**Ministerul Educaţiei și Cercetării al Republicii Moldova**
**Universitatea Tehnică a Moldovei**
**Facultatea Calculatoare, Informatică și Microelectronică**

# Laboratory work 2:

# Course: Formal Languages and Finite Automata
# Topic: Determinism in Finite Automata. Conversion from NFA to DFA. Chomsky Hierarchy.

Elaborated:
st. gr. FAF-221 Chichioi Iuliana

Verified:

asist. univ. Cretu Dumitru

Chişinău - 2023

## TABLE OF CONTENTS

**THEORY**....................2
**OBJECTIVES**....................3
**IMPLEMENTATION**....................3
**RESULTS**....................7
**GRAPH REPRESENTATION**....................8
**CONCLUSION**....................8
Achieved Objectives:....................8

## THEORY

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

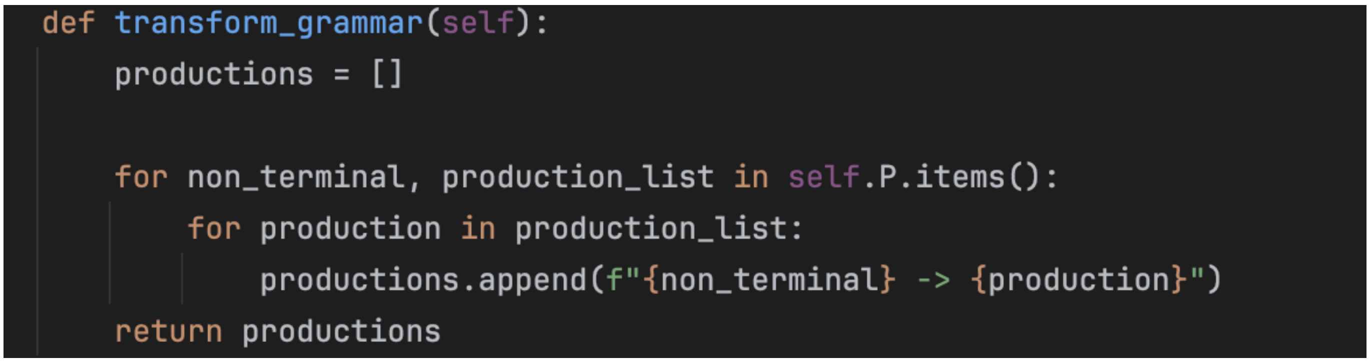
1. Understand what an automaton is and what it can be used for.
2. Continuing the work in the same repository and the same project, the following need to be added:
   a. Provide a function in your grammar type/class that could classify the grammar based on Chomsky hierarchy.
   b. For this you can use 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

For the implementation, **Python** was chosen as the programming language due to its familiarity and ease of use. In my previous Lab_1.py I added 2 new functionalities.

**Transforming Grammar:**

I implemented a method called transform_grammar in the Grammar class.
This method iterates over the productions and transforms them into a list of strings representing the productions in the form of non_terminal -> production.

