# COMPILER DESIGN CO-302
# LAB FILE

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING DELHI TECHNOLOGICAL UNIVERSITY**

**(Formerly Delhi College of Engineering)**

**Bawana Road, Delhi- 110042**

**SUBMITTED BY:**
Saurav Chaurasiya
(2K22/CO/421)

**SUBMITTED TO:**
Prof. Pooja Gupta

## INDEX

## Experiment - 1

**Aim:** Write a program to scan and count the number of characters, words and lines in a file.

**Theory:**

When working with text files in programming, analyzing the file's content to gather basic statistics is a common task. These statistics typically include:

- **Characters**: This refers to every single character in the file, including letters, digits, punctuation marks, whitespaces, and newline characters. It provides a measure of the total data in the file.
- **Words**: A word is defined as a sequence of characters separated by spaces or newline characters. Counting words helps in estimating the content size and understanding the structure of the text.
- **Lines**: Each line in a file ends with a newline character (\n). Counting lines gives insight into how the content is organized, especially in formatted documents or code files.

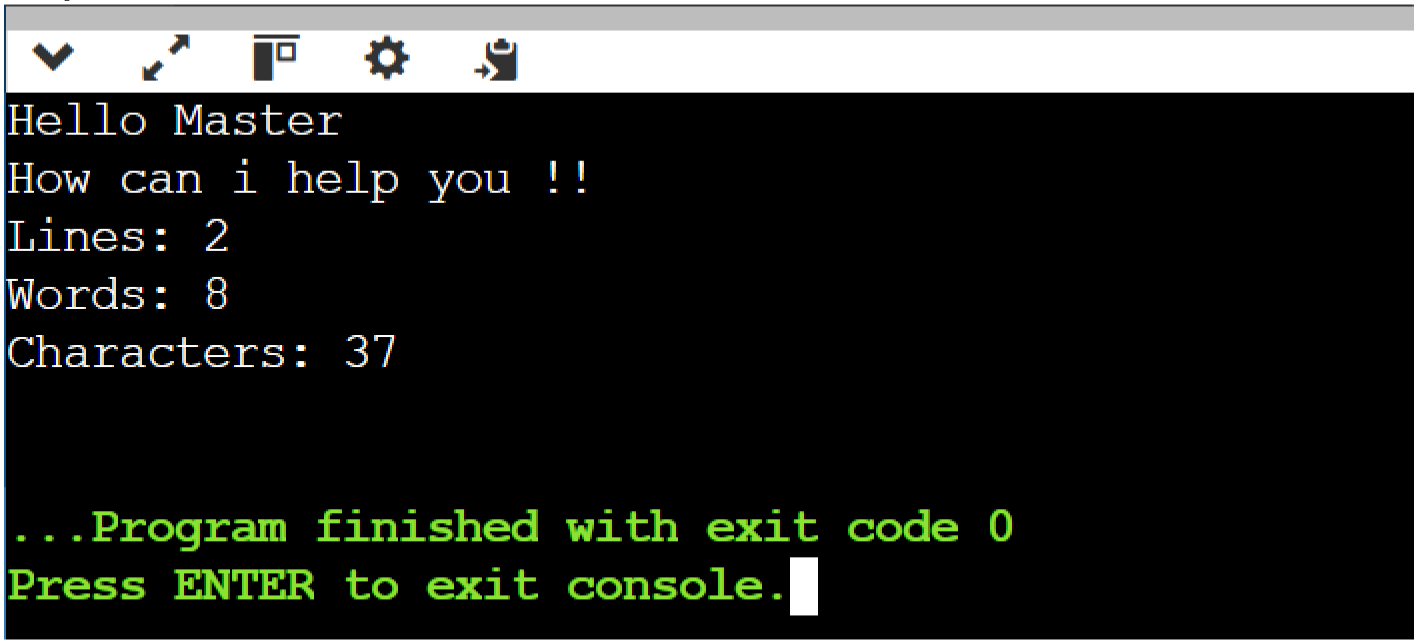
In C++, file handling is efficiently performed using the **ifstream** class, which allows us to read data from a file. To analyze the content, we open the file using ifstream and read it line by line using the getline() function. Each line is processed to update the line count and character count (including the newline character). To count words, we use the stringstream class to break each line into words.
This process provides a simple yet effective way to extract basic file statistics, which can be useful in applications such as word processors, compilers, or data parsers.

**Code:**

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

int main() {
  ifstream file("input.txt");
  if (!file) {
    cerr << "File could not be opened." << endl;
    return 1;
  }
```

```
    string line;
    int lineCount = 0, wordCount = 0, charCount = 0;

    while (getline(file, line)) {
      lineCount++;
      charCount += line.length() + 1; // +1 for newline character
      stringstream ss(line);
      string word;
      while (ss >> word) wordCount++;
    }

    file.close();

    cout << "Lines: " << lineCount << endl;
    cout << "Words: " << wordCount << endl;
    cout << "Characters: " << charCount << endl;

    return 0;
}
```

**Output:**



```
Hello Master
How can i help you !!
Lines: 2
Words: 8
Characters: 37


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

**Learning Outcomes:**

## Experiment - 2

**Aim:** Write a program to convert NFA to DFA.

**Theory:**

An **NFA (Non-deterministic Finite Automaton)** allows multiple transitions for the same input symbol from a given state, and may include ε (epsilon) transitions. A **DFA (Deterministic Finite Automaton)**, on the other hand, has **exactly one transition per input symbol per state**.

To convert an NFA to a DFA, we use the **subset construction method**, also known as the **powerset construction**. The idea is to treat **each DFA state as a set of NFA states** and build a transition table accordingly.

**Key Steps:**

1. Start with the **epsilon closure** of the initial state.
2. For each input symbol, compute the **set of NFA states** reachable from the current set of states.
3. Continue until all reachable subsets are processed.
4. Mark DFA states that contain NFA final states as **accepting**.

