

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. 2*

**Formal Languages & Finite Automata**

**Elaborated by:** Gorcea Alexandrina,  
FAF-223

**Verified by:** Dumitru Crețu,  
univ. assist.

Chișinău, 2024

### Theoretical considerations:

A finite automaton is a mechanism used to represent processes of different kinds. It can be compared to a state machine as they both have similar structures and purpose as well. The word finite signifies the fact that an automaton comes with a starting and a set of final states. In other words, for process modeled by an automaton has a beginning and an ending.

Based on the structure of an automaton, there are cases in which with one transition multiple states can be reached which causes non determinism to appear. In general, when talking about systems theory the word determinism characterizes how predictable a system is. If there are random variables involved, the system becomes stochastic or non-deterministic.

That being said, the automata can be classified as non-/deterministic, and there is in fact a possibility to reach determinism by following algorithms which modify the structure of the automaton.

### Objectives:

1. Understand what an automaton is and what it can be used for.
2. Continuing the work in the same repository and the same project, the following need to be added:
  - a. Provide a function in your grammar type/class that could classify the grammar based on Chomsky hierarchy.
  - b. For this you can use the variant from the previous lab.
3. According to your variant number (by universal convention it is register ID), get the finite automaton definition and do the following tasks:
  - a. Implement conversion of a finite automaton to a regular grammar.
  - b. Determine whether your FA is deterministic or non-deterministic.
  - c. Implement some functionality that would convert an NDFA to a DFA.
  - d. Represent the finite automaton graphically (Optional, and can be considered as a *bonus point*):
    - You can use external libraries, tools or APIs to generate the figures/diagrams.
    - Your program needs to gather and send the data about the automaton and the lib/tool/API return the visual representation.

### Implementation description

The **Grammar** class, encapsulates the essence of formal grammars and their relation to finite automata. The class represents a grammar using variables (**VN**), terminals (**VT**), and production rules (**P**). Notably, the class offers functionalities to generate strings based on the grammar, convert the grammar to a finite automaton, and identify the type of the grammar in Chomsky Hierarchy.

The **generate\_strings** method generates a specified number of strings adhering to the grammar rules, utilizing the **generate\_recursive** helper function. This recursive function explores the grammar's production rules to construct strings, ensuring diversity by preventing duplicates.

```
class Grammar:
    # AlexandrinaGorcea
    def __init__(self):
        self.VN = set()
        self.VT = set()
        self.P = {}

    # AlexandrinaGorcea
    def generate_strings(self, num_strings=5) -> list:
        ans = []
        while len(ans) < num_strings:
            current_word = self.generate_recursive('S', '')
            if current_word not in ans:
                ans.append(current_word)
        return ans

    # AlexandrinaGorcea
    def generate_recursive(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_recursive(s, current_string)
            return current_string
```

The **to\_finite\_automaton** method facilitates the transformation of the grammar into a finite automaton. It defines states, alphabet, and transitions based on the grammar's variables, terminals, and production rules, respectively.

```
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():
        finite_automaton.delta.update(
            (
                (non_terminal, production[0], production[1])
                if len(production) > 1
                else (
```

```

        (non_terminal, production, 'X')
        if non_terminal in finite_automaton.F or production in {'b', 'd'}
        else (non_terminal, production, production)
    )
)
for production in productions
)

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

return finite_automaton

```

The **identify\_grammar\_type** method determines the Chomsky Hierarchy type of the grammar. It considers the structure of production rules to classify the grammar as Type-0 (Unrestricted), Type-1 (Context-Sensitive), Type-2 (Context-Free), or Type-3 (Regular). This method employs various conditions, such as the presence of epsilon productions or the length of productions, to make informed decisions about the grammar's classification within the Chomsky Hierarchy.

```

def identify_grammar_type(self):
    start_symbol = next(iter(self.P), None)
    has_epsilon = any('ε' in production for productions in self.P.values() for production in productions)

    for non_terminal, productions in self.P.items():
        for production in productions:
            if len(production) > 2:
                return "Type-0 (Unrestricted)"
            elif len(production) == 2 and production[0] in self.VN and production[1] in self.VT:
                return "Type-1 (Context-Sensitive)"
            elif len(production) == 1 and production[0] in self.VT:
                return "Type-3 (Regular)"

    if start_symbol and not has_epsilon:
        return "Type-2 (Context-Free)"

    return "Type-0 (Unrestricted)"

```

The second class, named **FiniteAutomaton**, plays a crucial role in language recognition through finite automata. The class encompasses methods for checking if a given string is in the language accepted by the automaton, converting the automaton to a regular grammar, determining whether the automaton is deterministic, and transforming it into a deterministic finite automaton.

