MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software și Automatică

MARTINIUC ARTIOM 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:
    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)

3. 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 13:

VN={S, B, D},

VT={a, b, c},

P={

S → aB

B → aD

B → bB

D → aD

D → bS

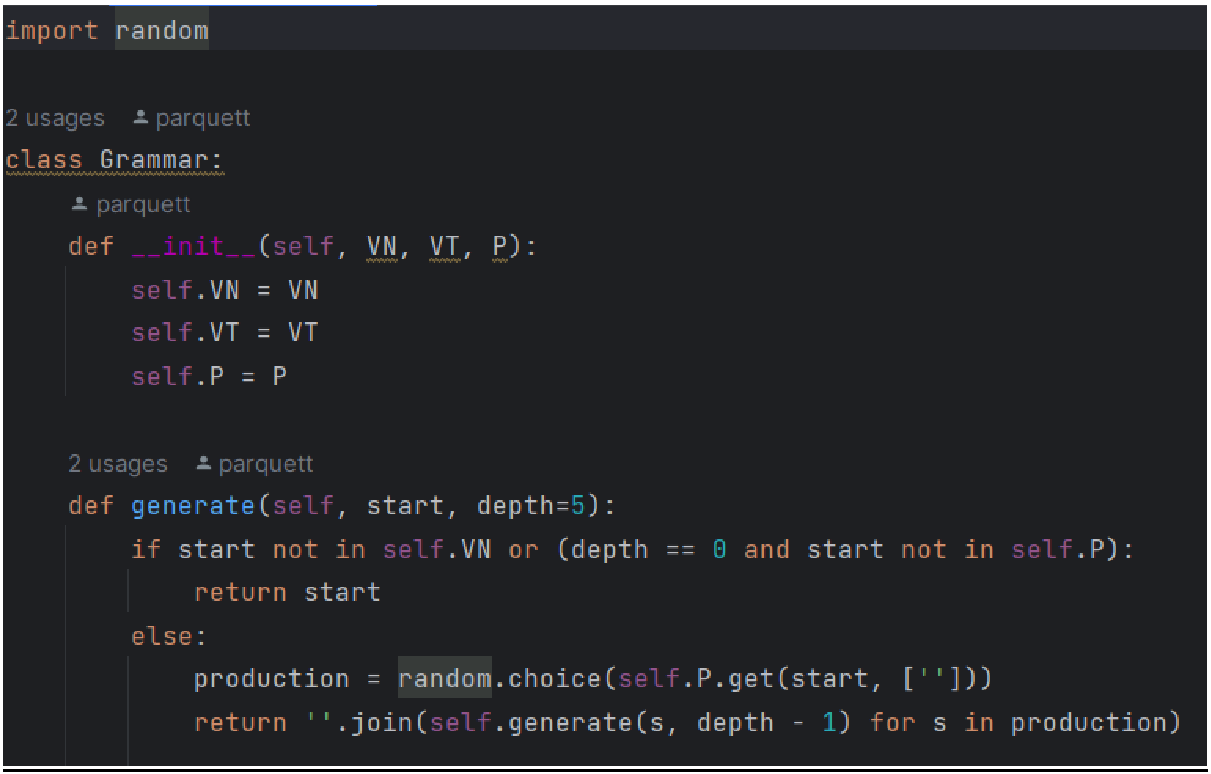
B → cS

D → c

}

Implementation:

First of all, we define a class Grammar which represents a context-free grammar



```
import random

2 usages  parquett
class Grammar:
    parquett
    def __init__(self, VN, VT, P):
        self.VN = VN
        self.VT = VT
        self.P = P

    2 usages  parquett
    def generate(self, start, depth=5):
        if start not in self.VN or (depth == 0 and start not in self.P):
            return start
        else:
            production = random.choice(self.P.get(start, ['']))
            return ''.join(self.generate(s, depth - 1) for s in production)
```

First, we have the constructor method for the Grammar class. It initializes instances of the class with three parameters: VN: A set of non-terminal symbols (variables) in the grammar. VT: A set of terminal symbols (terminals) in the grammar. P: A dictionary representing the productions of the grammar. Keys are non-terminal symbols, and values are lists of strings representing possible expansions of those symbols.

