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

Formal Languages and Finite Automata

Laboratory work 1 : Regular Grammars and Finite Automata

Elaborated:
st.gr. FAF-211 Nistor Stefan
Verified:
asist.univ. Vasile Drumea

Chișinău, 2023

Content

Introduction

Regular Grammars and Finite Automata are fundamental concepts in theoretical computer science and play a crucial role in the study of formal languages and automata theory.

A Regular Grammar is a set of production rules used to generate strings in a formal language. A formal language is a set of strings made up of symbols from a given alphabet, and regular grammars describe a subset of those languages that can be recognized by Finite Automata. Regular grammars are defined by a set of production rules, which specify how to rewrite a symbol in the language to a string of other symbols. The production rules are applied repeatedly to generate all the valid strings in the language.

A Finite Automaton is a mathematical model used to recognize patterns in strings, and it consists of a set of states, an initial state, a set of accepting states, and a transition function that maps from a current state and input symbol to a next state. Finite automata are classified as either deterministic or non-deterministic, depending on whether or not multiple transitions are possible for a given input symbol and state.

The relationship between regular grammars and finite automata is very close, and they are often used interchangeably. In particular, every regular grammar can be associated with a deterministic finite automaton, and every deterministic finite automaton can be associated with a regular grammar.

Regular grammars and finite automata have numerous applications in computer science, including parsing, pattern matching, lexical analysis, and text processing. They provide a foundation for more advanced topics, such as context-free grammars, pushdown automata, and Turing machines, which are essential in the study of computational complexity and algorithm design.

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:
   (a) Create a local remote repository of a VCS hosting service (let us all use Github to avoid unnecessary headaches);
   (b) Choose a programming language, and my suggestion would be to choose one that supports all the main paradigms;
   (c) 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:
   (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;

1 Implementation

1. You can use 2 classes in order to represent the 2 main object which are the grammar and finite automaton. Additional data model, helper classes etc. can be added but should be used (i.e. you shouldn't have source code file that are not used).
2. In order to show the execution you can implement a client class/type, which is just a "Main" class/type in which you can instantiate the types/classes. Another approach would be to write unit tests if you are familiar with them.

1.1 Code:

```
from FiniteAutomaton import FiniteAutomaton
from Grammar import Grammars
class Main:


    def __init__(self):
        self.productions = {
            'S': ['aA', 'aB'],
            'A': ['bS'],
            'B': ['aC'],
            'C': ['a', 'bS'],
        }
        self.start_symbol = 'S'
        self.grammar = Grammars(self.productions, self.start_symbol)
        self.finite_automaton = self.grammar.to_finite_automaton()
        self.automaton = FiniteAutomaton


    def generate_strings(self, num_strings):
        for i in range(num_strings):
            string = self.grammar.generate_string()


            print(string)


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

```
        main.generate_strings(5)
        automatons = main.grammar.to_finite_automaton()
        automaton = {
            'states': {'q0', 'q1', 'q2', 'q3', 'q4', 'q5'},
            'alphabet': {'a', 'b'},
            'transitions': {
                'q0': {'a': 'q1'},
                'q1': {'b': 'q2', 'a': 'q3'},
                'q2': {'a': 'q4'},
                'q3': {'a': 'q1', 'b': 'q5'},
                'q4': {'a': 'q5', 'b': 'q5'},
                'q5': {'a': 'q5', 'b': 'q5'}
            },
            'start_state': 'q0',
            'final_states': {'q5'}
        }
        checker = FiniteAutomaton(automaton)
        checker.check_strings(['aaa', 'abaaa', 'ababaa', 'aa', 'abababa'])
        print(automatons)
```

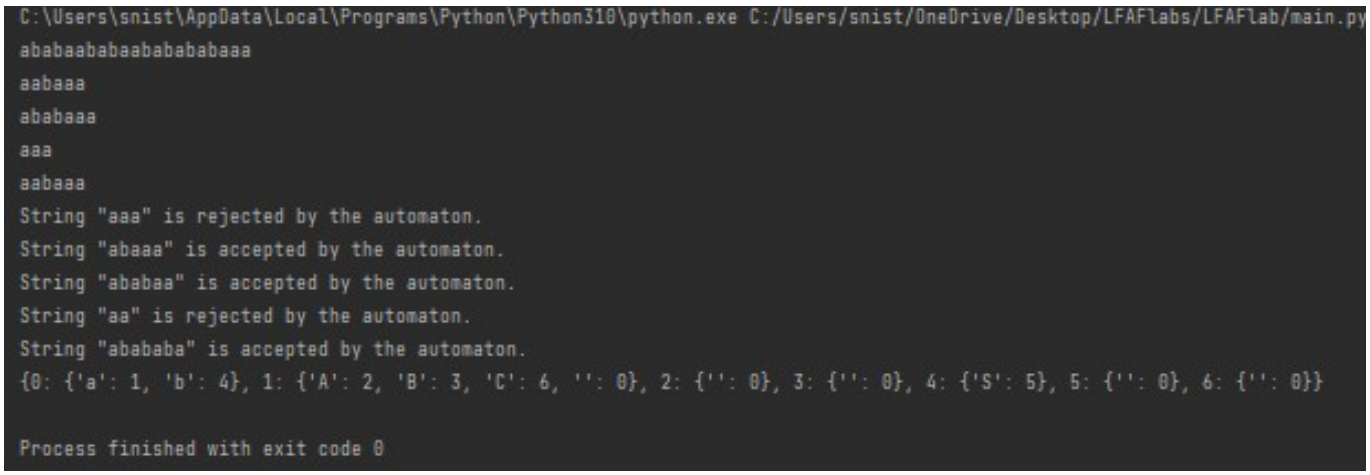
Here is the 'Grammar' class and the 'FiniteAutomaton' class

