Ministry of Education, Culture and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work *No. 1*

Formal Languages & Finite Automata

Elaborated by: Gorcea Alexandrina,
FAF-223
Verified by: Dumitru Crețu,
univ. assist.

Chișinău, 2024

Theoretical considerations:

Formal languages and finite automata form the cornerstone of theoretical computer science, providing a framework to understand and analyze the nature of computation. At the heart of this theory lies the concept of formal languages, which serve as mathematical abstractions to represent sets of strings over a given alphabet. These languages are classified into various types, including regular, context-free, and recursively enumerable, each possessing distinct generative or recognition power.

Finite automata, on the other hand, are computational devices designed to recognize and accept or reject strings from a formal language. They come in various forms such as deterministic finite automata (DFA) and nondeterministic finite automata (NFA), each defining a specific set of computationally solvable problems.

The connection between formal languages and finite automata is profound. Regular languages, the simplest type of formal languages, can be precisely recognized by finite automata, either deterministic or nondeterministic. Context-free languages, a more expressive class, can be recognized by pushdown automata.

Objectives:

- Understand 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. 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.
- 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.

Implementation description

In the initial phase of this project, I took a meticulous approach to establish a solid foundation. The creation of a dedicated GitHub repository marked the commencement, laying the groundwork for a structured and version-controlled development environment.

The **Grammar** class encapsulates the essential components of a formal grammar, featuring nonterminals, terminals, and production rules. Its **generate_strings** method bellow is crucial as it orchestrates the generation of strings by iteratively replacing nonterminals in the initial word with randomly chosen productions, ensuring each generated string consists solely of terminal symbols. The process continues until the desired number of strings is achieved, and the result is returned.

```
def generate_strings(self, num_strings=5) -> list:
    ans = []
    while len(ans) < num_strings:
        current_word = self.start
        while not self.terminal_word(current_word):
            for char in current_word:
                if not self.terminal_char(char):
                    production = self.__replace(self.language[char])
                    current_word = current_word.replace(char, production, 1)
        if current_word not in ans:
            ans.append(current_word)
    return ans
```

Next, in the code bellow is the **terminal_word** method, which determines whether all characters in a given word are terminals by checking if each character is present in the set of terminals. The **terminal_char** method checks if a specific character is a terminal by verifying its presence in the set of terminals. The private `__replace` method randomly selects a production from a list of possible productions associated with a given nonterminal character.

```
def terminal_word(self, word: str) -> bool:
    # Check if all char from word are terminals
    return all(char in self.terminals for char in word)

def terminal_char(self, char: str) -> bool:
    # Check if char is terminal
    return char in self.terminals

def __replace(self, value: list) -> str:
    # Choose random from values of dictionary of given char
    return random.choice(value)
```

The **FiniteAutomaton** class initializes a finite automaton using information from a given context-free grammar (`Grammar`). It includes the set of states (`Q`), the alphabet of symbols (`Alphabet`), the initial state (`q0`), the transition function (`delta`), and the set of accepting states (`F`). Additionally, the

accepting state is extended with a unique symbol `"X"`, likely representing the end of a production in the context of the grammar.

```
class FiniteAutomaton:
    AlexandrinaGorcea
    def __init__(self, grammar: Grammar, _delta: dict) -> None:
        self.Q = grammar.nonterminals + ["X"]
        self.Alphabet = grammar.terminals
        self.q0 = grammar.start
        self.delta = _delta
        self.F = ["X"]
```

In this class, I have the method **grammar_to_DFA** that converts a context-free grammar into a finite automaton by mapping the grammar's nonterminals and their productions to states and transitions in the automaton. The resulting automaton has states, transitions, an initial state (`q0`), and an accepting state (`"X"`) representing the end of a production.

```
def grammar_to_DFA(grammar: Grammar) -> 'FiniteAutomaton':
    delta = {}

    for nonterminal in grammar.language:
        for production in grammar.language[nonterminal]:
            if len(production) > 1:
                transition = production[0]
                result_state = production[1]
                delta.setdefault(nonterminal, {})[transition] = result_state
            else:
                transition = production
                result_state = "X"
                delta.setdefault(nonterminal, {})[transition] = result_state

    return FiniteAutomaton(grammar, delta)
```

```
def word_can_be_created(self, string: str) -> bool:
    current_state = self.q0

    for char in string:
        if char not in self.Alphabet or char not in self.delta[current_state]:
            return False
        current_state = self.delta[current_state][char]

    return current_state in self.F
```

The method **word_can_be_created** from above checks whether the given string can be generated by the finite automaton. It iterates through each character of the string, transitioning between states based on the automaton's delta function, and returns `True` if the final state is an accepting state (`self.F`), otherwise `False`.

Conclusion:

Point B:

```
babca
baa
bbbaa
aca
bbaca
```

Point C:

```
Q: ['S', 'F', 'D', 'X']
Alphabet: ['a', 'b', 'c']
q0: S
F: ['X']
Delta:
(S, a) -> F
(S, b) -> S
(F, b) -> F
(F, c) -> D
(F, a) -> X
(D, c) -> S
(D, a) -> X
```

Point D (example):

```
The following string 'abca' belongs to the language: True
```

In conclusion, during this laboratory I made a well-structured and versatile implementation of formal language theory concepts. The **Grammar** class efficiently generates strings adhering to specified grammar rules, while the **FiniteAutomaton** class seamlessly transforms context-free grammars into finite automata. This code provides a valuable foundation for exploring computational linguistics, exemplifying key principles in the representation and manipulation of formal languages. The strings provided in Point B illustrate examples that can be generated or accepted by the implemented finite automaton or grammar. Point C provides a succinct representation of the finite automaton's structure and transition rules, and the example in Point D, 'abca,' demonstrates the correct recognition of strings within the defined language, collectively showcasing the effectiveness of the implemented code in formal language manipulation.

Bibliography:

1. **Formal Languages and Automata**
   https://web.mei.edu/textual?FilesData=An_Introduction_To_Formal_Languages_And_Automata.pdf&digit=Y92z602