**MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII**

**AL REPUBLICII MOLDOVA**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică şi Microelectronică**

**Departamentul Inginerie Software și Automatică**

**Mihalachi Mihai FAF-221**

# Report

*Laboratory work no.2*

## *of Formal Languages & Finite Automata*

**Chișinău – 2024**

# Theory

A finite automaton is a mathematical model used in computer science and theoretical computer science to describe computation and control processes. It is a simplified abstraction of a computing device with a limited amount of memory, known as states.

A finite automaton consists of the following components:

- States: A finite set of states that the automaton can be in. Each state represents a specific condition or situation.
- Alphabet: A finite set of symbols or inputs that the automaton can read. These are the inputs that trigger state transitions.
- Transitions: A set of rules that define how the automaton transitions from one state to another based on input symbols. Each transition specifies the current state, the input symbol, and the next state.
- Initial State: The starting state of the automaton.
- Final States: Some states are designated as accepting states. If the automaton reaches an accepting state after processing a sequence of inputs, it is said to accept that input sequence.

Finite automata are classified into two main types:

- Deterministic Finite Automaton (DFA): In a DFA, for each state and input symbol, there is exactly one next state. The transition from one state to another is uniquely determined.
- Nondeterministic Finite Automaton (NFA): In an NFA, there can be multiple possible next states for a given state and input symbol. It has a more flexible transition structure.

Finite automata are used in various areas of computer science, including compiler design, formal language theory, and the modeling of systems with a finite number of possible states and transitions. They provide a formal and concise way to represent and analyze the behavior of systems with limited memory and discrete states.

# 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.

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.

# Implementation description:

1. The method used that classifies the grammar based on Chomsky hierarchy looks like this:

```
def check_type(self):
        if all(string in (set(self.VN) | self.VT) and key in self.VN for key, array in
    self.P.items() for string in array):
            return "Grammar is not valid"
        value = "Type 0 - Recursively Enumerable Grammar"
        if all((len(string) == 2 and string[0].islower() and string[1].isupper()) or
               (len(string) == 1 and string.islower()) for array in self.P.values() for
    string in array):
            return "Type 3 - Right Regular Grammar"
        elif all((len(string) == 2 and string[0].isupper() and string[1].islower()) or
                 (len(string) == 1 and string.islower()) for array in self.P.values() for
    string in array):
            return "Type 3 - Left Regular Grammar"
        if any((sum(symbol.isupper() for symbol in string) > 1 or len(string) > 2
                  or sum(symbol.islower() for symbol in string) > 1 or string == '')
                    for array in self.P.values() for string in array):
            value = "Type 2 - Context-Free Grammar"
        if (any(len(state) > 1 for state in self.P.keys()) and
                all(len(state.strip()) <= len(string) for state in self.P.keys() for
    string in self.P[state])):
            value = "Type 1 - Context-Sensitive Grammar"
            return value
```

   I went from checking the Type 3 to Type 1 (if it's not Type 1 then by default it's Type 0). I checked the main rule of Type 3, then going to the next type I used a rule that is not present on the higher type(e.g Type 2 can have empty string on the right side).

2. The method that convers finite automaton to grammar looks like this:

```
      def to_grammar(self):
              VN = list(self.Q)
              VT = self.Σ
              P = {}
              for key in self.δ.keys():
                      if key[0] not in P.keys():
                              P[key[0]] = []
                      for symbol in self.δ[key]:
                              if symbol == key[1]:
                                      production = symbol
                              else:
                                      production = key[1] + symbol
                              P[key[0]].append(production)
              return g.Grammar(VN, VT, P)
```

For the grammar conversion I did the reverse of the conversion to finite automaton method where I combine the transition symbol back to the state it transitions. Also for final state where I have the non-terminal symbol as a transition state, I check for it and not include it.

3. For checking if a FA is NFA or DFA I simply just check if the transitions dictionary has at least one set with length bigger than 1:

```
def is_nfa(self) -> bool:
        return any(len(production) > 1 for production in self.δ.values())
```

4. For converting NFA to DFA I did it manually, because some things just couldn't get working in code. The following images show the graph for the NFA, the table representation, then the table representation for the converted NFA to DFA, then it's graph:

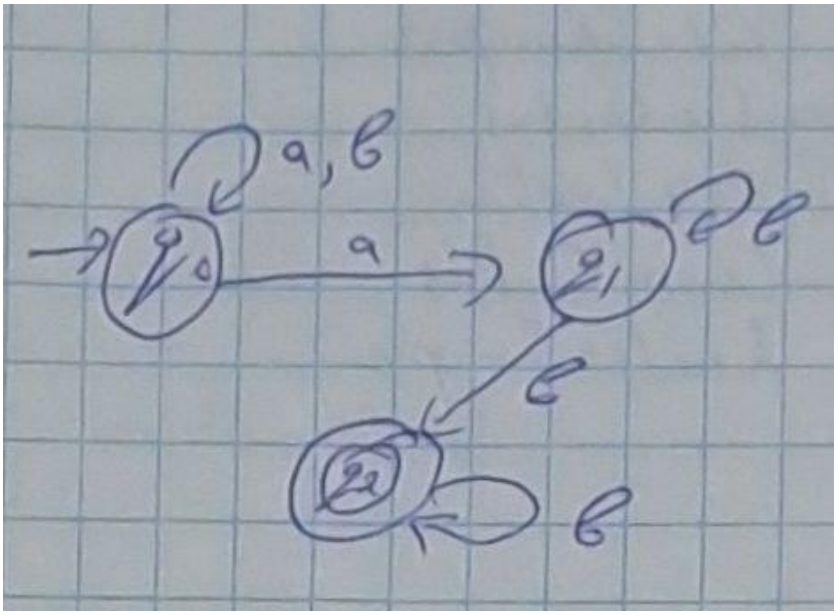


Figure 1: NFA graph

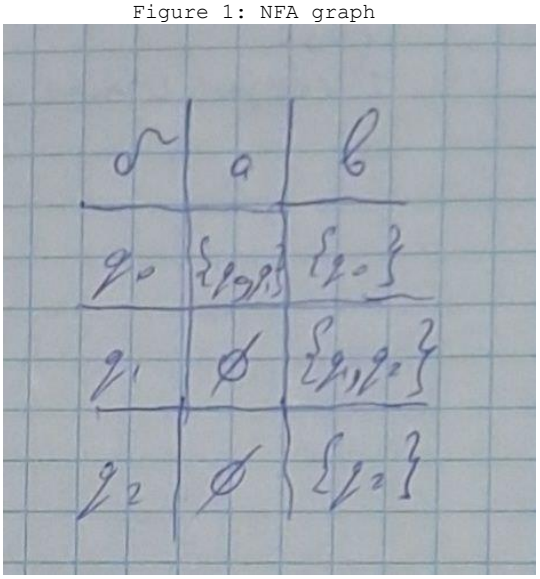


Figure 2: NFA table

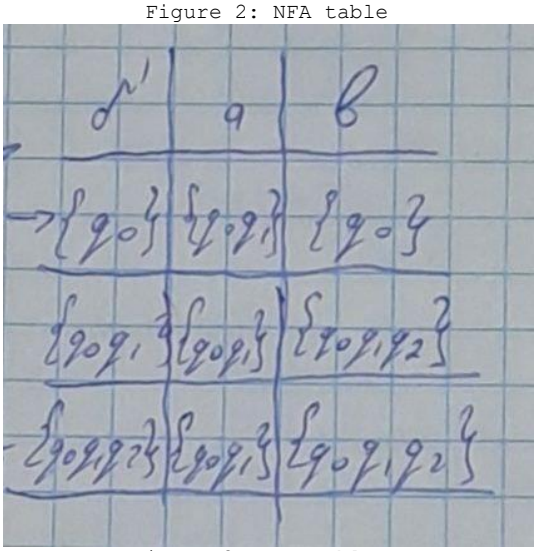


Figure 3: DFA table

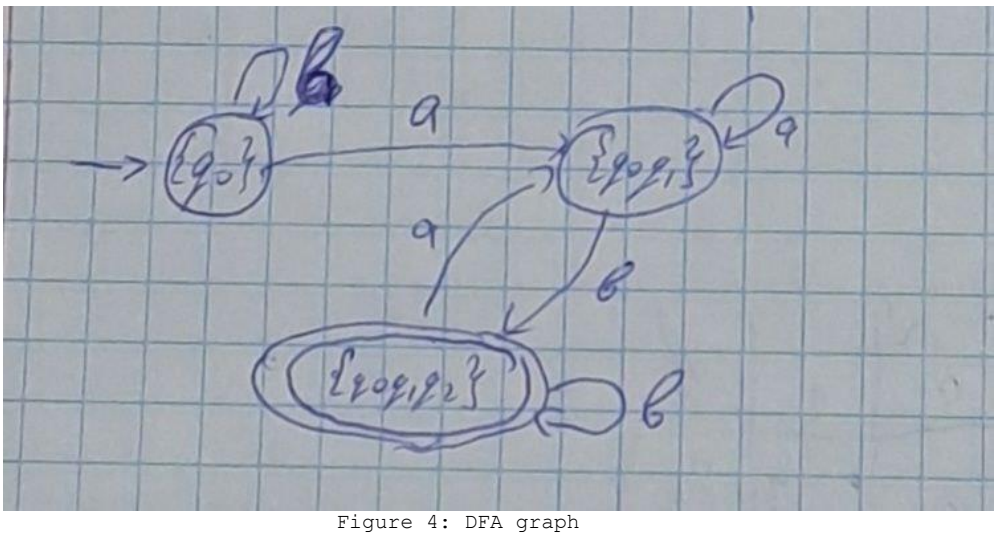

Figure 4: DFA graph

# Conclusion

This project involving the practice of finite automaton offers a valuable opportunity to deepen understanding and practical application of theoretical computer science concepts. The project's outcomes include a better grasp of formal language theory, and improved proficiency in translating abstract concepts into practical solutions.

Furthermore, this laboratory fosters creativity, encouraging students to explore different approaches in designing finite automata to achieve specific objectives. The skills acquired during this laboratory can prove beneficial in broader contexts within computer science, where finite automata play a role in modeling and solving computational problems. Overall, such a project provides a valuable learning experience that combines theoretical knowledge with practical implementation, contributing to the development of well-rounded computer science skills.