**MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA TECHNICAL UNIVERSITY OF MOLDOVA FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS**

**Laboratory work 1:**

**Regular Grammars & Finite Automata**

**Course: Formal Languages & Finite Automata**

**Author:Ostafi Eugen, FAF-222**

**Chișinău, 2024**

**THEORY**

Exploring formal languages and automata is a key part of computer science theory, helping us understand how strings of text are processed and recognized by computers. At the heart of this area are regular grammars and finite automata, which are tools that help describe and identify patterns in text. Regular grammars use a set of straightforward rules to outline how strings in languages are formed, using specific symbols and a starting point to generate these strings. Meanwhile, finite automata, which come in two types—deterministic (DFA) and nondeterministic (NFA)—act like machines that scan through strings to check if they follow the language's rules. The process of turning a regular grammar into a finite automaton shows how closely related they are, as both can represent the same language patterns. This connection reveals how the rules for creating language and the steps for checking it work together, deepening our understanding of how computers process language. This knowledge is not just academic; it has practical uses in creating software that processes text, showing the real-world value of these theoretical concepts.

**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

• According to variant 1, get the grammar definition and do the following:

a. Implement a type/class for your grammar;

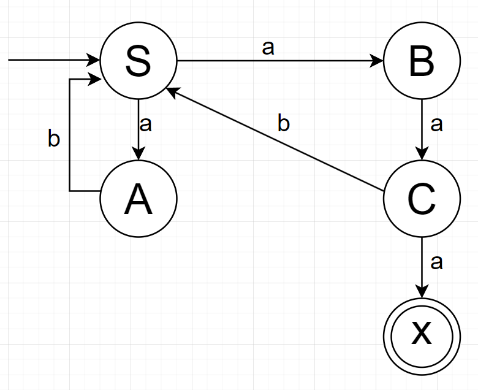
b. Add one function that would generate 5 valid strings from the language expressed by your given grammar;

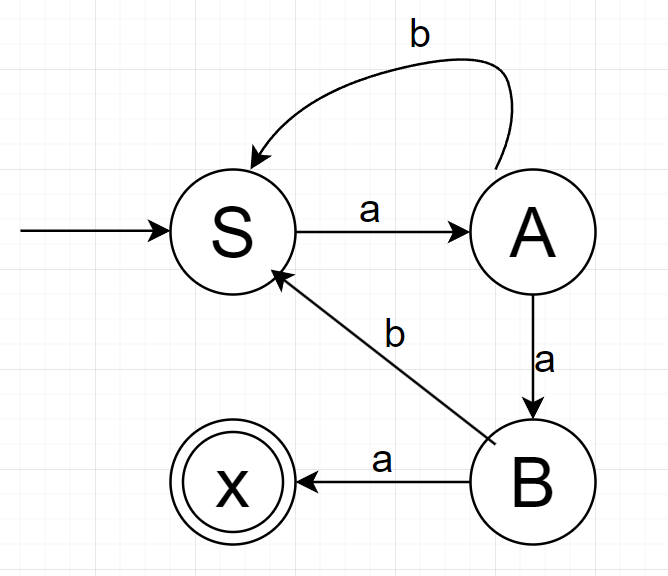
c. Implement some functionality that would convert and object of type Grammar to one of type Finite Automaton;

d. For the Finite Automaton, please add a method that checks if an input string can be obtained via the state transition from it;

