Experiment No: 07
Experiment Name: Finding Follow of any Grammar

Objective:
The objective of this experiment is to implement an algorithm to find the FOLLOW set of any given grammar.

Problem Structure:
In this experiment, we are given a context-free grammar (CFG) defined by a set of productions. Each production consists of a non-terminal symbol followed by a sequence of symbols (terminals and non-terminals). The task is to compute the FOLLOW set for each non-terminal symbol in the grammar.

Procedure:
1. Input Grammar:
The given grammar is represented using a set of classes and data structures in C++. The `Grammar` class represents the grammar, and productions are added using the `addProduction` method. The start symbol of the grammar is set using the `setStartSymbol` method.

2. Computing FIRST Set:
The FIRST set for each non-terminal symbol is computed using the `FirstSet` class. The `computeFirst` method computes the FIRST set of each non-terminal symbol recursively, considering the FIRST sets of its production rules. The `computeFirstOfNonTerminal` method is responsible for computing the FIRST set of a single non-terminal symbol. It handles epsilon productions and updates the FIRST set accordingly.

3. Computing FOLLOW Set:
The FOLLOW set for each non-terminal symbol is computed using the `FollowSet` class. The `computeFollow` method iteratively computes the FOLLOW set for each non-terminal symbol in the grammar. It considers the FIRST sets of subsequent symbols in the production rules and updates the FOLLOW set accordingly.

Code:
```
#include <iostream>
#include <vector>
#include <map>
#include <set>
#include <string>
#include <algorithm>
#include <sstream>
#include <iterator>

using namespace std;

class Grammar {
public:
    map<string, vector<string>> productions;
    set<string> nonTerminals;
    set<string> terminals;
```

```
    string startSymbol;

    Grammar() {
        startSymbol = "";
    }

    void addProduction(const string& nonTerminal, const string& production) {
        productions[nonTerminal].push_back(production);
        nonTerminals.insert(nonTerminal);
        istringstream iss(production);

        string symbol;
        while (iss >> symbol) {
            if (!isupper(symbol[0]) && symbol != "@") {
                terminals.insert(symbol);
            }
        }
    }

    void setStartSymbol(const string& symbol) {
        startSymbol = symbol;
    }
};

//()

class FirstSet {
public:
    map<string, set<string>> first;

    void computeFirst(Grammar &grammar) {
        for (const string& nt : grammar.nonTerminals) {
            first[nt] = computeFirstOfNonTerminal(nt, grammar);
        }
    }

    set<string> computeFirstOfNonTerminal(const string& symbol, Grammar
&grammar) {
        set<string> result;
        if (grammar.terminals.count(symbol)) {
            result.insert(symbol);
            return result;
        }
        //E->"T E'"
        //F -> "( E )", "id"

        for (const string& production : grammar.productions[symbol]) {
            if (production == "@") {
```

```
                result.insert("@");
            } else {
                istringstream iss(production);
                vector<string> symbols{istream_iterator<string>{iss},
istream_iterator<string>{}};
                for (const string& sym : symbols) {
                    set<string> firstOfSym = computeFirstOfNonTerminal(sym,
grammar);
                    result.insert(firstOfSym.begin(), firstOfSym.end());
                    if (firstOfSym.find("@") == firstOfSym.end()) {
                        break;
                    } else {
                        result.erase("@");
                    }
                }
            }
        }
        return result;
    }

    void printFirst() {
        for (const auto &pair : first) {
            cout << "FIRST(" << pair.first << ") = { ";
            for (const string& ch : pair.second) {
                cout << ch << " ";
            }
            cout << "}\n";
        }
    }
};
class FollowSet {
public:
    map<string, set<string>> follow;

    void computeFollow(Grammar &grammar, FirstSet &firstSet) {
        for (const string& nt : grammar.nonTerminals) {
            follow[nt] = {};
        }

        follow[grammar.startSymbol].insert("$");

        bool changed;
        do {
            changed = false;
            for (const auto &production : grammar.productions) {
                const string &nt = production.first;
                for (const string &prod : production.second) {
                    istringstream iss(prod);
```

```
                    vector<string> symbols{istream_iterator<string>{iss},
istream_iterator<string>{}};
                    for (size_t i = 0; i < symbols.size(); ++i) {
                        const string& B = symbols[i];
                        if (grammar.nonTerminals.count(B)) {
                            set<string> followB = follow[B];
                            if (i + 1 < symbols.size()) {
                                set<string> firstBeta =
firstSet.computeFirstOfNonTerminal(symbols[i + 1], grammar);
                                set<string> temp = follow[B];
                                temp.insert(firstBeta.begin(),
firstBeta.end());
                                temp.erase("@");
                                if (temp.size() > follow[B].size()) {
                                    follow[B] = temp;
                                    changed = true;
                                }
                            }
                            if (i + 1 == symbols.size() ||
firstSet.computeFirstOfNonTerminal(symbols[i + 1], grammar).count("@")) {
                                set<string> temp = follow[B];
                                temp.insert(follow[nt].begin(),
follow[nt].end());
                                if (temp.size() > follow[B].size()) {
                                    follow[B] = temp;
                                    changed = true;
                                }
                            }
                        }
                    }
                }
            }
        } while (changed);
    }

    void printFollow() {
        for (const auto &pair : follow) {
            cout << "FOLLOW(" << pair.first << ") = { ";
            for (const string& ch : pair.second) {
                cout << ch << " ";
            }
            cout << "}\n";
        }
    }
};
int main() {
    Grammar grammar;
    grammar.addProduction("E", "T E'");
```

```
    grammar.addProduction("E'", "+ T E'");
    grammar.addProduction("E'", "@");
    grammar.addProduction("T", "F T'");
    grammar.addProduction("T'", "* F T'");
    grammar.addProduction("T'", "@");
    grammar.addProduction("F", "( E )");
    grammar.addProduction("F", "id");

    grammar.setStartSymbol("E");

    FirstSet firstSet;
    firstSet.computeFirst(grammar);
    cout << "First Set:" << endl;
    firstSet.printFirst();
    cout << endl << endl;
    FollowSet followSet;
    cout << "Follow Set:" << endl;
    followSet.computeFollow(grammar, firstSet);
    followSet.printFollow();

    return 0;
}
```

Conclusion:

In this experiment, we successfully implemented an algorithm to find the FOLLOW set of any given grammar. The algorithm efficiently handles epsilon productions and computes the FOLLOW sets iteratively based on the FIRST sets of subsequent symbols. The computed FIRST and FOLLOW sets provide valuable insights into the grammar's structure and are essential for various parsing and analysis tasks in compiler design and natural language processing.

Output:

First Set:
FIRST(E) = { (id }
FIRST(E') = { + @ }
FIRST(F) = { (id }
FIRST(T) = { (id }
FIRST(T') = { * @ }

Follow Set:
FOLLOW(E) = { $) }
FOLLOW(E') = { $) }
FOLLOW(F) = { $) * + }
FOLLOW(T) = { $) + }
FOLLOW(T') = { $) + }