MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII

AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software și Automatică

Grebennicova Ecaterina FAF-223

Report

Determinism in Finite Automata. Conversion from NDFA 2 DFA. Chomsky Hierarchy.

of Formal Languages & Finite Automata

Checked by:

Cretu Dumitru, *university assistant*

Chișinău – 2023

1. **Theory**

A finite automaton (FA), also known as a finite state machine, is a theoretical model of computation used in computer science to simulate sequential logic and computational processes. It consists of a finite number of states, including at least one starting state, and can transition from one state to another in response to some inputs. The transitions between states are determined by a set of rules or a transition function.

Finite automata are categorized into two types: deterministic finite automata (DFA), where each state has exactly one transition for each possible input, and nondeterministic finite automata (NFA), where a state can have zero, one, or multiple transitions for each input. They are used in various applications, including text processing, compiler design, and network protocols, to model systems that can be in a finite number of different states.

The Chomsky hierarchy is a classification of formal languages based on their generative grammars, proposed by linguist Noam Chomsky. It divides languages into four levels: Type 0 (recursively enumerable), Type 1 (context-sensitive), Type 2 (context-free), and Type 3 (regular), with each type being strictly more restrictive than the one before it. This hierarchy helps in understanding the computational complexity of different languages and the power of various computational models.

2. **Objectives**

- Understand what an automaton is and what it can be used for
- Provide a function in your grammar type/class that could classify the grammar based on Chomsky hierarchy. For this to use the variant from the previous lab
- Implement conversion of a finite automaton to a regular grammar
- Determine whether your FA is deterministic or non-deterministic
- Implement some functionality that would convert an NDFA to a DFA

3. **Implementation Description**

Point 2 - implemented

In this program we have to classify a given set of production rules of a grammar into one of the four types of the Chomsky hierarchy: Type 3 (Regular Grammar), Type 2 (Context-Free Grammar), Type 1 (Context-Sensitive Grammar), and Type 0 (Unrestricted Grammar).

The code:

```
String[] productions = {
    "S -> aB",
    "B -> aD",
    "B -> bB",
    "D -> aD",
    "D -> bS",
    "B -> cS",
    "D -> c"
};
```

This array of strings represent the production rule of the grammar given from the previous lab. It is written that way to be able to split the production rules and do check the left sides and right sides separately.

```
for (String production : productions) {
    // Split the production rule into left and right parts based on "->"
    String[] parts = production.split("->");
    String leftSide = parts[0].trim();
    String rightSide = parts[1].trim();

    // Check for Type 3: Regular Grammar
    if (!rightSide.matches("^[a-c]*[SBD]?$")) {
        isType3 = false;
    }

    // Check for Type 2: Context-Free Grammar
    if (leftSide.length() != 1 || !leftSide.matches("[SBD]")) {
        isType2 = false;
    }

    // Check for Type 1: Context-Sensitive Grammar
    if (rightSide.length() < leftSide.length()) {
        isType1 = false;
        break;
    }
}
```

This code snippet shows the rules for checking the grammar type according to Chomsky.

Firstly, we check if the right side of the production matches the regular expression ^[a-c]*[SBD]?$. This regex checks for a sequence of terminals (a, b or c)

optionally followed by a single non-terminal (S, B, or D). If any production rule does not match this pattern, then it is not Type3.

Secondly, we check if we have a single non-terminal on the left side, exactly one character long and matches any of the non-terminals (S, B, or D). If not, then it is not Type2.

Thirdly, we check if the right side is not shorter than the left side. If the length of the right side of any production rule is less than the length of its left side, then it is not Type1.

If it is not Type 1, then it is Type 0.

Output:

```
The grammar is of type 3
```

Point 3 a, b - implemented

In this program we check if the given FA is deterministic or non-deterministic, and convert it to its equivalent regular grammar. The program is divided into three main parts: initialization and setup of the FA, checking determinism, and conversion to regular grammar. For initialization we are using HashMap, as it is easier to associate states with their corresponding transitions. We use this structure where the key is the string that represents the state, and the value is another map. In that map the key is the character given for transition (a or b) and the list of strings represent the state to which it makes the transit. We initialize these transitions and add them to the map.

```
// Initialize a map to hold the transitions of the FA
Map<String, Map<Character, List<String>>> transitions = new HashMap<>();

// Define transitions for state q0
Map<Character, List<String>> q0Transitions = new HashMap<>();
q0Transitions.put('a', List.of("q0"));
q0Transitions.put('b', List.of("q1"));

// Define transitions for state q1
Map<Character, List<String>> q1Transitions = new HashMap<>();
q1Transitions.put('a', Arrays.asList("q1", "q2"));
```

```
        q1Transitions.put('b', List.of("q3"));

        // Define transitions for state q2
        Map<Character, List<String>> q2Transitions = new HashMap<>();
        q2Transitions.put('a', List.of("q2"));
        q2Transitions.put('b', List.of("q3"));

        // Adding transitions to the map
        transitions.put("q0", q0Transitions);
        transitions.put("q1", q1Transitions);
        transitions.put("q2", q2Transitions);
```

