Laboratory work nr.1
Regular Grammars

Course: Formal Languages & Finite Automata

Student: Schipschi Daniil

Objectives:

1. Discover what a language is and what it needs to have in order to be considered a formal one;

2. Provide the initial setup for the evolving project that you will work on during this semester. You can deal with each laboratory work as a separate task or project to demonstrate your understanding of the given themes, but you also can deal with labs as stages of making your own big solution, your own project. Do the following:

    a. Create GitHub repository to deal with storing and updating your project;

    b. Choose a programming language. Pick one that will be easiest for dealing with your tasks, you need to learn how to solve the problem itself, not everything around the problem (like setting up the project, launching it correctly and etc.);

    c. Store reports separately in a way to make verification of your work simpler (duh)

3. According to your variant number, get the grammar definition and do the following:

    a. Implement a type/class for your grammar;

    b. Add one function that would generate 5 valid strings from the language expressed by your given grammar;

    c. Implement some functionality that would convert and object of type Grammar to one of type Finite Automaton;

    d. For the Finite Automaton, please add a method that checks if an input string can be obtained via the state transition from it;

Implementation:

To carry out this laboratory assignment, the initial step involved the creation of the Grammar class using Python. This class serves as a blueprint for various Grammars, featuring attributes with clear and representative names. The important method within this class is `generateString()`.

```java
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.ThreadLocalRandom;

public class Grammar {
    // Instance variables to hold the grammar components
    private final String SS; // Starting symbol
    private final Set<String> VN; // Set of non-terminal symbols
    private final Map<String, List<String>> P; // Production rules
    private final FiniteAutomaton FA; // Finite Automaton representation of the grammar

    // Constructor to initialize the Grammar object with the provided grammar components.
    public Grammar(String SS, Set<String> VN, Set<String> VT, Map<String, List<String>> P) {
        this.SS = SS; // Initialize the starting symbol
        this.VN = VN; // Initialize the set of non-terminal symbols
        this.P = P; // Initialize the production rules
        // Convert the grammar to a Finite Automaton for validation.
        // The Finite Automaton will use the terminal symbols and transitions built from the production rules.
        this.FA = new FiniteAutomaton(VT, buildTransitions(P), SS);
    }

    // Method to generate and print a specified number of strings and their validity.
    public void generateNeededStrings(int num) {
        System.out.println("Strings + Validity:");
        // Loop to generate and check validity for each string
        for (int i = 0; i < num; i++) {
            // Generate a random string according to the grammar.
            String generatedString = generateString();
            // Check if the generated string is valid and print its validity.
            System.out.println((i + 1) + ": " + generatedString + " " + (isValid(generatedString) ? "is valid" : "is
not valid"));
        }
    }

    // Method to check if a given word is valid according to the grammar.
    public boolean isValid(String word) {
        // Delegate the validation to the Finite Automaton representation of the grammar.
        return FA.isValid(word);
    }

    // Method to generate a random string according to the grammar.
    public String generateString() {
        // Start with the starting symbol.
        String result = getRandomProduction(SS);
        boolean containsNonTerminal = true;

        // Continue replacing non-terminals with their productions until no more non-terminals remain.
        while (containsNonTerminal) {
            containsNonTerminal = false;
            // Iterate over each character in the current string.
            for (char entry : result.toCharArray()) {
                String entryString = String.valueOf(entry);
                // If the character is a non-terminal, replace it with a random production.
                if (VN.contains(entryString)) {
                    result = result.replaceFirst(entryString, getRandomProduction(entryString));
                    containsNonTerminal = true;
                }
            }
        }
        return result;
    }

    // Method to get a random production for a given non-terminal symbol.
    private String getRandomProduction(String nonTerminal) {
        // Get the list of productions for the given non-terminal and select one randomly.
        return P.get(nonTerminal).get(ThreadLocalRandom.current().nextInt(P.get(nonTerminal).size()));
    }

