Name : Aadil Mohamed Puthiyaveetil
Reg No : 22BCE2436
Course Name : Compiler Design Lab
Course Code : VL2024250102359

COMPILER DESIGN LAB DA-3

Question

1. Given a CFG, display the following:
   a) Each Non-terminal's First and Follow set.
   b) The LL(1) parse Table
2. Display step-by-step stack operation of parsing inputs using the Parse table created.
   Take one input that can be generated by the Grammar and another input that can not be generated by the Grammar.

CODE

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

class CFG {
public:
   std::set<char> terminals;
   std::set<char> non_terminals;
   std::map<char, std::vector<std::string>> productions;
   char start_symbol;
   std::map<char, std::set<char>> first_sets;
   std::map<char, std::set<char>> follow_sets;
   std::map<char, std::map<char, std::string>> parse_table;

   CFG(const std::set<char>& terminals, const std::set<char>& non_terminals,
      const std::map<char, std::vector<std::string>>& productions, char start_symbol)
      : terminals(terminals), non_terminals(non_terminals), productions(productions),
        start_symbol(start_symbol) {
      for (char nt : non_terminals) {
         first_sets[nt] = {};
         follow_sets[nt] = {};
```

```
        parse_table[nt] = {};
        for (char t : terminals) {
            parse_table[nt][t] = "";
        }
        parse_table[nt]['$'] = "";
    }
}

void compute_first() {
    for (char terminal : terminals) {
        first_sets[terminal] = { terminal };
    }

    bool changed = true;
    while (changed) {
        changed = false;
        for (char nt : non_terminals) {
            for (const std::string& production : productions[nt]) {
                std::set<char> first_set = get_first_of_string(production);

                size_t initial_len = first_sets[nt].size();
                first_sets[nt].insert(first_set.begin(), first_set.end());
                for (auto i: first_set) {
                    if (i != ' ')
                        first_sets[nt].insert(i);
                }
                if (first_sets[nt].size() > initial_len) {
                    changed = true;
                }
            }
        }
    }
}

std::set<char> get_first_of_string(const std::string& symbols) {
    std::set<char> first_set;

    if (symbols == ""){
        first_set.insert(' '); // Use space to represent epsilon
        return first_set;
    }

    bool nullable = true;

    for (char symbol : symbols) {
        std::set<char> symbol_first_set = first_sets[symbol];
        first_set.insert(symbol_first_set.begin(), symbol_first_set.end());
```

```
      if (symbol_first_set.find(' ') == symbol_first_set.end()) {
        nullable = false;
        break;
      }
    }

    if (nullable) {
      first_set.insert(' '); // If all symbols can produce epsilon
    }

    return first_set;
}


void compute_follow() {
    follow_sets[start_symbol].insert('$');

    bool changed = true;
    while (changed) {
      changed = false;
      for (const auto& prod : productions) {
        char nt = prod.first;
        for (const std::string& production : prod.second) {
          std::set<char> follow = follow_sets[nt];
          for (int i = production.size() - 1; i >= 0; --i) {
            char symbol = production[i];
            if (non_terminals.find(symbol) != non_terminals.end()) {
              size_t initial_len = follow_sets[symbol].size();
              follow_sets[symbol].insert(follow.begin(), follow.end());
              if (first_sets[symbol].find(' ') != first_sets[symbol].end()) {
                follow.insert(first_sets[symbol].begin(), first_sets[symbol].end());
                follow.erase(' ');
              } else {
                follow = first_sets[symbol];
              }
              if (follow_sets[symbol].size() > initial_len) {
                changed = true;
              }
            } else {
              follow = first_sets[symbol];
            }
          }
        }
      }
    }
}
```

```
void create_parse_table() {
    for (char nt : non_terminals) {
        for (const std::string& production : productions[nt]) {
            std::set<char> first_set = get_first_of_string(production);
            for (char terminal : first_set) {
                if (terminal != ' ') {
                    parse_table[nt][terminal] = production;
                }
            }
            if (first_set.find(' ') != first_set.end()) {
                for (char terminal : follow_sets[nt]) {
                    parse_table[nt][terminal] = production;
                }
                if (follow_sets[nt].find('$') != follow_sets[nt].end()) {
                    parse_table[nt]['$'] = production;
                }
            }
        }
    }
}

void display_first_follow_sets() {
    std::cout << "First Sets:\n";
    for (const auto& entry : first_sets) {
        std::cout << "First(" << entry.first << ") = { ";
        for (char c : entry.second) {
            if (c == ' ') {
                std::cout << "ε ";
            } else {
                std::cout << c << " ";
            }
        }
        std::cout << "}\n";
    }

    std::cout << "\nFollow Sets:\n";
    for (const auto& entry : follow_sets) {
        std::cout << "Follow(" << entry.first << ") = { ";
        for (char c : entry.second) {
            if (c == ' ') {
                std::cout << "ε ";
            } else {
                std::cout << c << " ";
            }
        }
        std::cout << "}\n";
```