**IMPLEMENTATION DESCRIPTION**

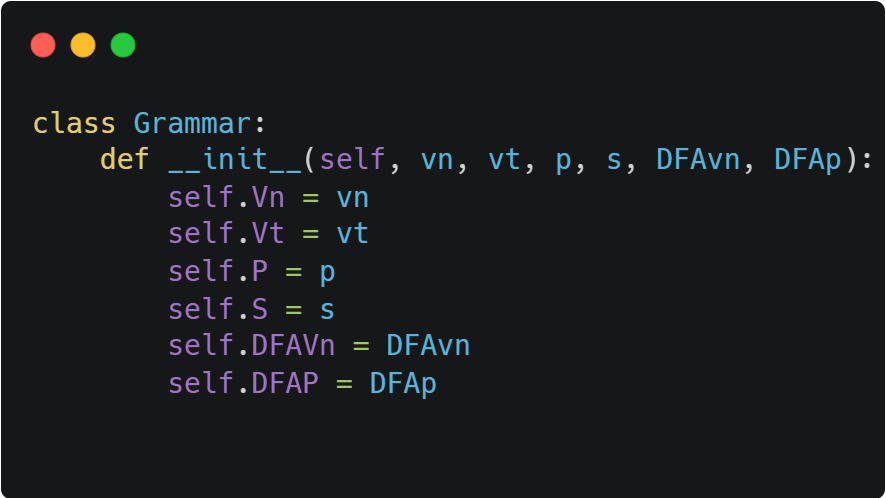
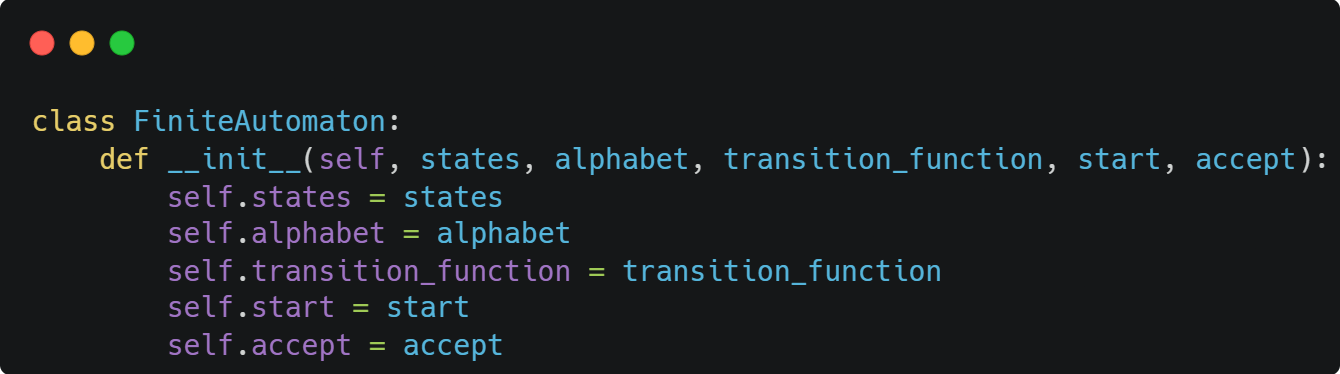
To avoid some problems in my code I transform my NFA automaton to DFA automaton:

 NFA: DFA:



------------------>

For this laboratory I chose to work in python. The implementation comprises two main classes: Grammar and FiniteAutomaton:

****

Next, I add to the Grammar class a method that generates five words based on the NFA grammar/

****

Here's a breakdown of what it does:

It initializes an empty list called words.

It enters a while loop that runs until len(words) is equal to 5.

Inside this loop, it initializes a string variable str with the value "S".

It enters another while loop that runs as long as the lowercase version of str is not equal to str. This seems to be a check to ensure that the string str contains at least one lowercase character.

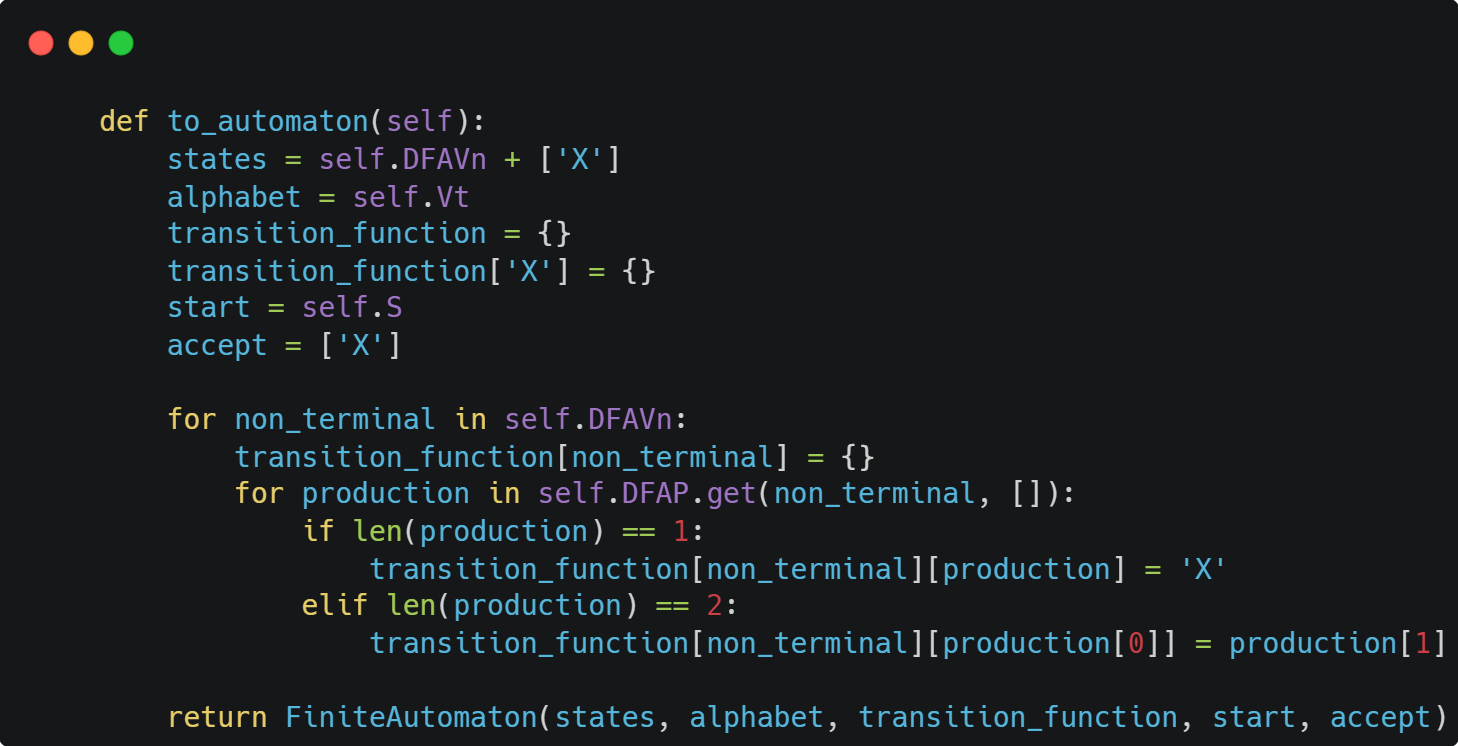
Within this inner loop, it replaces the last character of the string str with a randomly chosen character from self.P[str[len(str) - 1]]. The replacement character is chosen using random.randint(0, len(self.P[str[len(str) - 1]]) - 1).

