

**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.1*

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

Checked by:

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

FCIM, UTM

**Chișinău – 2025**

1. **Theory:**

### Formal Languages and Finite Automata

Alright, so a **formal language** is basically a structured way of representing information. It’s not like natural languages (English, Spanish, etc.), which have tons of irregularities. Instead, formal languages follow strict rules to ensure clarity and avoid misinterpretation. They’re often used in computing, logic, and mathematics.

A formal language consists of three main parts:

* **Alphabet**: The set of symbols you can use (e.g., {0,1} in binary, {a, b, c} in some toy language).
* **Vocabulary**: All the valid words you can form using the alphabet.
* **Grammar**: The rules that determine which sequences of words are allowed and how they can be structured.

For example, in programming, Python has its own formal language where the alphabet includes letters, numbers, and symbols, the vocabulary consists of valid keywords like if, while, def, and the grammar dictates how these can be combined to form valid programs.

### Finite Automata (FA)

Now, formal languages often need to be processed by some kind of system, and that’s where **Finite Automata (FA)** come in. These are abstract mathematical models used to recognize patterns and decide if a given input string belongs to a certain language. They’re super important in areas like compiler design, text processing, and even AI.

A **Finite Automaton** consists of:

* **A finite set of states** (including at least one start state and one or more final/accepting states).
* **An alphabet** (the symbols the automaton reads).
* **A transition function** (which tells the automaton where to move based on input).

#### Types of Finite Automata

There are two major types of FA:

1. **Deterministic Finite Automaton (DFA)**
   * For each state and input, there’s exactly **one** possible next state.
   * It’s predictable and easy to implement but can require a lot of states.
2. **Nondeterministic Finite Automaton (NFA)**
   * Allows multiple possible next states for a given input.
   * Can have transitions that don’t consume input (ε-transitions).
   * More flexible than a DFA, but in practice, both have the same computational power (you can always convert an NFA into a DFA).

#### Composition of Finite Automata

A finite automaton is typically represented as a **5-tuple**:  
**FA = (Q, Σ, δ, q0, F)**where:

* Q is the set of states.
* Σ (sigma) is the alphabet.
* δ (delta) is the transition function.
* q0 is the start state.
* F is the set of accepting states.

1. **Objectives:**

-Discover what a language is and what it needs to have in order to be considered a formal one;

-Provide the initial setup for the evolving project that you will work on during this semester. You can deal with each laboratory work as a separate task or project to demonstrate your understanding of the given themes, but you also can deal with labs as stages of making your own big solution, your own project. Do the following:  
  
a. Create GitHub repository to deal with storing and updating your project;  
b. Choose a programming language. Pick one that will be easiest for dealing with your tasks, you need to learn how to solve the problem itself, not everything around the problem (like setting up the project, launching it correctly and etc.);  
c. Store reports separately in a way to make verification of your work simpler (duh)

-According to your variant number, get the grammar definition and do the following:  
a. Implement a type/class for your grammar;  
b. Add one function that would generate 5 valid strings from the language expressed by your given grammar;  
c. Implement some functionality that would convert and object of type Grammar to one of type Finite Automaton;  
d. For the Finite Automaton, please add a method that checks if an input string can be obtained via the state transition from it;

-Variant 9:

VN={S, B, D, Q},

VT={a, b, c, d},

P={

S → aB

S → bB

B → cD

D → dQ

Q → bB

D → a

Q → dQ

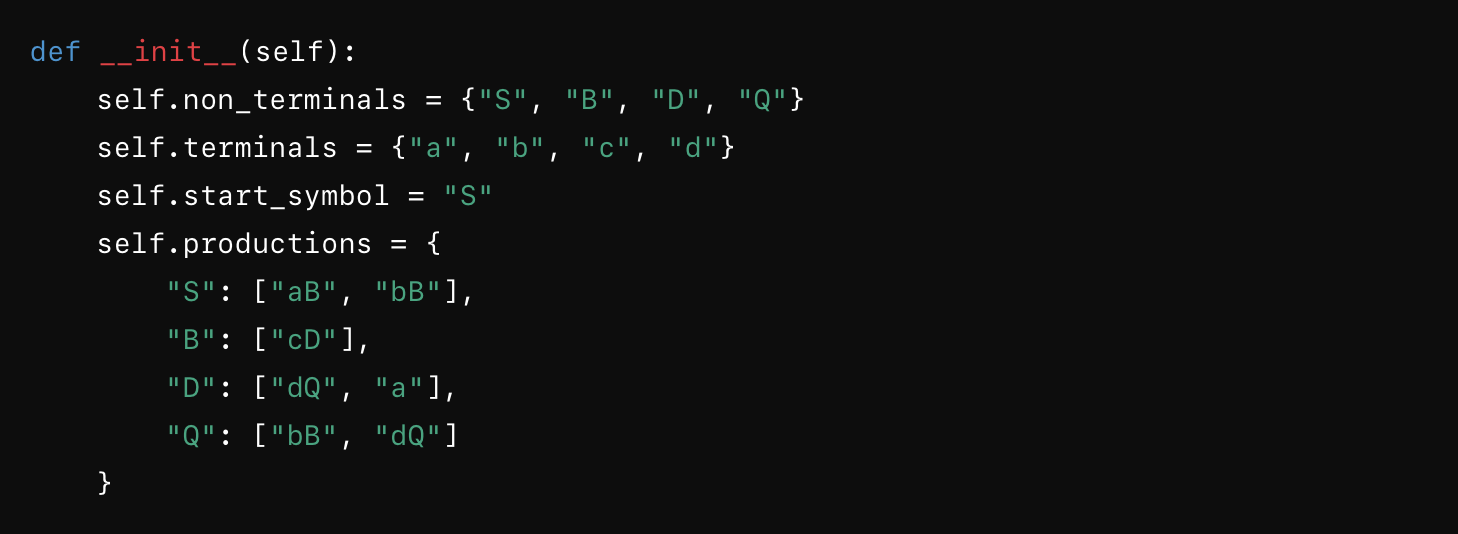
}

1. **Implementation Description:**

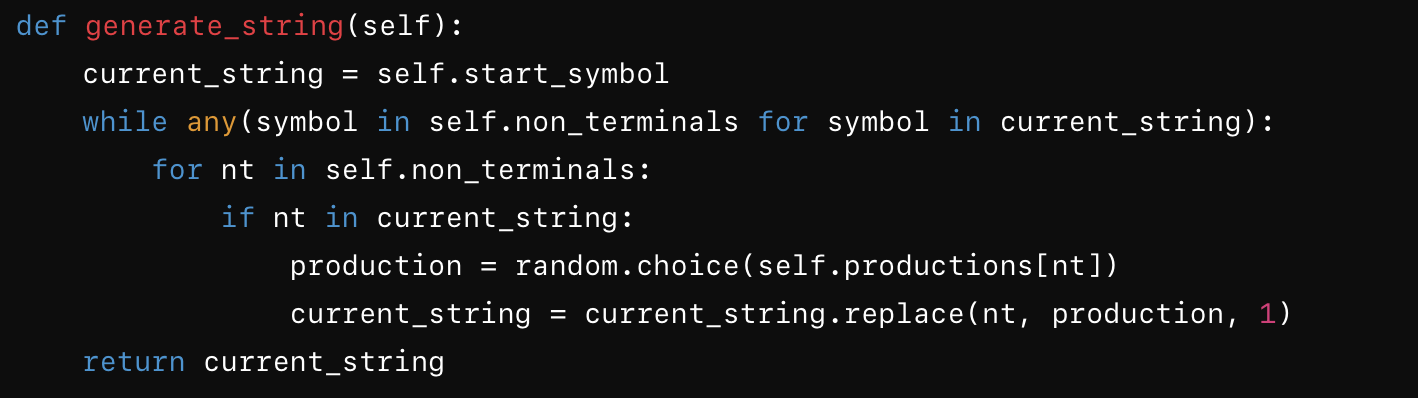
This implementation consists of two main classes: Grammar and FiniteAutomaton. The Grammar class defines a context-free grammar with production rules, generates valid strings based on the grammar, and converts itself into a FiniteAutomaton. The FiniteAutomaton class represents a finite state machine that determines whether a given string belongs to the language defined by the grammar.

