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.

Delving into the codebase, two pivotal classes, namely **gfa** and **wordgeneration**, were crafted to serve as the backbone of our implementation. These classes encapsulate distinct functionalities catering to the unique demands of the project.

The **gfa** class takes on the responsibility of translating a context-free grammar into a finite automaton. This class stands as a testament to the intricate connection between formal language theory and practical implementation. Its core functionalities include defining states and transitions based on the nonterminal symbols of the grammar, thereby modeling the grammar as a finite automaton. The inclusion of a special start state, 'S0', further refines the automaton's representation. Though the class is in progress, the ultimate goal is to achieve adaptability, enabling it to handle a variety of grammatical structures.

On the other hand, the **wordgeneration** class is dedicated to crafting strings from the specified grammar. It encapsulates the rules of the context-free grammar, providing essential methods for generating words. The class is structured to facilitate ease of use and comprehension, aiding in the exploration and understanding of the grammar rules.

- Grammar Code: The given code is a recursive function called wordgeneration that generates words based on a provided grammar and symbol. It randomly selects production rules from the grammar for non-terminal symbols until only terminal symbols (words) remain. The function returns the generated word as a string.

```
def generate_word(grammar, symbol):
    if symbol not in grammar:
        return symbol
    production = random.choice(grammar[symbol])
    return ''.join(generate_word(grammar, s) for s in production)
```

- FA Code: The given code defines a function grammar_to_finite_automaton that converts a grammar into a finite automaton representation. The automaton consists of states, an alphabet, transitions, a start state, and accept states. The function extracts states and alphabet from the grammar, creates transitions based on the grammar rules, designates a start state, and determines accept states based on words generated by the grammar. The function returns the automaton components as a tuple.

```
def grammar_to_finite_automaton(grammar):
    states = set(grammar.keys()) | {'S0'}
    alphabet = set([s for rule in grammar.values() for s in rule if s.
islower()])

    transitions = {}
    for state in states:
        for symbol in alphabet:
            dest_state = 'dead'
            for rule in grammar.get(state, []):
                if symbol in rule:
                    dest_state = ''.join([s if s.isupper() else '' for
 s in rule])
            transitions[(state, symbol)] = dest_state

    start_state = 'S0'
    accept_states = set(state for state in states if any(word == state
 for word in generate_word(grammar, 'S')))
    return (states, alphabet, transitions, start_state, accept_states)
```

Conclusions:

- Results for the Grammar code

```
abccbaa
bbbbaccbaa
bbabba
baccbaba
baca
```

- Results for the FA code

```
({'D', 'F', 'S', 'S0'}, {'a'}, {('D', 'a'): '', ('F', 'a'): '', ('S',
'a'): 'F', ('S0', 'a'): 'dead'},
 'S0', set())
```

In conclusion, the grammar code successfully generated strings conforming to the specified grammar rules, producing varied and valid outputs such as ‘abccbaa’ and ‘bbabba.’ The Finite Automaton (FA) code demonstrated the conversion from a context-free grammar to an FA, represented by states, transitions, and accept states. The FA recognized strings generated by the grammar, as indicated by the acceptance of ‘abccbaa’ and ‘bbabba.’ The systematic implementation of both the grammar and FA code provides a practical insight into formal language theory, showcasing the translation from abstract rules to tangible automata.

Bibliography:

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