The **is\_string\_in\_language** method utilizes the automaton's transition function to determine if a given input string is accepted by the automaton. It iterates over the symbols of the input string, updating the current state according to the transitions defined by the automaton, and finally checks if the final state is in the set of accepting states.

```

def is_string_in_language(self, input_string):
    # AlexandrinaGorcea
    def transition(current_state, symbol):
        next_states = {next_state for (state, input_symbol, next_state) in self.delta
                        if state == current_state and input_symbol == symbol}
        return next_states.pop() if next_states else None
    final_state = reduce(transition, input_string, self.q0)
    return final_state in self.F

```

The **to\_regular\_grammar** method below converts the finite automaton to a regular grammar. It captures the essential components of the automaton, including states, input symbols, and transitions, in the form of production rules for each state in the regular grammar.

```

def to_regular_grammar(self):
    regular_grammar = Grammar()
    regular_grammar.VN = self.Q
    regular_grammar.VT = self.Sigma
    regular_grammar.P = {state: [] for state in self.Q}

    for state, input_symbol, next_state in self.delta:
        if next_state != 'X':
            production = f"{input_symbol}-{next_state}"
            regular_grammar.P[state].append(production)

    return regular_grammar

```

The **is\_deterministic** method assesses whether the finite automaton is deterministic by inspecting the transitions for each state and input symbol. If any state has multiple transitions on the same symbol or there are unreachable states, the automaton is deemed nondeterministic.

The **to\_deterministic\_FiniteAutomaton** method transforms the nondeterministic finite automaton (NFA) into a deterministic finite automaton (DFA) using the subset construction technique. It computes the epsilon closure of states and systematically constructs the DFA's transition table.

These methods collectively offer a versatile set of tools for working with finite automata, providing functionalities for language recognition, grammar conversion, and deterministic automaton transformation.

```

def is_deterministic(self):
    visited_states = set()

    for state in self.Q:
        for symbol in self.Sigma:
            next_states = {next_state for (_, input_symbol, next_state) in self.delta
                             if _ == state and input_symbol == symbol}

            if len(next_states) != 1:
                return False


            visited_states.update(next_states)

    return len(visited_states) == len(self.Q) # Ensure all states are reachable

```

```

def to_deterministic_FiniteAutomaton(self):
    dfa = FiniteAutomaton()
    dfa.Sigma = self.Sigma
    dfa.q0 = frozenset([self.q0]) # Initial state is the epsilon closure of the original initial state
    dfa.F = set()
    dfa.Q = set([dfa.q0]) # Initialize set of states
    dfa.delta = set()

    # Function to compute epsilon closure of a state in the NFA
     AlexandrinaGorcea
    def epsilon_closure(state):
        closure = {state}
        stack = list(state)
        closure.update(next_state for (_, input_symbol, next_state) in self.delta
                        if input_symbol == 'ε' and next_state not in closure)
        return frozenset(closure)

    unprocessed_states = [dfa.q0]

    while unprocessed_states:
        current_state = unprocessed_states.pop(0)

```

```

    for symbol in dfa.Sigma:
        next_state = frozenset(
            next_state
            for state in current_state
            for (_, input_symbol, next_state) in self.delta
            if state in current_state and input_symbol == symbol
        )

        next_state_closure = epsilon_closure(next_state)

        if next_state_closure:
            dfa.delta.add((current_state, symbol, next_state_closure))

            if next_state_closure not in dfa.Q:
                dfa.Q.add(next_state_closure)
                unprocessed_states.append(next_state_closure)

            if any(state in self.F for state in next_state_closure):
                dfa.F.add(next_state_closure)

    return dfa