The Grammar class initializes the following elements:

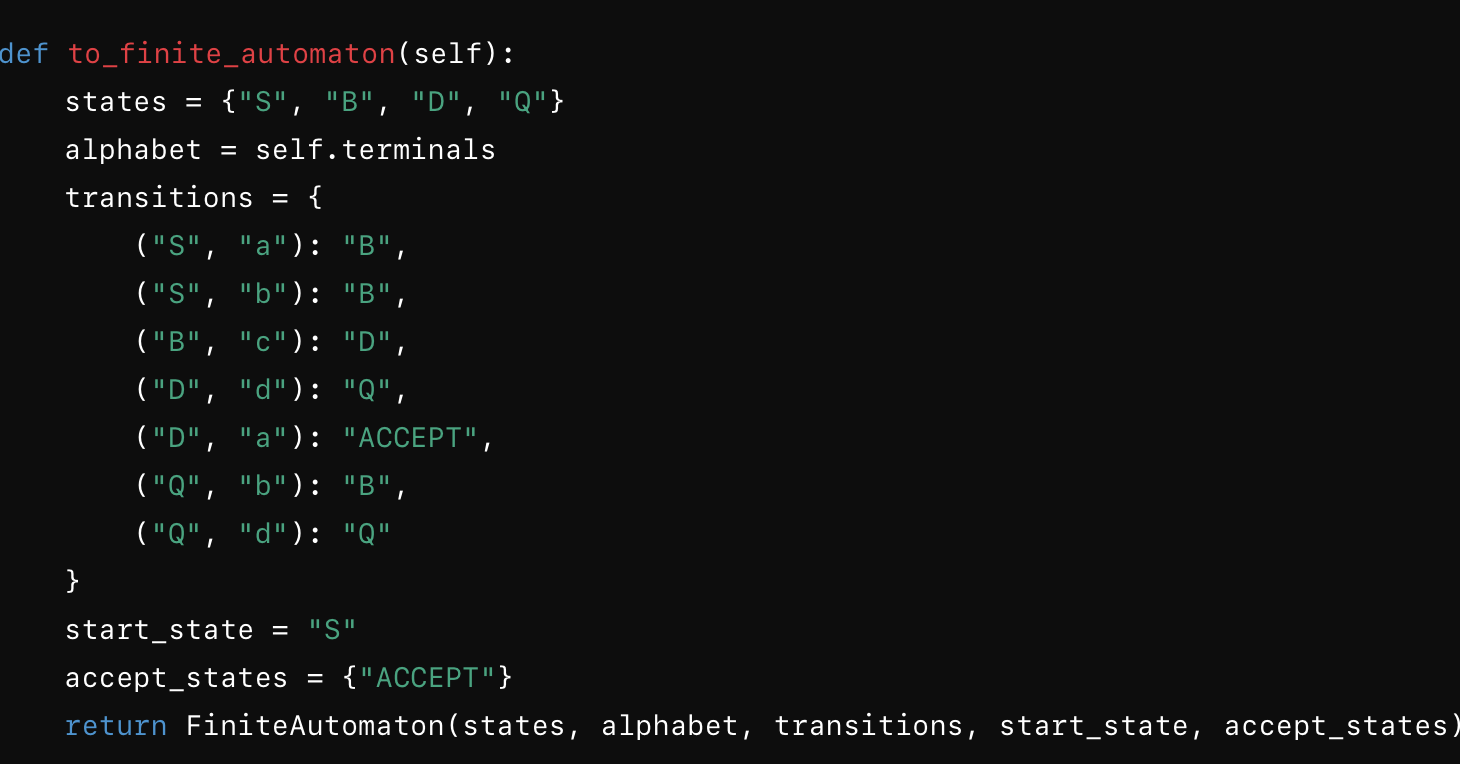
* **Non-Terminals (VN)**: {S, B, D, Q}
* **Terminals (VT)**: {a, b, c, d}
* **Production Rules (P)**: Defined in a dictionary where each non-terminal maps to possible right-hand side replacements.
* **Start Symbol**: "S"

  
figure 1: This constructor initializes the grammar components needed for string generation and conversion to a finite automaton.