**Code:**

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

int nfa[10][2][10];
int nfa_count[10][2];
set<set<int>> dfa_states;
map<set<int>, int> dfa_state_ids;
int dfa[100][2];
int dfa_final[100];
int dfa_state_count = 0;

set<int> get_next_set(set<int> current, int input) {
   set<int> result;
   for (int state : current) {
      for (int i = 0; i < nfa_count[state][input]; i++) {
         result.insert(nfa[state][input][i]);
      }
   }
   return result;
```

```
}

int main() {
   int n, final_state;
   cout << "Enter number of NFA states: ";
   cin >> n;

   cout << "Enter transitions (format: state input next_state). Enter -1 to stop:\n";
   while (true) {
      int from, input, to;
      cin >> from;
      if (from == -1) break;
      cin >> input >> to;
      nfa[from][input][nfa_count[from][input]++] = to;
   }
   cout << "Enter final state of NFA: ";
   cin >> final_state;

   queue<set<int>> q;
   set<int> start = {0};
   dfa_states.insert(start);
   dfa_state_ids[start] = dfa_state_count++;
   q.push(start);

   while (!q.empty()) {
      set<int> current = q.front();
      q.pop();
      int current_id = dfa_state_ids[current];

      for (int i = 0; i < 2; i++) {
         set<int> next = get_next_set(current, i);
         if (next.empty()) continue;
         if (dfa_state_ids.find(next) == dfa_state_ids.end()) {
            dfa_state_ids[next] = dfa_state_count++;
            dfa_states.insert(next);
            q.push(next);
         }
         dfa[current_id][i] = dfa_state_ids[next];
      }
      if (current.find(final_state) != current.end()) {
         dfa_final[current_id] = 1;
      }
```

```
    }
    cout << "\nDFA Transition Table:\n";
    for (int i = 0; i < dfa_state_count; i++) {
        cout << "State " << i << " -- 0 --> " << dfa[i][0] << " , 1 --> " << dfa[i][1];
        if (dfa_final[i]) cout << " (Final)";
        cout << endl;
    }

    return 0;
}
```

**Output:**

```
Enter number of NFA states: 3
Enter transitions (format: state input next_state). Enter -1 to stop:
0 0 0
0 1 0
0 1 1
1 0 2
-1
Enter final state of NFA: 2

DFA Transition Table:
State 0 -- 0 --> 0 , 1 --> 1
State 1 -- 0 --> 2 , 1 --> 1
State 2 -- 0 --> 0 , 1 --> 1 (Final)
```

**Learning Outcome:**

# Experiment - 3

**Aim:** Write a program for acceptance of strings by DFA.

**Theory:**

A Deterministic Finite Automaton (DFA) is a computational model used to recognize patterns within input strings. It operates by transitioning through a series of defined states based on the symbols it reads from the input. A DFA consists of:

- A limited collection of states.
- A defined set of input characters (also called the alphabet).
- A transition function that takes a current state and an input symbol and returns exactly one next state.
- A designated starting state where processing begins.
- One or more accepting (or final) states.

The fundamental characteristic of a DFA is that for each state and input symbol, there exists exactly one valid transition. This deterministic nature ensures a single unique path of computation for each input string.

When a DFA processes a string, it reads each character in sequence, moving between states as defined by the transition function. Once the entire input is consumed, the automaton checks if it has landed in an accepting state. If so, the string is accepted; otherwise, it is rejected.

**Code:**

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

int main() {
  int n, m;
  cout << "States? "; cin >> n;
  cout << "Symbols? "; cin >> m;

  unordered_map<string, unordered_map<char, string>> trans;
  cout << "Transitions (from symbol to):" << endl;
  for (int i = 0; i < n * m; i++) {
    string from, to; char sym;
    cin >> from >> sym >> to;
    trans[from][sym] = to;
  }

  string start;
  cout << "Start state? "; cin >> start;

  int f; unordered_set<string> finals;
  cout << "Final states? "; cin >> f;
  while (f--) {
    string fs; cin >> fs;
    finals.insert(fs);
  }

  string input, state = start;
  cout << "Input string? "; cin >> input;

  for (char c : input) {
    if (trans[state].count(c)) state = trans[state][c];
    else { cout << "Rejected" << endl; return 0; }
  }

  cout << (finals.count(state) ? "Accepted" : "Rejected") << endl;
  return 0;
}
```

**Output:**

```
States? 3
Symbols? 2
Transitions (from symbol to):
q0 a q1
q0 b q0
q1 a q1
q1 b q2
q2 a q1
q2 b q0
Start state? q0
Final states? 1
q2
Input string? aab
Accepted
```

**Learning Outcomes:**

## Experiment - 4

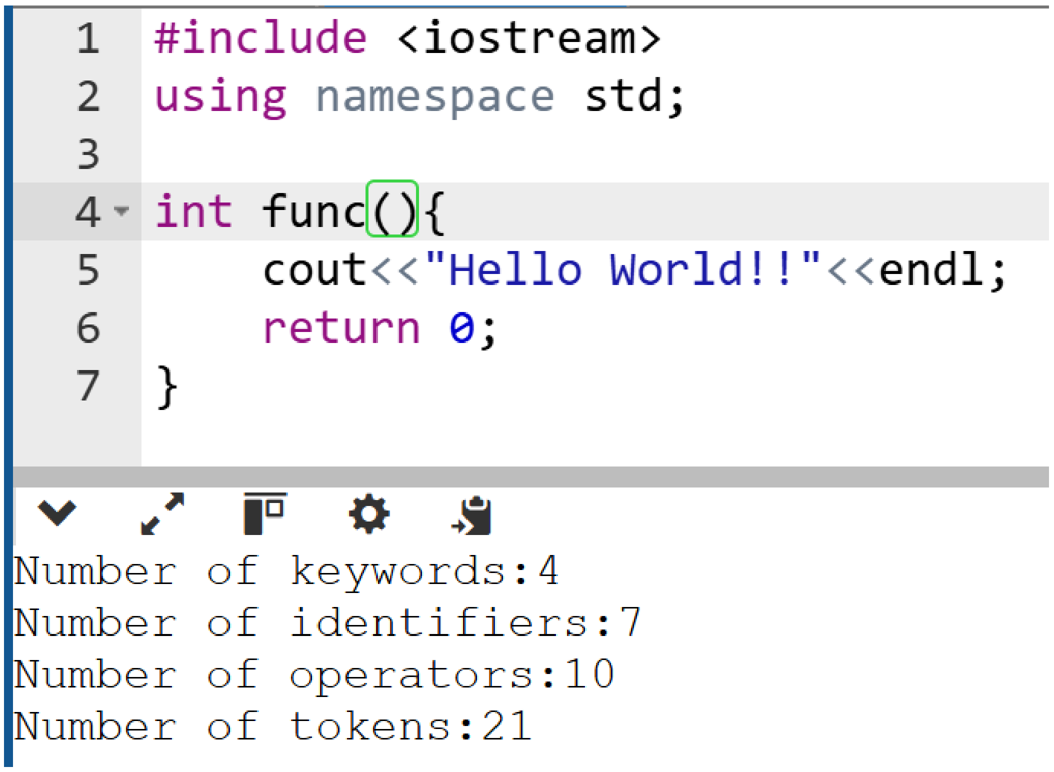
**Aim:** Write a program to find tokens in a given program.

**Theory:**
As it is known that Lexical Analysis is the first phase of compiler also known as scanner. It converts the input program into a sequence of tokens. A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal or a symbol.

