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: Bostan Victor, 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.

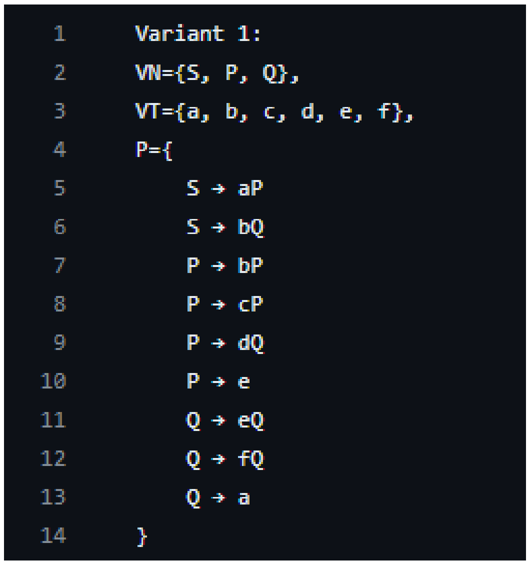
OBJECTIVES

- 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

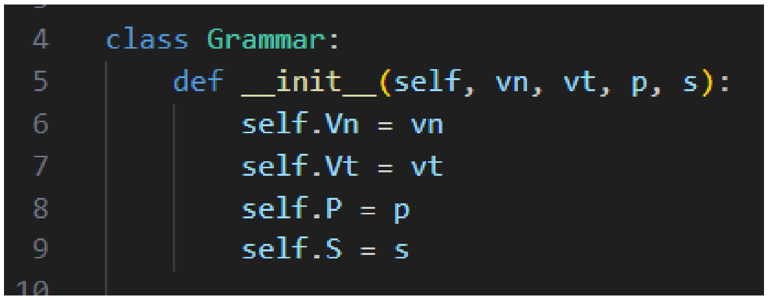
For this laboratory I chose to work in python. The implementation comprises two main classes: Grammar and FiniteAutomaton.

- The Grammar class encapsulates the components of a regular grammar: non-terminal and terminal symbols, production rules, and a start symbol. It includes a method generate_strings to produce valid strings and a method to_finite_automaton to convert the grammar into a finite automaton.
- The FiniteAutomaton class represents a finite automaton with states, an alphabet, a transition function, a start state, and accept states. It includes methods to print its configuration (print_automaton) and to determine if a given string is accepted by the automaton (is_word_accepted).



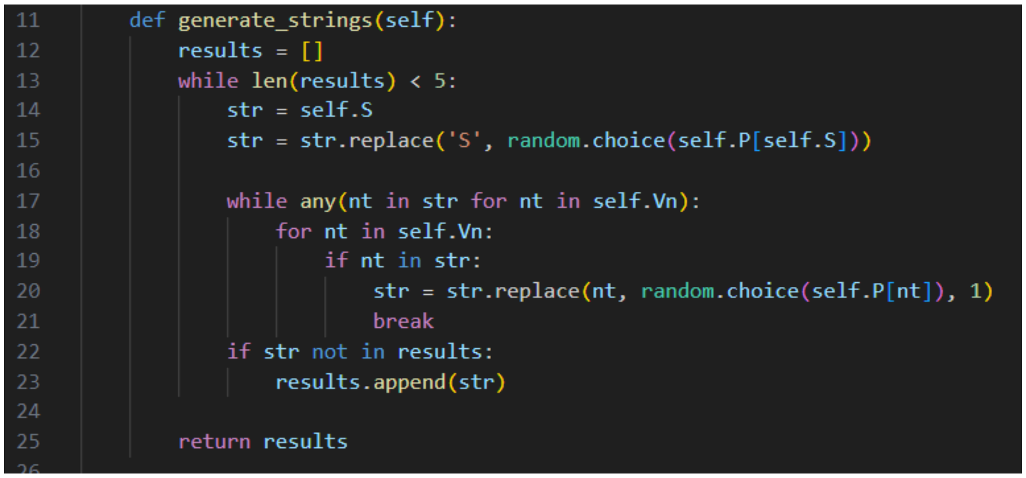
```
Variant 1:
VN={S, P, Q},
VT={a, b, c, d, e, f},
P={
    S → aP
    S → bQ
    P → bP
    P → cP
    P → dQ
    P → e
    Q → eQ
    Q → fQ
    Q → a
}
```

