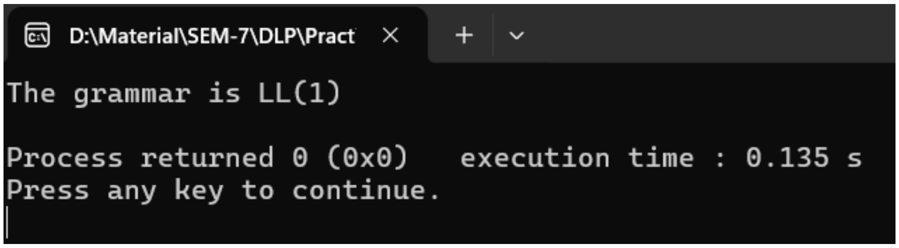
Practical-5

Aim: Implement a program to validate that given grammar is LL(1) or not.

Code:

```
#include <iostream>
#include <map>
#include <set>
#include <vector>
#include <string>

using namespace std;

// Function to calculate the FIRST set for a given grammar
void calculateFirst(const map<char, vector<string>>& grammar, map<char,
set<char>>& firstSets) {
    bool changed = true;
    while (changed) {
        changed = false;
        for (auto& rule : grammar) {
            char nonTerminal = rule.first;
            for (const string& production : rule.second) {
                char firstSymbol = production[0];
                if (isupper(firstSymbol)) {  // Non-terminal
                    for (char f : firstSets[firstSymbol]) {
                        if (firstSets[nonTerminal].insert(f).second) {
                            changed = true;
                        }
                    }
                } else {  // Terminal or epsilon
                    if (firstSets[nonTerminal].insert(firstSymbol).second) {
                        changed = true;
                    }
                }
            }
        }
    }
}

// Function to calculate the FOLLOW set for a given grammar
void calculateFollow(const map<char, vector<string>>& grammar, map<char,
set<char>>& firstSets, map<char, set<char>>& followSets, char startSymbol) {
    followSets[startSymbol].insert('$');  // End of input marker

    bool changed = true;
    while (changed) {
        changed = false;
        for (auto& rule : grammar) {
            char nonTerminal = rule.first;
            for (const string& production : rule.second) {
                for (int i = 0; i < production.size(); i++) {
                    char symbol = production[i];
                    if (isupper(symbol)) {
                        set<char> follow;
                        if (i + 1 < production.size()) {
                            char nextSymbol = production[i + 1];
                            if (isupper(nextSymbol)) {
                                follow = firstSets[nextSymbol];
                                follow.erase('~'); // '~' represents epsilon (ε)
                            } else {
                                follow.insert(nextSymbol);
                            }
                        } else {
                            follow = followSets[nonTerminal];
                        }

                        for (char f : follow) {
                            if (followSets[symbol].insert(f).second) {
                                changed = true;
                            }
                        }
                    }
                }
            }
        }
    }
}

// Function to check if the grammar is LL(1)
bool isLL1(const map<char, vector<string>>& grammar, map<char, set<char>>&
firstSets, map<char, set<char>>& followSets) {
    for (auto& rule : grammar) {
        char nonTerminal = rule.first;
        set<char> checked;
        for (const string& production : rule.second) {
            set<char> first = firstSets[production[0]];
            if (first.find('~') != first.end()) { // '~' represents epsilon (ε)
                first.insert(followSets[nonTerminal].begin(), followSets[nonTerminal].end());
            }
            for (char f : first) {
                if (checked.find(f) != checked.end()) {
                    return false;  // Conflict in predictive parsing table
                }
                checked.insert(f);
            }
        }
    }
    return true;
}

int main() {
    // Example Grammar
    map<char, vector<string>> grammar = {
        {'E', {"TA"}},
        {'A', {"+TA", "~"}},  // '~' represents epsilon (ε)
        {'T', {"FB"}},
        {'B', {"*FB", "~"}},  // '~' represents epsilon (ε)
        {'F', {"(E)", "a"}}
    };

    char startSymbol = 'E';

    // Calculate FIRST and FOLLOW sets
    map<char, set<char>> firstSets;
    map<char, set<char>> followSets;
    for (auto& rule : grammar) {
        firstSets[rule.first] = set<char>();
        followSets[rule.first] = set<char>();
    }

    // Calculate FIRST sets
    calculateFirst(grammar, firstSets);

    // Calculate FOLLOW sets
    calculateFollow(grammar, firstSets, followSets, startSymbol);

    // Check if the grammar is LL(1)
    if (isLL1(grammar, firstSets, followSets)) {
        cout << "The grammar is LL(1)" << endl;
    } else {
        cout << "The grammar is NOT LL(1)" << endl;
    }

    return 0;
}
```

Output:



```
The grammar is LL(1)

Process returned 0 (0x0)   execution time : 0.135 s
Press any key to continue.
```

Questions

1) How to create predictive parsing table?

Ans: **Step 1:** Calculate the FIRST set for each production.

Step 2: Calculate the FOLLOW set for each non-terminal.

Step 3: For each production, place it in the table under the columns for terminals in its FIRST set. If it can derive ε, add it in the columns for terminals in the FOLLOW set.

Step 4: Ensure no two productions for the same non-terminal share a table cell.

2) How to verify that given grammar is LL(1) or not?

Ans: Condition 1: The grammar should have no left recursion.

Condition 2: The FIRST sets of a non-terminal's productions must not overlap. If any production derives ε, its FOLLOW set should be disjoint from the other FIRST sets.

Verification: Build the predictive parsing table and check for conflicts (multiple entries in the same cell).

Student sign Marks/Grade Faculty Sign