Point a - Conversion of FA to grammar

```
public static void convertToRegularGrammar(Map<String, Map<Character,
List<String>>> transitions, String acceptingState) {
    for (String state : transitions.keySet()) {
        Map<Character, List<String>> stateTransitions = transitions.get(state);
        for (Map.Entry<Character, List<String>> transition :
stateTransitions.entrySet()) {
            char input = transition.getKey();
            for (String nextState : transition.getValue()) {
                System.out.println(state + " -> " + input + nextState);
            }
        }
        // If the state is an accepting state, it also goes to ε
        if (state.equals(acceptingState)) {
            System.out.println(state + " -> ε");
        }
    }
}
```

This method transforms a FA represented by its transitions into an equivalent regular grammar. For each state in the FA, it iterates over all possible transitions and prints out a corresponding production rule in the format “state -> input + nextState” for each transition. If a state is designated as the accepting state, it additionally prints a production rule “state -> ε”, indicating that the state can transition to the empty string (ε).

Point b - Determine if FA is deterministic or non-deterministic

```
public static boolean isDeterministicFA(Map<String, Map<Character, List<String>>>
transitions) {
```

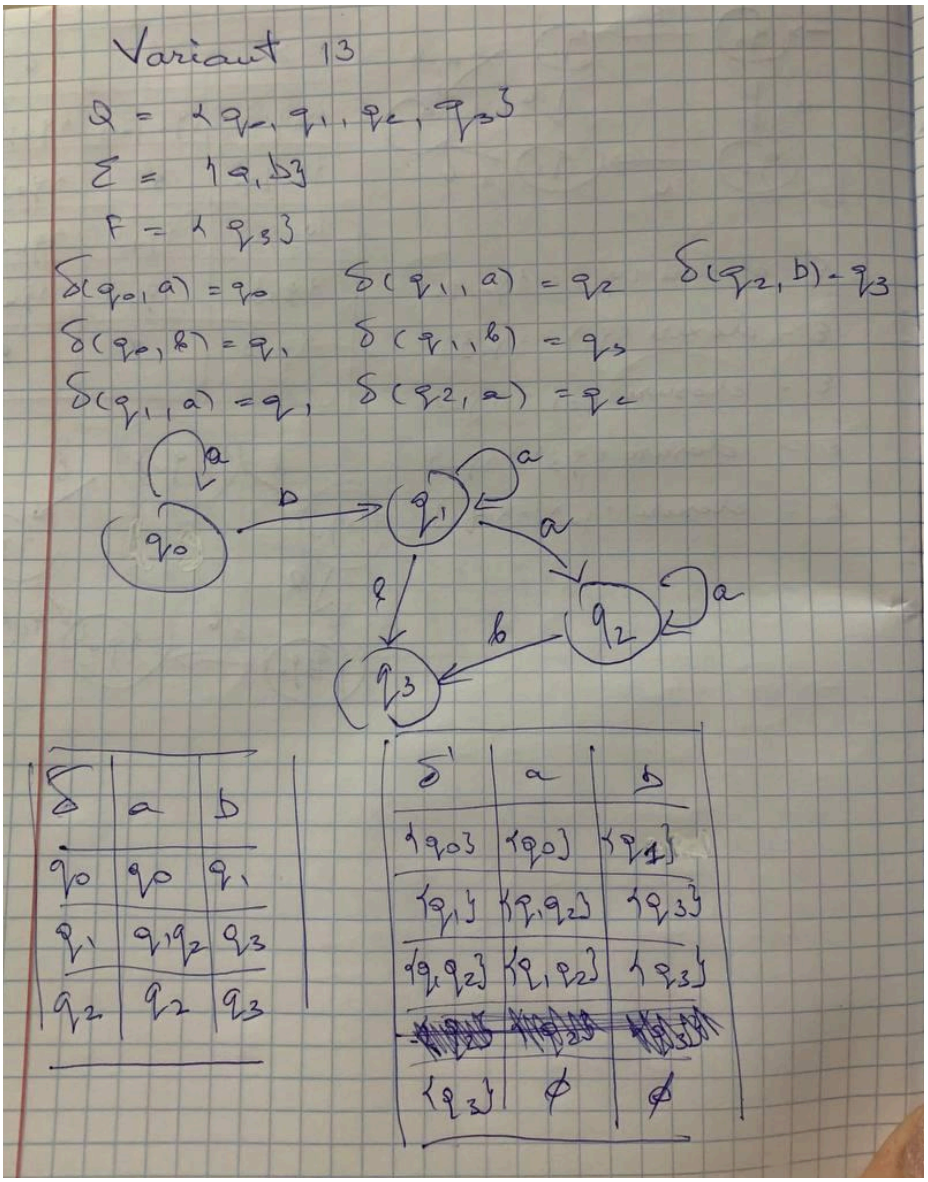
```
    for (Map<Character, List<String>> stateTransitions : transitions.values()) {
        for (List<String> targetStates : stateTransitions.values()) {
            if (targetStates.size() > 1) {
                // If any symbol leads to more than one state, it's
non-deterministic
                return false;
            }
        }
    }
    // If no input symbol leads to more than one state, the FA is deterministic
    return true;
}
```

