# Experiment No. 6: Write a program in to remove the left recursion from grammar

**SOURCE CODE**

#include <iostream> #include <cstring> #include <cstdio>

using namespace std; int main() {

char input[100], lhs[50], rhs[50], temp[10], productions[25][50]; int i = 0, j = 0, flag = 0, consumed = 0;

cout << "Enter the production (example: A->Aa|b): "; scanf("%s -> %s", lhs, rhs);

cout << "The grammar is: " << lhs << " -> " << rhs << endl; while (sscanf(rhs + consumed, "%[^|]", temp) == 1 && consumed <

strlen(rhs)) {

consumed += strlen(temp) + 1;

lhs);

if (temp[0] == lhs[0]) { // Check if there is left recursion flag = 1;

sprintf(productions[i++], "%s'->%s%s'", lhs, temp + 1,

} else {

// Non-left recursion case

sprintf(productions[i++], "%s->%s%s'", lhs, temp, lhs);

}

}

if (flag == 1) {

sprintf(productions[i++], "%s'->ε", lhs); // ε denotes

epsilon (empty production)

cout << "The productions after eliminating Left Recursion are:" << endl;

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

cout << productions[j] << endl;

}

} else {

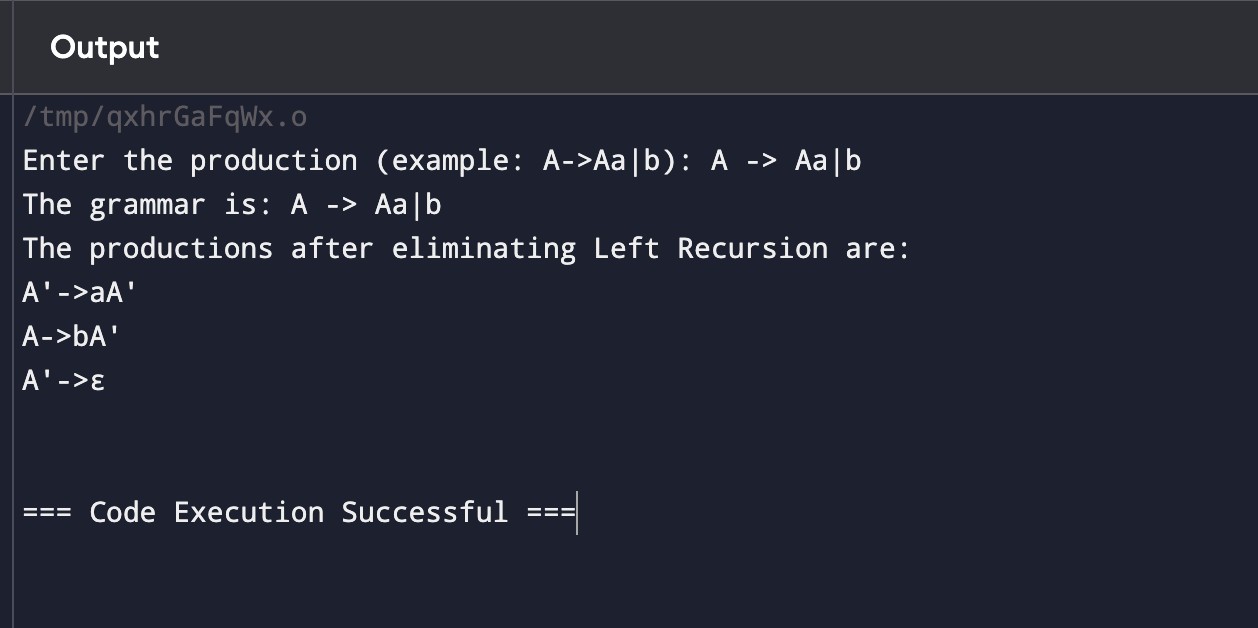
cout << "The Given Grammar has no Left Recursion." << endl;

}

return 0;

}

## OUTPUT



**Experiment No. 7:** Write a program in C++to perform left factoring on a grammar.

### Elimination of Left Factoring in Compiler Design

What is Left Factoring?