```

Next is **main.py**, which showcases the application of the **Grammar** and **FiniteAutomaton** classes.

```

grammar = Grammar()
grammar.VN = {'S', 'F', 'D'}
grammar.VT = {'a', 'b', 'c'}
grammar.P = {
    'S': ['aF', 'bS'],
    'F': ['bF', 'cD', 'a'],
    'D': ['cS', 'a']
}

# Check the type of the grammar
print("Grammar Type:", grammar.identify_grammar_type())

```

This part initializes a context-free grammar with variables (**VN**), terminals (**VT**), and production rules (**P**). It then identifies and prints the Chomsky Hierarchy type of the grammar. From here, we obtain the following output:

```
Grammar Type: Type-3 (Regular)
```

This part initializes a finite automaton and converts it into a regular grammar, capturing states, symbols, and transitions as production rules for each state in the regular grammar.

```
finite_automaton = FiniteAutomaton()
finite_automaton.Q = {'q0', 'q1', 'q2', 'q3'}
finite_automaton.Sigma = {'a', 'b', 'c'}
finite_automaton.delta = {('q0', 'b', 'q0'), ('q0', 'a', 'q1'), ('q1', 'c', 'q1'),
                          ('q1', 'a', 'q2'), ('q3', 'a', 'q1'), ('q3', 'a', 'q3'), ('q2', 'a', 'q3')}
finite_automaton.q0 = 'q0'
finite_automaton.F = {'q2'}

# Convert finite automaton to regular grammar
regular_grammar = finite_automaton.to_regular_grammar()

# Print the regular grammar productions
print("\nRegular Grammar Productions for the NDFA:")
productions_output = "\n".join(f"{non_terminal} -> {production}" for non_terminal, productions in regular_grammar.P.items())
print(productions_output)

# Determine if the finite automaton is deterministic
is_deterministic = finite_automaton.is_deterministic()
print("The NDFA is deterministic." if is_deterministic else "The NDFA is non-deterministic.")

# Convert finite automaton to deterministic finite automaton
dfa = finite_automaton.to_deterministic_FiniteAutomaton()

# Check if the resulting DFA is deterministic
is_deterministic_dfa = dfa.is_deterministic()
print("The converted DFA is deterministic." if is_deterministic_dfa else "The converted DFA is non-deterministic.")
```

This section prints the production rules of the regular grammar obtained from the finite automaton, showcasing the mapping of states to their respective production rules, checks if the original finite automaton is deterministic and prints the result, transforms the original nondeterministic finite automaton (NFA) into a deterministic finite automaton (DFA) using the subset construction technique, verifies if the resulting deterministic finite automaton (DFA) is indeed deterministic and prints the outcome. Here is the output:

```
Regular Grammar Productions for the NDFA:
q3 -> aq3
q3 -> aq1
q0 -> aq1
q0 -> bq0
q1 -> cq1
q1 -> aq2
q2 -> aq3
The NDFA is non-deterministic.
The converted DFA is non-deterministic.
```



## **Conclusion:**

In summary, the laboratory exploration delves into the fundamental concepts of formal language theory, exemplifying the practical applications of context-free grammars (CFG) and finite automata (FA) using Python. The initial phase defines a CFG and identifies its Chomsky Hierarchy type, elucidating the structural complexities of recognized patterns.

Subsequently, the conversion of a non-deterministic finite automaton (NFA) into a deterministic finite automaton (DFA) showcases the importance of determinism in computational linguistics. This transformative process highlights the intricacies of language recognition and the strategies employed to enhance computational efficiency.

The laboratory not only provides theoretical insights into the Chomsky Hierarchy but also offers hands-on experience in translating between formal language representations. The scripting exercises not only deepen the understanding of language structures but also underscore the significance of determinism in automaton design.

Overall, the laboratory serves as a valuable exploration, bridging theory and practical implementation, and fostering a holistic understanding of formal language theory concepts through engaging Python programming exercises. Through this experiential learning, students gain a deeper appreciation for the complexities and applications of formal languages and automata in the realm of computer science and linguistics.

## **Bibliography:**

### **1. Formal Languages and Automata**

[https://web.mei.edu/textual?FilesData=An\\_Introduction\\_To\\_Formal\\_Languages\\_And\\_Automata.pdf&digit=Y92z602](https://web.mei.edu/textual?FilesData=An_Introduction_To_Formal_Languages_And_Automata.pdf&digit=Y92z602)