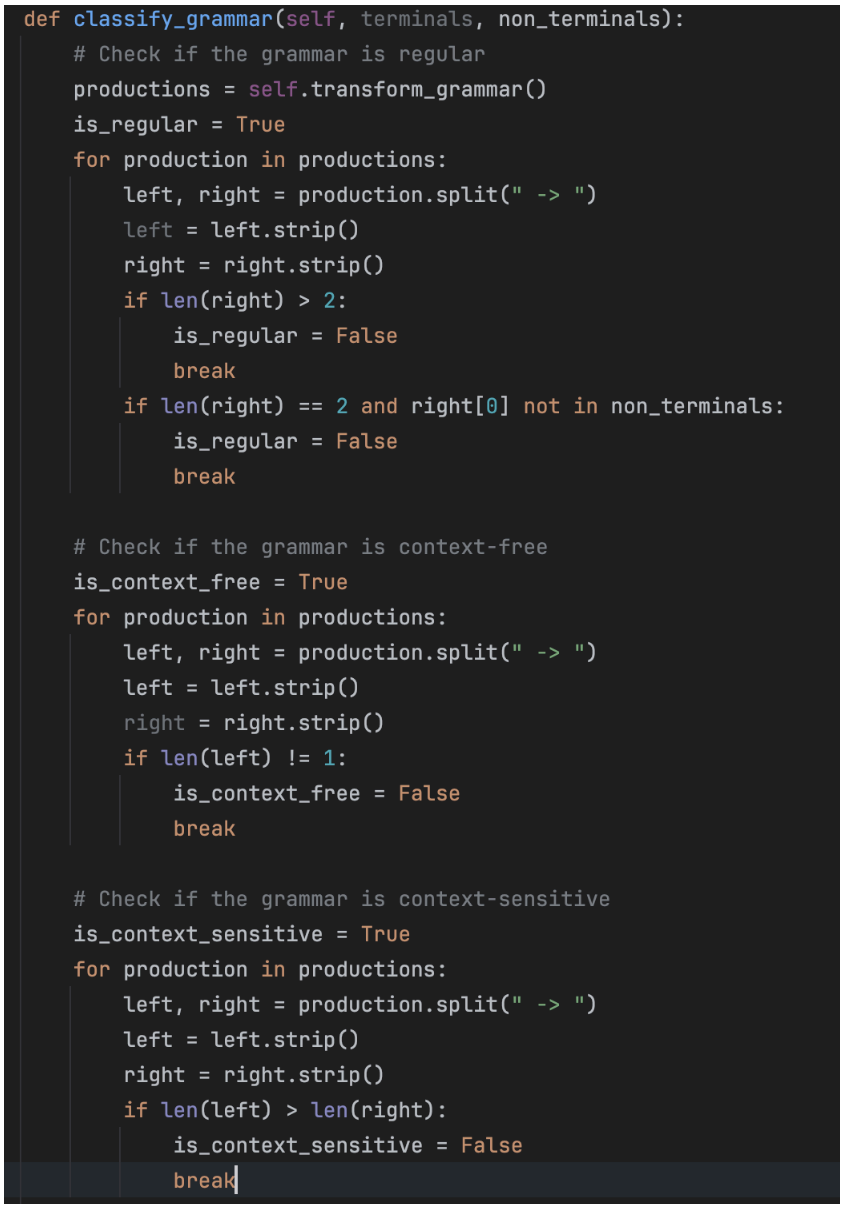


```python
def transform_grammar(self):
    productions = []

    for non_terminal, production_list in self.P.items():
        for production in production_list:
            productions.append(f"{non_terminal} -> {production}")
    return productions
```

*Figure 1 transform_grammar*

**Classifying the Grammar**

I implemented a method called classify_grammar in the Grammar class.This method checks if the grammar is regular, context-free, context-sensitive, or unrestricted based on the Chomsky hierarchy rules.Finally, I returned the classification of the grammar.