Left factoring is taking out the regular left factor that shows up in two productions of the equivalent non-terminal. It is done to keep away from back-tracking by the parser.

Consider an example below

A -> aP/iiQ

where A, P, Q are non-terminals and o is a common factor after left factoring the grammar will be:

A -> oS' S' -> P/Q

Left Factoring is basically a grammar transformation technique. It has "factoring out" prefixes which are common to two or more productions or in other words Left factoring is a process of transformation, in which the grammar turns from a left-recursive form to an equivalent non-left-recursive form.

### CODE:

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

string longestCommonPrefix(const string& s1, const string& s2) { int minLength = min(s1.size(), s2.size());

int i = 0;

while (i < minLength && s1[i] == s2[i]) { i++;

}

return s1.substr(0, i);

}

// Function to perform left factoring on a given grammar

void leftFactoring(string nonTerminal, vector<string>& productions) { vector<string> factoredProductions;

bool leftFactoringNeeded = false; while (!productions.empty()) {

string prefix = productions[0];

vector<string> currentGroup; vector<string> remainingProductions;

// Group productions with the same common prefix

for (size\_t i = 0; i < productions.size(); ++i) { string commonPrefix = longestCommonPrefix(prefix,

productions[i]);

if (!commonPrefix.empty() && commonPrefix == prefix.substr(0, commonPrefix.size())) {

currentGroup.push\_back(productions[i]);

} else {

remainingProductions.push\_back(productions[i]);

}

}

// If there's a common prefix, apply left factoring if (currentGroup.size() > 1) {

leftFactoringNeeded = true;

string newNonTerminal = nonTerminal + "'"; // Create a new non-terminal

factoredProductions.push\_back(nonTerminal + " -> " + prefix + newNonTerminal);

cout << newNonTerminal << " -> ";

for (size\_t j = 0; j < currentGroup.size(); ++j) {

string suffix = currentGroup[j].substr(prefix.size());

if (suffix.empty()) suffix = "ε"; // ε denotes empty

production (epsilon)

cout << suffix;

if (j < currentGroup.size() - 1) cout << " | ";

}

cout << endl;

} else {

factoredProductions.push\_back(nonTerminal + " -> " + productions[0]);

}

productions = remainingProductions;

}

if (!leftFactoringNeeded) {

cout << "No left factoring needed." << endl;

} else {

// Display the factored grammar

cout << "After Left Factoring:" << endl;

for (const auto& production : factoredProductions) { cout << production << endl;

}

}

}

int main() {

string nonTerminal; int n;

cout << "Enter the Parent Non-Terminal: "; cin >> nonTerminal;

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

vector<string> productions(n);

cout << "Enter the productions for " << nonTerminal << " (one per line):" << endl;

for (int i = 0; i < n; ++i) { cout << nonTerminal << " -> "; cin >> productions[i];

