

**MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică şi Microelectronică**

**Departamentul Inginerie Software și Automatică**

**Tabanschi Nichita FAF-222**

**Report**

*Laboratory work n.1*

***of Limbaje Formale și Automate***

Checked by:

**Dumitru Cretu**, *university assistant*

DISA, FCIM, UTM

**Chișinău – 2024**

**Topic: Intro to formal languages. Regular grammars. Finite Automata.**

**Overview**

A formal language can be considered to be the media or the format used to convey information from a sender entity to the one that receives it. The usual components of a language are:

The alphabet: Set of valid characters;

The vocabulary: Set of valid words;

The grammar: Set of rules/constraints over the lang.

Now these components can be established in an infinite amount of configurations, which actually means that whenever a language is being created, it's components should be selected in a way to make it as appropriate for it's use case as possible. Of course sometimes it is a matter of preference, that's why we ended up with lots of natural/programming/markup languages which might accomplish the same thing.

**Objectives:**

1. Discover what a language is and what it needs to have in order to be considered a formal one;
2. 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:
   1. Create GitHub repository to deal with storing and updating your project;
   2. 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.);
   3. Store reports separately in a way to make verification of your work simpler (duh)
3. According to your variant number, get the grammar definition and do the following:
   1. Implement a type/class for your grammar;
   2. Add one function that would generate 5 valid strings from the language expressed by your given grammar;
   3. Implement some functionality that would convert and object of type Grammar to one of type Finite Automaton;
   4. For the Finite Automaton, please add a method that checks if an input string can be obtained via the state transition from it;

Variant 28:

