1) Implement push and pop operation of stack

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

struct Stack{
   int size;
   int top;
   int *s;// pointer to array for dynamically allocating size of array at the time of creation.
};

void create(struct Stack *st){
   printf("\nEnter the size of stack: ");
   scanf("%d",&st->size);
   st->top=-1;
   st->s=(int *)malloc(st->size*sizeof(int));
}

void push(struct Stack *st,int val){
   if (st->top==st->size-1)
   {
      printf("\nOverflow Error");
   }else{
      st->top++;
      st->s[st->top]=val;
      printf("\n%d was Inserted successfully.",val);
   }
}
 void pop(struct Stack *st){
   if (st->top==-1)
   {
      printf("\nUnderflow Error");
```

```
    }else{
        st->top--;
        printf("\nPopped successfully.");
    }
}
Int main(){
            struct Stack trialStack;
            create(&trialStack);
              push(trialStack,value);
              pop(trialStack);
return 0;
}
```

2) Identify characters and digits as token in a given string:

```
#include <iostream>
#include <cctype>

int main() {
    std::string inputString;

    // Get input from the user
    std::cout << "Enter a string: ";
    std::getline(std::cin, inputString);

    // Identify characters and digits as tokens
    std::cout << "Tokens in the given string:\n";

    for (char ch : inputString) {
        if (std::isalpha(ch)) {
            std::cout << "Character: " << ch << '\n';
        } else if (std::isdigit(ch)) {
            std::cout << "Digit: " << ch << '\n';
        }
    }

    return 0;
}
```

3) Identify alphanumeric characters and operators in a given string

```
#include <iostream>
#include <cctype>
```

```
int main() {
    std::string inputString;

    // Get input from the user
    std::cout << "Enter a string: ";
    std::getline(std::cin, inputString);

    // Identify alphanumeric characters and operators
    std::cout << "Alphanumeric characters and operators in the given string:\n";

    for (char ch : inputString) {
        if (std::isalnum(ch)) {
            std::cout << "Alphanumeric character: " << ch << '\n';
        } else if (ispunct(ch) && ch != '_') {
            std::cout << "Operator: " << ch << '\n';
        }
    }

    return 0;
}
```

4) Implement infix to prefix of a given grammar

```
#include <iostream>

#include <stack>

#include <algorithm>


bool isOperator(char c) {

    return (c == '+' || c == '-' || c == '*' || c == '/');

}


int getPrecedence(char op) {

    if (op == '+' || op == '-') {

        return 1;

    } else if (op == '*' || op == '/') {

        return 2;

    }

    return 0;

}


std::string infixToPrefix(const std::string& infix) {

    std::stack<char> operators;

    std::string prefix;


    // Reverse the infix expression

    std::string reversedInfix = infix;

    std::reverse(reversedInfix.begin(), reversedInfix.end());


    for (char ch : reversedInfix) {

        if (isalnum(ch)) {

            // Operand

            prefix += ch;

        } else if (ch == ')') {
            // Right parenthesis
            operators.push(ch);
        } else if (ch == '(') {
            // Left parenthesis
            while (!operators.empty() && operators.top() != ')') {
                prefix += operators.top();
                operators.pop();
            }
            operators.pop(); // Pop the corresponding ')'
        } else if (isOperator(ch)) {
            // Operator
            while (!operators.empty() && getPrecedence(operators.top()) >= getPrecedence(ch)) {
                prefix += operators.top();
                operators.pop();
            }
            operators.push(ch);
        }
    }

    // Pop any remaining operators from the stack
    while (!operators.empty()) {
        prefix += operators.top();
        operators.pop();
    }

    // Reverse the final result to get the prefix expression
    std::reverse(prefix.begin(), prefix.end());

    return prefix;
}
```

```cpp
int main() {
    std::string infixExpression;

    // Get input from the user
    std::cout << "Enter an infix expression: ";
    std::getline(std::cin, infixExpression);

    // Convert infix to prefix
    std::string prefixExpression = infixToPrefix(infixExpression);

    // Display the result
    std::cout << "Prefix expression: " << prefixExpression << std::endl;

    return 0;
}
```

5) Implement prefix to postfix for the given grammar

```
#include <iostream>
#include <stack>
#include <algorithm>

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

int getPrecedence(char op) {
    if (op == '+' || op == '-') {
        return 1;
    } else if (op == '*' || op == '/') {
        return 2;
    }
    return 0;
}

std::string infixToPostfix(const std::string& infix) {
    std::stack<char> operators;
    std::string postfix;

    for (char ch : infix) {
        if (isalnum(ch)) {
            // Operand
            postfix += ch;
        } else if (ch == '(') {
            // Left parenthesis
            operators.push(ch);
        } else if (ch == ')') {
            // Right parenthesis
            while (!operators.empty() && operators.top() != '(') {
                postfix += operators.top();
                operators.pop();
            }
            operators.pop(); // Pop the corresponding '('
        } else if (isOperator(ch)) {
            // Operator
            while (!operators.empty() && getPrecedence(operators.top()) >= getPrecedence(ch)) {
                postfix += operators.top();
                operators.pop();
            }
            operators.push(ch);
        }
    }