First task I had was to implement the Grammar according to variant 1 into a class. Upon creating the class, in the constructor method I declared all the parameters a Grammar must have(set of non-terminal symbols, set of terminal symbols, productions and start symbol):



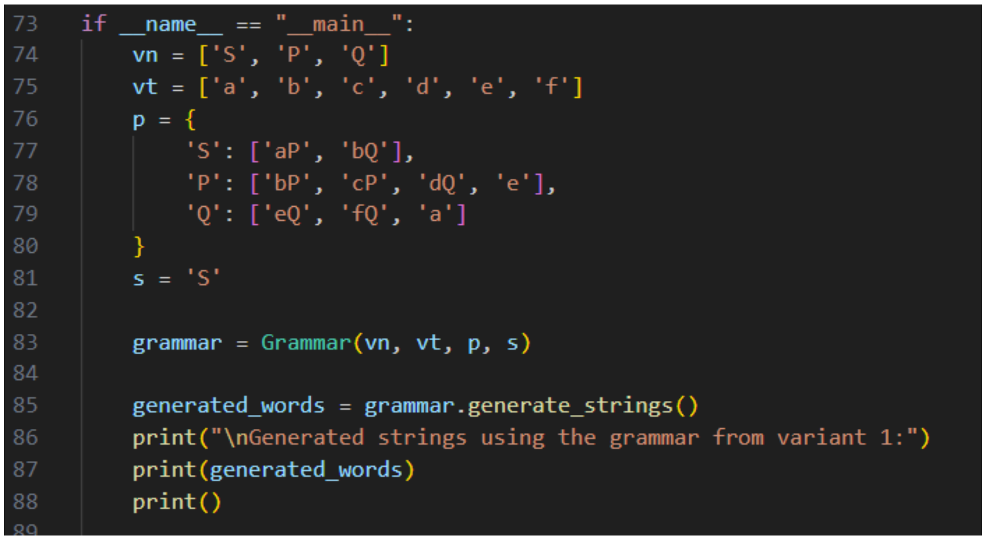
```python
class Grammar:
    def __init__(self, vn, vt, p, s):
        self.Vn = vn
        self.Vt = vt
        self.P = p
        self.S = s
```

Next, I created the *generate_strings* method, which returns 5 randomly generated words based on the grammar from my variant. I did this with the help of the python library random, which allowed me to choose a random production when substituting a non-terminal symbol:



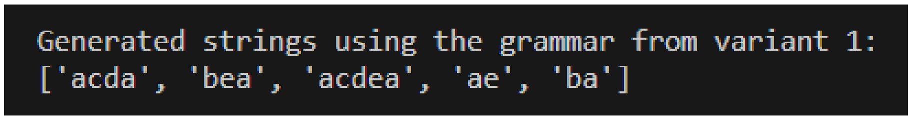
```python
    def generate_strings(self):
        results = []
        while len(results) < 5:
            str = self.S
            str = str.replace('S', random.choice(self.P[self.S]))

            while any(nt in str for nt in self.Vn):
                for nt in self.Vn:
                    if nt in str:
                        str = str.replace(nt, random.choice(self.P[nt]), 1)
                        break
            if str not in results:
                results.append(str)

        return results
```

Now, in order to test the method, it is needed to create an object of type *Grammar* with all the parameters that correspond to my variant. And then just run the function:



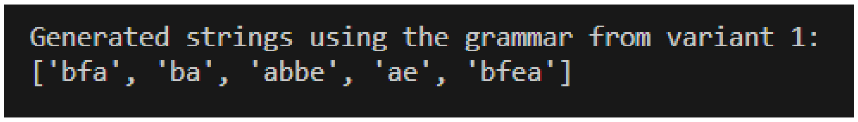
```python
if __name__ == "__main__":
    vn = ['S', 'P', 'Q']
    vt = ['a', 'b', 'c', 'd', 'e', 'f']
    p = {
        'S': ['aP', 'bQ'],
        'P': ['bP', 'cP', 'dQ', 'e'],
        'Q': ['eQ', 'fQ', 'a']
    }
    s = 'S'

    grammar = Grammar(vn, vt, p, s)

    generated_words = grammar.generate_strings()
    print("\nGenerated strings using the grammar from variant 1:")
    print(generated_words)
    print()
```

And so the result looks like this:

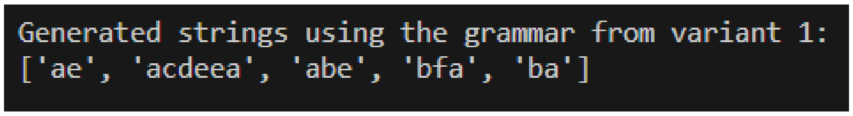


```
Generated strings using the grammar from variant 1:
['acda', 'bea', 'acdea', 'ae', 'ba']
```

We can clearly see that these are all valid words according to the grammar. Let's run it a couple more times to make sure that the results are consistently correct:

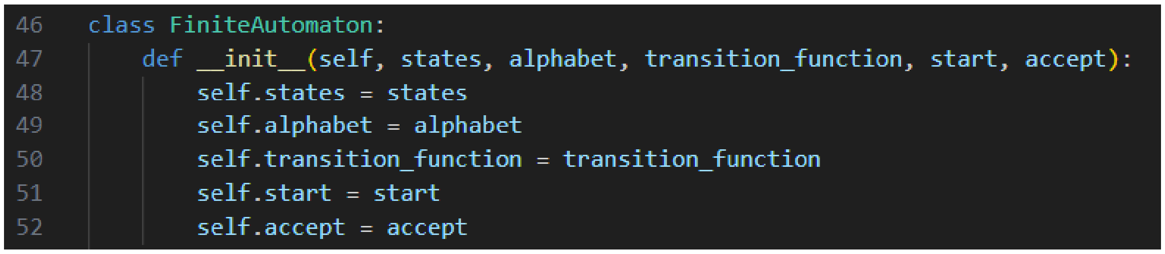


```
Generated strings using the grammar from variant 1:
['bfa', 'ba', 'abbe', 'ae', 'bfea']
```



```
Generated strings using the grammar from variant 1:
['ae', 'acdeea', 'abe', 'bfa', 'ba']
```

The next task was to implement the functionality to convert an Object of type Grammar into Finite Automaton. So, the first thing I did was to create a class named *FiniteAutomaton* with all the necessary parameters in the contructor(States, Alphabet, Transition Function, Start State and Accept State):



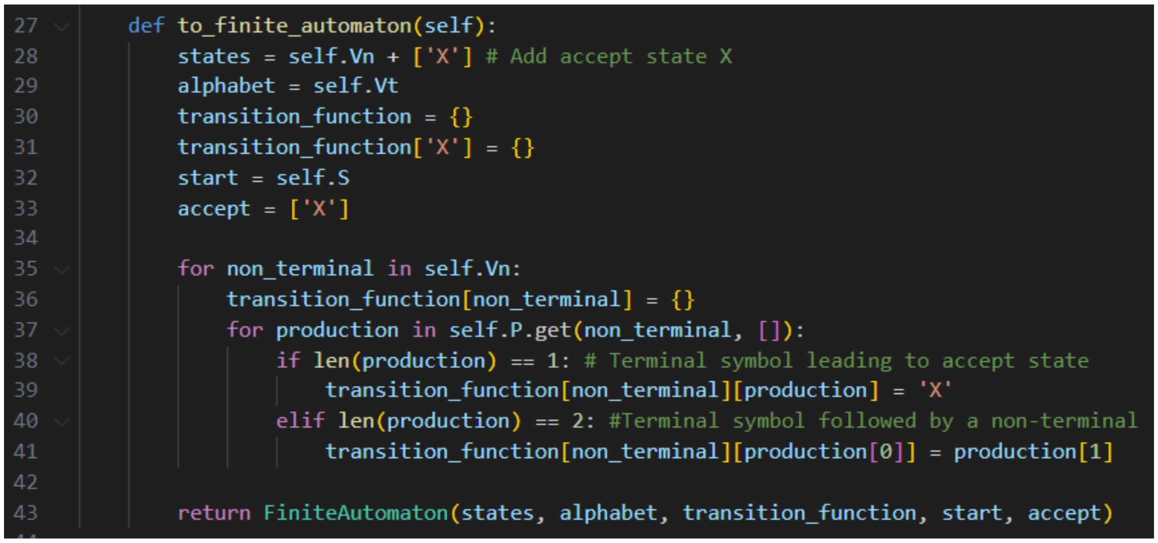
```python
class FiniteAutomaton:
    def __init__(self, states, alphabet, transition_function, start, accept):
        self.states = states
        self.alphabet = alphabet
        self.transition_function = transition_function
        self.start = start
        self.accept = accept
```