The method generate\_string() creates a valid string from the grammar by recursively replacing non-terminals with their production rules until only terminals remain. It selects replacement rules randomly to ensure variation.

  
figure 2:This method ensures that a fully formed string adhering to the grammar is produced.

The function generate\_multiple\_strings(count=5) simply calls generate\_string() five times to produce a set of different valid strings.

The to\_finite\_automaton() method constructs a corresponding finite automaton from the grammar by mapping states and transitions:  
  
  
  
figure 3: This method transforms the grammar into a finite automaton representation where states correspond to non-terminals, and transitions represent valid symbol replacements.

The FiniteAutomaton class represents the automaton's components:

* **States**: Set of all states ({S, B, D, Q, ACCEPT})
* **Alphabet**: The set of valid symbols ({a, b, c, d})
* **Transitions**: A dictionary mapping (state, symbol) -> next\_state
* **Start State**: "S"
* **Accept States**: {ACCEPT} (indicating valid final strings)

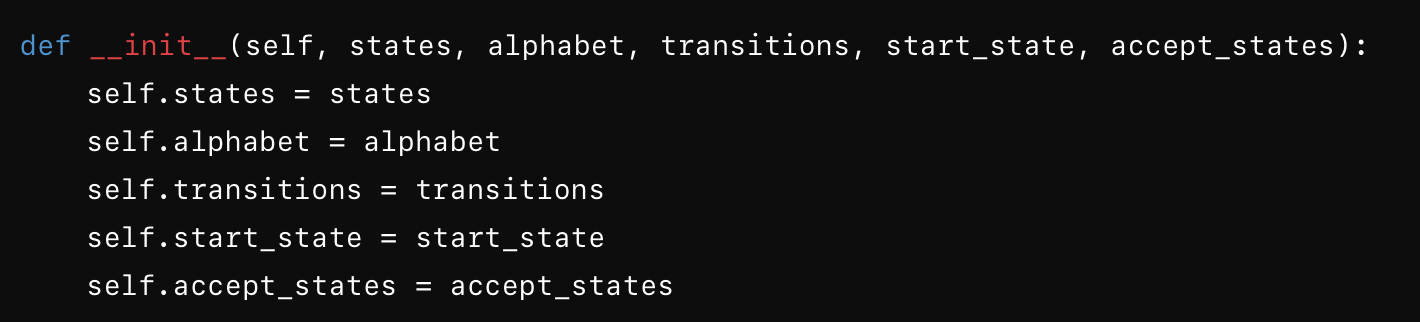
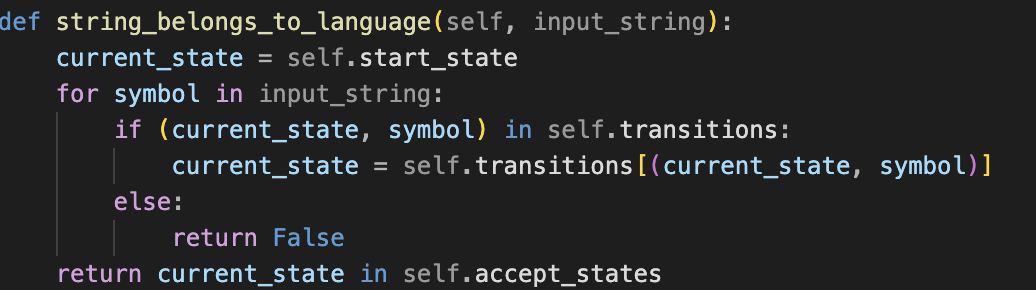
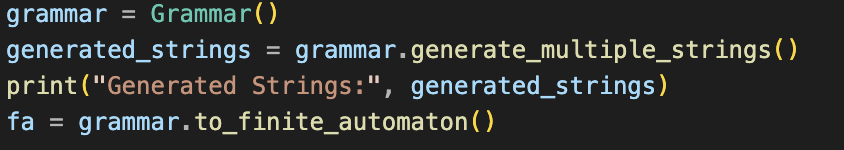
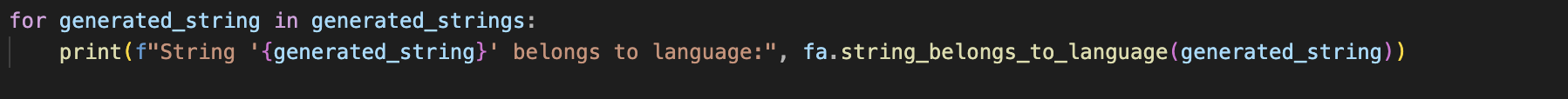


