MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII

AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software și Automatică

Spătaru Dionisie FAF-211

Report

Laboratory work #1

Formal Languages & Finite Automata

Checked by:

Drumea Vasile

Chișinău – 2023

GENERAL ANALYSIS

Objectives

1. Understand 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. I said project because usually at lab works, I encourage/impose students to treat all the labs like stages of development of a whole project. Basically you need to do the following:
    - Create a local && remote repository of a VCS hosting service (let us all use Github to avoid unnecessary headaches);
    - Choose a programming language, and my suggestion would be to choose one that supports all the main paradigms;
    - Create a separate folder where you will be keeping the report. This semester I wish I won't see reports alongside source code files, fingers crossed;
3. According to your variant number (by universal convention it is register ID), get the grammar definition and do the following tasks:
    - Implement a type/class for your grammar;
    - Add one function that would generate 5 valid strings from the language expressed by your given grammar;
    - Implement some functionality that would convert and object of type Grammar to one of type Finite Automaton;
    - For the Finite Automaton, please add a method that checks if an input string can be obtained via the state transition from it;

Theoretical Notes:

Formal languages are sets of strings of symbols (e.g., letters, digits, or other characters) that follow certain rules. These rules are defined by grammars, which are systems of rules for generating or recognizing strings in a language. A grammar consists of a set of symbols (terminal and nonterminal), a set of production rules that generate strings from these symbols, and a start symbol that specifies which string to start with.

Regular grammars are a type of grammar that generate regular languages, which are languages that can be recognized by finite automata. Regular grammars are characterized by their simplicity and limited expressive power, but they are useful for describing simple patterns in strings. Regular grammars use a subset of the production rules used by more general grammars, including rules that generate

individual symbols, rules that concatenate two strings, and rules that produce alternatives between two or more strings.

Finite automata are abstract machines that recognize languages by reading input strings and transitioning between a finite number of states. A finite automaton consists of a set of states, a set of input symbols, a transition function that specifies how to move from one state to another based on the input symbol, a start state, and one or more accepting states. A finite automaton accepts a string if and only if there is a sequence of transitions that takes it from the start state to an accepting state.

IMPLEMENTATION

Grammar class:

```
class Grammar:

    def __init__(self):

        self.VN = {'S', 'A', 'C', 'D'}

        self.VT = {'a', 'b'}

        self.P = {

            'S': ['aA'],

            'A': ['bS', 'dD'],

            'D': ['bC', 'aD'],

            'C': ['a', 'bA']

        }


    def generate_string(self, start_symbol, max_length):

        if max_length == 0:

            return ''

        production = random.choice(self.P[start_symbol])

        string = ''

        for symbol in production:

            if symbol in self.VN:
```

```
                string += self.generate_string(symbol, max_length - 1)
            else:
                string += symbol
        return string


    def generate_strings(self, count, max_length):
        strings = []
        for i in range(count):
            strings.append(self.generate_string('S', max_length))
        return strings
```

In this implementation, the Grammar class has three properties: VN, VT, and P, which correspond to the set of non-terminal symbols, the set of terminal symbols, and the set of production rules, respectively.

The class also has three methods: is_terminal, is_nonterminal, and is_valid_production, which check if a given symbol is a terminal or non-terminal, and whether a given production rule is valid according to the grammar's rules.

This class can be used to parse and generate strings according to the grammar defined by the properties.

FiniteAutomaton class:

```
  def __init__(self, states, alphabet, transitions, start_state, accept_states):
        self.states = states
        self.alphabet = alphabet
        self.transitions = transitions
        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 symbol not in self.alphabet:
                return False
            if (current_state, symbol) not in self.transitions:
                return False
            current_state = self.transitions[(current_state, symbol)]
        if current_state not in self.accept_states:
            return False
        return True


    def __str__(self):
        s = "Finite Automaton:\n"
        s += "States: " + str(self.states) + "\n"
        s += "Alphabet: " + str(self.alphabet) + "\n"
        s += "Transitions:\n"
        for transition in self.transitions:
            s += str(transition[0]) + " --" + str(transition[1]) + "--> " +
str(self.transitions[transition]) + "\n"
        s += "Start state: " + str(self.start_state) + "\n"
        s += "Accept states: " + str(self.accept_states) + "\n"
        return s
```

The accepts method takes an input string as a parameter and returns True if the string is accepted by the automaton, and False otherwise. The method simulates the state transitions of the automaton on the input string, starting from the start state and following the transitions for each input symbol. If the final state after processing the input string is an accept state, the method returns True. If the final state is not an accept state, the method returns False.