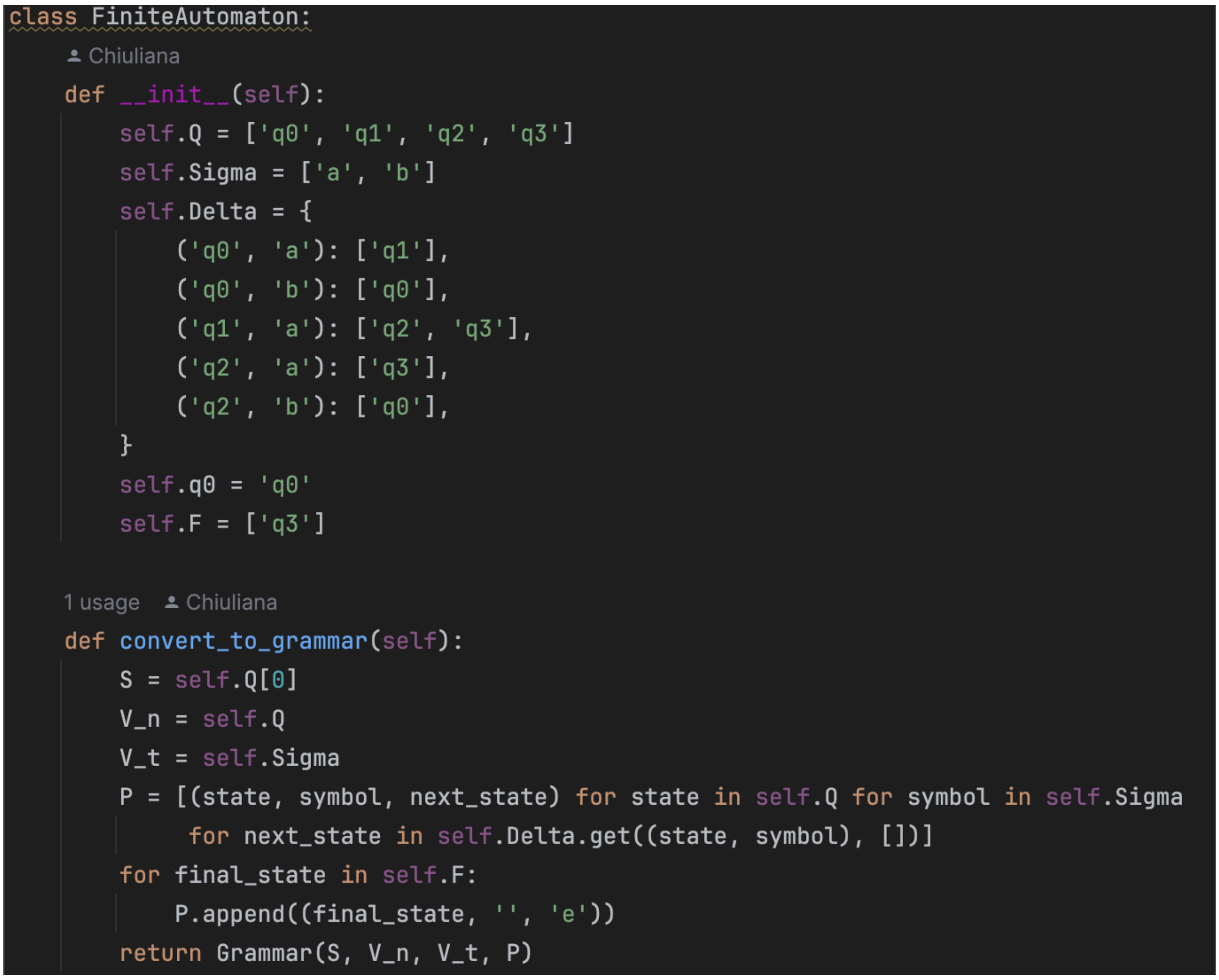
```python
def classify_grammar(self, terminals, non_terminals):
    # Check if the grammar is regular
    productions = self.transform_grammar()
    is_regular = True
    for production in productions:
        left, right = production.split(" -> ")
        left = left.strip()
        right = right.strip()
        if len(right) > 2:
            is_regular = False
            break
        if len(right) == 2 and right[0] not in non_terminals:
            is_regular = False
            break

    # Check if the grammar is context-free
    is_context_free = True
    for production in productions:
        left, right = production.split(" -> ")
        left = left.strip()
        right = right.strip()
        if len(left) != 1:
            is_context_free = False
            break

    # Check if the grammar is context-sensitive
    is_context_sensitive = True
    for production in productions:
        left, right = production.split(" -> ")
        left = left.strip()
        right = right.strip()
        if len(left) > len(right):
            is_context_sensitive = False
            break
```

*Figure 2 classify_grammar*

In Lab_2.py implemented the rest of the laboratory work objectives. I will provide step by step guide how I achieved the desired results.

**Converting Finite Automaton to Grammar:**

I defined a method called convert_to_grammar in the FiniteAutomaton class to create a grammar representation of the finite automaton. This method generates productions based on transitions and states of the automaton.