VN={S, A, B, C},

VT={a, b},

P={

S → aA

A → bS

A → aB

B → bC

C → aA

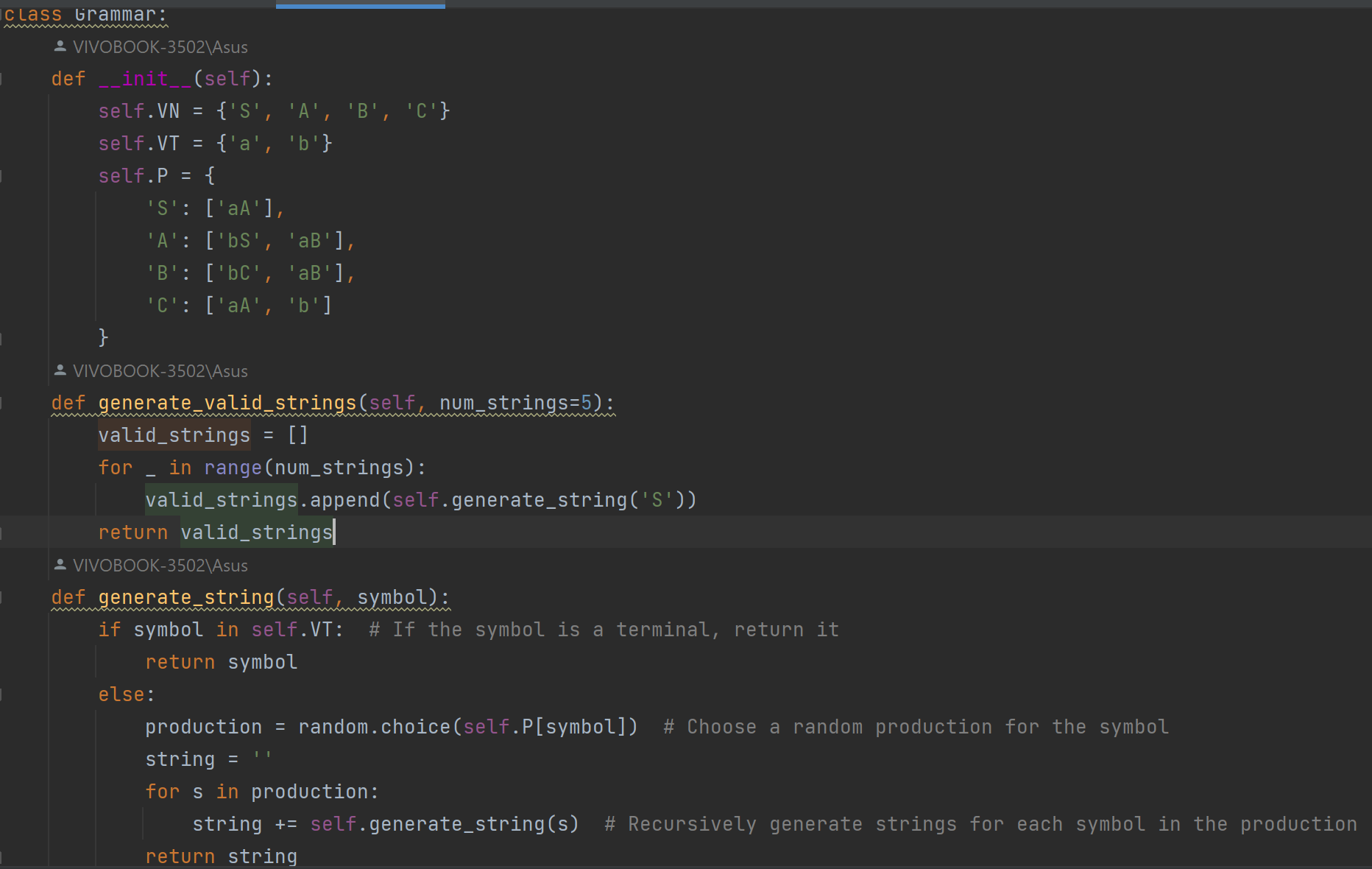
B → aB

C → b

}

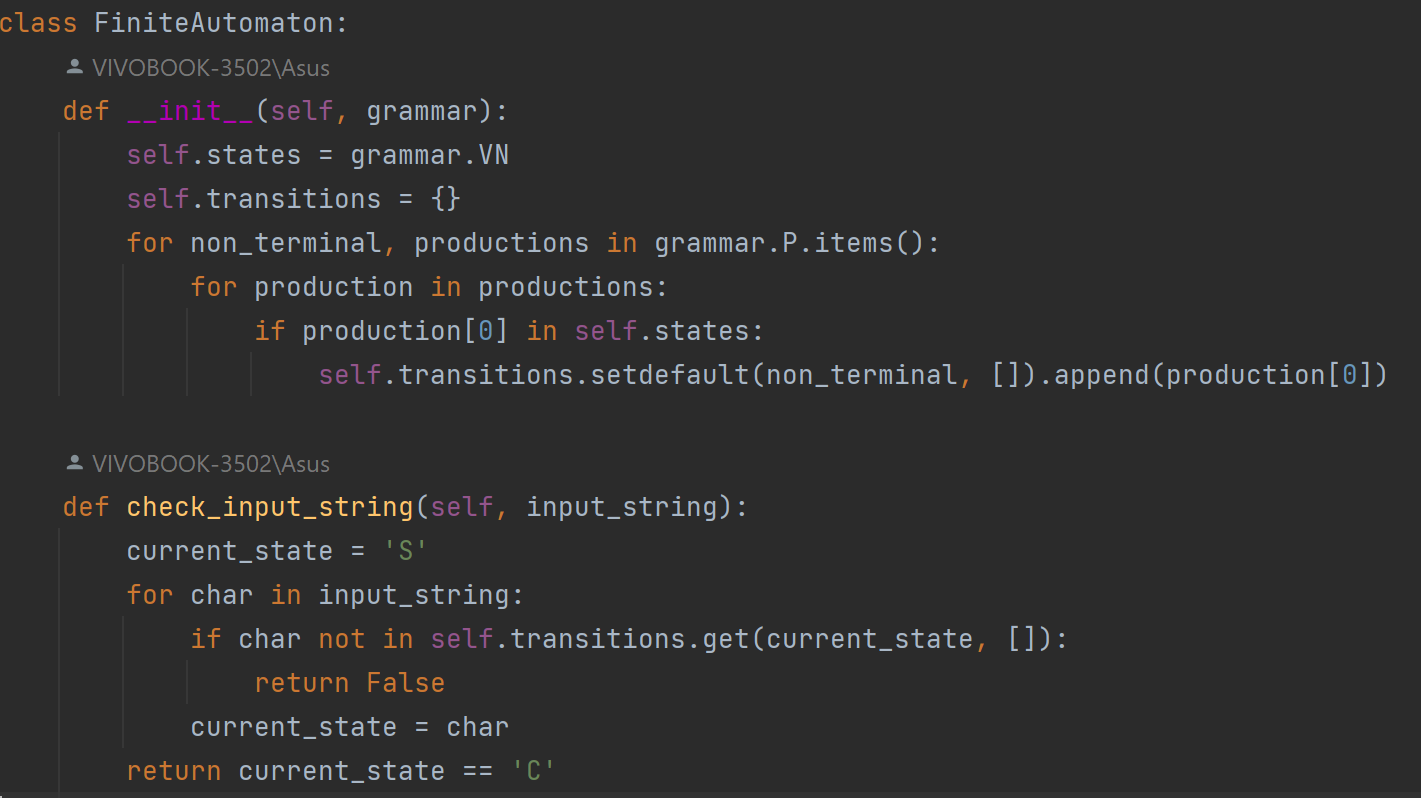
Grammar (grammar.py):