Then, inside the *Grammar* class I created a method named *to_finite_automaton*, which implements the algorithm to convert a Regular Grammar into Finite Automata. Here is the algorithm:

$$G = (V_N, V_T, P, S) \mid FA = (Q, \Sigma, \sigma, q_0, F)$$

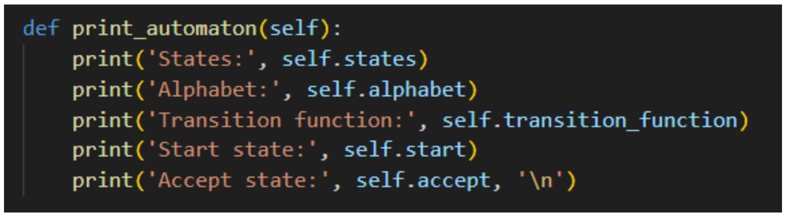
1) $Q = V_N \cup \{X\}$
2) $\Sigma = V_T$
3) $q_0 = \{S\}$
4) $F = \{X\}$
5) If we have A -> aB, $\sigma(A, a) = \{B\}$
6) If we have A -> a, $\sigma(A, a) = \{X\}$

And here is the implementation in python:

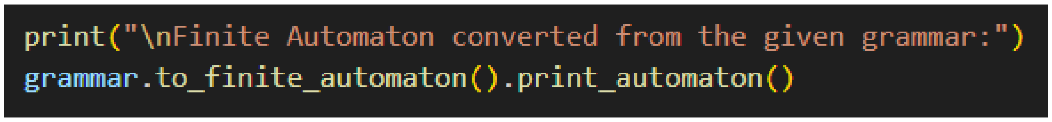


```python
    def to_finite_automaton(self):
        states = self.Vn + ['X'] # Add accept state X
        alphabet = self.Vt
        transition_function = {}
        transition_function['X'] = {}
        start = self.S
        accept = ['X']

        for non_terminal in self.Vn:
            transition_function[non_terminal] = {}
            for production in self.P.get(non_terminal, []):
                if len(production) == 1: # Terminal symbol leading to accept state
                    transition_function[non_terminal][production] = 'X'
                elif len(production) == 2: #Terminal symbol followed by a non-terminal
                    transition_function[non_terminal][production[0]] = production[1]

        return FiniteAutomaton(states, alphabet, transition_function, start, accept)
```

Inside the *FiniteAutomaton* class I also created a method to print the automaton so I could show the results of the conversion:

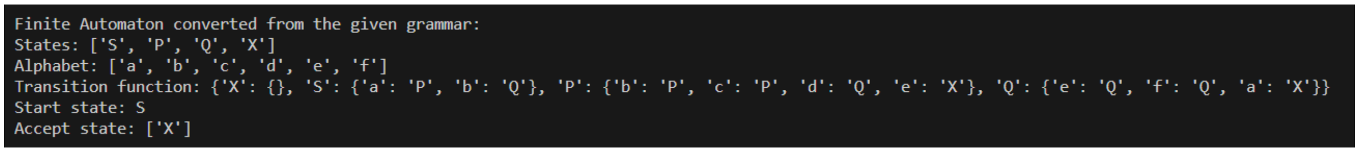


```python
def print_automaton(self):
    print('States:', self.states)
    print('Alphabet:', self.alphabet)
    print('Transition function:', self.transition_function)
    print('Start state:', self.start)
    print('Accept state:', self.accept, '\n')
```



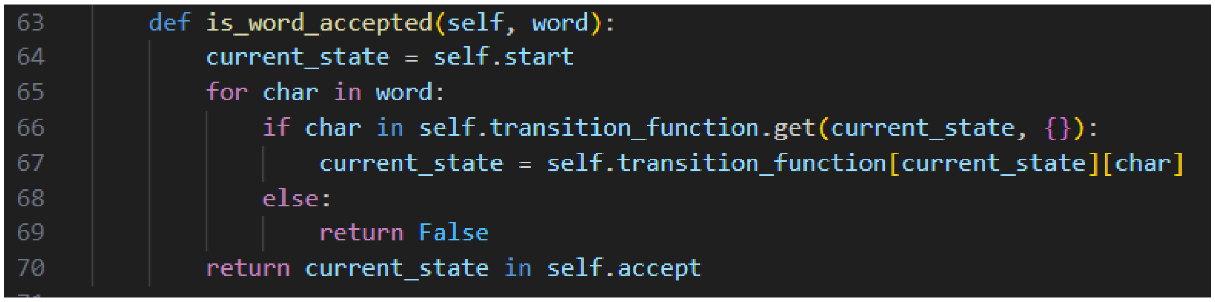
```python
print("\nFinite Automaton converted from the given grammar:")
grammar.to_finite_automaton().print_automaton()
```