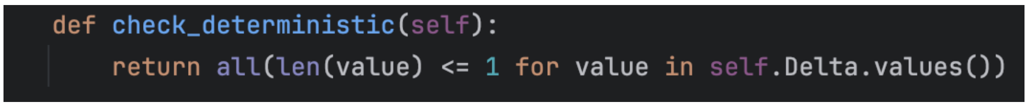


```python
class FiniteAutomaton:
    ≗ Chiuliana
    def __init__(self):
        self.Q = ['q0', 'q1', 'q2', 'q3']
        self.Sigma = ['a', 'b']
        self.Delta = {
            ('q0', 'a'): ['q1'],
            ('q0', 'b'): ['q0'],
            ('q1', 'a'): ['q2', 'q3'],
            ('q2', 'a'): ['q3'],
            ('q2', 'b'): ['q0'],
        }
        self.q0 = 'q0'
        self.F = ['q3']

    1 usage  ≗ Chiuliana
    def convert_to_grammar(self):
        S = self.Q[0]
        V_n = self.Q
        V_t = self.Sigma
        P = [(state, symbol, next_state) for state in self.Q for symbol in self.Sigma
             for next_state in self.Delta.get((state, symbol), [])]
        for final_state in self.F:
            P.append((final_state, '', 'e'))
        return Grammar(S, V_n, V_t, P)
```

*Figure 3 class FiniteAutomaton & classify_grammar*

**Checking Determinism of Finite Automaton:**

I implemented a method called check_deterministic in the FiniteAutomaton class to check if all transitions of the automaton are deterministic.



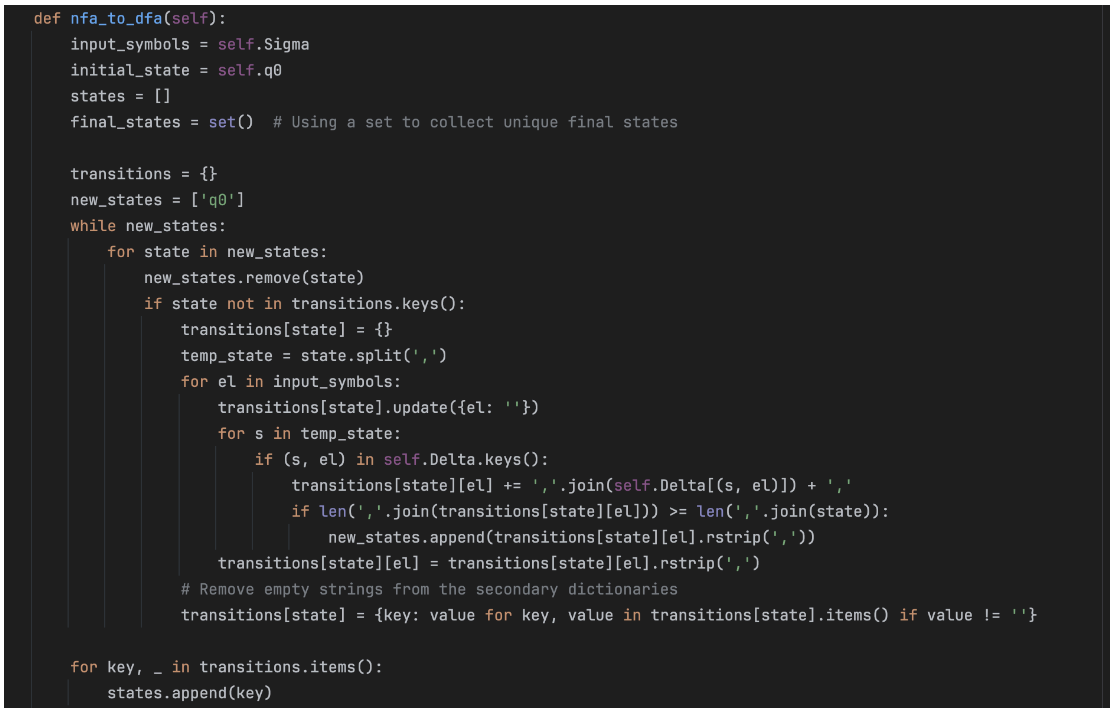
```python
def check_deterministic(self):
    return all(len(value) <= 1 for value in self.Delta.values())
```