    // Pop any remaining operators from the stack
    while (!operators.empty()) {
        postfix += operators.top();
        operators.pop();
    }

    return postfix;
}

int main() {
    std::string infixExpression;

    // Get input from the user
    std::cout << "Enter an infix expression: ";
    std::getline(std::cin, infixExpression);
    // Convert infix to postfix
    std::string postfixExpression = infixToPostfix(infixExpression);

    // Display the result
    std::cout << "Postfix expression: " << postfixExpression << std::endl;

    return 0;
}
```

6) Find FIRST of non-terminals

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

void calc_first(int x,int y,int N,vector<char> &first,vector<vector<char>> first_matrix,vector<string>
v){
    if(y<N && v[x][y]!='|' && !(v[x][y]>='A' && v[x][y]<='Z')){
        first.push_back(v[x][y]);
    }
    if(y<N && (v[x][y]>='A' && v[x][y]<='Z')){
        int n=first_matrix.size();
        for(int i=0;i<n;i++){
            if(first_matrix[i][0]==v[x][y]){
                int m=first_matrix[i].size();
                for(int j=1;j<m;j++){
                    if(first_matrix[i][j]=='^'){
                        if(y<N-1 && v[x][y+1]!='|'){
                            calc_first(x,y+1,N,first,first_matrix,v);
                        }
                        else{
                            first.push_back('^');
                        }
                    }
                    else{
                        first.push_back(first_matrix[i][j]);
                    }
                }
            }
        }
    }
}

void remove_dulpicates(int n,vector<vector<char>> &first_matrix){
    for(int i=0;i<n;i++){
        sort(first_matrix[i].begin()+1,first_matrix[i].end());
        cout<<"\nFirst("<<first_matrix[i][0]<<") = {";
        int m=first_matrix[i].size();
        for(int j=1;j<m;j++){
            if(first_matrix[i][j]!=first_matrix[i][j-1]){
                cout<<first_matrix[i][j]<<",";
            }
        }
        if(m>1){
            cout<<"\b}";
        }
        else{
            cout<<"}";
        }
    }
    return;
}

int main(){
    vector<string> productions;
    string s;
    cout<<"Enter the grammar (use ^ as null character)\nType 'end' to stop entering
productions\n\n";
    while(true){
        cin>>s;
        if(s=="end"){
            break;
        }
        if(s[0]<'A' || s[0]>'Z' || s[1]!='-' || s[2]!='>'){
            cout<<"Invalid grammar";
            break;
        }
        productions.push_back(s);
    }
    int n=productions.size();
    vector<vector<char>> first_matrix,first_matrix2;
    vector<char> first;
    for(int i=n-1;i>-1;i--){
        first.push_back(productions[i][0]);
        int m=productions[i].size();
        for(int j=1;j<m;j++){
            if(productions[i][j]=='>' || productions[i][j]=='|'){
                calc_first(i,j+1,m,first,first_matrix,productions);
            }
        }
        first_matrix.push_back(first);
        first.clear();
    }
    for(int i=n-1;i>-1;i--){
        first.push_back(productions[i][0]);
        int m=productions[i].size();
        for(int j=1;j<m;j++){
            if(productions[i][j]=='>' || productions[i][j]=='|'){
                calc_first(i,j+1,m,first,first_matrix,productions);
            }
        }
        first_matrix[n-i-1]=first;
        first.clear();
    }
    remove_dulpicates(n,first_matrix);
    return 0;
}
```

7) Remove left factoring

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

class grammar{
public:
    vector<vector<string>> table;
    grammar(vector<string> s);
    void show_table(){
        int n=table.size();
        for(int i=0;i<n;i++){
            int m=table[i].size();
            for(int j=0;j<m;j++){
                cout<<table[i][j]<<" ";
            }
            cout<<"\n";
        }
    }
};
grammar::grammar(vector<string> s){
    vector<string> v;
    int n=s.size();
    for(int i=0;i<n;i++){
        int j=0;
        int l=0;
        while(j<s[i].size()){
            if(s[i][j]=='-'){
                v.push_back(s[i].substr(l,j-l));
                l=j+2;
            }
            else if(s[i][j]=='|'){
                v.push_back(s[i].substr(l,j-l));
                l=j+1;
            }
            j++;
        }
        if(j-l>0){
            v.push_back(s[i].substr(l,j-l));
        }
        table.push_back(v);
        v.clear();
    }
}

void remove_lf(int x,vector<string> &productions,int &comma){
    int m=productions[x].size();
    vector<string> prod_rule;
    for(int j=0;j<m;j++){
        if(productions[x][j]=='>' || productions[x][j]=='|'){
```