So, here is the result:



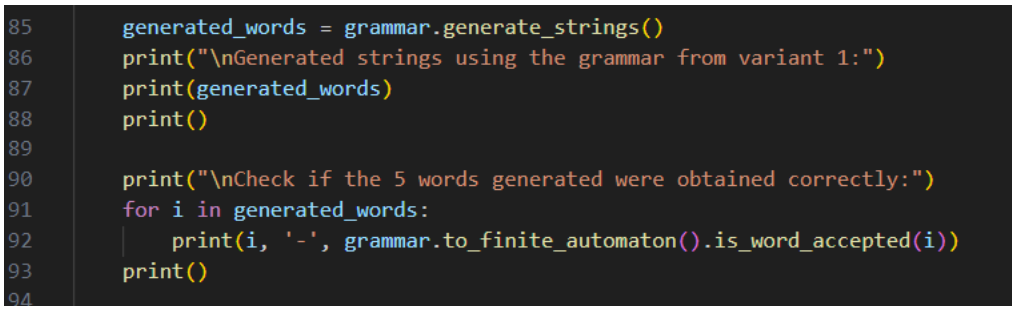
```
Finite Automaton converted from the given grammar:
States: ['S', 'P', 'Q', 'X']
Alphabet: ['a', 'b', 'c', 'd', 'e', 'f']
Transition function: {'X': {}, 'S': {'a': 'P', 'b': 'Q'}, 'P': {'b': 'P', 'c': 'P', 'd': 'Q', 'e': 'X'}, 'Q': {'e': 'Q', 'f': 'Q', 'a': 'X'}}
Start state: S
Accept state: ['X']
```

As we can see, all the transition correspond to the products of the grammar, meaning that the conversion was done correctly. But, in order to check with certinty that I created the correct automaton for the grammar, the next task is to implement the *is_word_accepted* method in the *FiniteAutomaton* class which would check using the automaton if an input string belongs to the language of the grammar in my variant. The algorithm here is simple, we just need to iterate through the word and check if each non-terminal symbol can be obtained correctly using our transition function, and also check if the last state is the accept state:

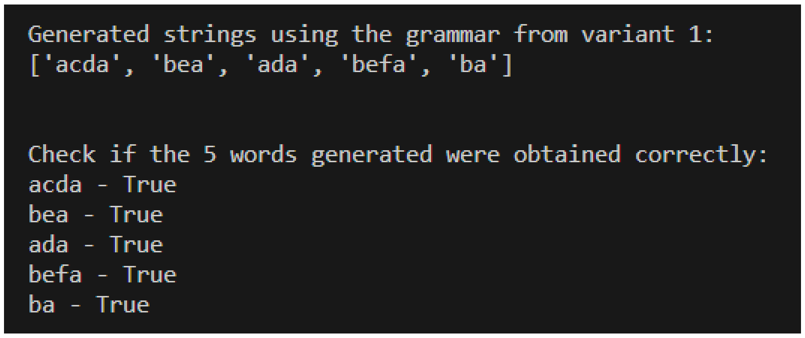


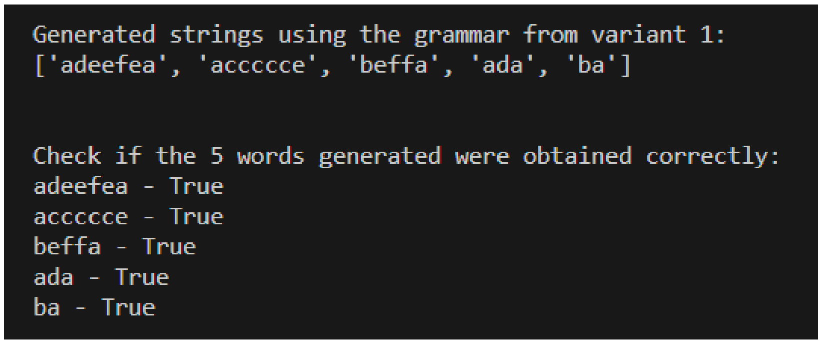
```python
    def is_word_accepted(self, word):
        current_state = self.start
        for char in word:
            if char in self.transition_function.get(current_state, {}):
                current_state = self.transition_function[current_state][char]
            else:
                return False
        return current_state in self.accept
```

