# Laboratory Work No. 1

# Course: Formal Languages & Finite Automata

# Student: Iordan Liviu, Group: FAF-223

## Overview

A formal language can 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 number of configurations, which means that whenever a language is being created, its components should be selected in a way to make it as appropriate for its use case as possible. Of course, sometimes it is a matter of preference, that is 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 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 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;

## Fulfillment of Conditions:

- **Understanding of Formal Languages:**
    - The code demonstrates an understanding of formal languages by implementing context-free grammar and finite automaton.
    - It defines the necessary components of context-free grammar and illustrates how to generate strings from it.
    - Additionally, it converts the grammar into a finite automaton, which is another formal language model used for pattern recognition in strings.
- **Initial Project Setup:**
    - **GitHub Repository:** A GitHub repository has been created to store and update the project code. https://github.com/XRRRA/LFA_LABS
    - **Programming Language:** Python has been chosen as the programming language due to its simplicity and suitability for solving the given tasks effectively.

## Description Implementation:

This file (**main.py**) serves as the entry point of the program and the place where all the tests are located.

- Verification of the first requirement: Generates 5 valid strings based on the grammar rules and prints them.

```
grammar = Grammar()
generated_strings = grammar.generate_string()
```

```
# Print the generated strings
for i, string in enumerate(generated_strings, start=1):
    print(f"Generated string {i}: {string}")
```

- Converts the grammar to a finite automaton and manually adds transitions.
- Verification of the second requirement: Prints attributes of the finite automaton, including states, alphabet, transitions, initial state, and accepting states.

```
# Convert Grammar to FiniteAutomaton
finite_automaton = grammar.to_finite_automaton()

# Verify attributes of the FiniteAutomaton object
print("\nStates (Q):", finite_automaton.Q)
print("Alphabet (Sigma):", finite_automaton.Sigma)
print("Transitions (delta):", finite_automaton.delta)
print("Initial state (q0):", finite_automaton.q0)
print("Accepting state (F):", finite_automaton.F)
```

- Verification of the third requirement: Testing the **string_belongs_to_language** method of the finite automaton with an input string.

```
class FiniteAutomaton:
    def __init__(self):
        self.Q = set()
        self.Sigma = set()
        self.delta = set()
        self.q0 = None
        self.F = set()

    def string_belongs_to_language(self, input_string):
        current_state = self.q0
        for symbol in input_string:
            next_states = {next_state for (state, input_symbol, next_state) in self.delta
                           if state == current_state and input_symbol == symbol}
            if not next_states:
                return False
            current_state = next_states.pop()
        return current_state in self.F
```

The file (**grammar.py**) contains the implementation of context-free grammar.

- Context-free grammar consists of non-terminals (symbols that can be replaced by sequences of other symbols), terminals (symbols that cannot be replaced), and production rules (specify how symbols can be replaced).
- The **Grammar** class is defined, which represents context-free grammar.
- The **__init__** method initializes the grammar by defining the set of non-terminals (**VN**), terminals (**VT**), and production rules (**P**).
- The **generate_string** method generates 5 random strings based on the grammar rules.

```
def __init__(self):
    self.VN = {'S', 'B', 'D'}
    self.VT = {'a', 'b', 'c', 'd'}
    self.P = {
        'S': ['aS', 'bB'],
        'B': ['cB', 'd', 'aD'],
        'D': ['aB', 'b']
    }

def generate_string(self):
    generated_strings = []
    for _ in range(5):
        generated_string = self._generate_string_helper('S', '')
        generated_strings.append(generated_string)
    return generated_strings
```

- The **_generate_string_helper** method is a recursive helper function used internally to generate strings based on production rules.
- The **to_finite_automaton** method converts the grammar to a finite automaton.

```
def _generate_string_helper(self, symbol, current_string):
    if symbol in self.VT:
        return current_string + symbol
    else:
        productions = self.P[symbol]
        chosen_production = random.choice(productions)
        for s in chosen_production:
            current_string = self._generate_string_helper(s, current_string)
        return current_string

def to_finite_automaton(self):
    finite_automaton = FiniteAutomaton()

    finite_automaton.Q = self.VN.union(self.VT)
    finite_automaton.Sigma = self.VT
```

```
    finite_automaton.delta = set()

    for non_terminal, productions in self.P.items():
        for production in productions:
            if len(production) > 1:
                current_state = production[0]
                next_state = production[1]
                finite_automaton.delta.add((non_terminal, current_state, next_state))
            else:
                # Handle terminal elements
                if non_terminal in finite_automaton.F:
                    finite_automaton.delta.add((non_terminal, production, 'X'))
                else:
                    # Update terminal values
                    if production == 'b':
                        finite_automaton.delta.add((non_terminal, production, 'X'))
                    elif production == 'd':
                        finite_automaton.delta.add((non_terminal, production, 'X'))
                    else:
                        finite_automaton.delta.add((non_terminal, production,
production))

    finite_automaton.q0 = 'S'
    finite_automaton.F = {'X'}

    return finite_automaton
```