**1) Keywords:** Examples-for,while,if etc.
**2) Identifier:** Examples-Variable name, function name etc.
**3) Operators:** Examples-'+','++','-'etc.

**Code:**
```
#include <bits/stdc++.h>
using namespace std;
int isKeyword(char buffer[]){
  char keywords[55][10] = {"auto", "break", "case", "char", "const", "continue", "default", "do",
"double", "else", "enum", "extern", "float", "for", "goto","if", "int", "long", "register", "return",
"short", "signed", "sizeof", "static", "struct", "switch", "typedef", "union","unsigned", "void",
"volatile", "while", "namespace", "bool", "explicit", "new", "catch", "false", "operator",
"template", "class", "friend", "private", "this", "inline", "public", "throw", "delete", "mutable",
"protected", "true", "try", "typeid", "typename", "using"};
  int i, flag = 0;
  for (i = 0; i < 55; ++i){
    if (strcmp(keywords[i], buffer) == 0){
      flag = 1; break;
    }
  }
  return flag;
}
int main(){
  char ch, buffer[15], operators[] = "+-*/%=(){,|?.><&}^~[]";
  fstream fin("test.cpp");
  int i, j = 0;
  if (!fin.is_open()){
    cout << "error while opening the file\n";
    exit(0);
  }
  int op = 0;
  int id = 0;
  int key = 0;
  while (!fin.eof()){
    ch = fin.get();
    for (i = 0; i < 19; ++i){
      if (ch == operators[i])
        op++;
    }
    if (isalnum(ch)){
      buffer[j++] = ch;
    }
    else if ((ch == ' ' || ch == '\n') && (j != 0)){
      buffer[j] = '\0';
      j = 0;
      if (isKeyword(buffer) == 1) key++;
      else id++;
    }
  }
  fin.close();
  cout << "Number of keywords:" << key << endl;
  cout << "Number of identifiers:" << id << endl;
  cout << "Number of operators:" << op << endl;
  cout << "Number of tokens:" << op+key+id << endl;
  return 0;
}
```

**Output:**



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

int func(){
     cout<<"Hello World!!"<<endl;
     return 0;
}
```

```
Number of keywords:4
Number of identifiers:7
Number of operators:10
Number of tokens:21
```

**Learning Outcome:**

# Experiment - 5

**Aim:** Write a program to implement Lexical Analyzer.

**Theory:**

A **Lexical Analyzer**, also called a **lexer** or **scanner**, is the **first phase of a compiler's front-end**. Its primary role is to read the source code character by character and **convert it into a sequence of tokens**. These tokens are the building blocks for further stages of compilation such as syntax analysis and semantic analysis.

Tokens represent logical units such as:

- **Identifiers** (e.g., variable names)
- **Keywords** (e.g., if, while, return)
- **Operators** (e.g., +, ==, &&)
- **Literals** (e.g., numeric or string constants)
- **Punctuation/Symbols** (e.g., ;, (, ))

**Working Process** of an lexical analyzer-

- The lexer **reads the source code** one character at a time.
- It **matches sequences** of characters using predefined **patterns or rules**.
- These patterns are often described using **regular expressions**, which are then converted into **finite automata** (DFA/NFA) for efficient recognition.
- When a match is found, the corresponding **token is generated** and passed to the next compiler phase (typically the parser).

**Code:**

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

bool isPunct(char ch) {
  return isspace(ch) || string("+-*/=><(){}[];,&|").find(ch) != string::npos;
}

bool isOperator(char ch) {
  return string("+-*/=><&|").find(ch) != string::npos;
}
```

```
bool isKeyword(const string& s) {
  vector<string> keywords = {
    "int", "float", "char", "double", "if", "else", "while", "do",
    "return", "void", "static", "struct", "switch", "case", "typedef",
    "enum", "union", "extern", "const", "bool", "long", "short",
    "sizeof", "continue", "break", "unsigned", "volatile", "cout"
  };
  for (auto& kw : keywords)
    if (s == kw) return true;
  return false;
}

bool isValidIdentifier(const string& s) {
  if (isdigit(s[0]) || isPunct(s[0])) return false;
  for (char ch : s)
    if (isPunct(ch)) return false;
  return true;
}

bool isNumber(const string& s) {
  int dot = 0;
  for (int i = 0; i < s.size(); i++) {
    if (s[i] == '.') dot++;
    else if (!isdigit(s[i]) && !(i == 0 && s[i] == '-')) return false;
  }
  return dot <= 1;
}

void parse(const string& line) {
  int left = 0, right = 0, len = line.length();
  while (right <= len) {
    if (!isPunct(line[right]) && right < len) right++;
    else {
      if (left == right && isOperator(line[right])) {
        cout << line[right] << " IS AN OPERATOR\n";
        right++;
        left = right;
      } else if (left != right) {
        string token = line.substr(left, right - left);
        if (isKeyword(token)) cout << token << " IS A KEYWORD\n";
        else if (isNumber(token)) cout << token << " IS A NUMBER\n";
```

```
                else if (isValidIdentifier(token)) cout << token << " IS A VALID IDENTIFIER\n";
                else cout << token << " IS NOT A VALID IDENTIFIER\n";
                left = right;
            } else {
                right++;
                left = right;
            }
        }
    }
}

int main() {
    cout<<"Enter the code to parse: ";
    string code;
    getline(cin, code);
    parse(code);
    return 0;
}
```

**Output:**

```
Enter the code to parse: int main () { int a = 0 ; cout << a << 'a' ; }
int IS A KEYWORD
main IS A VALID IDENTIFIER
int IS A KEYWORD
a IS A VALID IDENTIFIER
= IS AN OPERATOR
0 IS A NUMBER
cout IS A KEYWORD
< IS AN OPERATOR
< IS AN OPERATOR
a IS A VALID IDENTIFIER
< IS AN OPERATOR
< IS AN OPERATOR
'a' IS A VALID IDENTIFIER
```

**Learning Outcome:**

## Experiment - 6

**Aim:** Write a program to implement Recursive Descent Parser.

**Theory:**

**Recursive Descent Parsing** is a **top-down parsing technique** used in compilers to analyze and validate the syntax of source code. It starts with the start symbol of a grammar and works its way down to match tbhe input string using a set of **recursive procedures**, where each procedure typically represents a **non-terminal** in the grammar.

This method reads the input **from left to right (L2R)** and tries to derive the string using **context-free grammar** rules. The parser attempts to **match terminals directly** and calls functions for non-terminals.

In standard recursive descent, **backtracking** may be required when multiple grammar rules can match the same prefix. This happens especially when the grammar is **not left-factored**. To avoid this, a **predictive parser** (a type of recursive-descent parser that uses lookahead) is used, which can determine the correct production to use **without backtracking**, thus improving efficiency.