*Figure 4 check_deterministic*

**Converting Non-Deterministic Finite Automaton to Deterministic Finite Automaton (NFA to DFA):**

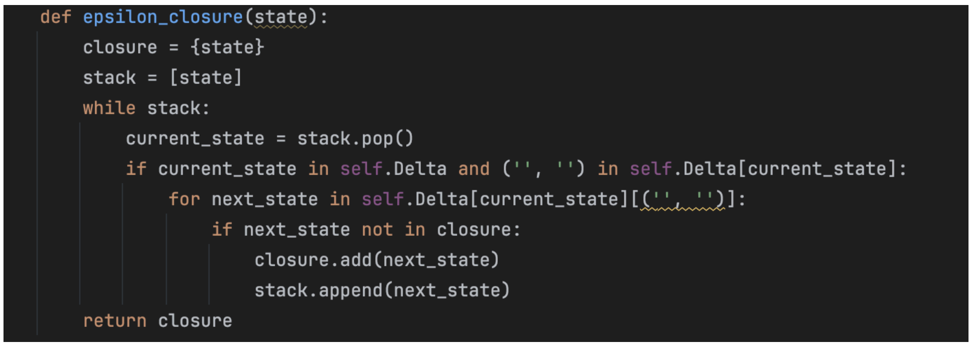
I implemented a method called nfa_to_dfa in the FiniteAutomaton class to convert the non-deterministic finite automaton to a deterministic finite automaton.

This method performs epsilon closure and generates transitions for the DFA.



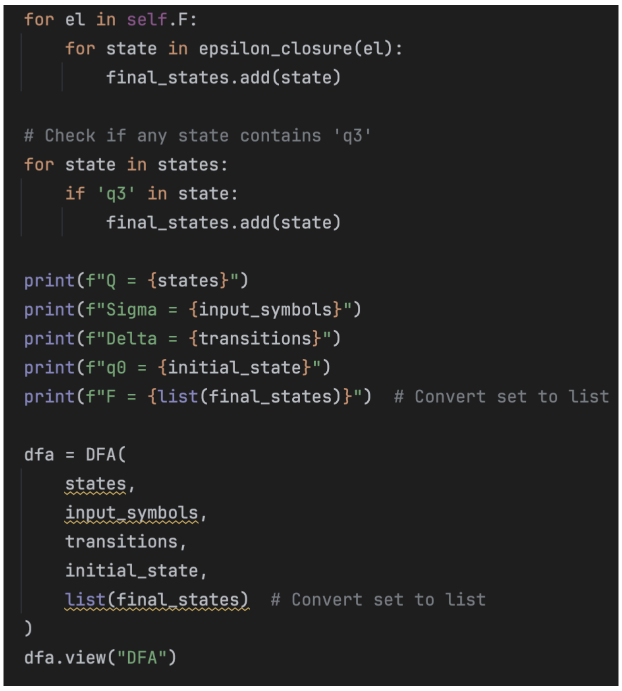
```python
def nfa_to_dfa(self):
    input_symbols = self.Sigma
    initial_state = self.q0
    states = []
    final_states = set()  # Using a set to collect unique final states

    transitions = {}
    new_states = ['q0']
    while new_states:
        for state in new_states:
            new_states.remove(state)
            if state not in transitions.keys():
                transitions[state] = {}
                temp_state = state.split(',')
                for el in input_symbols:
                    transitions[state].update({el: ''})
                    for s in temp_state:
                        if (s, el) in self.Delta.keys():
                            transitions[state][el] += ','.join(self.Delta[(s, el)]) + ','
                            if len(','.join(transitions[state][el])) >= len(','.join(state)):
                                new_states.append(transitions[state][el].rstrip(','))
                    transitions[state][el] = transitions[state][el].rstrip(',')
                # Remove empty strings from the secondary dictionaries
                transitions[state] = {key: value for key, value in transitions[state].items() if value != ''}

    for key, _ in transitions.items():
        states.append(key)
```

