

**MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică**

**Ciumacenco Pavel FAF-232**

**Report**

*Laboratory work nr.2*

***Course: Formal Languages & Finite Automata***

Checked by:

**Cretu Dumitru,** *university prof.*

FCIM, UTM

**Chișinău – 2025**

1. **Theory:**

### Introduction to Finite Automata

Finite automata are mathematical models of computation that recognize patterns within input strings. These automata serve as the foundation for regular languages and are widely used in various applications, such as lexical analysis in compilers, text processing, and artificial intelligence. Finite automata operate using a finite set of states, a finite alphabet, a transition function, an initial state, and one or more accepting states.

Finite automata can be classified into two main types:

* **Deterministic Finite Automaton (DFA):** Each state has exactly one transition for each symbol in the alphabet.
* **Nondeterministic Finite Automaton (NDFA/NFA):** States can have multiple transitions for the same input symbol, including epsilon () transitions that allow state changes without consuming input.

**Determinism in Finite Automata**

Determinism in finite automata means that for a given state and input symbol, the transition function specifies exactly one possible next state. This ensures predictable and unique execution paths. In contrast, nondeterministic finite automata allow multiple possible transitions, meaning that computation can branch into multiple paths.

Despite their structural differences, DFAs and NDFAs are equivalent in expressive power; any language recognized by an NDFA can also be recognized by a DFA. However, the conversion from NDFA to DFA may lead to an exponential increase in the number of states.

### Conversion from NDFA to DFA

The process of converting an NDFA to a DFA is known as the **subset construction method** or **powerset construction**. This method involves the following steps:

1. **Identify the Start State:** The start state of the DFA corresponds to the epsilon-closure of the start state of the NDFA.
2. **Compute Transitions:** For each DFA state (which is a set of NDFA states), determine the next set of states for each input symbol.
3. **Mark Accepting States:** Any DFA state that contains at least one accepting NDFA state is marked as an accepting state.
4. **Eliminate Unreachable States:** Some states in the DFA may not be reachable from the start state; these are removed.

This construction can lead to an exponential increase in the number of states, meaning that the resulting DFA may be much larger than the original NDFA.

### Chomsky Hierarchy

The Chomsky hierarchy classifies formal languages based on their generative power and the type of computational model required to recognize them. The hierarchy consists of four levels:

1. **Type 0 (Recursively Enumerable Languages):** Recognized by Turing machines, these languages include all computable languages but may not have a halting guarantee.
2. **Type 1 (Context-Sensitive Languages):** Recognized by linear-bounded automata, these languages require context-sensitive grammars where productions must not decrease the length of the string.
3. **Type 2 (Context-Free Languages):** Recognized by pushdown automata, context-free languages (CFLs) can be described by context-free grammars. Examples include programming language syntax.
4. **Type 3 (Regular Languages):** Recognized by finite automata (both DFA and NDFA), these languages have simple rules and can be expressed using regular expressions.

Each level in the hierarchy is a strict subset of the previous one, meaning that every regular language is also context-free, every context-free language is context-sensitive, and so on.

### Deterministic vs. Nondeterministic Models

Deterministic and nondeterministic computational models differ in how they process inputs:

* **Deterministic Models:** Given a state and input, there is exactly one possible transition, ensuring a single computational path.
* **Nondeterministic Models:** Given a state and input, multiple transitions are possible, leading to multiple computational paths.

While nondeterminism can simplify design and representation, deterministic models are often more practical for implementation. The key theoretical result is that for finite automata and pushdown automata, nondeterminism does not increase expressive power, though it may affect efficiency.

### Applications of Finite Automata

Finite automata are used in numerous applications, including:

* **Lexical Analysis:** Tokenizing input in programming languages using regular expressions.
* **Text Searching:** Algorithms like the Aho-Corasick algorithm use finite automata for efficient pattern matching.
* **Protocol Verification:** Modeling and verifying network protocols and software systems.
* **Artificial Intelligence:** Recognizing speech patterns and processing natural language.

1. **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.

Variant 9

Q = {q0,q1,q2,q3,q4},

∑ = {a,b,c},

F = {q4},

δ(q0,a) = q1,

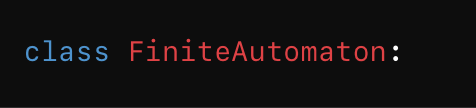
δ(q1,b) = q2,

δ(q2,c) = q0,

δ(q1,b) = q3,

δ(q3,a) = q4,

δ(q3,b) = q0.

