Laboratory work nr.2
Finite Automata

Course: Formal Languages & Finite Automata
Student: Schipschi Daniil

Objectives:
Understand what an automaton is and what it can be used for.

Continuing the work in the same repository and the same project, the following need to be added: a. Provide a function in your grammar type/class that could classify the grammar based on Chomsky hierarchy.

b. For this you can use the variant from the previous lab.

According to your variant number (by universal convention it is register ID), get the finite automaton definition and do the following tasks:

a. Implement conversion of a finite automaton to a regular grammar.

b. Determine whether your FA is deterministic or non-deterministic.

c. Implement some functionality that would convert an NDFA to a DFA.

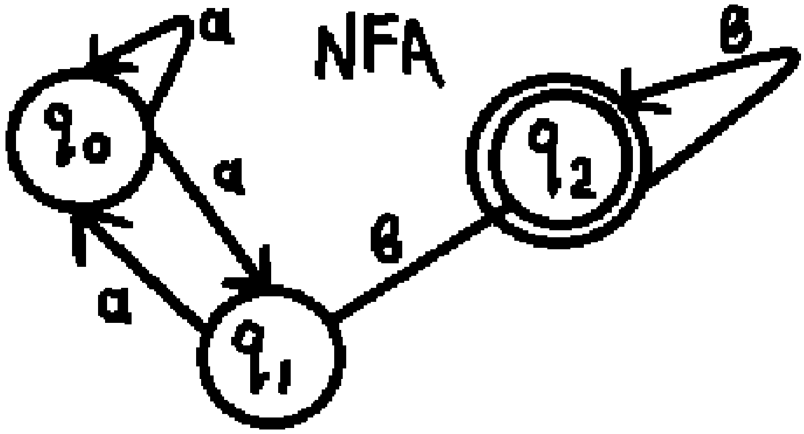
d. Represent the finite automaton graphically (Optional, and can be considered as a bonus point):

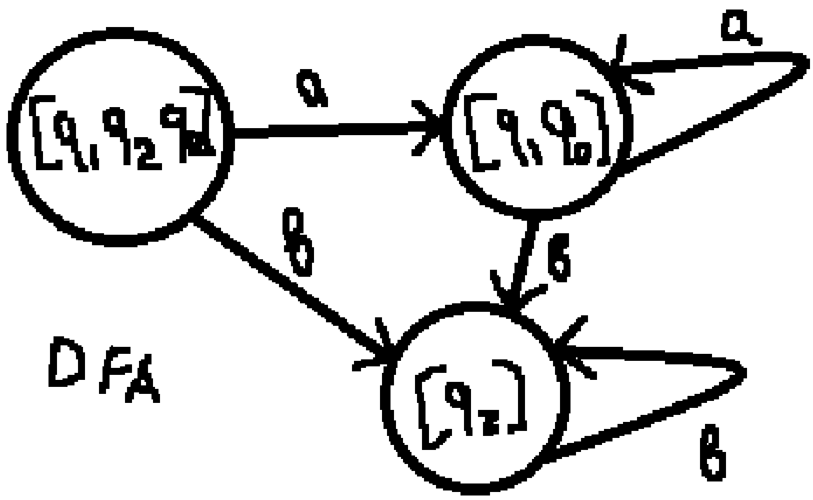
You can use external libraries, tools or APIs to generate the figures/diagrams.

Your program needs to gather and send the data about the automaton and the lib/tool/API return the visual representation.

Please consider that all elements of the task 3 can be done manually, writing a detailed report about how you've done the conversion and what changes have you introduced. In case if you'll be able to write a complete program that will take some finite automata and then convert it to the regular grammar - this will be a good bonus point.

```java
import util.Pair;
import java.util.*;

public class ManualFiniteAutomaton {

    private final List<String> stateList;
    private final Set<String> alphabet;
    private final Set<String> acceptingStates;
    private final Map<String, List<Pair>> T;

    public ManualFiniteAutomaton(List<String> stateList, Set<String> alphabet, Set<String> acceptingStates,
Map<String, List<Pair>> T) {
        this.stateList = stateList;
        this.alphabet = alphabet;
        this.acceptingStates = acceptingStates;
        this.T = T;
    }

    // Method to map original states to equivalent single-character states
    public Map<String, String> mapStates() {
        Map<String, String> mappedStates = new HashMap<>(); // Creating a HashMap to store mappings
        char mappedState = 'A'; // Starting mapping with character 'A'
        for (String state : stateList) { // Iterating through each state in stateList
            mappedStates.put(state, String.valueOf(mappedState)); // Mapping original state to a single character
            mappedState++; // Moving to the next character for mapping
        }
        return mappedStates; // Returning the mapped states
    }

    // Method to check if the automaton is deterministic
    public void isDeterministic() {
        for (Map.Entry<String, List<Pair>> entry : T.entrySet()) { // Iterating through each transition in T
            if (entry.getValue().size() > 1) { // If there are multiple transitions for a state and symbol
                System.out.println("It is not deterministic!"); // Print that the automaton is not deterministic
                return; // Exit the method
            }
        }
        System.out.println("It is deterministic!"); // Print that the automaton is deterministic
    }

    // Method to convert the NFA to DFA
    public void toDFA() {
        Map<Set<String>, Map<String, Set<String>>> dfaTransitions = new HashMap<>(); // Creating a map for DFA
transitions
        Queue<Set<String>> queue = new LinkedList<>(); // Queue to store sets of states

        Set<String> initialState = new HashSet<>(stateList); // Creating initial set of states containing all states
        queue.add(initialState); // Adding the initial state to the queue

        while (!queue.isEmpty()) { // While there are unprocessed states in the queue
            Set<String> currentState = queue.poll(); // Dequeue a state from the queue

            Map<String, Set<String>> transitions = new HashMap<>(); // Map to store transitions from current state
            for (String symbol : alphabet) { // For each symbol in the alphabet
                Set<String> nextStates = new HashSet<>(); // Set to store next states for the symbol
                for (String state : currentState) { // For each state in the current set of states
                    List<Pair> transitionsForState = T.get(state); // Get transitions for the current state
                    if (transitionsForState != null) { // If there are transitions for the state
                        for (Pair p : transitionsForState) { // For each transition from the state
                            if (p.first().equals(symbol)) { // If the transition symbol matches the current symbol
                                nextStates.add(p.second()); // Add the destination state to nextStates
                            }
                        }
                    }
                }
                transitions.put(symbol, nextStates); // Add transitions for the current symbol
                if (!nextStates.isEmpty() && !dfaTransitions.containsKey(nextStates)) { // If nextStates is not empty
and not already processed
                    queue.add(nextStates); // Enqueue nextStates for further processing
                }
            }

            dfaTransitions.put(currentState, transitions); // Add transitions from current state to DFA transitions
        }

        // Printing DFA transitions
        System.out.println("DFA Transitions:");
        for (Map.Entry<Set<String>, Map<String, Set<String>>> entry : dfaTransitions.entrySet()) { // For each DFA
state and its transitions
            Set<String> currentState = entry.getKey(); // Get the current DFA state
            Map<String, Set<String>> transitions = entry.getValue(); // Get its transitions
            System.out.println("State: " + currentState); // Print the current DFA state
            for (Map.Entry<String, Set<String>> transition : transitions.entrySet()) { // For each transition from the
state
                String inputSymbol = transition.getKey(); // Get the input symbol
                Set<String> nextState = transition.getValue(); // Get the next state(s)
                if (!nextState.isEmpty()) { // If there is a transition
                    System.out.println("    Input: " + inputSymbol + " --> " + nextState); // Print the transition
                }
            }
        }
    }

    // Method to convert the NFA to Grammar
    public void toGrammar() {
        Map<String, String> mappedStates = mapStates(); // Map original states to single-character states
        Map<String, List<String>> grammar = new HashMap<>(); // Create a map to store the grammar rules

        for(Map.Entry<String, List<Pair>> te : T.entrySet()) { // For each transition in the transition function
            List<String> maps = new ArrayList<>(); // Create a list to store grammar rules for the current state
            for (Pair p : te.getValue()) { // For each transition from the current state
                maps.add(p.first() + mappedStates.get(p.second())); // Add a grammar rule to the list
            }
            grammar.put(mappedStates.get(te.getKey()), maps); // Add the grammar rules to the grammar map
        }

        // Printing the grammar rules
        for (Map.Entry<String, List<String>> grammarEntry : grammar.entrySet()) { // For each state and its grammar
rules
            System.out.print(grammarEntry.getKey() + " -> "); // Print the state
            if (grammarEntry.getValue().size() > 1) { // If there are multiple rules
                System.out.print(grammarEntry.getValue().get(0)); // Print the first rule
                for (int i = 1; i < grammarEntry.getValue().size(); i++) { // For the remaining rules
                    System.out.print(" | " + grammarEntry.getValue().get(i)); // Print each rule
                }
            } else { // If there is only one rule
                System.out.print(grammarEntry.getValue().get(0)); // Print the rule
            }
            System.out.println(); // Move to the next line
        }
    }
}
```





Conclusions:
This lab served as a pivotal hands-on experience for me, allowing me to deeply comprehend the intricate concepts of formal languages and their properties through practical Java implementation. Crafting a Java class capable of constructing finite automata from specified transition data was an enlightening exercise. The subsequent conversion from Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA) underscored the nuances involved in streamlining computational models for efficiency and determinism.

Additionally, navigating the process of transforming automata into grammars provided crucial insights into the interplay between different formal language representations. This journey culminated in the rigorous examination of automata classification according to Chomsky's hierarchy, elucidating the structural characteristics that define their computational power.

By actively engaging in coding and refining the algorithms, I honed my understanding of formal language theory at a granular level. This hands-on immersion not only fortified my comprehension of abstract concepts but also equipped me with practical skills essential for navigating complex linguistic structures and their computational manifestations. As I venture into future lab assignments and real-world applications, the robust foundation laid in this lab will undoubtedly serve as a cornerstone for my continued growth and proficiency in formal language theory and its practical implementations.