Recursive descent is easy to implement and understand, making it ideal for **handwritten parsers**, especially when the grammar is simple and unambiguous.

**Grammar:**

We consider the following modified arithmetic expression grammar:

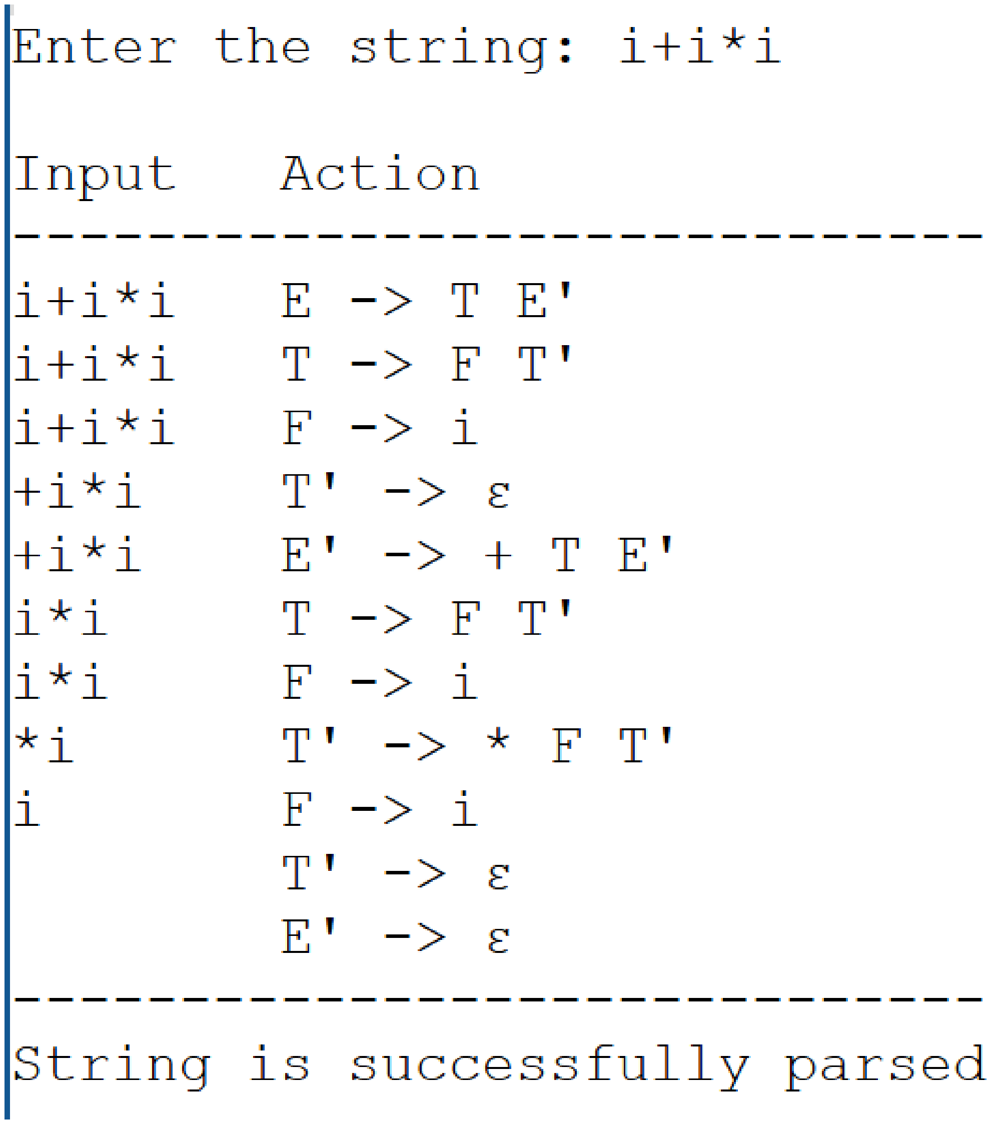
$$\begin{aligned} E &\rightarrow T\,E' \\ E' &\rightarrow +\,T\,E' \mid \varepsilon \\ T &\rightarrow F\,T' \\ T' &\rightarrow *\,F\,T' \mid \varepsilon \\ F &\rightarrow (E) \mid i \end{aligned}$$

**Code:**

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

const char *cursor;

bool E(), Edash(), T(), Tdash(), F();
bool E() {
  cout << cursor << "\tE -> T E'\n";
  return T() && Edash();
}

bool Edash() {
  if (*cursor == '+') {
    cout << cursor << "\tE' -> + T E'\n";
    cursor++;
    return T() && Edash();
  }
  cout << cursor << "\tE' -> ε\n";
  return true;
}

bool T() {
  cout << cursor << "\tT -> F T'\n";
  return F() && Tdash();
}

bool Tdash() {
  if (*cursor == '*') {
    cout << cursor << "\tT' -> * F T'\n";
    cursor++;
    return F() && Tdash();
  }
  cout << cursor << "\tT' -> ε\n";
  return true;
}

bool F() {
  if (*cursor == '(') {
    cout << cursor << "\tF -> ( E )\n";
    cursor++;
    if (E() && *cursor == ')') {
      cursor++;
      return true;
    }
    return false;
  } else if (*cursor == 'i') {
    cout << cursor << "\tF -> i\n";
    cursor++;
```

```
        return true;
    }
    return false;
}

int main() {
    char input[64];
    cout << "Enter the string: ";
    cin >> input;
    cursor = input;

    cout << "\nInput\tAction\n-----------------------------\n";
    if (E() && *cursor == '\0')
        cout << "-----------------------------\nString is successfully parsed \n";
    else
        cout << "-----------------------------\nError in parsing string\n";