```
    import random

class Grammars:
    def __init__(self, productions, start_symbol):
        self.productions = productions
        self.start_symbol = start_symbol

    def generate_string(self):
        return self._generate_string(self.start_symbol)

    def _generate_string(self, symbol):
        if symbol not in self.productions:
```

```
        return symbol
    production = random.choice(self.productions[symbol])
    return ''.join(self._generate_string(s) for s in production)

def to_finite_automaton(self):
    # Initialize the automaton with a single start state
    start_state = 0
    automatons = {start_state: {}}
    state_count = 1

    # Add transitions for each production rule
    for symbol in self.productions:
        for production in self.productions[symbol]:
            current_state = start_state
            for s in production:
                if s not in automatons[current_state]:
                    # Add a new state and transition
                    automatons[current_state][s] = state_count
                    automatons[state_count] = {}
                    state_count += 1
                current_state = automatons[current_state][s]
            # Add an epsilon transition to the final state for the last symbol in th
            if current_state not in automatons:
                automatons[current_state] = {}
            automatons[current_state][''] = start_state

    # Return the automaton
    return automatons

    class FiniteAutomaton:
def __init__(self, automaton):
    self.states = automaton['states']
    self.alphabet = automaton['alphabet']
    self.transitions = automaton['transitions']
```

```
        self.start_state = automaton['start_state']
        self.final_states = automaton['final_states']

    def check_string(self, string):
        current_state = self.start_state

        for symbol in string:
            try:
                current_state = self.transitions[current_state][symbol]
            except KeyError:
                return False

        return current_state in self.final_states

    def check_strings(self, strings):
        for string in strings:
            if self.check_string(string):
                print(f'String "{string}" is accepted by the automaton.')
            else:
                print(f'String "{string}" is rejected by the automaton.')
```

1.2 Screenshot:



Conclusions

The study of regular grammars and finite automata is crucial for computer scientists to design efficient algorithms for various tasks such as text processing, pattern matching, and lexical analysis. These algorithms are widely used in real-world applications, like data compression, search engines, and spam filters. Moreover, recognizing and generating regular languages is significant for computer systems to communicate in a standardized way, which is essential for network communication.

Studying regular grammars and finite automata is essential as it helps computer scientists in developing new computation models that can solve more complex problems, beyond the capabilities of these models. For instance, context-free grammars and pushdown automata can recognize more complex patterns than regular grammars and finite automata. These advanced models provide the basis for Turing machines, which can solve any problem that computers can solve.

Apart from computer science, regular grammars and finite automata have connections with other branches of mathematics, such as topology, algebra, and group theory. These connections give insight into computation's nature and its relationship with other mathematical concepts.

In summary, regular grammars and finite automata are foundational concepts in computer science, having practical applications in various areas, including natural language processing, artificial intelligence, and compilers. They help understand computer capabilities and limitations and have connections to other mathematical areas. The study of these concepts is an ongoing area of research, and they will play a crucial role in shaping the future of computer science.