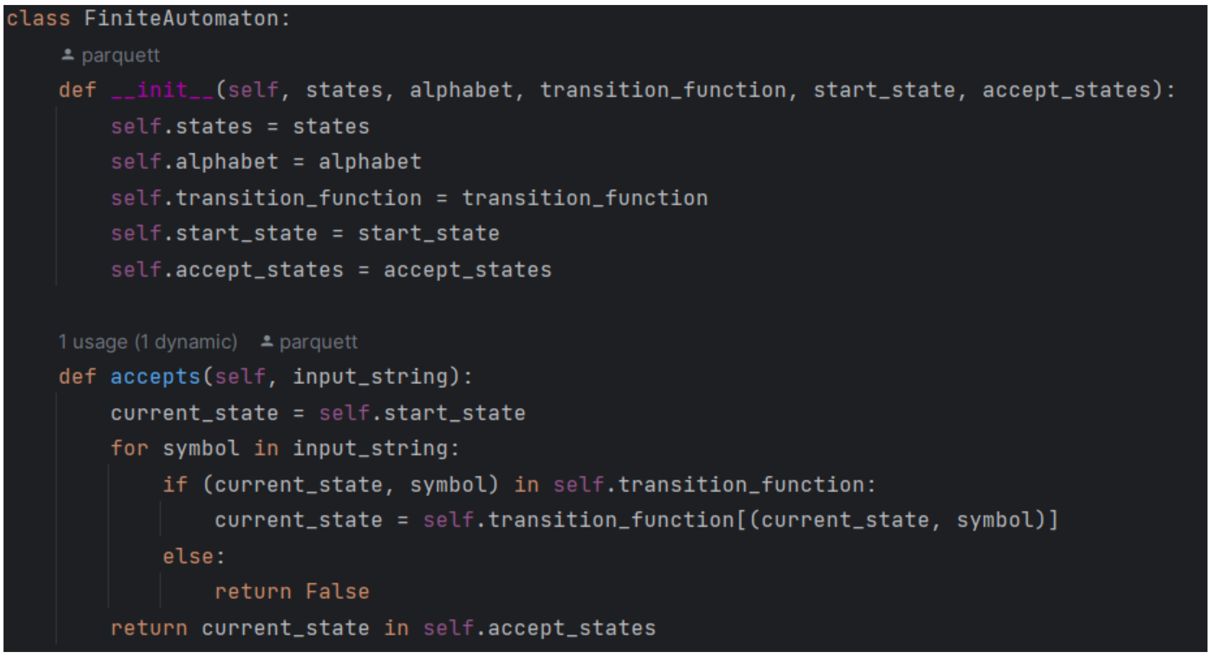
Then we define a method generate within the Grammar class. It generates a random string based on the grammar, starting from the given non-terminal symbol start. It has an optional parameter depth, which controls the depth of recursion to avoid infinite recursion.

We check if the start symbol is not in the set of non-terminal symbols (VN) or if the depth is zero and the start symbol is not in the productions (P). If either condition is met, it means the start symbol cannot be expanded further, so it returns the start symbol itself.

Otherwise, we select a random production rule for the given start symbol from the dictionary P. If start is not found in P, it defaults to an empty list. random.choice() then selects a random element from this list.

Finally, we recursively generate strings for each symbol in the selected production and concatenates them to form the final generated string. It iterates over each symbol s in the production list, calling the generate method recursively with s as the new starting symbol and depth - 1 as the new depth. The generated strings for each symbol are joined together using ''.join(). This process continues until the depth becomes zero or until there are no more non-terminal symbols to expand.

Next, we define a class FiniteAutomaton.