    return 0;
}
```

**Output:**



```
Enter the string: i+i*i

Input   Action
-----------------------------
i+i*i   E -> T E'
i+i*i   T -> F T'
i+i*i   F -> i
+i*i    T' -> ε
+i*i    E' -> + T E'
i*i     T -> F T'
i*i     F -> i
*i      T' -> * F T'
i       F -> i
        T' -> ε
        E' -> ε
-----------------------------
String is successfully parsed
```

**Learning Outcome:**

## Experiment - 7

**Aim:** Write a program to left factor the given grammar.

**Theory:**

In compiler design, left factorization is a technique that simplifies and improves a programming language's syntax. To reduce ambiguity and duplication in the grammar, it entails detecting common prefixes in productions of the grammar and factoring them out into independent productions. This lowers the parser's complexity and increases parsing efficiency.

The following outlines the fundamentals of left factorization and how to construct a C++ programme for it:

Basic Theory of Left Factorization:

- In a context-free grammar, left factorization is the process of identifying common prefixes in the right-hand sides of production rules.
- When multiple production rules share the same prefix, it can lead to ambiguity or redundancy during parsing.
- Left factorization aims to factor out these common prefixes into separate production rules to simplify the grammar and improve parsing efficiency.
- The resulting grammar should be unambiguous and capable of generating the same language as the original grammar.

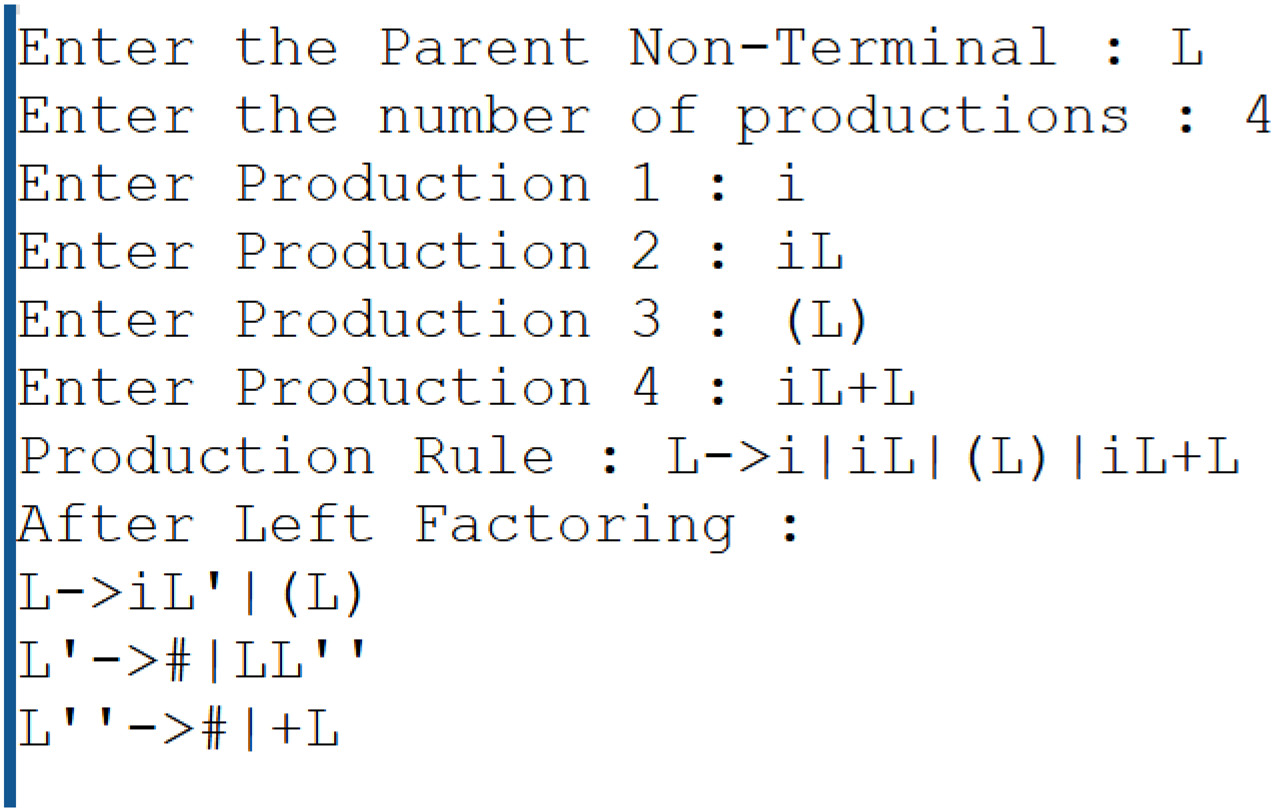
**Code:**

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

int main() {
  string ip, op1, op2, temp;
  int sizes[10] = {};
  char c;
  int n, j, l;

  cout << "Enter the Parent Non-Terminal : ";
  cin >> c;
  ip.push_back(c);
```

```
op1 += ip + "\'->";
op2 += ip + "\'\'->";
ip += "->";

cout << "Enter the number of productions : ";
cin >> n;

for (int i = 0; i < n; i++) {
  cout << "Enter Production " << i + 1 << " : ";
  cin >> temp;
  sizes[i] = temp.size();
  ip += temp;
  if (i != n - 1) ip += "|";
}

cout << "Production Rule : " << ip << endl;

char x = ip[3];
for (int i = 0, k = 3; i < n; i++) {
  if (x == ip[k]) {
    if (ip[k + 1] == '|') {
      op1 += "#";
      ip.insert(k + 1, 1, ip[0]);
      ip.insert(k + 2, 1, '\'');
      k += 4;
    } else {
      op1 += "|" + ip.substr(k + 1, sizes[i] - 1);
      ip.erase(k - 1, sizes[i] + 1);
    }
  } else {
    while (ip[k++] != '|');
  }
}

char y = op1[6];
for (int i = 0, k = 6; i < n - 1; i++) {
  if (y == op1[k]) {
    if (op1[k + 1] == '|') {
      op2 += "#";
      op1.insert(k + 1, 1, op1[0]);
      op1.insert(k + 2, 2, '\'');
      k += 5;
```

```
        } else {
          temp.clear();
          for (int s = k + 1; s < op1.length(); s++) temp.push_back(op1[s]);
          op2 += "|" + temp;
          op1.erase(k - 1, temp.length() + 2);
        }
      }
    }

    op2.erase(op2.size() - 1);
    cout << "After Left Factoring : " << endl;
    cout << ip << endl;
    cout << op1 << endl;
    cout << op2 << endl;

    return 0;
}
```

**Output:**



```
Enter the Parent Non-Terminal : L
Enter the number of productions : 4
Enter Production 1 : i
Enter Production 2 : iL
Enter Production 3 : (L)
Enter Production 4 : iL+L
Production Rule : L->i|iL|(L)|iL+L
After Left Factoring :
L->iL'|(L)
L'->#|LL''
L''->#|+L
```

**Learning Outcome:**

<u>**Experiment - 8**</u>

**Aim:** Write a program to convert left recursive grammar to right recursive grammar.

**Theory:**

Left recursion in a grammar occurs when a non-terminal appears on the left-most side of its own production. This can lead to infinite recursion in top-down parsers like recursive descent parsers.

A grammar is said to be left recursive if there exists a non-terminal A such that:

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

Where α is a sequence of grammar symbols and β does not begin with A.

To remove left recursion, we rewrite the grammar as:

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

This converts it to a right-recursive grammar which is more suitable for top-down parsers.