* This module defines a Grammar class responsible for generating valid strings based on a specified grammar.
* It has attributes:
  + VN: Set of non-terminal symbols.
  + VT: Set of terminal symbols.
  + P: Production rules of the grammar, represented as a dictionary where keys are non-terminal symbols and values are lists of strings representing possible productions for each non-terminal.
* It provides methods:
  + generate\_valid\_strings(num\_strings=5): Generates a specified number of valid strings using the grammar. It iterates num\_strings times and calls generate\_string() method to generate each string.
  + generate\_string(symbol): Recursively generates a string starting from a given non-terminal symbol according to the grammar rules. It selects a random production for the given symbol and recursively expands each symbol in the production until terminal symbols are reached.

****

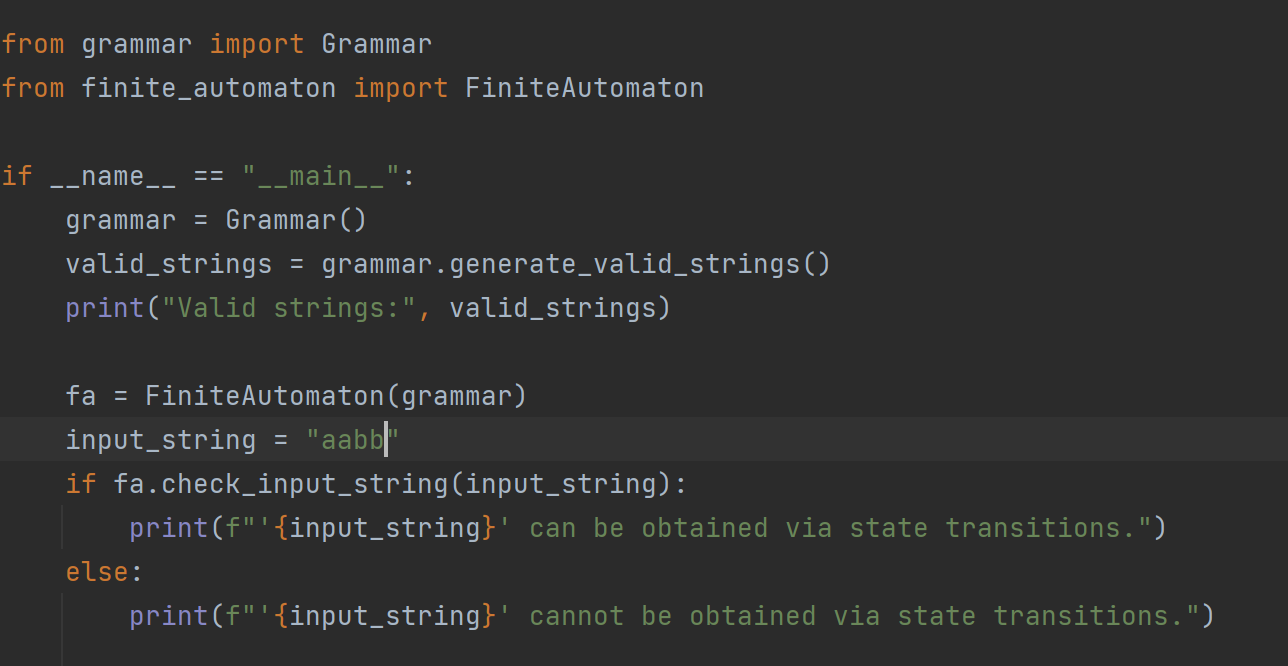
Finite Automaton (finite\_automaton.py):

* This module defines a FiniteAutomaton class responsible for representing a finite automaton and checking input strings against its transitions.
* It has attributes:
  + states: Set of states of the finite automaton.
  + transitions: Dictionary representing transitions of the finite automaton. Keys are non-terminal symbols, and values are lists of symbols representing transitions.
* It provides methods:
  + \_\_init\_\_(grammar): Initializes the finite automaton with states and transitions derived from the provided grammar.
  + check\_input\_string(input\_string): Checks if the given input string can be obtained by transitioning through the finite automaton's states according to its transitions. It iterates through each character of the input string and verifies if there exists a transition from the current state to the next state based on the input character.

****

Main Script (main.py):

* This script demonstrates the usage of the Grammar and FiniteAutomaton classes.
* It imports both classes from their respective modules (grammar.py and finite\_automaton.py).
* In the main block:
  + It creates an instance of the Grammar class to generate valid strings based on the grammar.
  + It prints the generated valid strings.
  + It creates an instance of the FiniteAutomaton class using the generated grammar.
  + It checks a sample input string against the transitions of the finite automaton and prints whether the input string can be obtained via state transitions or not.

****