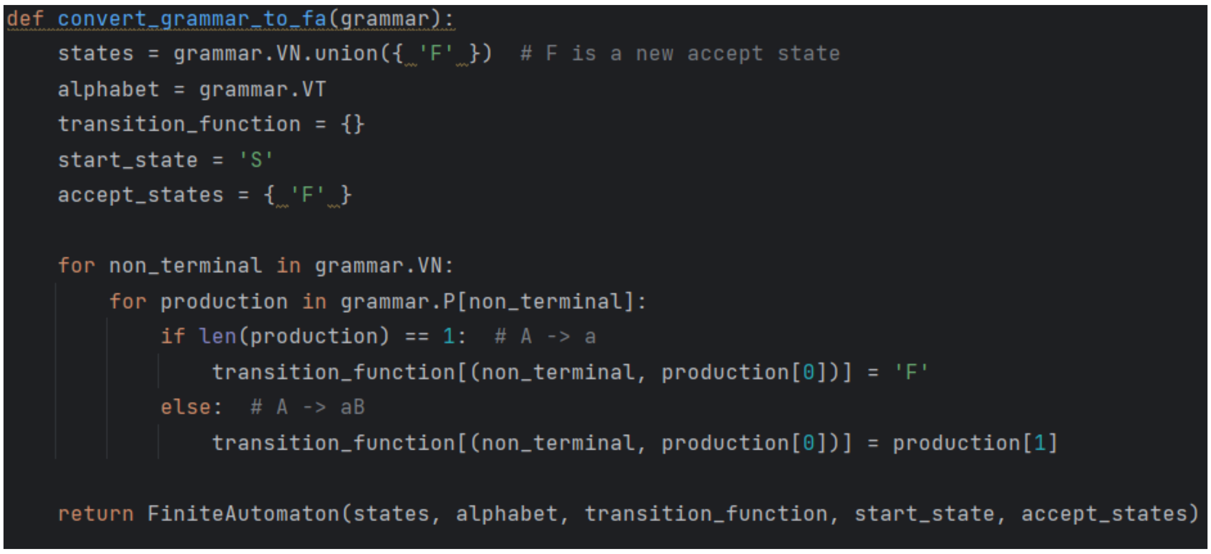
```python
class FiniteAutomaton:
    def __init__(self, states, alphabet, transition_function, start_state, accept_states):
        self.states = states
        self.alphabet = alphabet
        self.transition_function = transition_function
        self.start_state = start_state
        self.accept_states = accept_states

    def accepts(self, input_string):
        current_state = self.start_state
        for symbol in input_string:
            if (current_state, symbol) in self.transition_function:
                current_state = self.transition_function[(current_state, symbol)]
            else:
                return False
        return current_state in self.accept_states
```

The constructor method for initializes instances of the class with five parameters: A set of states in the finite automaton. A set of symbols representing the alphabet of the finite automaton. A dictionary representing the transition function of the finite automaton, where keys are tuples `(state, symbol)` and values are the resulting states. The start state of the finite automaton and a set of states representing the accept states of the finite automaton.

Afterwards, we define a method `accepts` within the class. It checks whether the given input string is accepted by the finite automaton. It iterates through each symbol in the input string, following the transitions defined by the transition function. If it reaches the end of the input string and the current state is one of the accept states, it returns `True`; otherwise, it returns `False`.

Basically, we check if there exists a transition defined for the current state and the current input symbol. If there is a transition defined, we update the current state according to the transition function.

This is a method to convert object of type Grammar to Finite Automaton



```python
def convert_grammar_to_fa(grammar):
    states = grammar.VN.union({ 'F' })  # F is a new accept state
    alphabet = grammar.VT
    transition_function = {}
    start_state = 'S'
    accept_states = { 'F' }

    for non_terminal in grammar.VN:
        for production in grammar.P[non_terminal]:
            if len(production) == 1:  # A -> a
                transition_function[(non_terminal, production[0])] = 'F'
            else:  # A -> aB
                transition_function[(non_terminal, production[0])] = production[1]

    return FiniteAutomaton(states, alphabet, transition_function, start_state, accept_states)
```

We create a set of states for the finite automaton, including all non-terminal symbols of the grammar (`grammar.VN`) and a new accept state `'F'`.

We set the alphabet of the finite automaton to be the set of terminal symbols of the grammar (`grammar.VT`).

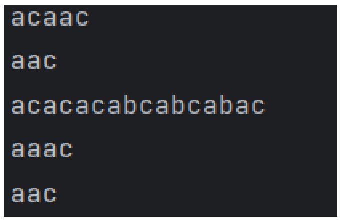
Next, initialize an empty dictionary to represent the transition function of the finite automaton.

In my case, the start state would be ‘S’. Then, define a set containing only the new accept state `'F'`.

The subsequent nested loops iterate over each non-terminal symbol in the grammar, and for each production, it sets up transitions in the transition function accordingly. If a production has a single symbol (terminal), it transitions to the accept state `'F'`, otherwise, it transitions to the next symbol in the production.

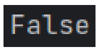
Finally, the function returns an instance of the FiniteAutomaton class initialized with the constructed states, alphabet, transition function, start state, and accept states.

These are 5 generated strings, based on my grammar.