**Code:**

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

#define MAX_PROD 10
#define MAX_LEN 20
#define MAX_NON_TERMINALS 10

void eliminateLeftRecursion(char non_terminal, char productions[MAX_PROD][MAX_LEN],
int num_productions) {
    char new_productions[MAX_PROD][MAX_LEN];
    char recursive_productions[MAX_PROD][MAX_LEN];
    int new_count = 0, recursive_count = 0;

    for (int i = 0; i < num_productions; i++) {
        if (productions[i][0] == non_terminal) {
            strcpy(recursive_productions[recursive_count], productions[i] + 1);
            strcat(recursive_productions[recursive_count], "'");
            recursive_count++;
        } else {
```

```
        strcpy(new_productions[new_count], productions[i]);
        strcat(new_productions[new_count], "'");
        new_count++;
      }
   }

   if (recursive_count > 0) {
      cout << non_terminal << " -> ";
      for (int i = 0; i < new_count; i++) {
         cout << new_productions[i];
         if (i != new_count - 1) cout << " | ";
      }
      cout << endl;

      cout << non_terminal << "' -> ";
      for (int i = 0; i < recursive_count; i++) {
         cout << recursive_productions[i] << " | ";
      }
      cout << "epsilon\n";
   } else {
      cout << "No left recursion detected for " << non_terminal << ".\n";
   }
}

int main() {
   char non_terminals[MAX_NON_TERMINALS];
   int num_non_terminals;
   int num_productions[MAX_NON_TERMINALS];
   char productions[MAX_NON_TERMINALS][MAX_PROD][MAX_LEN];

   cout << "Enter the number of non-terminals: ";
   cin >> num_non_terminals;

   for (int i = 0; i < num_non_terminals; i++) {
      cout << "\nEnter the non-terminal " << (i + 1) << ": ";
      cin >> non_terminals[i];

      cout << "Enter the number of productions for " << non_terminals[i] << ": ";
      cin >> num_productions[i];

      cout << "Enter the productions for " << non_terminals[i] << " (use 'epsilon' for ε):\n";
      for (int j = 0; j < num_productions[i]; j++) {
            cout << non_terminals[i] << " -> ";
            cin >> productions[i][j];
        }
    }

    for (int i = 0; i < num_non_terminals; i++) {
        cout << "\nTransformed Grammar for " << non_terminals[i] << ":\n";
        eliminateLeftRecursion(non_terminals[i], productions[i], num_productions[i]);
    }

    return 0;
}
```

**Output:**

```
Enter the number of non-terminals: 2

Enter the non-terminal 1: A
Enter the number of productions for A: 3
Enter the productions for A (use 'epsilon' for ε):
A -> Aa
A -> Ab
A -> c

Enter the non-terminal 2: B
Enter the number of productions for B: 2
Enter the productions for B (use 'epsilon' for ε):
B -> Bb
B -> d

Transformed Grammar for A:
A -> c'
A' -> a' | b' | epsilon

Transformed Grammar for B:
B -> d'
B' -> b' | epsilon
```

**Learning Outcome:**

## Experiment - 9

**Aim:** Write a program to compute the following:

- First set for each of the non terminal symbols
- Follow set for each of the non terminal symbol

**Theory:**

In compiler design, **FIRST** and **FOLLOW** sets are fundamental concepts used in the construc tion of predictive parsers such as LL(1) parsers. These sets help in syntax analysis and are used to build the parser's prediction table.

- FIRST Set

  FIRST for a symbol is the set of terminals that can appear at the beginning of some string derived from that symbol.
- FOLLOW Set

  FOLLOW for a non-terminal A is the set of terminals that can appear immediately after A

in

  some sentential form during the derivation of the input string.

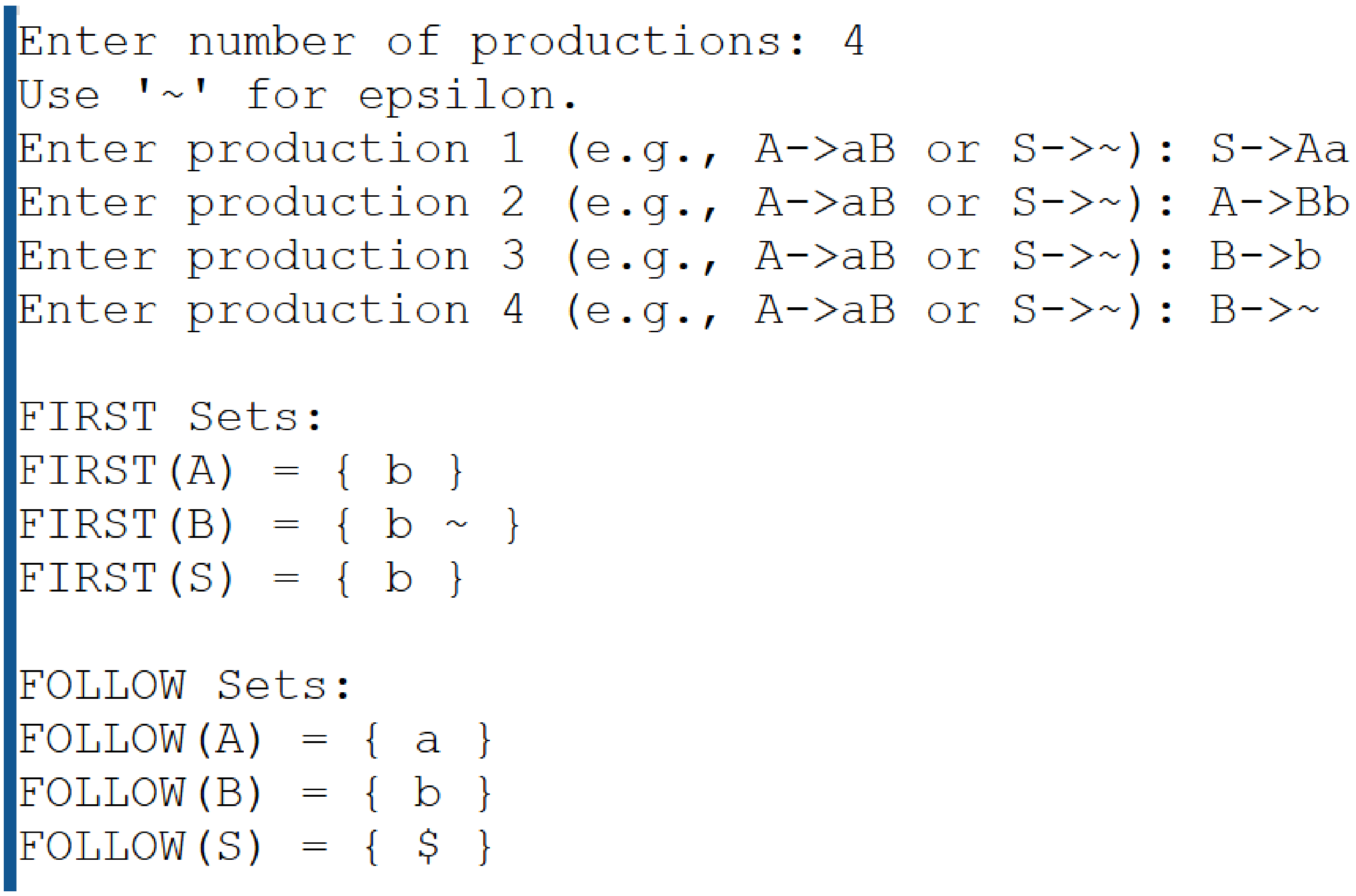
**Code:**

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

map<char, vector<string>> grammar;
map<char, set<char>> FIRST, FOLLOW;
set<char> nonterminals;
char startSymbol;

bool isNonTerminal(char c) {
  return isupper(c);
}

void computeFIRST(char nt, set<char>& visited) {
  if (visited.count(nt)) return;
  visited.insert(nt);

  for (const string& prod : grammar[nt]) {
    bool epsilonPossible = true;
    for (char symbol : prod) {
        if (!isNonTerminal(symbol)) {
          FIRST[nt].insert(symbol);
          epsilonPossible = false;
          break;
        } else {
          computeFIRST(symbol, visited);
          bool hasEpsilon = false;
          for (char ch : FIRST[symbol]) {
            if (ch == '~') hasEpsilon = true;
            else FIRST[nt].insert(ch);
          }
          if (!hasEpsilon) {
            epsilonPossible = false;
            break;
          }
        }
      }
      if (epsilonPossible) FIRST[nt].insert('~');
  }
}

void computeFOLLOW(char nt) {
  for (auto& [lhs, productions] : grammar) {
    for (string prod : productions) {
      for (int i = 0; i < prod.size(); ++i) {
        if (prod[i] == nt) {
          int j = i + 1;
          while (j < prod.size()) {
            char next = prod[j];
            if (!isNonTerminal(next)) {
              FOLLOW[nt].insert(next);
              break;
            } else {
              for (char ch : FIRST[next]) {
                if (ch != '~')
                  FOLLOW[nt].insert(ch);
              }
              if (FIRST[next].count('~'))
                j++;
              else break;
            }
          }
```

```
                if (j == prod.size() && lhs != nt) {
                    for (char ch : FOLLOW[lhs])
                        FOLLOW[nt].insert(ch);
                }
            }
        }
    }
  }
}

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

    cout << "Use '~' for epsilon.\n";
    for (int i = 0; i < n; ++i) {
        string input;
        cout << "Enter production " << i + 1 << " (e.g., A->aB or S->~): ";
        cin >> input;

        if (input.size() < 4 || input[1] != '-' || input[2] != '>') {
            cout << "Invalid format. Try again.\n";
            i--;
            continue;
        }

        char lhs = input[0];
        string rhs = input.substr(3);
        grammar[lhs].push_back(rhs);
        nonterminals.insert(lhs);
        if (i == 0) startSymbol = lhs;
    }

    for (char nt : nonterminals) {
        set<char> visited;
        computeFIRST(nt, visited);
    }

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

    for (char nt : nonterminals)
```

```
        computeFOLLOW(nt);

    cout << "\nFIRST Sets:\n";
    for (char nt : nonterminals) {
        cout << "FIRST(" << nt << ") = { ";
        for (char ch : FIRST[nt])
            cout << ch << " ";
        cout << "}\n";
    }

    cout << "\nFOLLOW Sets:\n";
    for (char nt : nonterminals) {
        cout << "FOLLOW(" << nt << ") = { ";
        for (char ch : FOLLOW[nt])
            cout << ch << " ";
        cout << "}\n";
    }

    return 0;
}
```