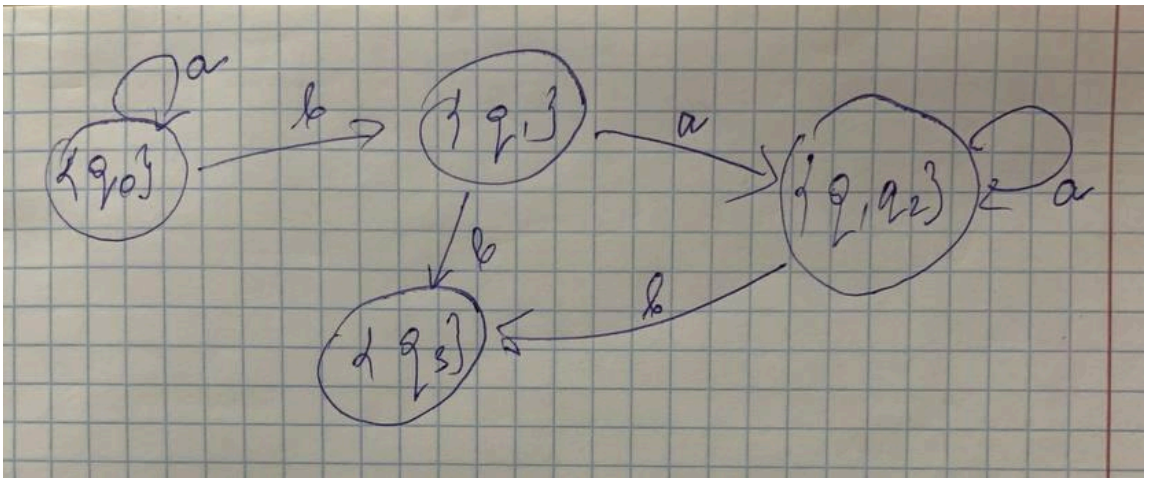
This method checks whether FA is deterministic or not. It iterates through all state transitions and if any input symbol for a state leads to more than one target state, the FA is classified as non-deterministic.. If no such case is found after checking all transitions, the FA is considered deterministic.

Output:

```
The FA is non-deterministic.
q1 -> aq1
q1 -> aq2
q1 -> bq3
q2 -> aq2
q2 -> bq3
q0 -> aq0
q0 -> bq1
```

Point c - Made manually on paper the conversion from NFA to DFA.





Here, we create the transition graphically in order to better visualize the transitions. After that we create the transition table listing all the states and their transitions for each input symbol. Then we create the second table for a new automaton, where we reconstruct the states and the transitions. If the initial state

leads to another state, we have to introduce this new state in the initial states of DFA. Proceeding this way we complete the table. Using the info about states, their transitions and inputs we recreate the graph.

4. Conclusion

By understanding better how the grammar and FA work, it is more clear how to implement the tasks. As well I spent time reading once again the Chomsky hierarchy and the types which represent the grammar, and their ways of identification. Now, it is easier to make the 2 lab as it has some similarities with the 1 one. The first created program is used to to classify a given set of production rules of a grammar into one of the four types of the Chomsky hierarchy: Type 3 (Regular Grammar), Type 2 (Context-Free Grammar), Type 1 (Context-Sensitive Grammar), and Type 0 (Unrestricted Grammar) using some specific simple rules. The second program is about checking if the given FA is deterministic or non-deterministic and its conversion to its equivalent regular grammar. And for the last point where we have to convert the NFA to DFA I made it manually the way the teacher told us at the course to draw it in a simple and logical way.