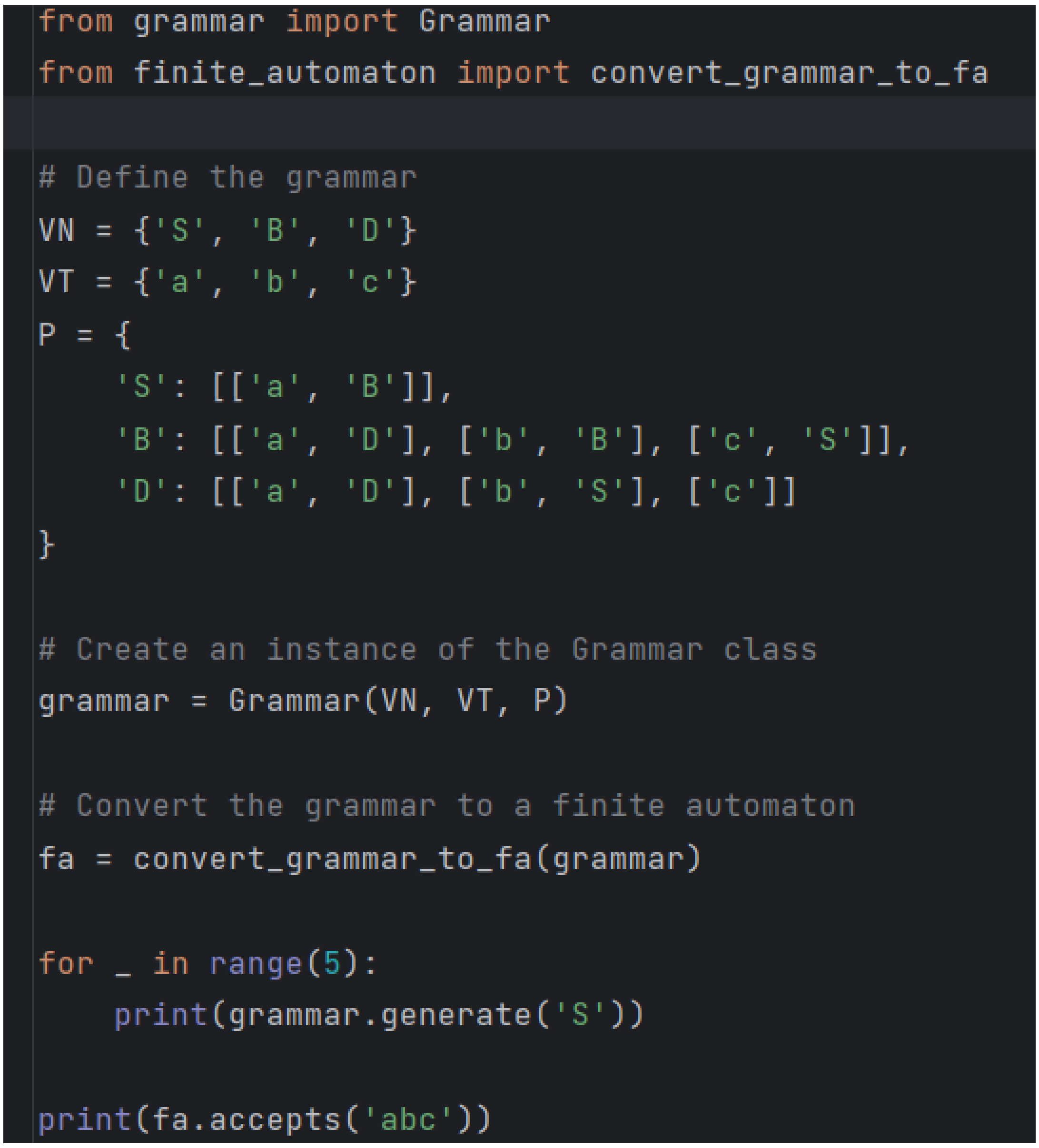
```
acaac
aac
acacacabcabcabac
aaac
aac
```

Also I check if FA accept string ‘abc’, and get this result:



```
False
```

For reference, below you can find my main.py.


```python
from grammar import Grammar
from finite_automaton import convert_grammar_to_fa

# Define the grammar
VN = {'S', 'B', 'D'}
VT = {'a', 'b', 'c'}
P = {
    'S': [['a', 'B']],
    'B': [['a', 'D'], ['b', 'B'], ['c', 'S']],
    'D': [['a', 'D'], ['b', 'S'], ['c']]
}

# Create an instance of the Grammar class
grammar = Grammar(VN, VT, P)

# Convert the grammar to a finite automaton
fa = convert_grammar_to_fa(grammar)

for _ in range(5):
    print(grammar.generate('S'))

print(fa.accepts('abc'))
```