INDEX

Experiment-1

Aim: Write a program which check if the given string is accepted by the regular expression $0^*1^+(0+1)$ or not.

Theory: Simple phrases known as Regular Expressions can simply explain the language that finite automata accept. It's the most efficient means of expressing any language. A regular expression is also known as a pattern sequence that defines a string. To match character combinations in strings, regular expressions are used. This pattern was employed by the string searching method to discover the operations on a string.

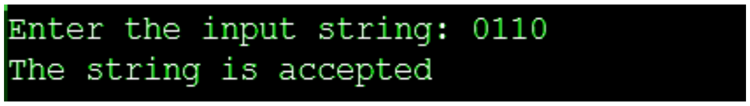
Code:

```
#include <bits/stdc++.h>


using namespace std;

int main() {

  string input;

  cout << "Enter Input String: ";

  cin >> input;

  int dfa = 0;

  bool isAccepted = false;

  for (char c: input) {

    switch (dfa) {

    case 0: {

      if (c == '0') {

        dfa = 1;

        isAccepted = 1;

      } else if (c == '1') {
        dfa = 2;
        isAccepted = 1;
    } else {
        cout << "Error! Invalid Input" << endl;
        return 0;
    }
    break;
}
case 1: {
    if (c == '0') {
        dfa = 1;
        isAccepted = 1;
    } else if (c == '1') {
        dfa = 2;
        isAccepted = 1;
    } else {
        cout << "Error! Invalid Input" << endl;
        return 0;
    }
    break;
}
case 2: {
    if (c == '0') {
        dfa = 3;
        isAccepted = 1;
    } else if (c == '1') {
        dfa = 2;
```

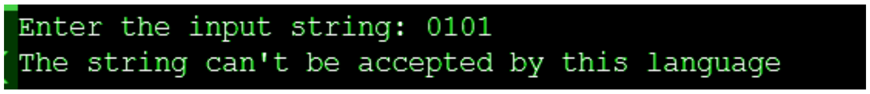
```
            isAccepted = 1;
        } else {
            cout << "Error! Invalid Input" << endl;
            return 0;
        }
        break;
    }
    case 3: {
        if (c == '0' || c == '1') {
            dfa = 4;
            isAccepted = 0;
        } else {
            cout << "Error! Invalid Input" << endl;
            return 0;
        }
        break;
    }
    case 4: {
        cout << "Not accepted" << endl;
        return 0;
    }
    break;
    }
}
if (isAccepted)
    cout << "Accepted" << endl;
else
```

```
        cout << "Accepted" << endl;
    return 0;
}
```

Output:



```
Enter the input string: 0110
The string is accepted
```



```
Enter the input string: 0101
The string can't be accepted by this language
```

Experiment-2

Aim: Write a program which check if the given string is accepted by the regular expression $a^+b^*(c+d)+a^*c^*$or not.

Theory:

Simple phrases known as Regular Expressions can simply explain the language that finite automata accept. It's the most efficient means of expressing any language. A regular expression is also known as a pattern sequence that defines a string. To match character combinations in strings, regular expressions are utilised. This pattern was employed by the string searching method to discover the operations on a string.

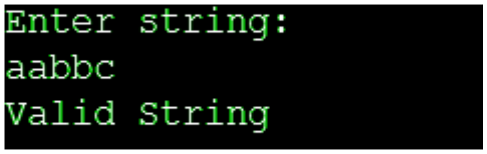
Code:

```
#include <bits/stdc++.h>

using namespace std;


int stringAcceptance(string input)

{

   int currentState = 0;

   bool isAccepted = false;


   for(char c: input)

   {

      switch(currentState)

      {

         case 0:

         {

            if(c == '0')

            {
```

```
        currentState = 0;

        isAccepted = true;

    }


    else if(c == '1')

    {

        currentState = 1;

        isAccepted = false;

    }


    else

    {

        cout<<"Error! Invalid input";

        isAccepted = false;

    }

    break;

}


case 1:

{

    if(c=='0')

    {

        currentState = 2;

        isAccepted = false;

    }


    else if(c=='1')
    {
        currentState = 3;
        isAccepted = false;
    }

    else
    {
        cout<<"Error! Invalid input";
        isAccepted = false;
    }

    break;
}

case 2:
{
    if(c=='0')
    {
        currentState = 4;
        isAccepted = false;
    }

    else if(c=='1')
    {
        currentState = 0;
        isAccepted = true;
    }
    else
    {
        cout<<"Error! Invalid input";
        isAccepted = false;
    }

    break;
}

case 3:
{
    if(c=='0')
    {
        currentState = 1;
        isAccepted = false;
    }

    else if(c=='1')
    {
        currentState = 2;
        isAccepted = false;
    }

    else
    {
        cout<<"Error! Invalid input";
        isAccepted = false;
    }
    break;
}

case 4:
{
    if(c=='0')
    {
        currentState = 3;
        isAccepted = false;
    }

    else if(c=='1')
    {
        currentState = 4;
        isAccepted = false;
    }

    else
    {
        cout<<"Error! Invalid input";
        isAccepted = false;
    }

    break;
}
```

```
        }
    }

    if(isAccepted)
    {
        cout<<"Accepted"<<endl;
    }

    else
    {
        cout<<"Not accepted"<<endl;
    }
    return 0;
}

int main()
{
    string input;

    cout<<"Enter Bin. String: ";
    cin>>input;
    stringAcceptance(input);
    return 0;
}
```

