**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-213 Sirghi Tudor

Verified:

asist.univ. Vasile Drumea

Chis, ina˘u, 2023

# Content

**Intr**[**oduction** .](#_bookmark0) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3**

**Objecti**[**ves** .](#_bookmark1) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **4**

[**Implementation** .](#_bookmark2) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5**

**1** [**Implementation** .](#_bookmark3) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5**

1.1 [Code: .](#_bookmark4) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.2 [Screenshot: .](#_bookmark5) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

[**Conclusions** .](#_bookmark5) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7**

# Introduction

A Regular Grammar is a set of rules that generate strings in a formal language. In other words, a regular grammar is a way of describing how to produce strings using a finite set of symbols from an alphabet. These symbols can be anything, such as letters, numbers, or punctuation marks. The formal language is defined as a set of strings made up of symbols from the given alphabet.

Regular grammars describe a subset of the formal languages that can be recognized by Finite Automata. A Finite Automaton is a mathematical model used to recognize patterns in strings. It consists of a set of states, an initial state, a set of accepting states, and a transition function that maps from the current state and input symbol to the next state.

There are two types of finite automata: deterministic and non-deterministic. In a deterministic finite automaton, there is only one possible transition for each input symbol and state. In contrast, a non-deterministic finite automaton may have multiple possible transitions for a given input symbol and state.

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

Regular grammars and finite automata have numerous practical applications in computer science. They are commonly used in parsing, which is the process of analyzing a text to determine its grammatical structure. They are also used in pattern matching, which involves searching for a specific pattern in a text. In addition, regular grammars and finite automata are essential in lexical analysis, which is the process of converting a text into a series of tokens for processing. Finally, these concepts are important in text processing, which involves manipulating text to perform various tasks.

Regular grammars and finite automata also serve as a foundation for more advanced topics, such as context-free grammars, pushdown automata, and Turing machines. These concepts are essential for the study of computational complexity and algorithm design. By understanding regular grammars and finite automata, computer scientists can develop more efficient algorithms and solve more complex problems.

# 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:
   1. Create a local remote repository of a VCS hosting service (let us all use Github to avoid unnec- essary headaches);
   2. Choose a programming language, and my suggestion would be to choose one that supports all the main paradigms;
   3. 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:
   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;

# 1 Implementation

1. You can use 2 classes in order to represent the 2 main object which are the grammar and finite automa- ton. 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. **Code:**

**Main.py**

|  |
| --- |
| import string  from FiniteAutomaton import FiniteAutomaton from Grammar import Grammars class Main:  def \_\_init\_\_(self):  self.productions = {  'S': ['aA'],  'A': ['aB', 'bS'],  'B': ['aB','bC'],  'B': ['aA', 'b'],  }  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)  print(main.generate\_strings(0))  auto = main.grammar.to\_finite\_automaton()  automaton = {  'states': {'q0', 'q1', 'q2', 'q3', 'q4'},  'alphabet': {'a', 'b'},  'transition': {  'q0': {'a': 'q1'},  'q1': {'a': 'q1', 'b': 'q2'},  'q2': {'a': 'q3', 'b': 'q4'},  'q3': {'a': 'q1', 'b': 'q2'},  'q4': {'a': 'q3', 'b': 'q4'},  },  'start\_state': 'q0',  'final\_states': {'q1', 'q2', 'q3', 'q4'}  }   checker = FiniteAutomaton(automaton)  checker.check\_strings(['aab', 'abbb', 'bab', 'aabb', 'abcdefghijklmnoprsqtvyz'])  print(auto) |

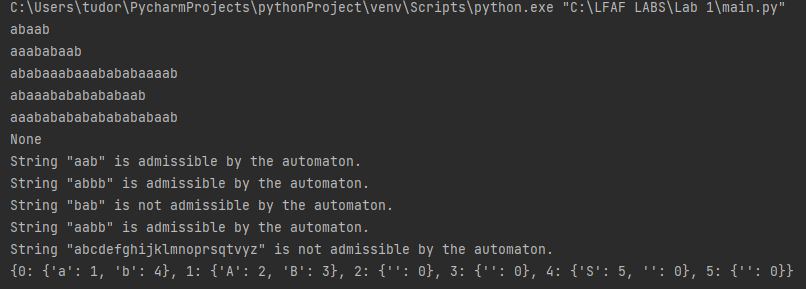
**Grammar.py**

|  |
| --- |
| 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):  start\_state = 0  auto = {start\_state: {}}  state\_count = 1   for symbol in self.productions:  for production in self.productions[symbol]:  current\_state = start\_state  for s in production:  if s not in auto[current\_state]:   auto[current\_state][s] = state\_count  auto[state\_count] = {}  state\_count += 1  current\_state = auto[current\_state][s]  if current\_state not in auto:  auto[current\_state] = {}  auto[current\_state][''] = start\_state  return auto |

**FiniteAutomaton.py**

|  |
| --- |
| class FiniteAutomaton:  def \_\_init\_\_(self, automaton):  self.states = automaton['states']  self.alphabet = automaton['alphabet']  self.transition = automaton['transition']  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.transition[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 admissible by the automaton.')  else:  print(f'String "{string}" is not admissible by the automaton.') |

* 1. **Screenshot:**



# Conclusions

The understanding of regular grammars and finite automata is crucial for comprehending the potential and constraints of computers. Computer scientists rely on these tools to design algorithms that are efficient in tasks such as text processing, pattern matching, and lexical analysis, which are widely used in real-world applications such as data compression, spam filters, and search engines. Furthermore, regular languages and automata enable computer systems to communicate with each other in a standardized way, which is vital for network communication.

The theoretical implications of the study of regular grammars and finite automata are also significant. By comprehending the limitations of these models, computer scientists can develop new models of computation that are more potent and can solve complex problems. For instance, context-free grammars and pushdown automata extend the capabilities of regular grammars and finite automata, allowing more complex patterns to be recognized. These models provide the basis for more advanced models such as Turing machines, which can solve any problem that a computer can solve.

Besides their practical applications in computer science, regular grammars and finite automata have links to other areas of mathematics such as group theory, topology, and algebra. These connections provide insight into the nature of computation and its relationship to other mathematical concepts.

In conclusion, regular grammars and finite automata are fundamental concepts in computer science with a wide range of practical applications in various fields such as natural language processing, compilers, and artificial intelligence. They are also essential tools for comprehending the capabilities and limitations of computers and have theoretical implications for computer science. The study of regular grammars and finite automata remains an active area of research and will undoubtedly shape the future of computer science.