**Output:**



```
Enter number of productions: 4
Use '~' for epsilon.
Enter production 1 (e.g., A->aB or S->~): S->Aa
Enter production 2 (e.g., A->aB or S->~): A->Bb
Enter production 3 (e.g., A->aB or S->~): B->b
Enter production 4 (e.g., A->aB or S->~): B->~

FIRST Sets:
FIRST(A) = { b }
FIRST(B) = { b ~ }
FIRST(S) = { b }

FOLLOW Sets:
FOLLOW(A) = { a }
FOLLOW(B) = { b }
FOLLOW(S) = { $ }
```

**Learning Outcome:**

## Experiment - 10

**Aim:** Write a program to construct LL(1) parsing table.

**Theory:**

An LL(1) parsing table is used by a predictive parser to decide which production to apply, based on the current non-terminal and the lookahead input symbol. The table is constructed using the FIRST and FOLLOW sets of the grammar.

**Steps to Construct the LL(1) Parsing Table:**

Let M[A, a] be the parsing table entry where:

- A is a non-terminal, and
- a is a terminal or $ (end of input symbol).

For each production of the form **A → α**:

1. **For each terminal a in FIRST(α)**, add the production **A → α** to **M[A, a]**.
2. **If ε (epsilon) is in FIRST(α)**:
    - For each terminal b in FOLLOW(A), add **A → α** to **M[A, b]**.
    - If **$** is in FOLLOW(A), add **A → α** to **M[A, $]**.

Note:

- **Single Production Per Cell**: If **more than one production** is assigned to a cell **M[A, a]**, then the grammar is **not LL(1)** (i.e., the parser would not know which rule to apply).
- **Epsilon Productions**: When epsilon (~ or ε) is in the FIRST set of a production, the FOLLOW set helps determine where the production can be used.

**Code:**

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

const char EPSILON = '~';

vector<pair<char, string>> grammarRules;
set<char> nonTerminals, terminals;
map<char, set<char>> firstMap, followMap;
map<pair<char, char>, int> parseTableMap;

set<char> computeFirst(char X) {
    if (!isupper(X)) return {X};
    if (!firstMap[X].empty()) return firstMap[X];
    for (auto &rule : grammarRules) {
        if (rule.first != X) continue;
        const string &rhs = rule.second;
        if (rhs[0] == EPSILON) {
            firstMap[X].insert(EPSILON);
        } else {
            bool allNullable = true;
            for (char symbol : rhs) {
                auto firstSetSymbol = computeFirst(symbol);
                for (char fch : firstSetSymbol) {
                    if (fch != EPSILON) firstMap[X].insert(fch);
                }
                if (!firstSetSymbol.count(EPSILON)) {
                    allNullable = false;
                    break;
                }
            }
            if (allNullable) firstMap[X].insert(EPSILON);
        }
    }
    return firstMap[X];
}

void computeFollow() {
    followMap[grammarRules[0].first].insert('$');
    bool updated = true;
    while (updated) {
        updated = false;
        for (auto &rule : grammarRules) {
            char A = rule.first;
            const string &rhs = rule.second;
            int len = rhs.size();
            for (int i = 0; i < len; ++i) {
                char B = rhs[i];
                if (!isupper(B)) continue;
                bool betaNullable = true;
                for (int j = i + 1; j < len; ++j) {
                    auto firstBeta = computeFirst(rhs[j]);
                    for (char f : firstBeta) {
                        if (f != EPSILON && !followMap[B].count(f)) {
```