Now let's run the program a couple of times, firstly generate 5 words and then check if they belong to the grammar, to see the results:

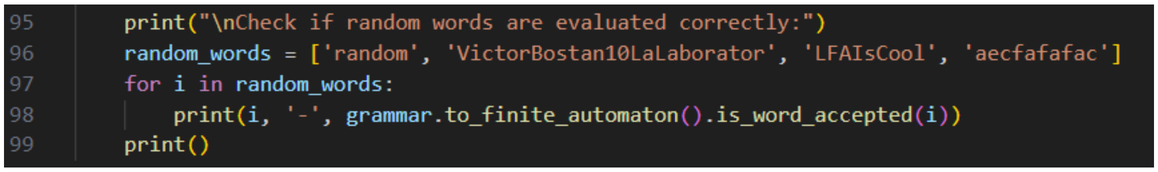


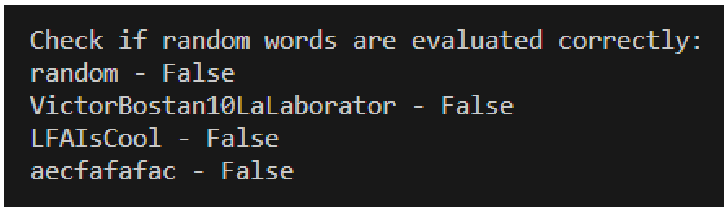
```python
    generated_words = grammar.generate_strings()
    print("\nGenerated strings using the grammar from variant 1:")
    print(generated_words)
    print()

    print("\nCheck if the 5 words generated were obtained correctly:")
    for i in generated_words:
        print(i, '-', grammar.to_finite_automaton().is_word_accepted(i))
    print()
```


```
Generated strings using the grammar from variant 1:
['acda', 'bea', 'ada', 'befa', 'ba']


Check if the 5 words generated were obtained correctly:
acda - True
bea - True
ada - True
befa - True
ba - True
```


```
Generated strings using the grammar from variant 1:
['adeefea', 'acccce', 'beffa', 'ada', 'ba']


Check if the 5 words generated were obtained correctly:
adeefea - True
acccce - True
beffa - True
ada - True
ba - True
```

We can see that each word that was created is evaluated as belonging to the grammar by the *is_word_accepted* method of our Finite Automaton. Finally, let's check if some random strings will be evaluated correctly:


```
    print("\nCheck if random words are evaluated correctly:")
    random_words = ['random', 'VictorBostan10LaLaborator', 'LFAIsCool', 'aecfafafac']
    for i in random_words:
        print(i, '-', grammar.to_finite_automaton().is_word_accepted(i))
    print()
```


```
Check if random words are evaluated correctly:
random - False
VictorBostan10LaLaborator - False
LFAIsCool - False
aecfafafac - False
```

CONCLUSION

The implementation successfully demonstrate the conversion of a regular grammar into a finite automaton. Generated strings from the grammar are correctly recognized by the automaton, validating the equivalence between the grammar and its corresponding automaton. Additionally, the automaton correctly rejects random strings not generated by the grammar, showcasing its ability to distinguish between valid and invalid strings within the language.

This laboratory underscores the foundational relationship between grammars and automata in the domain of formal languages, offering practical insights into their applications in compiler design, text processing, and beyond.

REFERENCES

1) Laboratory Work #1: Regular Grammars & Finite Automata. Authors – Cretu Dumitru, Vasile Drumea, Irina Cojuhari: https://github.com/filpatterson/DSL_laboratory_works/blob/master/1_RegularGrammars/task.md
2) Introduction to formal languages presentation, TUM, conf. univ., dr. Irina Cojuhari: https://else.fcim.utm.md/pluginfile.php/110457/mod_resource/content/0/Theme_1.pdf
3) Finite Automata presentation, TUM, conf. univ., dr. Irina Cojuhari: https://drive.google.com/file/d/1rBGyzDN5eWMXTNeUxLxmKsf7tyhHt9Jk/view