Output:



```
Enter string:
aabbc
Valid String
```

Experiment-3

Aim: Write a program to convert the infix expression into a postfix expression.

Theory:

Infix Expressions: The representation of the form a op b is infix form. When an operator is sandwiched between two operands.

Postfix Expressions: The expression of the form a b op is a postfix. When each pair of operands is followed by an operator.

Code:

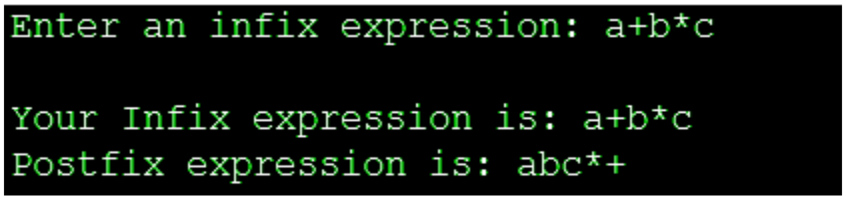
```
#include<bits/stdc++.h>

using namespace std;


bool IsOperator(char);

bool IsOperand(char);

bool eqlOrhigher(char, char);

string convert(string);


int main() {

  string infix_expression, postfix_expression;

  int ch;

  do {

    cout << " Enter an infix expression: ";

    cin >> infix_expression;

    postfix_expression = convert(infix_expression);

    cout << "\n Your Infix expression is: " << infix_expression;

    cout << "\n Postfix expression: " << postfix_expression;

    cout << "\n Do you want to enter infix expression(1/ 0)?";
```

```
        cin >> ch;
    } while (ch == 1);
    return 0;
}

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

bool IsOperand(char c) {
    if (c >= 'A' && c <= 'Z')
        return true;
    if (c >= 'a' && c <= 'z')
        return true;
    if (c >= '0' && c <= '9')
        return true;
    return false;
}

int precedence(char op) {
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    if (op == '^')
```

```
        return 3;
    return 0;
}

bool eqlOrhigher(char op1, char op2) {
    int p1 = precedence(op1);
    int p2 = precedence(op2);
    if (p1 == p2) {
        if (op1 == '^')
            return false;
        return true;
    }
    return (p1 > p2 ? true : false);
}

string convert(string infix) {
    stack < char > S;
    string postfix = "";
    char ch;

    S.push('(');
    infix += ')';

    for (int i = 0; i < infix.length(); i++) {
        ch = infix[i];

        if (ch == ' ')
```

```
            continue;
        else if (ch == '(')
            S.push(ch);
        else if (IsOperand(ch))
            postfix += ch;
        else if (IsOperator(ch)) {
            while (!S.empty() && eqlOrhigher(S.top(), ch)) {
                postfix += S.top();
                S.pop();
            }
            S.push(ch);
        } else if (ch == ')') {
            while (!S.empty() && S.top() != '(') {
                postfix += S.top();
                S.pop();
            }
            S.pop();
        }
    }
    return postfix;
}
```

Output:



```
Enter an infix expression: a+b*c

Your Infix expression is: a+b*c
Postfix expression is: abc*+
```

<u>**Experiment-4**</u>

Aim: Write a program to identify the total number of tokens present in a statement

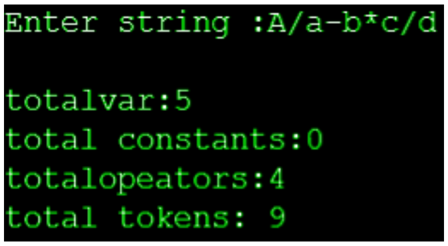
Theory:

Lexemes are defined as a token's sequence of characters (alphanumeric). Every lexeme must follow a set of rules in order to be recognised as a legitimate token. Grammar rules, in the form of a pattern, define these rules. A pattern describes what may be a token, and regular expressions are used to define these patterns. Tokens in programming languages include keywords, constants, identifiers, strings, integers, operators, and punctuation symbols.