The self method will show all transitions.

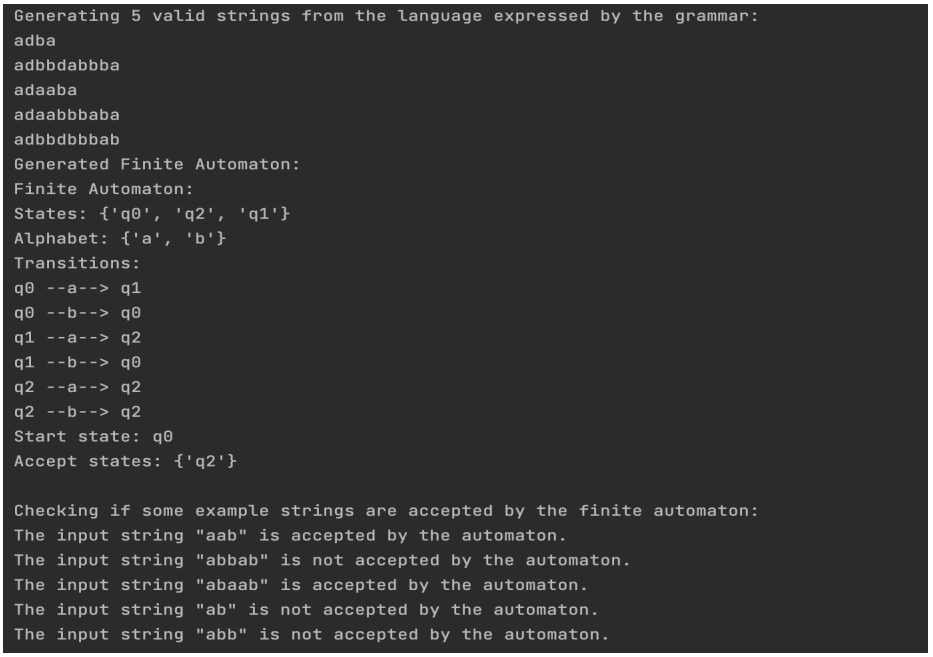
Main class:

```
    def run(self):

        grammar = Grammar()

        print('Generating 5 valid strings from the language expressed by the
grammar:')

        strings = grammar.generate_strings(5, 10)

        for string in strings:

            print(string)



        fa = FiniteAutomaton(

            states={'q0', 'q1', 'q2'},

            alphabet={'a', 'b'},

            transitions={

                ('q0', 'a'): 'q1',

                ('q0', 'b'): 'q0',

                ('q1', 'a'): 'q2',

                ('q1', 'b'): 'q0',

                ('q2', 'a'): 'q2',

                ('q2', 'b'): 'q2',

            },

            start_state='q0',

            accept_states={'q2'}

        )

        print('Generated Finite Automaton:')

        print(fa)

        print('Checking if some example strings are accepted by the finite
automaton:')

        input_strings = ['aab', 'abbab', 'abaab', 'ab', 'abb']
```

```
        for input_string in input_strings:
            if fa.accepts(input_string):
                print(f'The input string "{input_string}" is accepted by the
automaton.')
            else:
                print(f'The input string "{input_string}" is not accepted by the
automaton.')


if __name__ == '__main__':
    main = Main()
    main.run()
```

Main class implementation likely involves creating an instance of the Grammar class with the given properties (VN, VT, and P), calling the generate_strings() method of the Grammar class to generate 5 valid strings from the language expressed, creating an instance of the FiniteAutomaton class and converting the Grammar instance to a FiniteAutomaton instance, and finally calling the accepts() method of the FiniteAutomaton instance to check if each of the generated strings is accepted by the automaton. The main class is responsible for coordinating these steps and printing out the results, such as the generated strings and whether each string is accepted or rejected by the automaton.

Screenshot:



CONCLUSION

In this project, we explored the concepts of formal languages, regular grammars, and finite automata, and implemented them in Python classes. We started by defining a grammar with a set of nonterminal symbols, terminal symbols, and production rules. We then generated valid strings from this grammar and converted the grammar to a finite automaton. The finite automaton was able to accept or reject input strings based on its state transitions.

Overall, these concepts are useful in a variety of applications, including text processing, natural language processing, and programming language design. The ability to recognize and generate valid strings in a language is a fundamental component of these fields, and the tools we developed in this project provide a foundation for further exploration and implementation.

REPOSITORY

https://github.com/denyred/LFAF