*Figure 5 nfa_to_dfa Part 1*



```python
def epsilon_closure(state):
    closure = {state}
    stack = [state]
    while stack:
        current_state = stack.pop()
        if current_state in self.Delta and ('', '') in self.Delta[current_state]:
            for next_state in self.Delta[current_state][('', '')]:
                if next_state not in closure:
                    closure.add(next_state)
                    stack.append(next_state)
    return closure
```

*Figure 6 epsilon_closure*

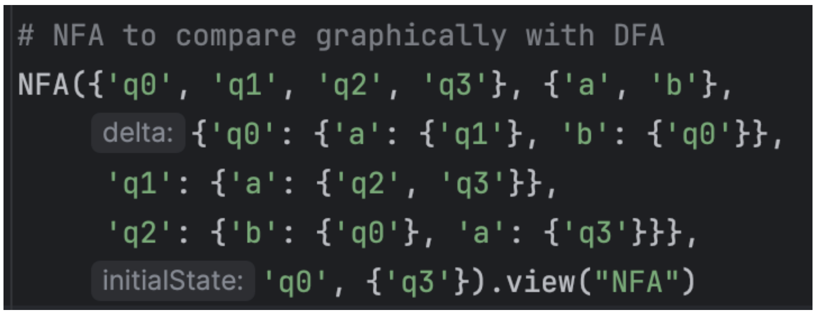


```python
for el in self.F:
    for state in epsilon_closure(el):
        final_states.add(state)

# Check if any state contains 'q3'
for state in states:
    if 'q3' in state:
        final_states.add(state)

print(f"Q = {states}")
print(f"Sigma = {input_symbols}")
print(f"Delta = {transitions}")
print(f"q0 = {initial_state}")
print(f"F = {list(final_states)}")  # Convert set to list

dfa = DFA(
    states,
    input_symbols,
    transitions,
    initial_state,
    list(final_states)  # Convert set to list
)
dfa.view("DFA")
```

*Figure 7 nfa_to_dfa Part 2*

**Displaying NFA for Comparison:**

Finally, I displayed the NFA graphically for comparison with the DFA.

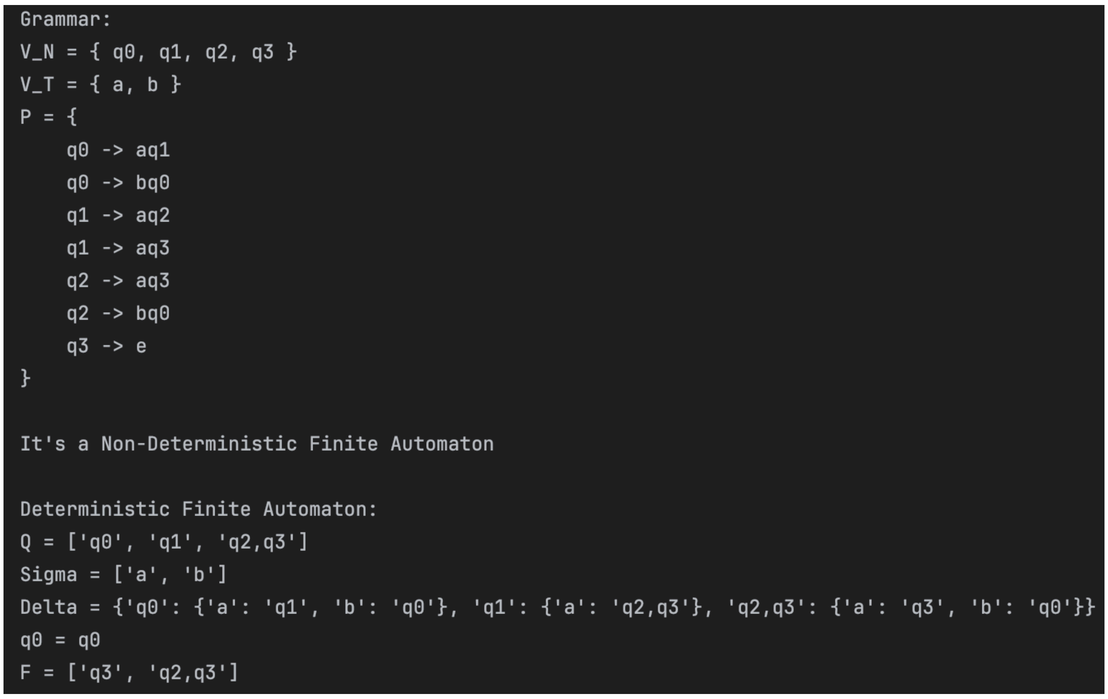


```
# NFA to compare graphically with DFA
NFA({'q0', 'q1', 'q2', 'q3'}, {'a', 'b'},
     delta: {'q0': {'a': {'q1'}, 'b': {'q0'}},
      'q1': {'a': {'q2', 'q3'}},
      'q2': {'b': {'q0'}, 'a': {'q3'}}},
     initialState: 'q0', {'q3'}).view("NFA")
```