Code:

```
#include<stdio.h>

#include<ctype.h>

#include<conio.h>

#include<string.h>

int main() {

    char str[50];

    int len;

    int i, a = 0, b = 0, d = 0, f = 0,

        var = 0, tokens = 0, constant = 0, oper = 0;

    printf("Enter string :");

    scanf("%s", str);

    len = strlen(str);

    for (i = 0; i < len; i++) {

        if (isalpha(str[i]))

            a++;

        if (isdigit(str[i])) {

            while (isdigit(str[i])) {
                i++;
            }
            d++;
        }
        if (str[i] == '%' || str[i] == '*' || str[i] == '/' || str[i] == '+' || str[i] == '-' || str[i] ==
'=')
            f++;
        else
            b++;
    }
    var = a;
    constant = d;
    oper = f;
    tokens =
        var +constant + oper;
    printf("\ntotalvar:%d ",
        var);
    printf("\ntotal constants:%d", constant);
    printf("\ntotalopeators:%d", oper);
    printf("\ntotal tokens: %d", tokens);
    return 0;
    getch();
}
```

Output:



```
Enter string :A/a-b*c/d

totalvar:5
total constants:0
totalopeators:4
total tokens: 9
```

Experiment-5

Aim: Write a program to implement a lexical analyzer.

Theory:

The initial phase of a compiler is lexical analysis. It uses modified source code written in the form of phrases from language preprocessors. By eliminating any whitespace or comments from the source code, the lexical analyzer breaks these syntaxes down into a sequence of tokens. The lexical analyzer gives an error if a token is found to be incorrect. The lexical analyzer and the syntax analyzer function in tandem. When the syntax analyzer requests it, it pulls character streams from the source code, verifies for legal tokens, and provides the data to it.

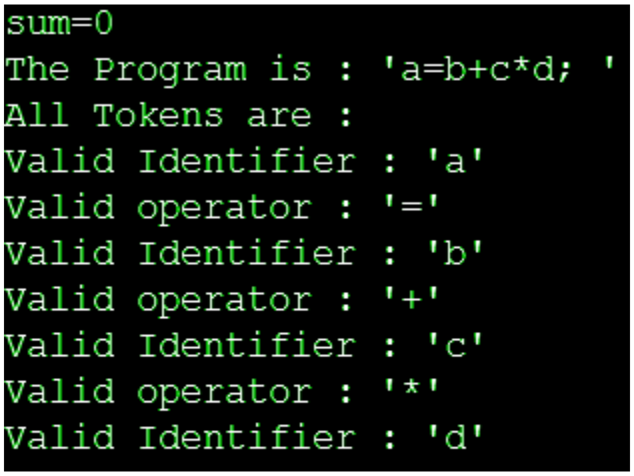
Code:

```
#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

bool isValidDelimiter(char ch) {

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

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

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

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

      return (true);

   return (false);

}

bool isValidOperator(char ch) {

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

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

      ch == '=')

      return (true);
```

```
    return (false);
}
bool isvalidIdentifier(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' || isValidDelimiter(str[0]) == true)
        return (false);
    return (true);
}
bool isValidKeyword(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);
}
bool isValidInteger(char * str) {
    int i, len = strlen(str);
    if (len == 0)
        return (false);
    for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2' && str[i] != '3' && str[i] != '4'
&& str[i] != '5' &&
            str[i] != '6' && str[i] != '7' && str[i] != '8' && str[i] != '9' || (str[i] == '-' && i
> 0))
            return (false);
    }
    return (true);
}
bool isRealNumber(char * str) {
    int i, len = strlen(str);
    bool hasDecimal = false;
    if (len == 0)
        return (false);
    for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2' && str[i] != '3' && str[i] != '4'
&& str[i] != '5' && str[i] != '6' && str[i] != '7' && str[i] != '8' &&
            str[i] != '9' && str[i] != '.' || (str[i] == '-' && i > 0))
            return (false);
        if (str[i] == '.')
            hasDecimal = true;
    }
    return (hasDecimal);
}
char * subString(char * str, int left, int right) {
    int i;
    char * subStr = (char * ) malloc(sizeof(char) * (right - left + 2));
    for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
    return (subStr);
```

```
}
void detectTokens(char * str) {
    int left = 0, right = 0;
    int length = strlen(str);
    while (right <= length && left <= right) {
        if (isValidDelimiter(str[right]) == false)
            right++;
        if (isValidDelimiter(str[right]) == true && left == right) {
            if (isValidOperator(str[right]) == true)
                printf("Valid operator : '%c'\n", str[right]);
            right++;
            left = right;
        } else if (isValidDelimiter(str[right]) == true && left != right || (right == length
&& left != right)) {
            char * subStr = subString(str, left, right - 1);
            if (isValidKeyword(subStr) == true)
                printf("Valid keyword : '%s'\n", subStr);
            else if (isValidInteger(subStr) == true)
                printf("Valid Integer : '%s'\n", subStr);
            else if (isRealNumber(subStr) == true)
                printf("Real Number : '%s'\n", subStr);
            else if (isvalidIdentifier(subStr) == true &&
                isValidDelimiter(str[right - 1]) == false)
                printf("Valid Identifier : '%s'\n", subStr);
            else if (isvalidIdentifier(subStr) == false &&
                isValidDelimiter(str[right - 1]) == false)
                printf("Invalid Identifier : '%s'\n", subStr);
            left = right;
        }
    }
    return;
}
int main() {
        int a, b, c, d;
        char str[100] = "a=b+c*d; ";
        printf("sum=%d \n", a);
        printf("The Program is : '%s' \n", str);
        printf("All Tokens are : \n");
        detectTokens(str);
        return (0);
```