```
        j++;
        int k=j;
        while(productions[x][k]!='|' && k<m){
            k++;
        }
        if(k-j>0){
            prod_rule.push_back(productions[x].substr(j,k-j));
        }
    }
}
sort(prod_rule.begin(),prod_rule.end());
int k=0;
while(productions[x][k]!='-'){
    k++;
}
string s1=productions[x].substr(0,k);
s1+="->";
int n=prod_rule.size();
if(n==1){
    s1.append(prod_rule[0]);
}
else{
    for(int i=0;i<n;i++){
        if(i<n-1 && prod_rule[i][0]!=prod_rule[i+1][0]){
            s1.append(prod_rule[i]);
            s1+="|";
        }
        else{
            if(i==n-1){
                if(prod_rule[i][0]!=prod_rule[i-1][0]){
                    s1.append(prod_rule[i]);
                }
                break;
            }
            s1+=prod_rule[i][0];
            s1+=productions[x].substr(0,k);
            string s2=productions[x].substr(0,k);
            for(int j=0;j<comma;j++){

            }
            s2+="'";
            s1+="'";
            s1+="|";
            s2+="->";
            char c=prod_rule[i][0];
            while(i<n && prod_rule[i][0]==c){
                if(prod_rule[i].size()>1){
                    s2.append(prod_rule[i].substr(1));
                }
                else{
                    s2+="^";
                }
                s2+="|";
                i++;
            }
            productions.insert(productions.begin()+x+1,s2);
            comma++;
            i--;
        }
    }
  }
  if(s1.back()=='|'){
    s1.pop_back();
  }
  if(s1!=productions[x]){
    productions.insert(productions.begin()+x+1,s1);
    productions.erase(productions.begin()+x);
  }
}

int main(){
  vector<string> productions;
  string s;
  cout<<"Enter the grammar (use ^ as null character)\nType 'end' to stop entering
productions\n\n";
  while(true){
    cin>>s;
    if(s=="end"){
      break;
    }
    if(s[0]<'A' || s[0]>'Z' || s[1]!='-' || s[2]!='>'){
      cout<<"Invalid grammar";
      return 0;
    }
    productions.push_back(s);
  }
  //int n=productions.size();
  int comma=1;
  for(int i=0;i<productions.size();i++){
    remove_lf(i,productions,comma);
  }
  cout<<"\nGrammar after left factoring:\n";
  int n=productions.size();
  for(int i=0;i<n;i++){
    int m=productions[i].size();
    for(int j=0;j<m;j++){
      cout<<productions[i][j];
    }
    cout<<"\n";
  }
  return 0;
}
```

8) Remove left recursion

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

void remove_lr(int x,vector<string> &productions){
   int m=productions[x].size();
   vector<string> alpha,beta;
   for(int j=0;j<m;j++){
      if(productions[x][j]=='>' || productions[x][j]=='|'){
         j++;
         if(productions[x][j]==productions[x][0]){
            int k=j+1;
            while(productions[x][k]!='|' && k<m){
               k++;
            }
            if(k-j-1>0){
               alpha.push_back(productions[x].substr(j+1,k-j-1));
            }
         }
         else{
            int k=j;
            while(productions[x][k]!='|' && k<m){
               k++;
            }
            if(k-j>0){
               beta.push_back(productions[x].substr(j,k-j));
            }
         }
      }
   }
   if(alpha.size()>0){
      string s1;
      s1=productions[x][0];
      s1+="'->";
      int n=alpha.size();
      for(int i=0;i<n;i++){
         s1.append(alpha[i]);
         s1+=productions[x][0];
         s1+="'|";
      }
      s1+="^";

      string s2=productions[x].substr(0,3);
      n=beta.size();
      for(int i=0;i<n;i++){
         s2.append(beta[i]);
         s2+=productions[x][0];
         s2+="'";
         if(i!=n-1){
                s2+="|";
            }
        }
        //productions.insert(productions.begin()+x+1,s2);
        //productions.insert(productions.begin()+x+1,s1);
        if(n>0){
            productions.push_back(s2);
        }
        productions.push_back(s1);
        productions.erase(productions.begin()+x);
    }
}

int main(){
    vector<string> productions;
    string s;
    cout<<"Enter the grammar (use ^ as null character)\nType 'end' to stop entering
productions\n\n";
    while(true){
        cin>>s;
        if(s=="end"){
            break;
        }
        if(s[0]<'A' || s[0]>'Z' || s[1]!='-' || s[2]!='>'){
            cout<<"Invalid grammar";
            return 0;
        }
        productions.push_back(s);
    }
    int n=productions.size();
    for(int i=n-1;i>-1;i--){
        remove_lr(i,productions);
    }
    cout<<"\nGrammar after removing left recursion:\n";
    n=productions.size();
    for(int i=0;i<n;i++){
        int m=productions[i].size();
        for(int j=0;j<m;j++){
            cout<<productions[i][j];
        }
        cout<<"\n";
    }
    return 0;
}
```