```
                    followMap[B].insert(f);
                    updated = true;
                  }
                }
                if (!firstBeta.count(EPSILON)) {
                  betaNullable = false;
                  break;
                }
              }
              if (betaNullable) {
                for (char f : followMap[A]) {
                  if (!followMap[B].count(f)) {
                    followMap[B].insert(f);
                    updated = true;
                  }
                }
              }
            }
          }
        }
      }

int main() {
  int prodCount;
  cout << "Enter number of productions: ";
  if (!(cin >> prodCount)) return 0;
  cout << "Enter productions (e.g., S->aB, use '~' for ε):\n";
  for (int i = 0; i < prodCount; ++i) {
    string prod;
    cin >> prod;
    char lhs = prod[0];
    string rhs = prod.substr(3);
    grammarRules.emplace_back(lhs, rhs);
    nonTerminals.insert(lhs);
  }

  for (auto &rule : grammarRules) {
    for (char ch : rule.second) {
      if (!isupper(ch) && ch != EPSILON)
        terminals.insert(ch);
    }
  }
```

```
terminals.insert('$');

for (char nt : nonTerminals) computeFirst(nt);
computeFollow();

cout << "\nFIRST sets:\n";
for (char nt : nonTerminals) {
  cout << "FIRST(" << nt << ") = { ";
  for (char ch : firstMap[nt]) cout << ch << " ";
  cout << "}\n";
}

cout << "\nFOLLOW sets:\n";
for (char nt : nonTerminals) {
  cout << "FOLLOW(" << nt << ") = { ";
  for (char ch : followMap[nt]) cout << ch << " ";
  cout << "}\n";
}

bool tableConflict = false;
for (int i = 0; i < grammarRules.size(); ++i) {
  char A = grammarRules[i].first;
  const string &rhs = grammarRules[i].second;
  set<char> firstAlpha;
  bool allNullable = true;
  for (char c : rhs) {
    auto firstC = computeFirst(c);
    for (char fc : firstC) if (fc != EPSILON) firstAlpha.insert(fc);
    if (!firstC.count(EPSILON)) {
      allNullable = false;
      break;
    }
  }
  if (allNullable) firstAlpha.insert(EPSILON);

  for (char a : firstAlpha) {
    if (a == EPSILON) {
      for (char b : followMap[A]) {
        pair<char, char> key = make_pair(A, b);
        if (parseTableMap.count(key)) {
          tableConflict = true;
          break;
```

```
                }
                parseTableMap[key] = i;
            }
        } else {
            pair<char, char> key = make_pair(A, a);
            if (parseTableMap.count(key)) {
                tableConflict = true;
                break;
            }
            parseTableMap[key] = i;
        }
    }
    if (tableConflict) break;
}

if (tableConflict) {
    cout << "\nGrammar is not LL(1) (parsing table conflict detected).\n";
    return 0;
}

cout << "\nLL(1) Parsing Table:\n\t";
for (char term : terminals) cout << term << "\t";
cout << "\n----------------------------------------\n";
for (char nt : nonTerminals) {
    cout << nt << "\t";
    for (char term : terminals) {
        auto it = parseTableMap.find({nt, term});
        if (it != parseTableMap.end())
            cout << grammarRules[it->second].first << "->" << grammarRules[it->second].second
<< "\t";
        else
            cout << "_\t";
    }
    cout << "\n";
}

cout << "\nEnter input string: ";
string inStr; cin >> inStr;
inStr.push_back('$');
stack<char> parseStack;
parseStack.push('$');
parseStack.push(grammarRules[0].first);
```

```
  cout << "\nParsing sequence (stack | input | action):\n";
  while (!parseStack.empty()) {
    string stackContent;
    stack<char> tempStack = parseStack;
    while (!tempStack.empty()) {
      stackContent.push_back(tempStack.top());
      tempStack.pop();
    }
    reverse(stackContent.begin(), stackContent.end());
    cout << setw(10) << stackContent << " | " << setw(10) << inStr << " | ";

    char topSym = parseStack.top(), currInput = inStr[0];
    if (topSym == currInput) {
      cout << "Match '" << currInput << "'\n";
      parseStack.pop();
      inStr.erase(0, 1);
    } else if (isupper(topSym)) {
      auto ruleIt = parseTableMap.find({topSym, currInput});
      if (ruleIt == parseTableMap.end()) {
        cout << "Error: no matching rule for [" << topSym << "," << currInput << "]\n";
        break;
      }
      auto &rule = grammarRules[ruleIt->second];
      cout << "Apply: " << rule.first << "->" << (rule.second == string(1, EPSILON) ? " " :
rule.second) << "\n";
      parseStack.pop();
      if (rule.second[0] != EPSILON) {
        for (int j = rule.second.size() - 1; j >= 0; j--) {
          parseStack.push(rule.second[j]);
        }
      }
    } else {
      cout << "Error: unexpected terminal '" << topSym << "'\n";
      break;
    }
  }

  if (parseStack.empty() && inStr.empty())
    cout << "\nParsing successful!\n";

  return 0;
```

```
}
```

**Output:**

```
Enter number of productions: 6
Enter productions (e.g., S->aB, use '~' for ε):
S->ACB
A->a
A->~
B->b
B->~
C->c

FIRST sets:
FIRST(A) = { a ~ }
FIRST(B) = { b ~ }
FIRST(C) = { c }
FIRST(S) = { a c }

FOLLOW sets:
FOLLOW(A) = { c }
FOLLOW(B) = { $ }
FOLLOW(C) = { $ b }
FOLLOW(S) = { $ }
```

```
LL(1) Parsing Table:
        $       a       b       c
-----------------------------------------
A       _       A->a    _       A->~
B       B->~    _       B->b    _
C       _       _       _       C->c
S       _       S->ACB  _       S->ACB

Enter input string: acb

Parsing sequence (stack | input | action):
        $S |       acb$ | Apply: S->ACB
      $BCA |       acb$ | Apply: A->a
      $BCa |       acb$ | Match 'a'
       $BC |        cb$ | Apply: C->c
       $Bc |        cb$ | Match 'c'
        $B |         b$ | Apply: B->b
        $b |         b$ | Match 'b'
         $ |          $ | Match '$'

Parsing successful!
```

**Learning Outcome:**