1. **Implementation Description:  
     
     
   **Figure 1: code snippet

This defines a class to represent a **Finite Automaton (FA)**, capable of handling both deterministic and non-deterministic versions. It encapsulates the properties and behavior of an FA, such as states, transitions, and conversion utilities.

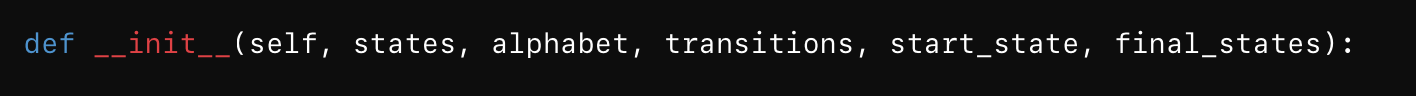
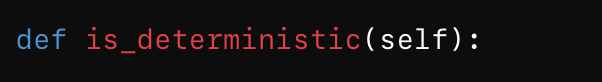
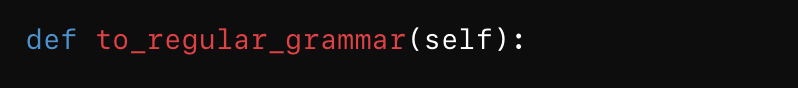
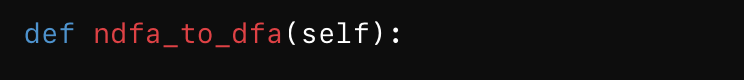


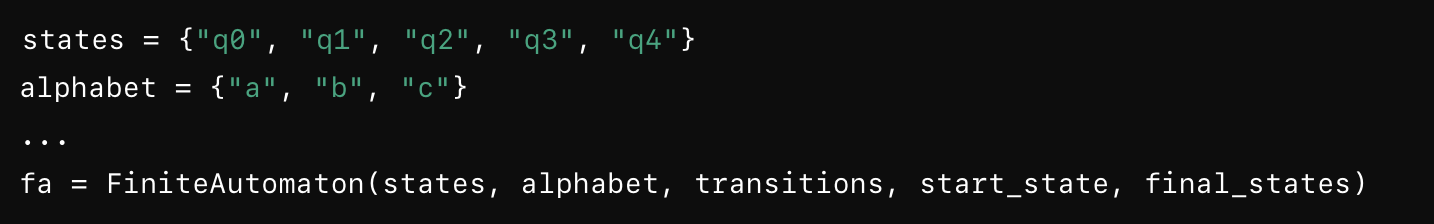
Figure 2: code snippet  
 The constructor initializes the automaton with a given set of states, an alphabet, a transitions dictionary, the start\_state, and the set of final\_states. This sets the foundational structure for the automaton’s behavior and transformations.

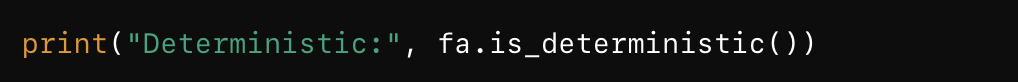
Figure 3: code snippet  
 This method checks whether the FA is deterministic by ensuring that each state has **at most one** transition for each symbol in the alphabet. If any symbol maps to more than one destination state from the same state, it returns False, indicating it's an NDFA.

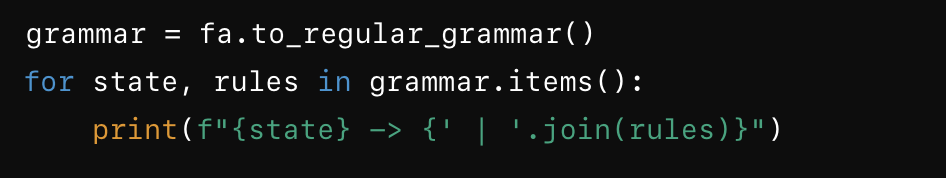
Figure 4: code snippet  
 This function converts the FA into a **Regular Grammar** representation using the transitions. For each transition from a state via a symbol to a destination state, it constructs a grammar rule in the form symbolDestination.

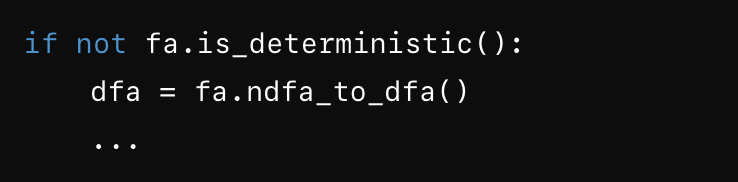
Figure 5: code snippet  
 This method converts a **Non-Deterministic Finite Automaton (NDFA)** into a **Deterministic Finite Automaton (DFA)** using the subset construction technique.

