Practical no. 4

Theory:

1. First Set

The First set of a non-terminal symbol in a grammar is the set of terminals that appear at the beginning of some string derived from that non-terminal.

- **For a terminal**: The First set of a terminal is the terminal itself.
- **For a non-terminal**: The First set of a non-terminal (A) includes:
  The First set of any terminal that appears directly in the production of (A).
  The First set of any non-terminal that appears at the beginning of a production for (A), with special handling if that non-terminal can derive epsilon (empty string).

2. Follow Set

The Follow set of a non-terminal symbol is the set of terminals that can appear immediately to the right of that non-terminal in some sentential form derived from the start symbol.

- **For the start symbol**: The Follow set of the start symbol includes the end-of-input marker (often represented as $).
- **For other non-terminals**: To compute the Follow set, you need to:
  Include the Follow set of a non-terminal (A) if (A) is followed by a non-terminal (B) in some production.
  Include the First set of the non-terminal (B) if (A) is followed by (B), excluding epsilon.
  If (A) is followed by a production that can derive epsilon, add Follow of (A) to Follow of (B).

Example

Consider the following simple grammar:

```
S → AB | a
A → a | ε
B → b
```

First Sets:

- **First(S):**
  For `S → AB`, start with `A`. `First(A)` is `{a, ε}`. Thus, `First(S)` includes `{a}` (since `A` can be `a` or `ε`, we take `First(B)` as well).
  For `S → a`, `First(S)` includes `a`.

Therefore, `First(S) = {a}`.

- **First(A):**
  For `A → a`, `First(A)` includes a.
  For `A → ε`, `First(A)` includes ε.
  Thus, `First(A) = {a, ε}`.
- **First(B):**
  For `B → b`, `First(B)` includes b.
  Thus, `First(B) = {b}`.

Follow Sets:

- **Follow(S):**
  Since `S` is the start symbol, `Follow(S)` includes `$` (end-of-input marker).
  Therefore, `Follow(S) = {$}`.
- **Follow(A):**
  In `S → AB`, after `A` comes `B`. So, `Follow(A)` includes `First(B)`, which is `{b}`.
  `A` can also derive ε, so include `Follow(S)` in `Follow(A)`.
  Thus, `Follow(A) = {b, $}`.
- **Follow(B):**
  In `S → AB`, `B` is at the end, so `Follow(B)` includes `Follow(S)`, which is `{ $ }`.
  Thus, `Follow(B) = { $ }`.

The computed First and Follow sets for the grammar are:

- **First Sets:**
  `First(S) = {a}`
  `First(A) = {a, ε}`
  `First(B) = {b}`
- **Follow Sets:**
  `Follow(S) = {$}`
  `Follow(A) = {b, $}`
  `Follow(B) = {$}`

Conclusion

This practical exercise demonstrated the computation of First and Follow sets for context-free grammars using C++. By implementing and running the provided code, we efficiently calculated these sets, which are essential for parser construction in compiler design. The successful execution confirms the effectiveness of the algorithms in processing grammar rules and generating accurate parsing information.

Code:

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

```cpp
#include <set>
#include <map>
#include <sstream>
#include <iterator>

using namespace std;

// Global variables
map<char, set<char>> first;
map<char, set<char>> follow;
map<char, vector<string>> grammar;
set<char> nullable;

// Helper function to split strings
vector<string> split(const string &s, char delimiter) {
    vector<string> tokens;
    stringstream ss(s);
    string item;
    while (getline(ss, item, delimiter)) {
        tokens.push_back(item);
    }
    return tokens;
}

// Compute First sets
void computeFirst(char nonTerminal) {
    if (!first[nonTerminal].empty())
        return;

    for (const string &production : grammar[nonTerminal]) {
        if (production == "ε") {
            first[nonTerminal].insert('ε');
            nullable.insert(nonTerminal);
        } else {
            bool nullableProd = true;
            for (char symbol : production) {
                if (symbol >= 'a' && symbol <= 'z') {
                    first[nonTerminal].insert(symbol);
                    nullableProd = false;
                    break;
                } else {
                    computeFirst(symbol);
                    first[nonTerminal].insert(first[symbol].begin(),
first[symbol].end());
```

```
                    first[nonTerminal].erase('ε');
                    if (first[symbol].find('ε') == first[symbol].end()) {
                        nullableProd = false;
                        break;
                    }
                }
            }
            if (nullableProd) {
                first[nonTerminal].insert('ε');
            }
        }
    }
}

// Compute Follow sets
void computeFollow(char nonTerminal) {
    if (!follow[nonTerminal].empty())
        return;

    if (nonTerminal == grammar.begin()->first) {
        follow[nonTerminal].insert('$');
    }

    for (const auto &rule : grammar) {
        for (const string &production : rule.second) {
            size_t pos = production.find(nonTerminal);
            while (pos != string::npos) {
                if (pos + 1 < production.length()) {
                    char nextSymbol = production[pos + 1];
                    if (nextSymbol >= 'a' && nextSymbol <= 'z') {
                        follow[nonTerminal].insert(nextSymbol);
                    } else {
                        computeFirst(nextSymbol);
                        follow[nonTerminal].insert(first[nextSymbol].begin(),
first[nextSymbol].end());
                        follow[nonTerminal].erase('ε');
                        if (first[nextSymbol].find('ε') != first[nextSymbol].end()) {
                            computeFollow(nextSymbol);
                            follow[nonTerminal].insert(follow[nextSymbol].begin(),
follow[nextSymbol].end());
                        }
                    }
                } else {
                    computeFollow(rule.first);
```

```
                    follow[nonTerminal].insert(follow[rule.first].begin(),
follow[rule.first].end());
                }
                pos = production.find(nonTerminal, pos + 1);
            }
        }
    }
}

int main() {
    // Define grammar
    grammar['S'] = {"AB", "a"};
    grammar['A'] = {"a", "ε"};
    grammar['B'] = {"b"};

    // Compute First sets
    for (const auto &rule : grammar) {
        computeFirst(rule.first);
    }

    // Compute Follow sets
    for (const auto &rule : grammar) {
        computeFollow(rule.first);
    }

    // Print First sets
    cout << "First sets:" << endl;
    for (const auto &entry : first) {
        cout << entry.first << " -> {";
        copy(entry.second.begin(), entry.second.end(), ostream_iterator<char>(cout,
", "));
        cout << "\b\b}" << endl;
    }

    // Print Follow sets
    cout << "\nFollow sets:" << endl;
    for (const auto &entry : follow) {
        cout << entry.first << " -> {";
        copy(entry.second.begin(), entry.second.end(), ostream_iterator<char>(cout,
", "));
        cout << "\b\b}" << endl;
    }
```

```
    return 0;
}
```