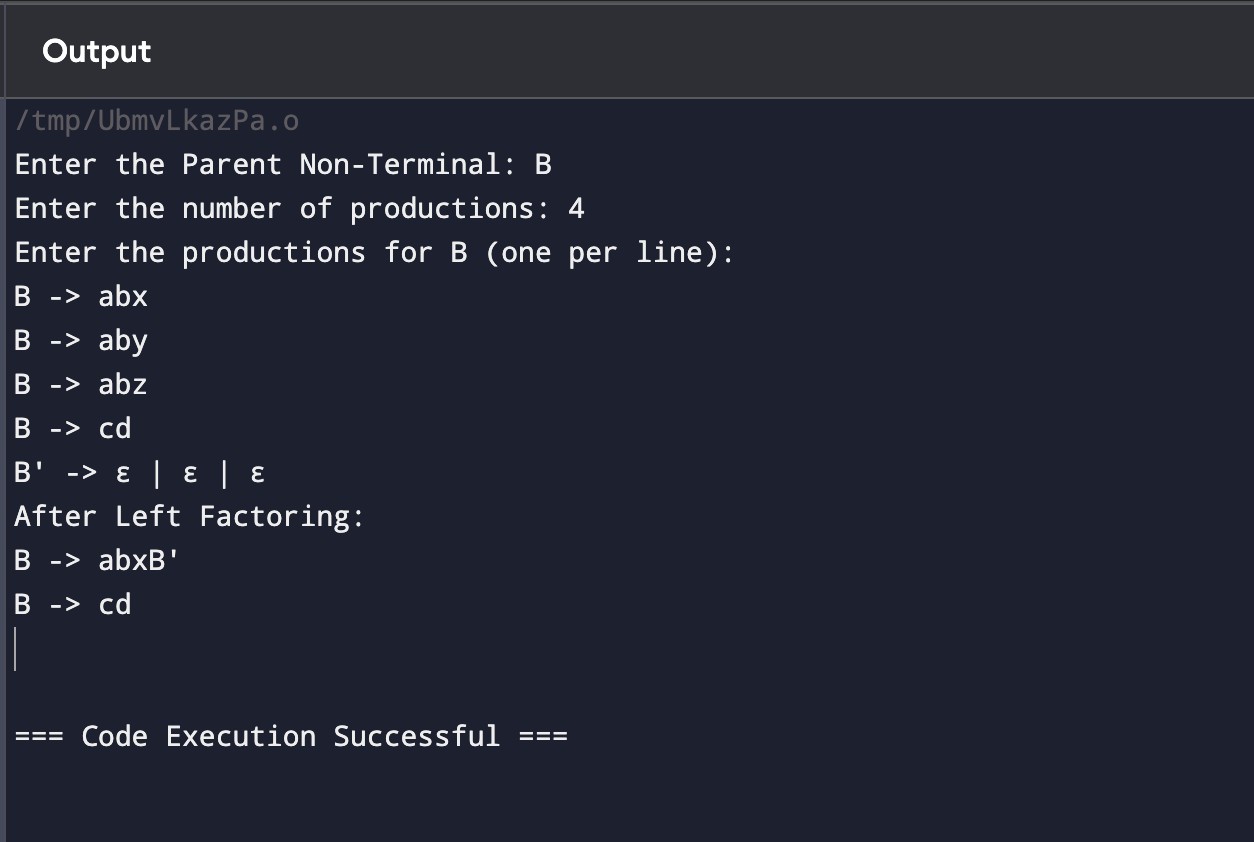
}

leftFactoring(nonTerminal, productions);

return 0;

}

OUTPUT



# Experiment No. 8: Write a program to check whether a string is a keyword or not

**SOURCE CODE:**

#include <iostream> #include <string> #include <unordered\_set> using namespace std;

bool isKeyword(const string& word) {

// Set of C++ keywords unordered\_set<string> keywords = {

"auto", "break", "case", "char", "const", "continue",

"default", "do", "double", "else",

"enum", "extern", "float", "for", "goto", "if", "inline", "int", "long", "register",

"return", "short", "signed", "sizeof", "static", "struct", "switch", "typedef", "union",

"unsigned", "void", "volatile", "while", "class", "public", "private", "protected",

"template", "this", "throw", "try", "virtual", "namespace", "new", "delete", "bool",

"true", "false", "using", "nullptr", "constexpr", "friend",

"mutable", "operator",

"static\_cast", "dynamic\_cast", "reinterpret\_cast", "const\_cast", "typeid"

};

return keywords.find(word) != keywords.end();

}

int main() {

string input;

cout << "Enter a word to check if it's a keyword: "; cin >> input;

if (isKeyword(input)) {

cout << input << " is a keyword." << endl;

} else {

cout << input << " is not a keyword." << endl;

}

return 0;

}

## OUTPUT:

