Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work nr. 2
Course: Formal languages and finite automata
Topic: Determinism in Finite Automata. Conversion from NDFA 2 DFA. Chomsky Hierarchy.

Elaborated:
st. gr. FAF-221 Revenco Victor

Verified:
asist. univ. Cretu Dumitru

Chişinău - 2024

Theory

In the context of finite automata, determinism refers to the property that for every state in the automaton, there is at most one transition defined for each possible input symbol. A deterministic finite automaton (DFA) is defined by a 5-tuple (Q, Σ, δ, q0, F), where Q is a finite set of states, Σ is a finite alphabet of input symbols, δ is the transition function $\delta: Q \times \Sigma \rightarrow Q$, q0 is the initial state, and F is a set of final states. If for every state q in Q and input symbol a in Σ, the transition function δ(q, a) is defined and leads to exactly one state, the DFA is deterministic.

The conversion from a non-deterministic finite automaton (NDFA) to a DFA involves constructing a DFA that recognizes the same language as the given NDFA. This process requires considering all possible subsets of states in the NDFA to determine the behavior of the DFA. The resulting DFA will have a state for each subset of states in the NDFA, and the transitions in the DFA are determined by the transitions of the NDFA.

The Chomsky hierarchy is a classification of formal grammars, organized into four types, based on their expressive power. These types are Type-0 (recursively enumerable languages), Type-1 (context-sensitive languages), Type-2 (context-free languages), and Type-3 (regular languages). Each type of grammar corresponds to a specific type of automaton that can recognize the languages generated by the grammar. Regular languages are recognized by finite automata, context-free languages by pushdown automata, context-sensitive languages by linear-bounded automata, and recursively enumerable languages by Turing machines.

Objectives:

1. Understand what an automaton is and what it can be used for.
2. 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.
3. 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.

Implementation Description

For this laboratory I chose Java. I was variant 26.

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        // Define the finite automaton
        FiniteAutomaton fa = new FiniteAutomaton();
        fa.addState("q0");
        fa.addState("q1");
        fa.addState("q2");
        fa.addState("q3");
        fa.setStartState("q0");
        fa.addFinalState("q3");
        fa.addTransition("q0", 'a', "q1");
        fa.addTransition("q1", 'b', "q1");
        fa.addTransition("q1", 'a', "q2");
        fa.addTransition("q0", 'a', "q0");
        fa.addTransition("q2", 'c', "q3");
        fa.addTransition("q3", 'c', "q3");

        // Convert FA to regular grammar
        Grammar regularGrammar = fa.convertToRegularGrammar();
        System.out.println("Regular Grammar:");
        System.out.println(regularGrammar);
        System.out.println("This grammar is:" +
regularGrammar.classifyGrammar());

        // Determine if FA is deterministic or non-deterministic
        String determinism = fa.isDeterministic() ?
"Deterministic" : "Non-deterministic";
        System.out.println("Finite Automaton is " +
determinism);
```

```
        // Convert NDFA to DFA
        FiniteAutomaton dfa = fa.convertToDFA();
        System.out.println("NDFA -> DFA:");
        System.out.println(dfa);

    }
}
```

This Java class, `Main`, demonstrates the use of a `FiniteAutomaton` class to work with finite automata. It starts by defining a finite automaton (`fa`) with states `q0`, `q1`, `q2`, and `q3`, setting `q0` as the start state and `q3` as a final state. Transitions are added between states based on input symbols (`a`, `b`, `c`).

The program then converts the finite automaton to a regular grammar using `convertToRegularGrammar` method and prints the result. It also determines if the automaton is deterministic or non-deterministic and prints the result. Finally, it converts the non-deterministic finite automaton `fa` to a deterministic finite automaton `dfa` and prints `dfa`.