It creates new DFA states represented by sets of NDFA states (as frozenset) and builds new transitions accordingly. Final states in the DFA are any composite state that contains an original NDFA final state.

Figure 6: code snippet  
 These lines define the components of the given FA: states, alphabet symbols, transition functions, start state, and final states. An instance of FiniteAutomaton is created using these values.

Figure 7: code snippet  
 This prints whether the given FA is deterministic by calling the is\_deterministic() method. It's a basic diagnostic check to determine if conversion to DFA is needed.

Figure 8: code snippet  
 This block converts the FA to a regular grammar and prints each state's production rules. It displays the grammar in a familiar notation, useful for theoretical analysis or further processing.

Figure 9: code snippet  
 If the original automaton is not deterministic, this section performs the conversion to DFA. It outputs the new DFA’s states, final states, and the full transition table.

**4.Conclusions.Screenshots.Results.**

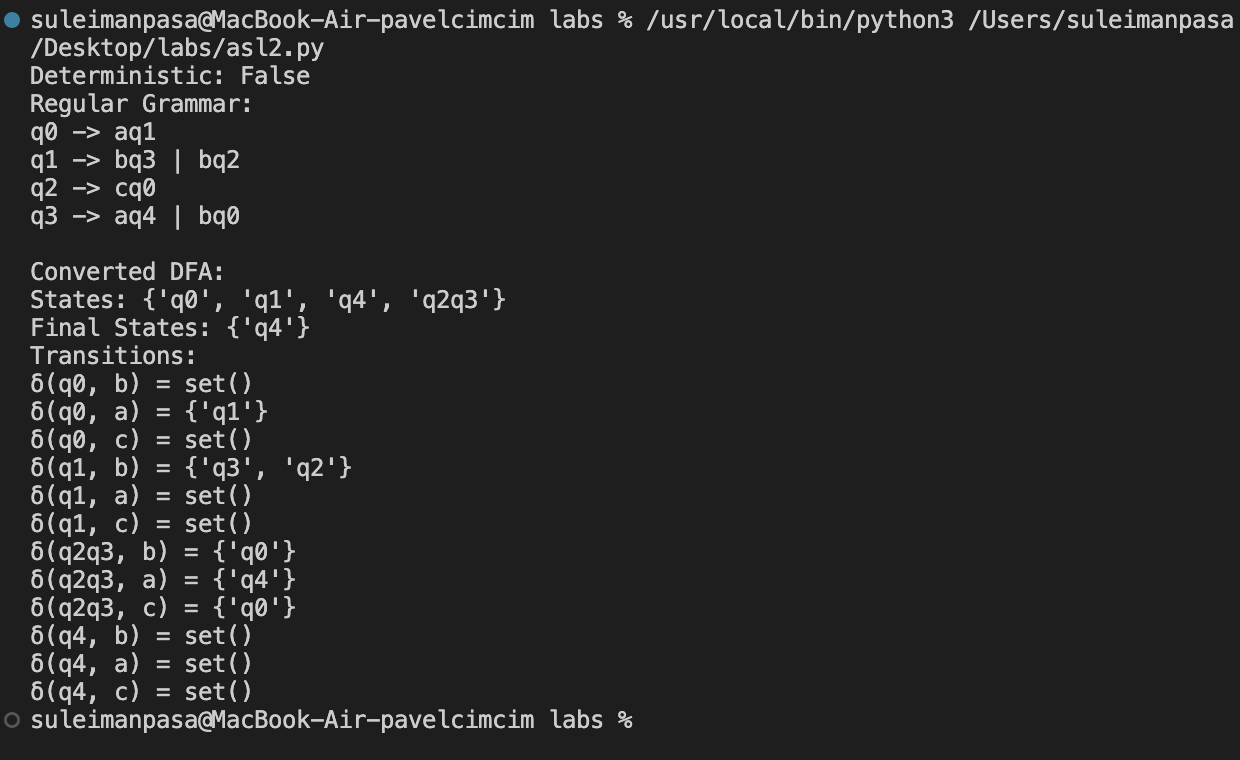
This implementation of a Finite Automaton (FA) effectively models both deterministic (DFA) and non-deterministic (NDFA) automata. It provides clear functionality to check determinism, convert an FA into a regular grammar, and transform an NDFA into an equivalent DFA using the subset construction method. The code is modular and extensible, making it a strong foundation for deeper exploration of automata theory and formal language processing. Overall, it demonstrates how theoretical computer science concepts can be translated into practical, executable Python logic.  


Figure 10: Results Picture