*Figure 8 classify_grammar*

## RESULTS

The grammar consists of non-terminals (V_N), terminals (V_T), and productions (P), where each production maps a non-terminal to a sequence of symbols. The grammar represents a Non-Deterministic Finite Automaton (NFA), implying multiple choices for transitions at certain states. The NFA is converted into a Deterministic Finite Automaton (DFA) with unique transitions for each state and input symbol. The DFA is described by its states, alphabet, transitions, initial state, and final states, providing a clear representation of the language it accepts.



```
Grammar:
V_N = { q0, q1, q2, q3 }
V_T = { a, b }
P = {
    q0 -> aq1
    q0 -> bq0
    q1 -> aq2
    q1 -> aq3
    q2 -> aq3
    q2 -> bq0
    q3 -> e
}

It's a Non-Deterministic Finite Automaton

Deterministic Finite Automaton:
Q = ['q0', 'q1', 'q2,q3']
Sigma = ['a', 'b']
Delta = {'q0': {'a': 'q1', 'b': 'q0'}, 'q1': {'a': 'q2,q3'}, 'q2,q3': {'a': 'q3', 'b': 'q0'}}
q0 = q0
F = ['q3', 'q2,q3']
```

*Figure 9 results*

## GRAPH REPRESENTATION

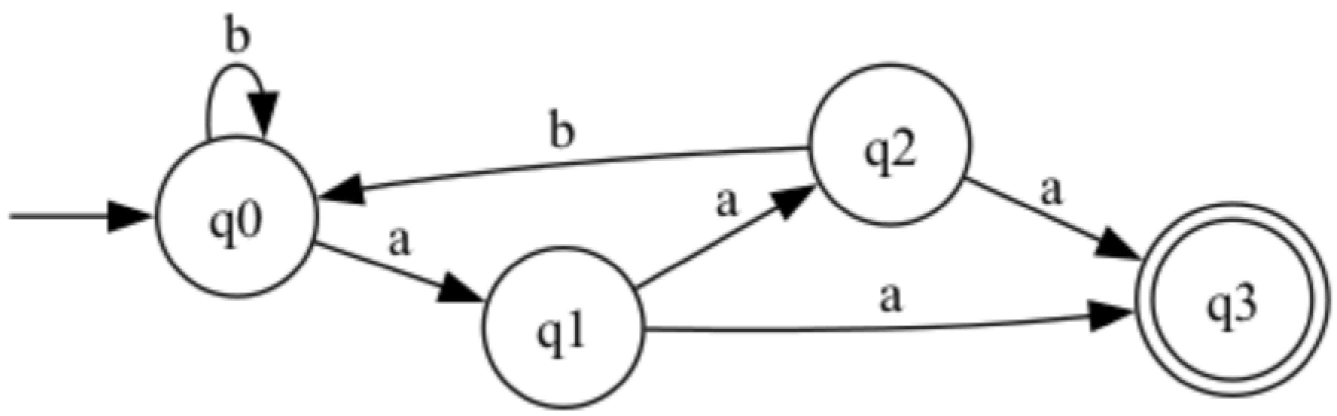


*Figure 10 graph representation of NFA*

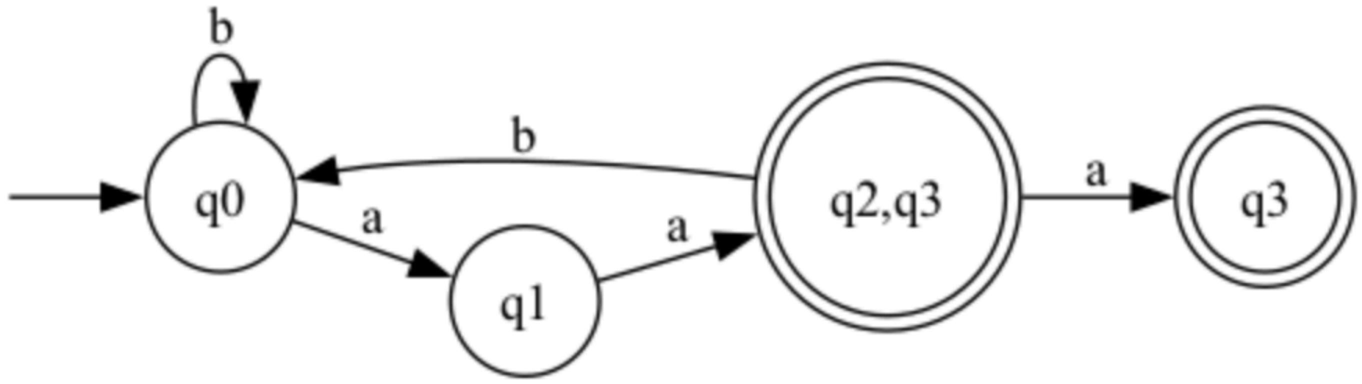


*Figure 11 graph representation of DFA*

## CONCLUSION

In conclusion, this laboratory work focused on exploring the concepts of determinism in finite automata and the conversion process from a Non-Deterministic Finite Automaton (NFA) to a Deterministic Finite Automaton (DFA), within the context of Formal Languages and Finite Automata.

Achieved Objectives:

**Understanding Automata:**Through the implementation and analysis of finite automata, I gained a deeper understanding of their role in modeling processes and languages.

**Classification of Grammar:**I successfully implemented a function to classify grammars based on the Chomsky hierarchy rules, providing insights into the nature and complexity of the grammars.

**Conversion to Grammar:**The conversion of a finite automaton to a regular grammar was achieved by generating productions based on the transitions and states of the automaton.

**Determinism Analysis:**Determinism in finite automata was explored through the implementation of methods to determine whether an automaton is deterministic or non-deterministic.

**NFA to DFA Conversion:**The process of converting a Non-Deterministic Finite Automaton (NFA) to a Deterministic Finite Automaton (DFA) was successfully implemented. This involved performing epsilon closure and generating unique transitions for each state and input symbol.

The results obtained from the laboratory work provided a clear representation of the grammar, NFA, and DFA. Through descriptive analysis and graphical representation, I gained insights into the structure and behavior of these formal language models.

In conclusion, the laboratory work enabled me to deepen my understanding of formal languages and finite automata, particularly focusing on determinism and conversion processes. By implementing and analyzing various algorithms and methods, I gained practical experience in handling and manipulating formal language structures. Overall, the knowledge and skills acquired through this laboratory work are valuable in the study and application of computational linguistics and automata theory.