Output:



```
sum=0
The Program is : 'a=b+c*d; '
All Tokens are :
Valid Identifier : 'a'
Valid operator : '='
Valid Identifier : 'b'
Valid operator : '+'
Valid Identifier : 'c'
Valid operator : '*'
Valid Identifier : 'd'
```

Experiment-6

Aim: Write a program to remove left recursion.

Theory:

A Grammar G (V, T, P, S) is left recursive if it has a production in the form.

$A \rightarrow A\ \alpha\ |\beta.$

The above Grammar is left recursive because the left of production is occurring at a first position on the right side of production. It can eliminate left recursion by replacing a pair of production with

$A \rightarrow \beta A'$

$A \rightarrow \alpha A'|\epsilon$

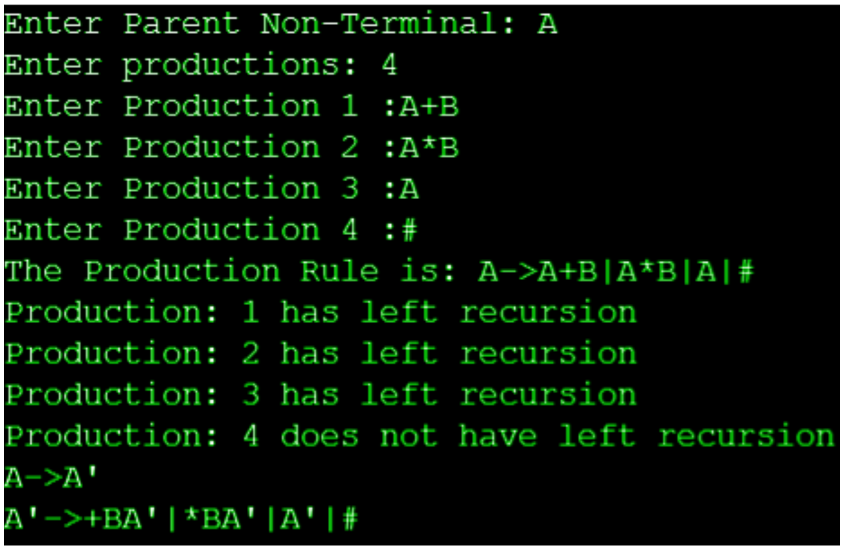
Code:

```
#include <iostream>


#include <string>


using namespace std;

int main() {

    int n, j, l, i, k;

    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: A
Enter productions: 4
Enter Production 1 :A+B
Enter Production 2 :A*B
Enter Production 3 :A
Enter Production 4 :#
The Production Rule is: A->A+B|A*B|A|#
Production: 1 has left recursion
Production: 2 has left recursion
Production: 3 has left recursion
Production: 4 does not have left recursion
A->A'
A'->+BA'|*BA'|A'|#
```

Experiment-7

Aim: Write a program to compute First and Follow for the given grammar

Theory:

FIRST and FOLLOW are two functions associated with grammar that help us fill in the entries of an M-table.

FIRST ()− It is a function that gives the set of terminals that begin the strings derived from the production rule.

FOLLOW()- Follow (A) is defined as the collection of terminal symbols that occur directly to the right of A.