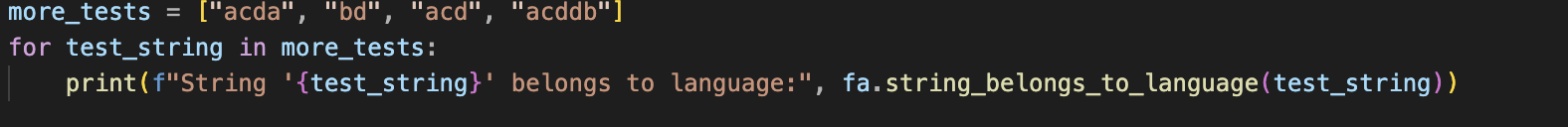
figure 4: This constructor initializes the finite automaton’s structure.  
  
The `string\_belongs\_to\_language()` method determines if a given string can be derived using state transitions.  
  
figure 5: This function starts at the initial state and follows transitions based on input symbols.

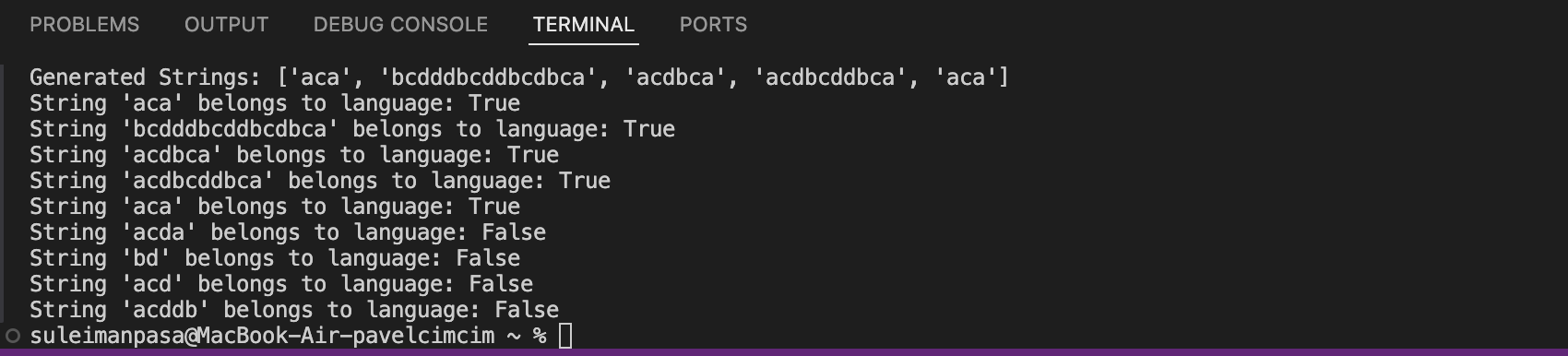
This function starts at the initial state and follows transitions based on input symbols. If a transition doesn't exist, the function returns `False`. If it reaches an accept state at the end, it returns `True`.  
To demonstrate functionality:

1. The `Grammar` object generates five valid strings:  


2. The finite automaton checks whether given strings belong to the language:

3. Additional hardcoded test cases are checked:

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

This implementation effectively models a context-free grammar and a corresponding finite automaton.   
  
 It allows for:The generation of valid strings from the grammar, the conversion of the grammar into a finite automaton, and the verification of whether a string can be derived through state transitions. This approach provides a structured way to analyze and process formal languages using both grammatical rules and automata-based recognition.