```
import java.util.*;
import java.util.stream.Collectors;

class FiniteAutomaton {
    private Set<String> states;
    private Set<Character> alphabet;
    private Map<String, Map<Character, Set<String>>>
transitions;
    private String startState;
    private Set<String> finalStates;

    public FiniteAutomaton() {
        states = new HashSet<>();
        alphabet = new HashSet<>();
        transitions = new HashMap<>();
        finalStates = new HashSet<>();
    }

    public void addState(String state) {
        states.add(state);
    }

    public void setStartState(String state) {
        startState = state;
    }

    public void addFinalState(String state) {
```

```
        finalStates.add(state);
    }

    public void addTransition(String fromState, char symbol,
String toState) {
        alphabet.add(symbol);
        transitions.computeIfAbsent(fromState, k -> new
HashMap<>())
                .computeIfAbsent(symbol, k -> new HashSet<>())
                .add(toState);
    }

    public Grammar convertToRegularGrammar() {
        Grammar grammar = new Grammar();

        // Add non-terminals based on states
        grammar.setNonTerminals(states);

        // Add terminals based on alphabet

grammar.setTerminals(alphabet.stream().map(String::valueOf).coll
ect(Collectors.toSet()));

        Map<String, Set<String>> productionRules = new
HashMap<>();

        // Add transitions as production rules
        for (String state : states) {
            for (char symbol : alphabet) {
                if (transitions.containsKey(state) &&
transitions.get(state).containsKey(symbol)) {
                    for (String nextState :
transitions.get(state).get(symbol)) {
                        String rule = String.format("%s ->
%s%s", state, symbol, nextState);
                        productionRules.computeIfAbsent(state, k
-> new HashSet<>()).add(rule);
                    }
                }
            }
        }

        grammar.setProductionRules(productionRules);
        return grammar;
    }
```

```
    public boolean isDeterministic() {
        for (String state : states) {
            if (transitions.containsKey(state)) {
                for (char symbol : alphabet) {
                    if
(transitions.get(state).containsKey(symbol) &&
transitions.get(state).get(symbol).size() > 1) {
                        return false;
                    }
                }
            }
        }
        return true;
    }

    public FiniteAutomaton convertToDFA() {
        FiniteAutomaton dfa = new FiniteAutomaton();
        dfa.setStartState(startState);
        Set<Set<String>> newStates = new HashSet<>();
        Set<String> initialState = new HashSet<>();
        initialState.add(startState);
        Map<Set<String>, String> stateMapping = new HashMap<>();
        newStates.add(initialState);
        stateMapping.put(initialState, startState);
        dfa.addState(startState);
        if (initialState.containsAll(finalStates)) {
            dfa.addFinalState(startState);
        }

        Queue<Set<String>> queue = new LinkedList<>();
        queue.add(initialState);
        while (!queue.isEmpty()) {
            Set<String> currentState = queue.poll();
            for (char symbol : alphabet) {
                Set<String> nextState = new HashSet<>();
                for (String state : currentState) {
                    if (transitions.containsKey(state) &&
transitions.get(state).containsKey(symbol)) {

nextState.addAll(transitions.get(state).get(symbol));
                    }
                }
                if (!newStates.contains(nextState)) {
                    newStates.add(nextState);
                    String stateName =
nextState.toString().replaceAll("[\\[\\],\\s]", "");
```

```
                    stateMapping.put(nextState, stateName);
                    dfa.addState(stateName);
                    if (nextState.containsAll(finalStates)) {
                        dfa.addFinalState(stateName);
                    }
                    queue.add(nextState);
                }

dfa.addTransition(stateMapping.get(currentState), symbol,
stateMapping.get(nextState));
            }
        }
        return dfa;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("States: ").append(states.isEmpty() ? "{}" :
states).append("\n");
        sb.append("Alphabet: ").append(alphabet).append("\n");
        sb.append("Transitions: \n");
        for (Map.Entry<String, Map<Character, Set<String>>>
entry : transitions.entrySet()) {
            String state = entry.getKey();
            sb.append(state).append(":\n");
            Map<Character, Set<String>> transitionMap =
entry.getValue();
            if (!transitionMap.isEmpty()) {
                for (Map.Entry<Character, Set<String>> trans :
transitionMap.entrySet()) {
                    char symbol = trans.getKey();
                    Set<String> nextStates = trans.getValue();
                    sb.append("\t").append(symbol).append(" ->
").append(nextStates.isEmpty() ? "{}" :
nextStates).append("\n");
                }
            } else {
                sb.append("\t{}").append("\n");
            }
        }
        sb.append("Start State:
").append(startState).append("\n");
        sb.append("Final States: ").append(finalStates.isEmpty()
? "{}" : finalStates).append("\n");
        return sb.toString();
    }
```