Code:

```
#include<stdio.h>

#include<ctype.h>

#include<string.h>

void followfirst(char, int, int);

void follow(char c);

void findfirst(char, int, int);

int count, n = 0;

char calc_first[10][100];

char calc_follow[10][100];

int m = 0;

char production[10][10];

char f[10], first[10];

int k;

char ck;

int e;
int main(int argc, char ** argv) {
    int jm = 0;
    int km = 0;
    int i, choice;
    char c, ch;
    count = 8;

    strcpy(production[0], "E=TR");
    strcpy(production[1], "R=+TR");
    strcpy(production[2], "R=#");
    strcpy(production[3], "T=FY");
    strcpy(production[4], "Y=*FY");
    strcpy(production[5], "Y=#");
    strcpy(production[6], "F=(E)");
    strcpy(production[7], "F=i");

    int kay;
    char done[count];
    int ptr = -1;

    for (k = 0; k < count; k++) {
        for (kay = 0; kay < 100; kay++) {
            calc_first[k][kay] = '!';
        }
    }
    int point1 = 0, point2, xxx;
```

```
for (k = 0; k < count; k++) {
    c = production[k][0];
    point2 = 0;
    xxx = 0;

    for (kay = 0; kay <= ptr; kay++)
        if (c == done[kay])
            xxx = 1;

    if (xxx == 1)
        continue;

    findfirst(c, 0, 0);
    ptr += 1;

    done[ptr] = c;
    printf("\n First(%c) = { ", c);
    calc_first[point1][point2++] = c;

    for (i = 0 + jm; i < n; i++) {
        int lark = 0, chk = 0;

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

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

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

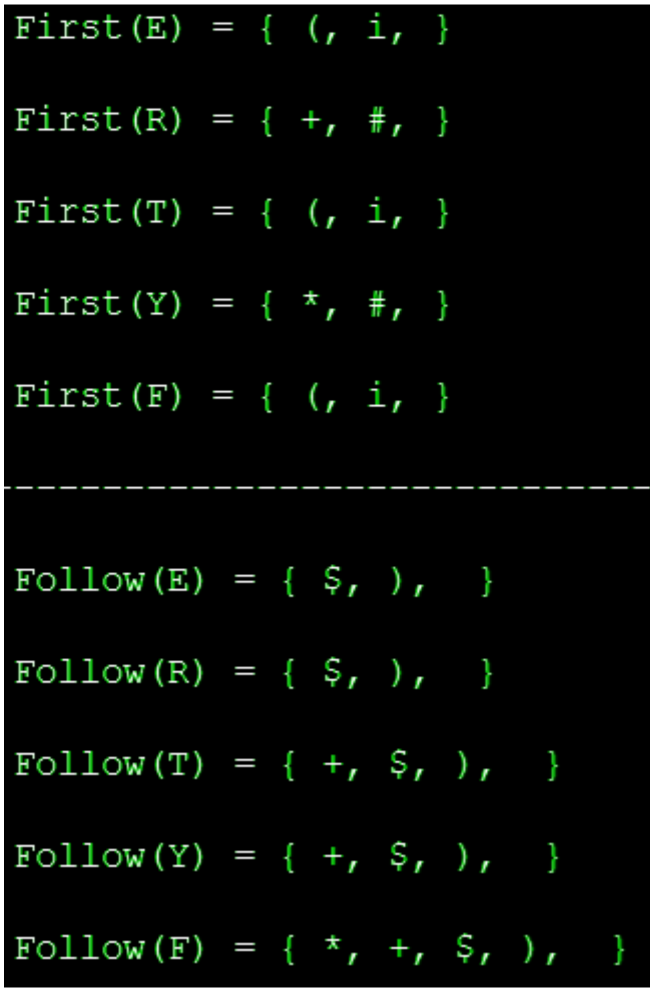
```
for (kay = 0; kay <= ptr; kay++)
    if (ck == donee[kay])
        xxx = 1;

if (xxx == 1)
    continue;
land += 1;

follow(ck);
ptr += 1;

donee[ptr] = ck;
printf(" Follow(%c) = { ", ck);
calc_follow[point1][point2++] = ck;

for (i = 0 + km; i < m; i++) {
    int lark = 0, chk = 0;
    for (lark = 0; lark < point2; lark++) {
        if (f[i] == calc_follow[point1][lark]) {
            chk = 1;
            break;
        }
    }
    if (chk == 0) {
        printf("%c, ", f[i]);
        calc_follow[point1][point2++] = f[i];
```

```
            }
        }
        printf(" }\n\n");
        km = m;
        point1++;
    }
}

void follow(char c) {
    int i, j;

    if (production[0][0] == c) {
        f[m++] = '$';
    }
    for (i = 0; i < 10; i++) {
        for (j = 2; j < 10; j++) {
            if (production[i][j] == c) {
                if (production[i][j + 1] != '\0') {
                    followfirst(production[i][j + 1], i, (j + 2));
                }

                if (production[i][j + 1] == '\0' && c != production[i][0]) {
                    follow(production[i][0]);
                }
            }
        }
    }
```

```
}

void findfirst(char c, int q1, int q2) {
    int j;

    if (!(isupper(c))) {
        first[n++] = c;
    }
    for (j = 0; j < count; j++) {
        if (production[j][0] == c) {
            if (production[j][2] == '#') {
                if (production[q1][q2] == '\0')
                    first[n++] = '#';
                else if (production[q1][q2] != '\0' &&
                    (q1 != 0 || q2 != 0)) {
                    findfirst(production[q1][q2], q1, (q2 + 1));
                } else
                    first[n++] = '#';
            } else if (!isupper(production[j][2])) {
                first[n++] = production[j][2];
            } else {
                findfirst(production[j][2], j, 3);
            }
        }
    }
}
```