```
    }
}

void display_parse_table() {
    std::cout << "\nParse Table:\n";
    std::cout << "NT/Terminal  | ";
    for (char t : terminals) {
        std::cout << t << " | ";
    }
    std::cout << "$\n";

    for (char nt : non_terminals) {
        std::cout << nt << "           | ";
        for (char t : terminals) {
            std::string value = parse_table[nt][t];
            if (value == "") {
                std::cout << "ε | ";
            } else if (value.empty()) {
                std::cout << "- | ";
            } else if (value == " ") {
                std::cout << "ε | ";
            } else {
                std::cout << value << " | ";
            }
        }
        std::string value = parse_table[nt]['$'];
        if (value.empty()) {
            std::cout << "-";
        } else if (value == " ") {
            std::cout << "ε";
        } else {
            std::cout << value;
        }
        std::cout << "\n";
    }
}

void predictive_parsing(const std::string& input_string) {
    std::string input = input_string + '$';
    std::string stack = "$" + std::string(1, start_symbol);
    size_t input_ptr = 0;

    std::cout << "Stack              Input              Action\n";

    while (!stack.empty()) {
        char stack_top = stack.back();
        char current_input = input[input_ptr];
```

```
            std::cout << stack << "        " << input.substr(input_ptr) << " ";

            if (stack_top == current_input) {
                std::cout << "Match " << current_input << "\n";
                stack.pop_back();
                input_ptr++;
            } else if (terminals.find(stack_top) != terminals.end()) {
                std::cout << "Error: Terminal mismatch\n";
                break;
            } else if (non_terminals.find(stack_top) != non_terminals.end()) {
                std::string production = parse_table[stack_top][current_input];
                if (!production.empty() || production == "") {
                    if (production == " ") {  // Epsilon production
                        std::cout << "Output " << stack_top << " -> ε (epsilon)\n";
                        stack.pop_back();
                    } else {
                        std::cout << "Output " << stack_top << " -> " << production << "\n";
                        stack.pop_back();
                        std::reverse(production.begin(), production.end());
                        stack += production;
                    }
                } else {
                    std::cout << "Error: No production rule for " << stack_top << " on input " <<
current_input << "\n";
                    break;
                }
            } else {
                std::cout << "Error: Invalid symbol on stack\n";
                break;
            }
        }

        if (stack.empty() && input_ptr == input.size()) {
            std::cout << "Input successfully parsed!\n";
        } else {
            std::cout << "Parsing error!\n";
        }
    }
};

int main() {
    std::set<char> terminals = { 'a', 'b', 'c' };
    std::set<char> non_terminals = { 'S', 'A', 'B' };
    std::map<char, std::vector<std::string>> productions = {
        {'S', {"AB"}},
        {'A', {"aA", ""}},
```

```
        {'B', {"bB", "c"}}
    };
    char start_symbol = 'S';

    CFG cfg(terminals, non_terminals, productions, start_symbol);
    cfg.compute_first();
    cfg.compute_follow();
    cfg.create_parse_table();

    cfg.display_first_follow_sets();
    cfg.display_parse_table();

    std::string input_string = "abc";
    cfg.predictive_parsing(input_string);

    return 0;
}
```

OUTPUT

```
/tmp/jHaQTv0B4Y.o
First Sets:
First(A) = { ε a }
First(B) = { b c }
First(S) = { ε a b c }
First(a) = { a }
First(b) = { b }
First(c) = { c }

Follow Sets:
Follow(A) = { b c }
Follow(B) = { $ }
Follow(S) = { $ }

Parse Table:
NT/Terminal  | a | b | c | $
A            | aA | ε | ε | -
B            | ε | bB | c | -
S            | AB | AB | AB | AB
Stack                Input                Action
$S         abc$ Output S -> AB
$BA         abc$ Output A -> aA
$BAa         abc$ Match a
$BA         bc$ Output A ->
$B         bc$ Output B -> bB
$Bb         bc$ Match b
$B         c$ Output B -> c
$c         c$ Match c
$        $ Match $
Input successfully parsed!


=== Code Execution Successful ===
```