```
}
```

This Java class, FiniteAutomaton, defines a finite automaton and provides methods to work with it. It includes functionality to convert the automaton to a regular grammar, check if it is deterministic, and convert a non-deterministic automaton to a deterministic one.

The class has instance variables for states, alphabet, transitions, start state, and final states. It uses sets and maps to store these elements.

The constructor initializes these variables, and methods like addState, setStartState, addFinalState, and addTransition are provided to modify the automaton's structure.

The convertToRegularGrammar method constructs a regular grammar from the automaton's transitions, representing states as non-terminals and transitions as production rules.

The isDeterministic method checks if the automaton is deterministic by ensuring that each state has at most one transition for each input symbol.

The convertToDFA method converts a non-deterministic automaton to a deterministic one using the subset construction algorithm. It creates new states for each subset of states in the original automaton reachable by epsilon transitions.

Additional methods handle epsilon closures, which are sets of states reachable by epsilon transitions.

The toString method provides a string representation of the automaton, including its states, alphabet, transitions, start state, and final states.

```
import java.util.*;

public class Grammar {
    private Set<String> nonTerminals;
    private Set<String> terminals;
    private Map<String, Set<String>> productionRules;

    public Grammar() {
        nonTerminals = new HashSet<>();
        terminals = new HashSet<>();
        productionRules = new HashMap<>();
    }

    public Set<String> getNonTerminals() {
        return nonTerminals;
    }

    public void setNonTerminals(Set<String> nonTerminals) {
```

```
        this.nonTerminals = nonTerminals;
    }

    public Set<String> getTerminals() {
        return terminals;
    }

    public void setTerminals(Set<String> terminals) {
        this.terminals = terminals;
    }

    public Map<String, Set<String>> getProductionRules() {
        return productionRules;
    }

    public void setProductionRules(Map<String, Set<String>>
productionRules) {
        this.productionRules = productionRules;
    }

    public String classifyGrammar() {
        boolean isRegular = true;
        boolean isContextFree = true;
        boolean isContextSensitive = true;
        boolean isUnrestricted = true;

        for (String nonTerminal : productionRules.keySet()) {
            for (String production :
productionRules.get(nonTerminal)) {
                // Check if production is regular
                if (production.length() > 2 ||
(production.length() == 2 &&
!Character.isLowerCase(production.charAt(1)))) {
                    isRegular = false;
                }

                // Check if production is context-free
                if (production.length() > 2) {
                    isContextFree = false;
                }

                // Check if production is context-sensitive
                if (!production.equals("ε") &&
production.length() < nonTerminal.length()) {
                    isContextSensitive = false;
                }
            }
```