```
void followfirst(char c, int c1, int c2) {
    int k;

    if (!(isupper(c)))
        f[m++] = c;
    else {
        int i = 0, j = 1;
        for (i = 0; i < count; i++) {
            if (calc_first[i][0] == c)
                break;
        }

        while (calc_first[i][j] != '!') {
            if (calc_first[i][j] != '#') {
                f[m++] = calc_first[i][j];
            } else {
                if (production[c1][c2] == '\0') {

                    follow(production[c1][0]);
                } else {
                    followfirst(production[c1][c2], c1, c2 + 1);
                }
            }
            j++;
        }
    }
}
```

Output:



```
First(E) = { (, i, }

First(R) = { +, #, }

First(T) = { (, i, }

First(Y) = { *, #, }

First(F) = { (, i, }

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

Follow(E) = { $, ),  }

Follow(R) = { $, ),  }

Follow(T) = { +, $, ),  }

Follow(Y) = { +, $, ),  }

Follow(F) = { *, +, $, ),  }
```

Experiment-8

Aim: Design the parser which accepts string of grammar:

S -> ABC, A -> abA | ab, B -> b | BC, C -> c | cC

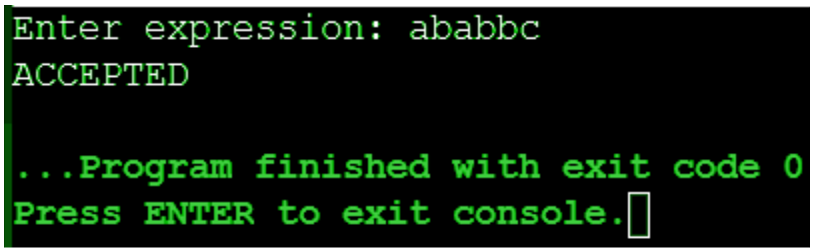
Theory:

The parser is the part of the compiler that accepts a token string as input and turns it into the matching Intermediate Representation using existing grammar. Syntax Analyzer is another name for the parser. Top-down Parser and Bottom-up Parser are the two primary types of parser.

Code:

```
#include<iostream>

#include<string.h>

using namespace std;

int i, error;

void S(char * input);

void A(char * input);

void Aprime(char * input);

void Adprime(char * input);

void B(char * input);

void Bprime(char * input);

void C(char * input);

void Cprime(char * input);

int main() {

    char input[10];

    cout << "Enter expression: ";

    cin >> input;

    i = 0;

    error = 0;
    S(input);
    if (strlen(input) == i && error == 0)
        cout << "ACCEPTED";
    else
        cout << "REJECTED";
}
void S(char * input) {
    A(input);
    B(input);
    C(input);
}
void A(char * input) {
    if (input[i] == 'a') {
        i++;
        Aprime(input);
    }
}
void Aprime(char * input) {
    if (input[i] == 'b') {
        i++;
        Adprime(input);
    }
}
void Adprime(char * input) {
    A(input);
}
void B(char * input) {
    if (input[i] == 'b') {
        i++;
        Bprime(input);
    }
}
void Bprime(char * input) {
    C(input);
}
void C(char * input) {
    if (input[i] == 'c') {
        i++;
        Cprime(input);
    }
}
void Cprime(char * input) {
    C(input);
}
```

Output:



```
Enter expression: ababbc
ACCEPTED

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

Experiment-9

Aim: Write a program to compute Leading and Trailing for the given grammar

Theory:

Leading:

If production is of form A → aα or A → Ba α where B is Non-terminal, and α can be any string, then the first terminal symbol on R.H.S is

Leading(A) = {a}

If production is of form A → Bα, if a is in LEADING (B), then a will also be in LEADING (A).

Trailing:

If production is of form A→ αa or A → αaB where B is Non-terminal, and α can be any string then,

