first follow:

```
 #include <bits/stdc++.h>
using namespace std;

map<char, set<char>> firstSet, followSet;
map<char, vector<string>> productions;
char startSymbol;


void computeFirst(char symbol) {
    if (!firstSet[symbol].empty()) return;

    for (const string &rule : productions[symbol]) {
        for (size_t i = 0; i < rule.size(); i++) {
            char current = rule[i];
            if (islower(current) || current == '(' || current == ')') {
                firstSet[symbol].insert(current);
                break;
            } else if (isupper(current)) {
                computeFirst(current);
                firstSet[symbol].insert(firstSet[current].begin(), firstSet[current].end());
                if (!firstSet[current].count('#')) break;
            } else if (current == '#') {
                firstSet[symbol].insert('#');
                break;
            }
        }
    }
}


void computeFollow(char symbol) {
    for (auto &[nt, rules] : productions) {
        for (const string & rule : rules) {
            for (size_t i = 0; i < rule.size(); i++) {
                if (rule[i] == symbol) {
                    if (i + 1 < rule.size()) {
                        char next = rule[i + 1];
                        if (islower(next) || next == '(' || next == ')') {
                            followSet[symbol].insert(next);
                        } else {
                            followSet[symbol].insert(firstSet[next].begin(), firstSet[next].end());
                            followSet[symbol].erase('#');
                            if (firstSet[next].count('#')) {
                                followSet[symbol].insert(followSet[nt].begin(), followSet[nt].end());
                            }
                        }
```

```
                } else {
                    followSet[symbol].insert(followSet[nt].begin(), followSet[nt].end());
                }
            }
        }
    }
}

int main() {
    int numProductions;
    cout << "Enter the number of productions: ";
    cin >> numProductions;

    cout << "Enter productions (e.g., S->AB|a):" << endl;
    for (int i = 0; i < numProductions; i++) {
        string input;
        cin >> input;
        char nonTerminal = input[0];
        string rules = input.substr(3);
        stringstream ss(rules);
        string rule;
        while (getline(ss, rule, '|')) {
            productions[nonTerminal].push_back(rule);
        }
        if (i == 0) startSymbol = nonTerminal;
    }


    for (auto &[nonTerminal, _] : productions) {
        computeFirst(nonTerminal);
    }


    followSet[startSymbol].insert('$');


    bool changed = true;
    while (changed) {
        changed = false;
        map<char, set<char>> oldFollowSet = followSet;

        for (auto &[nonTerminal, _] : productions) {
            computeFollow(nonTerminal);
        }

        if (followSet != oldFollowSet) {
            changed = true;
```

```
        }
    }


    cout << "\nFirst sets:" << endl;
    for (auto &[symbol, set] : firstSet) {
        cout << symbol << ": { ";
        for (char c : set) cout << c << " ";
        cout << "}" << endl;
    }


    cout << "\nFollow sets:" << endl;
    for (auto &[symbol, set] : followSet) {
        cout << symbol << ": { ";
        for (char c : set) cout << c << " ";
        cout << "}" << endl;
    }

    return 0;
}
```

Input: 3
S->AB|a
A->a|#
B->b|#

Output: First sets:
S: { a b # }
A: { a # }
B: { b # }

Follow sets:
S: { $ }
A: { b }
B: { $ }

Left Recursion Elimination

```
#include <iostream>
#include <vector>
```

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

int main()
{
   int n;
   cout<<"\nEnter number of non terminals: ";
   cin>>n;
   cout<<"\nEnter non terminals one by one: ";
   int i;
   vector<string> nonter(n);
   vector<int> leftrecr(n,0);
   for(i=0;i<n;++i) {
         cout<<"\nNon terminal "<<i+1<<" : ";
      cin>>nonter[i];
   }
   vector<vector<string> > prod;
   cout<<"\nEnter '^' for null";
   for(i=0;i<n;++i) {
      cout<<"\nNumber of "<<nonter[i]<<" productions: ";
      int k;
      cin>>k;
      int j;
      cout<<"\nOne by one enter all "<<nonter[i]<<" productions";
      vector<string> temp(k);
      for(j=0;j<k;++j) {
         cout<<"\nRHS of production "<<j+1<<": ";
         string abc;
         cin>>abc;
         temp[j]=abc;

if(nonter[i].length()<=abc.length()&&nonter[i].compare(abc.substr(0,nonter[i].length()))==0)
               leftrecr[i]=1;
      }
      prod.push_back(temp);
   }
   for(i=0;i<n;++i) {
      cout<<leftrecr[i];
   }
   for(i=0;i<n;++i) {
      if(leftrecr[i]==0)
         continue;
      int j;
      nonter.push_back(nonter[i]+"'");
      vector<string> temp;
      for(j=0;j<prod[i].size();++j) {
```

