){
    cout<<"Enter the starting symbol: ";
    cin>>start;
    cout<<"\nEnter number of productions: ";
    cin>>num;
    string prod[num];
    for(i=0;i<num;i++){
        cout<<"\nEnter production "<<(i+1)<<" : ";
        cin>>prod[i];
    }
    cout<<"\nThe grammar is: "<<endl;
    for(i=0;i<num;i++){
        cout<<start<<"->"<<prod[i]<<endl;;
    }
    cout<<"\nEnter the string"<<endl;
```

```
        cin>>s;
        n=s.length();
        for(i=n-1;i>=0;i--){
            a.push_back(s[i]);
        }
        for(i=0;i<n;i++){
            st.push_back(a[i]);
            a[i]=' ';
            check(prod); }
        if(st.size()==1 && st[0]==start)
            cout<<"String accepted!!!"<<endl;
        else
            cout<<"String rejected!!!"<<endl;
    }
```

**Output**-



```
Enter the starting symbol: S

Enter number of productions: 3

Enter production 1 : aS

Enter production 2 : bS

Enter production 3 : ab

The grammar is:
S->aS
S->bS
S->ab

Enter the string
aba
String rejected!!!


...Program finished with exit code 0
Press ENTER to exit console.
```

# Program-11

**Aim-** Write a program to evaluate mathematical expression by using parser.

**Theory-**

Shift Reduce Parser is a type of Bottom-Up Parser. It generates the Parse Tree from Leaves to the Root. In Shift Reduce Parser, the input string will be reduced to the starting symbol. This reduction can be produced by handling the rightmost derivation in reverse, i.e., from starting symbol to the input string.

Shift Reduce Parser requires two Data Structures

Input Buffer

Stack

There are the various steps of Shift Reduce Parsing which are as follows −

There are the various steps of Shift Reduce Parsing which are as follows −

It uses a stack and an input buffer.

Insert $ at the bottom of the stack and the right end of the input string in Input Buffer.

Stack
$
Input String
w
$

**Shift** − Parser shifts zero or more input symbols onto the stack until the handle is on top of the stack.

**Reduce** − Parser reduce or replace the handle on top of the stack to the left side of production, i.e., R.H.S. of production is popped, and L.H.S is pushed.

**Accept** − Step 3 and Step 4 will be repeated until it has detected an error or until the stack includes start symbol (S) and input Buffer is empty, i.e., it contains $.

Stack
$
S
Input String
$

**Code-**

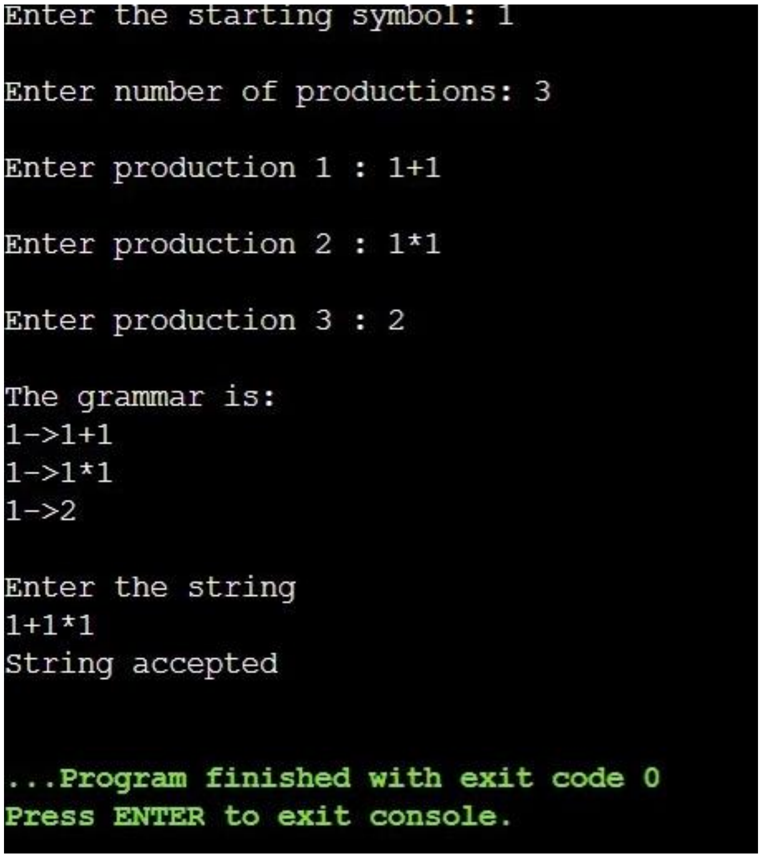
```
#include<iostream>

#include<vector>

using namespace std;


vector<char> a;

vector<char> st;

char start;

string s;

int num,n,i;


void check(string prod[]){

    string str;

    string pr;

    bool val=false;

    for(int i=0;i<st.size();i++){

        for(int j=i;j<st.size();j++){

            str=str+st[j];

            for(int k=0;k<num;k++){

              if(str==prod[k]){

                  pr=prod[k];

                val=true;

                break;

              }
```

```
                }
                if(val==true){
                    for(int k=0;k<str.length();k++){
                        st.pop_back();
                    }
                    st.push_back(start);
                    val=false;
                    check(prod);
                }
            }
            str.clear();
        }
    }
    int main(){
        cout<<"Enter the starting symbol: ";
        cin>>start;
        cout<<"\nEnter number of productions: ";
        cin>>num;
        string prod[num];
        for(i=0;i<num;i++){
            cout<<"\nEnter production "<<(i+1)<<" : ";
            cin>>prod[i];
        }
        cout<<"\nThe grammar is: "<<endl;
```

```
for(i=0;i<num;i++){
    cout<<start<<"->"<<prod[i]<<endl;;
}
cout<<"\nEnter the string"<<endl;
cin>>s;
int flag = 0;
for(int i = 0;i<s.length();i++){
    if(int(s[i]) >= 48 && int(s[i]) <= 57 || (int(s[i]) >= 33 && int(s[i]) <= 47)){
        flag = 1;
    }
    else{
        break;
    }
}
if(flag == 0){
    return 0;
}
n=s.length();
for(i=n-1;i>=0;i--){
    a.push_back(s[i]);
}
for(i=0;i<n;i++){
    st.push_back(a[i]);
    a[i]=' ';
```

```
        check(prod);

    }

    if(st.size()==1 && st[0]==start)

        cout<<"String accepted"<<endl;

    else

        cout<<"String rejected"<<endl;

}
```

**<u>Output-</u>**



```
Enter the starting symbol: 1

Enter number of productions: 3

Enter production 1 : 1+1

Enter production 2 : 1*1

Enter production 3 : 2

The grammar is:
1->1+1
1->1*1
1->2

Enter the string
1+1*1
String accepted


...Program finished with exit code 0
Press ENTER to exit console.
```