Trailing(A) = {a}

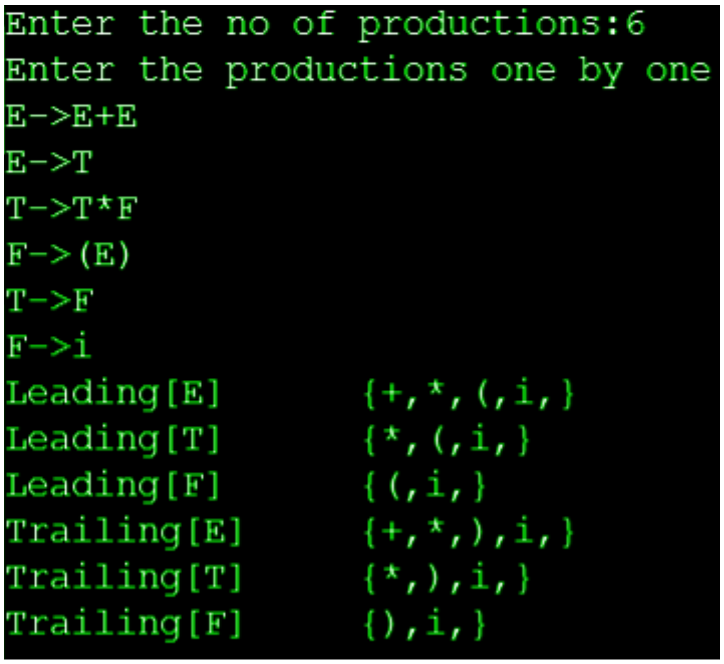
Code:

```
#include<iostream>

#include<string.h>

using namespace std;

int nt, t, top = 0;

char s[50], NT[10], T[10], st[50], l[10][10], tr[50][50];

int searchnt(char a) {

   int count = -1, i;

   for (i = 0; i < nt; i++) {

      if (NT[i] == a)

         return i;

   }
```

```
    return count;
}
int searchter(char a) {
    int count = -1, i;
    for (i = 0; i < t; i++) {
        if (T[i] == a)
            return i;
    }
    return count;
}
void push(char a) {
    s[top] = a;
    top++;
}
char pop() {
    top--;
    return s[top];
}
void installl(int a, int b) {
    if (l[a][b] == 'f') {
        l[a][b] = 't';
        push(T[b]);
        push(NT[a]);
    }
}
void installt(int a, int b) {
    if (tr[a][b] == 'f') {
        tr[a][b] = 't';
        push(T[b]);
        push(NT[a]);
    }
}
int main() {
    int i, s, k, j, n;
    char pr[30][30], b, c;
    cout << "Enter the no of productions:";
    cin >> n;
    cout << "Enter the productions one by one\n";
    for (i = 0; i < n; i++)
        cin >> pr[i];
    nt = 0;
    t = 0;
    for (i = 0; i < n; i++) {
        if ((searchnt(pr[i][0])) == -1)
            NT[nt++] = pr[i][0];
    }
    for (i = 0; i < n; i++) {
        for (j = 3; j < strlen(pr[i]); j++) {
            if (searchnt(pr[i][j]) == -1) {
                if (searchter(pr[i][j]) == -1)
                    T[t++] = pr[i][j];
            }
        }
    }
```

```
for (i = 0; i < nt; i++) {
    for (j = 0; j < t; j++)
        l[i][j] = 'f';
}
for (i = 0; i < nt; i++) {
    for (j = 0; j < t; j++)
        tr[i][j] = 'f';
}
for (i = 0; i < nt; i++) {
    for (j = 0; j < n; j++) {
        if (NT[(searchnt(pr[j][0]))] == NT[i]) {
            if (searchter(pr[j][3]) != -1)
                installl(searchnt(pr[j][0]), searchter(pr[j][3]));
            else {
                for (k = 3; k < strlen(pr[j]); k++) {
                    if (searchnt(pr[j][k]) == -1) {
                        installl(searchnt(pr[j][0]), searchter(pr[j][k]));
                        break;
                    }
                }
            }
        }
    }
}
while (top != 0) {
    b = pop();
    c = pop();
    for (s = 0; s < n; s++) {
        if (pr[s][3] == b)
            installl(searchnt(pr[s][0]), searchter(c));
    }
}
for (i = 0; i < nt; i++) {
    cout << "Leading[" << NT[i] << "]" << "\t{";
    for (j = 0; j < t; j++) {
        if (l[i][j] == 't')
            cout << T[j] << ",";
    }
    cout << "}\n";
}

top = 0;
for (i = 0; i < nt; i++) {
    for (j = 0; j < n; j++) {
        if (NT[searchnt(pr[j][0])] == NT[i]) {
            if (searchter(pr[j][strlen(pr[j]) - 1]) != -1)
                installt(searchnt(pr[j][0]), searchter(pr[j][strlen(pr[j]) - 1]));
            else {
                for (k = (strlen(pr[j]) - 1); k >= 3; k--) {
                    if (searchnt(pr[j][k]) == -1) {
                        installt(searchnt(pr[j][0]), searchter(pr[j][k]));
                        break;
                    }
                }
                }
            }
        }
    }
    while (top != 0) {
        b = pop();
        c = pop();
        for (s = 0; s < n; s++) {
            if (pr[s][3] == b)
                installt(searchnt(pr[s][0]), searchter(c));
        }
    }
    for (i = 0; i < nt; i++) {
        cout << "Trailing[" << NT[i] << "]" << "\t{";
        for (j = 0; j < t; j++) {
            if (tr[i][j] == 't')
                cout << T[j] << ",";
        }
        cout << "}\n";
    }
}
```