```
if(nonter[i].length()<=prod[i][j].length()&&nonter[i].compare(prod[i][j].substr(0,nonter[i].length()
))==0) {
                string
abc=prod[i][j].substr(nonter[i].length(),prod[i][j].length()-nonter[i].length())+nonter[i]+"'";
                temp.push_back(abc);
                prod[i].erase(prod[i].begin()+j);
                --j;
            }
            else {
                prod[i][j]+=nonter[i]+"'";
            }
        }
        temp.push_back("^");
        prod.push_back(temp);
    }
    cout<<"\n\n";
    cout<<"\nNew set of non-terminals: ";
    for(i=0;i<nonter.size();++i)
        cout<<nonter[i]<<" ";
    cout<<"\n\nNew set of productions: ";
    for(i=0;i<nonter.size();++i) {
        int j;
        for(j=0;j<prod[i].size();++j) {
            cout<<"\n"<<nonter[i]<<" -> "<<prod[i][j];
        }
    }
    return 0;
}
```

Input: 1
Non-terminal 1: A
Number of A productions: 2
RHS of production 1: Aa
RHS of production 2: b

Output: New set of non-terminals: A A'
New set of productions:
A -> bA'
A' -> aA' | ^

Left Factoring:

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

int main()
{
   char gram[20], part_1[20], part_2[20], modified_Gram[20], new_Gram[20],
temp_Gram[20];
   int i, j = 0, k = 0, l = 0, pos;

   printf("Enter the Production: A->");
   fgets(gram, sizeof(gram), stdin);
   gram[strcspn(gram, "\n")] = 0; // Remove trailing newline if present

   for (i = 0; gram[i] != '|'; i++, j++)
      part_1[j] = gram[i];
   part_1[j] = '\0';

   for (j = ++i, i = 0; gram[j] != '\0'; j++, i++)
      part_2[i] = gram[j];
   part_2[i] = '\0';

   for (i = 0; i < strlen(part_1) || i < strlen(part_2); i++) {
      if (part_1[i] == part_2[i]) {
         modified_Gram[k] = part_1[i];
         k++;
         pos = i + 1;
      }
   }

   for (i = pos, j = 0; part_1[i] != '\0'; i++, j++) {
      new_Gram[j] = part_1[i];
   }
   new_Gram[j++] = '|';

   for (i = pos; part_2[i] != '\0'; i++, j++) {
      new_Gram[j] = part_2[i];
   }

   modified_Gram[k] = 'X';
   modified_Gram[++k] = '\0';
   new_Gram[j] = '\0';
```

```
    printf("\nGrammar Without Left Factoring is: \n");
    printf(" A->%s", modified_Gram);
    printf("\n X->%s\n", new_Gram);

    return 0;
}
```

Input: A->ab|ac
Output: Grammar Without Left Factoring is:
A->aX
X->b|c

Parser:

```
#include <bits/stdc++.h>
using namespace std;

bool isProduction(const string &str) {
    if (str == "E+E" || str == "E*E" || str == "id") {
        return true;
    }
    return false;
}

bool reduce(stack<string> &parseStack) {
    vector<string> elements;
    while (!parseStack.empty()) {
        elements.push_back(parseStack.top());
        parseStack.pop();
    }
    reverse(elements.begin(), elements.end());

    for (size_t i = 0; i < elements.size(); ++i) {
        string candidate;

        for (size_t j = i; j < elements.size(); ++j) {
            candidate += elements[j];
            if (isProduction(candidate)) {

                elements.erase(elements.begin() + i, elements.begin() + j + 1);
                elements.insert(elements.begin() + i, "E");
                for (int k = elements.size() - 1; k >= 0; --k) {
                    parseStack.push(elements[k]);
```

```
            }
            cout << "Reduced: " << candidate << " → E\n";
            return true;
        }
    }
  }
  for (int i = elements.size() - 1; i >= 0; --i) {
    parseStack.push(elements[i]);
  }

  return false;
}

int main() {
  string input;
  cout << "Enter the input string: ";
  cin >> input;
  input += "$";
  stack<string> parseStack;
  int inputPtr = 0;

  cout << "\nSteps:\n";

  while (true) {
    cout << "Stack: ";
    stack<string> tempStack = parseStack;
    vector<string> stackElements;
    while (!tempStack.empty()) {
      stackElements.push_back(tempStack.top());
      tempStack.pop();
    }
    reverse(stackElements.begin(), stackElements.end());
    for (const auto &elem : stackElements) {
      cout << elem << " ";
    }
    cout << "  |  Input: " << input.substr(inputPtr) << "\n";
    if (parseStack.size() == 1 && parseStack.top() == "E" && input[inputPtr] == '$') {
      cout << "Input is successfully parsed!\n";
      break;
    }
    if (input[inputPtr] == 'i' && input[inputPtr + 1] == 'd') {
      parseStack.push("id");
      inputPtr += 2;
      cout << "Shifted: id\n";
    } else if (input[inputPtr] == '+' || input[inputPtr] == '*') {
      parseStack.push(string(1, input[inputPtr]));
      inputPtr++;
      cout << "Shifted: " << parseStack.top() << "\n";
```

```
        } else if (input[inputPtr] == '$') {

            if (!reduce(parseStack)) {
                cout << "Parsing failed at end marker!\n";
                break;
            }
        } else {

            if (!reduce(parseStack)) {
                cout << "Parsing failed!\n";
                break;
            }
        }
    }

    return 0;
}
```

Input: id+id*id
Output:
Steps:
Stack: | Input: id+id*id$
Shifted: id
Stack: id | Input: +id*id$
Shifted: +
Stack: id + | Input: id*id$
Shifted: id
Stack: id + id | Input: *id$
Reduced: id → E
Stack: E + | Input: *id$
Shifted: *
Stack: E + * | Input: id$
Shifted: id
Stack: E + * id | Input: $
Reduced: id → E
Reduced: E * E → E
Reduced: E + E → E
Input is successfully parsed!