If the resulting string str is not already in the words list, it appends str to the list.

Once the words list contains 5 unique strings, it returns the list.

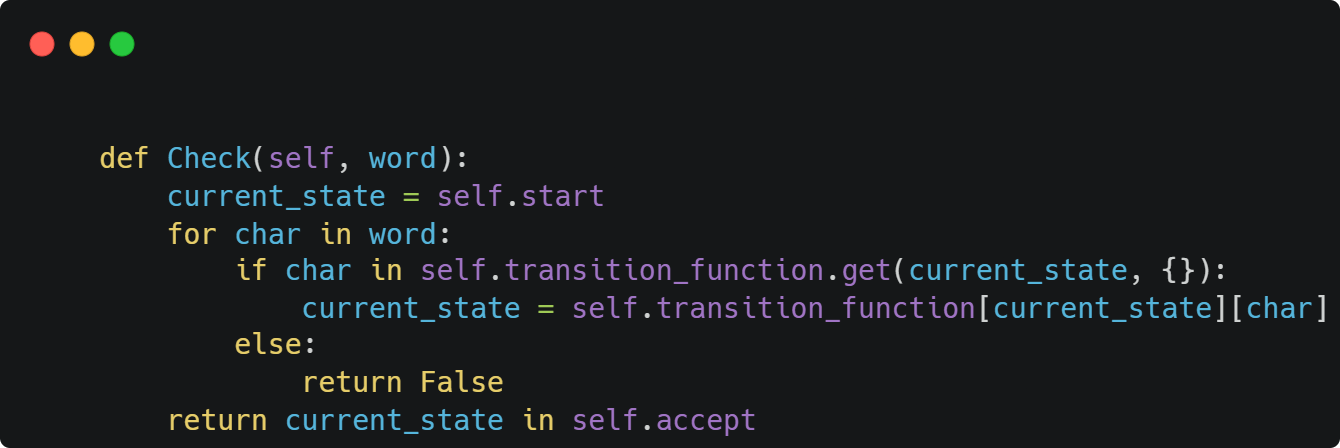
The function seems to be generating 5 strings by iteratively replacing the last character of the string with a random character from some dictionary self.P. The process continues until each generated string contains at least one lowercase character. Once 5 unique strings meeting this condition are generated, they are returned as a list**.**

I've augmented the Grammar class with a function that accepts an instance of the Grammar class as input and converts it into a FiniteAutomaton class object.

****

The Grammar class has been augmented with a new method, to\_automaton(), designed to facilitate the transformation of a grammar class object into a Finite Automaton class representation. This method serves as a bridge between the two formalisms, allowing for seamless conversion and interoperability.

The to\_automaton() function operates by extracting relevant components from the grammar object, such as states, alphabet, transition functions, start state, and acceptance states, and utilizes them to construct an equivalent Finite Automaton representation.



The Check method serves as a validator for words based on the DFA grammar represented by the class. It traverses the DFA's transition function to determine whether the input word satisfies the language defined by the DFA, ultimately returning a boolean value indicating its acceptance or rejection.

**Conclusion**

In conclusion, the integration of various functions and classes within the context of DFA (Deterministic Finite Automaton) grammar processing presents a comprehensive toolkit for linguistic analysis and validation.

The Grammar class, fortified with methods like generate\_strings() and to\_automaton(), provides a robust framework for generating strings based on NFA (Nondeterministic Finite Automaton) grammar and seamlessly transforming grammatical representations into Finite Automaton equivalents, respectively.

Moreover, the Check method, nestled within the Grammar class, offers a powerful mechanism for word validation against the DFA grammar's rules. By meticulously traversing the DFA's transition function, it effectively discerns the acceptability of input words, contributing to precise language recognition and validation.

Collectively, these functions and classes synergize to offer a cohesive suite of tools for grammar manipulation, string generation, and linguistic validation within the realm of formal language processing. Their integration not only enhances the versatility and utility of the framework but also underscores its efficacy in diverse computational linguistics applications.