Output:



```
Enter the no of productions:6
Enter the productions one by one
E->E+E
E->T
T->T*F
F->(E)
T->F
F->i
Leading[E]      {+,*,(,i,}
Leading[T]      {*,(,i,}
Leading[F]      {(,i,}
Trailing[E]     {+,*,),i,}
Trailing[T]     {*,),i,}
Trailing[F]     {),i,}
```

<u>**Experiment-10**</u>

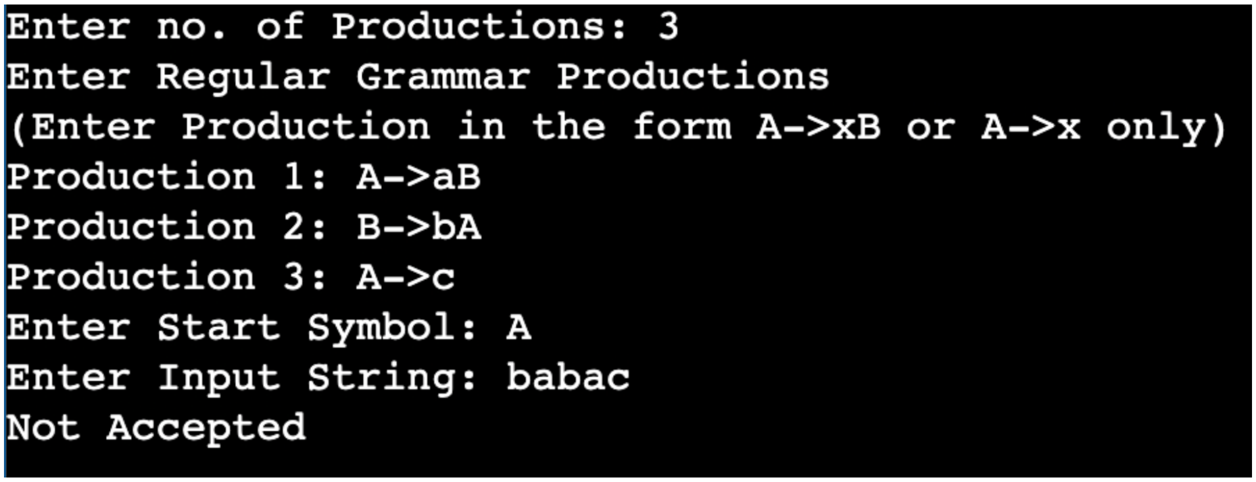
Aim: Write a program which accepts a regular grammar with no LR and no null production and check would it parse the string or not

Code:

```
#include<iostream>

#include<string.h>

using namespace std;

int num_prods;

string prods[100][100];

char nt[100];

int s = 0, error = 0;

void S(char * input);

void A(char * input);

void getProperGrammar() {

  for (int x = num_prods - 1; x > 0; x--) {

    if (prods[x][0] == prods[x - 1][0]) {

      prods[x - 1][2] = prods[x - 1][2] + " | " + prods[x][2];

      for (int l = 0; l < num_prods; l++) {

        for (int y = 0; y < 3; y++) {

          if (l == x) {

            prods[l][y] = "";

          }

        }

      }

    }

  }
```

```
}
int main() {
    cout << endl;
    cout << endl;
    cout << "Enter the number of productions: " << endl;
    cin >> num_prods;
    int i = 0;
    while (i < num_prods) {
        string lhs, rhs;
        cout << "Enter the LHS of production: " << endl;
        cin >> lhs;
        nt[i] = lhs.at(0);
        cout << "Enter the RHS of production: " << endl;
        cin >> rhs;
        if (lhs.at(0) == rhs.at(0) || rhs.at(0) == 'e') {
            cout << endl;
            cout << "ERROR: The grammar is either left recursive or contains a null
production: " << endl;
            cout << endl;
            return 0;
        }
        prods[i][0] = lhs;
        prods[i][1] = "-->";
        prods[i][2] = rhs;
        i++;
    }
    getProperGrammar();
    char input[10];
    cout << "Enter the string:" << endl;
    cin >> input;
    S(input);
    if (strlen(input) == s && error == 0)
        cout << "ACCEPTED";
    else
        cout << "REJECTED";
}
void S(char * input) {
    if (input[s] == 'c') {
        s++;
        A(input);
        if (input[s] == 'd')
            s++;
        else
            error = 1;
    } else
        error = 1;
}
void A(char * input) {
    if (input[s] == 'a' && input[s + 1] == 'b') {
        s++;
        if (input[s] == 'b')
            s++;
        else
            error = 1;
    } else if (input[s] == 'a')
        s++;
    else
        error = 1;
}
```

Output:



```
Enter no. of Productions: 3
Enter Regular Grammar Productions
(Enter Production in the form A->xB or A->x only)
Production 1: A->aB
Production 2: B->bA
Production 3: A->c
Enter Start Symbol: A
Enter Input String: babac
Not Accepted
```