    // Method to build transitions for the Finite Automaton representation of the grammar.
    private Map<String, Map<String, String>> buildTransitions(Map<String, List<String>> productions) {
        // Initialize a map to hold the transitions for each state (non-terminal symbol).
        Map<String, Map<String, String>> transitions = new HashMap<>();
        // Iterate over each production rule.
        for (Map.Entry<String, List<String>> entry : productions.entrySet()) {
            String state = entry.getKey(); // Get the non-terminal symbol as the state.
            List<String> productionList = entry.getValue(); // Get the list of productions for the state.
            Map<String, String> stateTransitions = new HashMap<>();
            // Iterate over each production.
            for (String production : productionList) {
                String symbol = production.substring(0, 1); // Get the first character as the symbol.
                // Get the next state as the rest of the production, or "OK" if it's empty.
                String nextState = production.length() > 1 ? production.substring(1) : "OK";
                stateTransitions.put(symbol, nextState); // Add the transition to the state's transitions.
            }
            transitions.put(state, stateTransitions); // Add the state and its transitions to the overall transitions
map.
        }
        return transitions;
    }

}
```

```java
import java.util.Map;
import java.util.Set;

public class FiniteAutomaton {
    private final Set<String> ALPHABET; // Set of symbols in the alphabet
    private final Map<String, Map<String, String>> P; // Transition function represented as a map of maps
    private final String SS; // Starting state

    // Constructor to initialize the FiniteAutomaton with the alphabet, transition function, and starting state
    public FiniteAutomaton(Set<String> ALPHABET, Map<String, Map<String, String>> P, String SS) {
        this.ALPHABET = ALPHABET; // Initialize the alphabet
        this.P = P; // Initialize the transition function
        this.SS = SS; // Initialize the starting state
    }

    // Method to validate an input string against the finite automaton
    public boolean isValid(String input) {
        String currentState = SS; // Start from the initial state

        // Iterate over each symbol in the input string
        for (char symbol : input.toCharArray()) {
            String symbolStr = String.valueOf(symbol); // Convert the symbol to a string

            // Check if the symbol is in the alphabet
            if (!ALPHABET.contains(symbolStr)) return false; // If not, the input is invalid

            // Check if there is a transition defined for the current state and symbol
            if (P.containsKey(currentState) && P.get(currentState).containsKey(symbolStr))
                currentState = P.get(currentState).get(symbolStr); // Move to the next state
            else
                return false; // If no transition is defined, the input is invalid
        }
        // Check if the final state is the "OK" state, indicating the input is valid
        return currentState.equals("OK");
    }
}
```

```java
import java.util.*;

public class Main {
    // Define the starting symbol, non-terminal symbols, terminal symbols, and production rules
    public static final String SS = "S"; // Starting symbol
    public static final Set<String> VN = Set.of("S", "B", "C"); // Non-terminal symbols
    public static final Set<String> VT = Set.of("a", "b", "c"); // Terminal symbols
    public static final Map<String, List<String>> P = Map.of( // Production rules
            "S", List.of("aB"), // Production rule for S
            "B", List.of("aC", "bB"), // Production rule for B
            "C", List.of("bB", "c", "aS")); // Production rule for C

    public static void main(String[] args) {
        String toCheck = "aac"; // Word to check

        // Create a Grammar object with the defined grammar
        Grammar grammar = new Grammar(SS, VN, VT, P);

        // Generate strings needed for the validation process
        grammar.generateNeededStrings(5);

        // Check if the given word is valid according to the grammar
        System.out.println("Is " + toCheck + " valid? " + grammar.isValid(toCheck));
    }
}
```

Conclusions:

This lab provided a valuable hands-on opportunity to grasp the foundational concepts of formal languages and their properties through practical examples. Throughout the lab, we diligently implemented grammars using sets of non-terminal and terminal symbols, along with production rules, enabling us to generate strings that adhere to predefined rules. We further explored the transformation of grammars into finite automata, gaining insight into how computational models recognize and validate the correctness of strings within a language. By actively engaging in implementing and refining the codebase, we solidified our understanding of formal language theory and its practical applications, preparing us for upcoming lab assignments and beyond.