```
        }

        if (isRegular) {
            return "Regular Grammar";
        } else if (isContextFree) {
            return "Context-Free Grammar";
        } else if (isContextSensitive) {
            return "Context-Sensitive Grammar";
        } else {
            return "Unrestricted Grammar";
        }
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Non-terminals:
").append(nonTerminals.isEmpty() ? "{}" :
nonTerminals).append("\n");
        sb.append("Terminals: ").append(terminals.isEmpty() ?
"{}" : terminals).append("\n");
        sb.append("Productions: \n");
        for (Map.Entry<String, Set<String>> entry :
productionRules.entrySet()) {
            String nonTerminal = entry.getKey();
            sb.append(nonTerminal).append(" : ");
            Set<String> productions = entry.getValue();
            if (!productions.isEmpty()) {
                Iterator<String> iterator =
productions.iterator();
                sb.append(iterator.next());
                while (iterator.hasNext()) {
                    sb.append(" | ").append(iterator.next());
                }
            } else {
                sb.append("{}");
            }
            sb.append("\n");
        }
        return sb.toString();
    }
}
```

This Java class, Grammar, represents a context-free grammar and provides methods to work with it. It includes functionality to classify the grammar (whether it is regular, context-free, context-sensitive, or unrestricted) based on the form of its production rules.

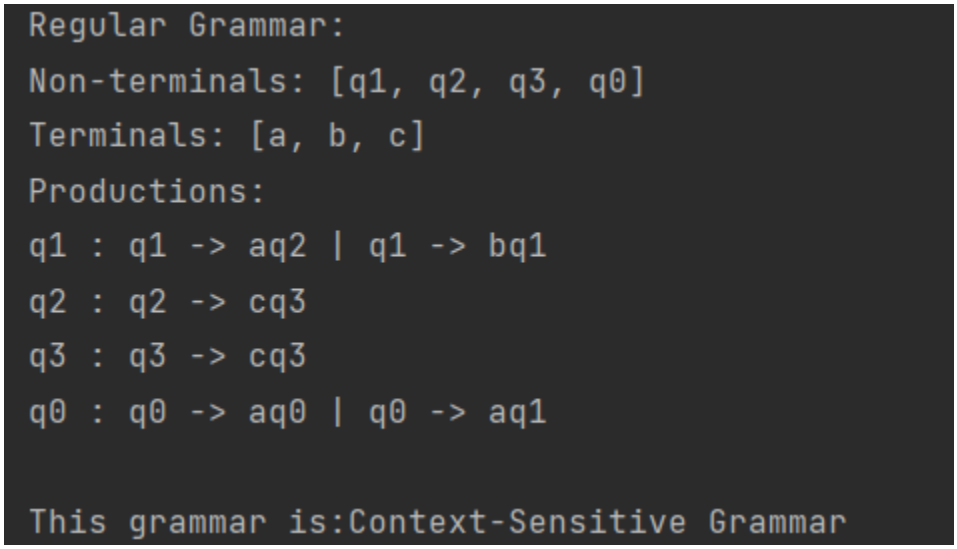
The class has instance variables for non-terminals, terminals, and production rules. Non-terminals

are represented as a set of strings, terminals are represented as a set of strings, and production rules are represented as a map where each non-terminal maps to a set of strings representing its production rules.

The constructor initializes these variables, and getter and setter methods are provided for each variable to allow access and modification.

The classifyGrammar method analyzes the production rules to determine the classification of the grammar. It checks the length and format of each production rule to determine if the grammar is regular, context-free, context-sensitive, or unrestricted. The method returns a string indicating the classification of the grammar.

Screenshots



```
Finite Automaton is Non-deterministic
NDFA -> DFA:
States: [, q1, q2, q3, q1q2q0, q1q0, q0]
Alphabet: [a, b, c]
Transitions:
q1:
    a -> [q2]
    b -> [q1]
    c -> []
:
    a -> []
    b -> []
    c -> []
q2:
    a -> []
    b -> []
    c -> [q3]
q3:
    a -> []
    b -> []
    c -> [q3]
q1q2q0:
    a -> [q1q2q0]
    b -> [q1]
    c -> [q3]
```

```
q1q0:
    a -> [q1q2q0]
    b -> [q1]
    c -> []
q0:
    a -> [q1q0]
    b -> []
    c -> []
Start State: q0
Final States: [q3]
```

Conclusions

Throughout this lab, we delved into the fundamental concepts of finite automata, grammars, and their interplay. By implementing a Java program, we constructed a robust `FiniteAutomaton` class to model finite automata and a versatile `Grammar` class to represent grammars. These classes allowed us to perform various operations, including converting a finite automaton to a regular grammar, determining determinism, and transforming a non-deterministic finite automaton into a deterministic one.

Through this practical exercise, we not only applied theoretical concepts but also gained valuable insights into their practical implications. By converting finite automata to regular grammars, we explored the relationship between the languages recognized by these structures. Additionally, by checking determinism, we deepened our understanding of the characteristics that define deterministic and non-deterministic automata. Finally, by converting a non-deterministic automaton to a deterministic one, we witnessed the importance of determinism in automata theory and its impact on computational complexity.

Overall, this lab provided a hands-on exploration of finite automata and grammars, highlighting their significance in formal language theory and computational theory. It underscored the practical application of these concepts in designing and analyzing algorithms and systems in various fields of computer science.