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

Laboratory work nr. 2
Course: Formal languages and finite automata
Topic: Determinism in Finite Automata. Conversion from NDFA to DFA. Chomsky Hierarchy.

Elaborated:
st. gr. FAF-221 Berco Andrei

Verified:
asist. univ. Cretu Dumitru

Chişinău - 2024

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 Description

For implementation I chose to use Python, because it is a familiar language. First, I implemented 2 classes: Grammar and FiniteAutomatom. Below will be the code snippets and short explanation of their functionalities.

```
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

    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

        # Check if the grammar is unrestricted
        is_unrestricted = True

        # Determine the type of grammar
        if is_regular:
            return "Regular Grammar"
        elif is_context_free:
            return "Context-Free Grammar"
        elif is_context_sensitive:
            return "Context-Sensitive Grammar"
        else:
            return "Unrestricted Grammar"
```

This code is a Python method classify_grammar which takes two parameters: terminals and non_terminals. It aims to classify the grammar based on its type: regular, context-free, context-sensitive, or unrestricted.
It first transforms the grammar using transform_grammar(), although that method isn't shown here.
It then checks if the grammar is regular by examining each production. A grammar is regular if each production has at most one non-terminal on the right-hand side and, if there are two symbols on the right-hand side, the first one must be a non-terminal. If any production violates these rules, the grammar is not regular.
It then checks if the grammar is context-free by ensuring that each left-hand side of a production consists of exactly one non-terminal symbol.
It checks if the grammar is context-sensitive by comparing the lengths of the left-hand side and right-hand side of each production. A grammar is context-sensitive if the left-hand side is longer than or equal to the right-hand side for every production.
Finally, if none of the above conditions are met, it considers the grammar to be unrestricted.

```
class FiniteAutomatom:
    def __init__(self):
        self.Q = ['q0','q1','q2','q3','q4']
        self.Sigma = ['a','b']
        self.Delta = {
            ('q0', 'a') : ['q1'],
            ('q1', 'b') : ['q1'],
            ('q1', 'a') : ['q2'],
            ('q2', 'b') : ['q2', 'q3'],
            ('q3', 'b') : ['q4'],
            ('q3', 'a') : ['q1']
        }
        self.q0 = 'q0'
        self.F = ['q4']

    def convert_to_grammar(self):
        S = self.Q[0]
        V_n = self.Q
        V_t = self.Sigma
        P = []
        for state in self.Q:
            for symbol in self.Sigma:
                if (state, symbol) in self.Delta:
                    if len(self.Delta[(state, symbol)]) > 1:
                        for i in range(len(self.Delta[(state, symbol)])):
                            next_state = self.Delta[(state, symbol)][i]
                            P.append((state, symbol, next_state))
                    else:
                        next_state = self.Delta[(state, symbol)][0]
                        P.append((state, symbol, next_state))
        for final_state in self.F:
            P.append((final_state, '', 'e'))
        return Grammar(S, V_n, V_t, P)

    def checkDeterministic(self):
        for _, value in self.Delta.items():
            if(len(value))>1:
                return False
        return True
```

This code defines a class FiniteAutomaton representing a finite automaton. The finite automaton is defined by its states (Q), input alphabet (Sigma), transition function (Delta), initial state (q0), and set of final states (F).

The convert_to_grammar method converts the finite automaton to a grammar. It does so by defining the start symbol (S), non-terminal symbols (V_n), terminal symbols (V_t), and production rules (P). It iterates over each state and symbol, checks the corresponding transition in the transition function, and generates production rules accordingly. Additionally, it adds epsilon transitions to represent transitions to final states. Finally, it returns an instance of the Grammar class with the generated grammar.

The checkDeterministic method checks whether the finite automaton is deterministic by inspecting the transition function. If any state has multiple transitions on the same input symbol, the automaton is considered non-deterministic, and the method returns False. Otherwise, it returns True.

```
def NFAtoDFA(self) :
        input_symbols = self.Sigma
        initial_state = self.q0
        states = []
        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)

        for el in states:
            if self.F[0] in el:
                final_states.append(el)

        print(f"Q = {states}")
        print(f"Sigma = {input_symbols}")
        print(f"Delta = {transitions}")
        print(f"q0 = {initial_state}")
        print(f"F = {final_states}")


        dfa = DFA(
            states,
            input_symbols,
            transitions,
            initial_state,
            final_states
        )
        dfa.view("DFA")
```

This code defines a method NFAtoDFA within the class, which converts a Non-Deterministic Finite Automaton (NFA) to a Deterministic Finite Automaton (DFA). It achieves this by simulating the powerset construction algorithm.
It initializes variables such as input symbols (input_symbols), initial state (initial_state), and empty lists for states (states) and final states (final_states).
It creates an empty dictionary transitions to store the transition functions for the DFA.
It initializes a list new_states with the initial state (q0) of the NFA.
It enters a while loop that iterates as long as new_states is not empty.
Inside the loop, it iterates over each state in new_states, removing it from the list.
It checks if the current state is already in the transitions dictionary. If not, it adds it with an empty dictionary as the value.
It then simulates the NFA transitions for each input symbol. For each input symbol, it gathers all possible states that can be reached from the current state in the NFA and updates the transition function accordingly.
It trims any trailing commas from the concatenated states and removes empty strings from the secondary dictionaries.
After processing all states in new_states, it updates new_states with the newly generated states.
Once all states have been processed, it populates the states list with all states from the transitions

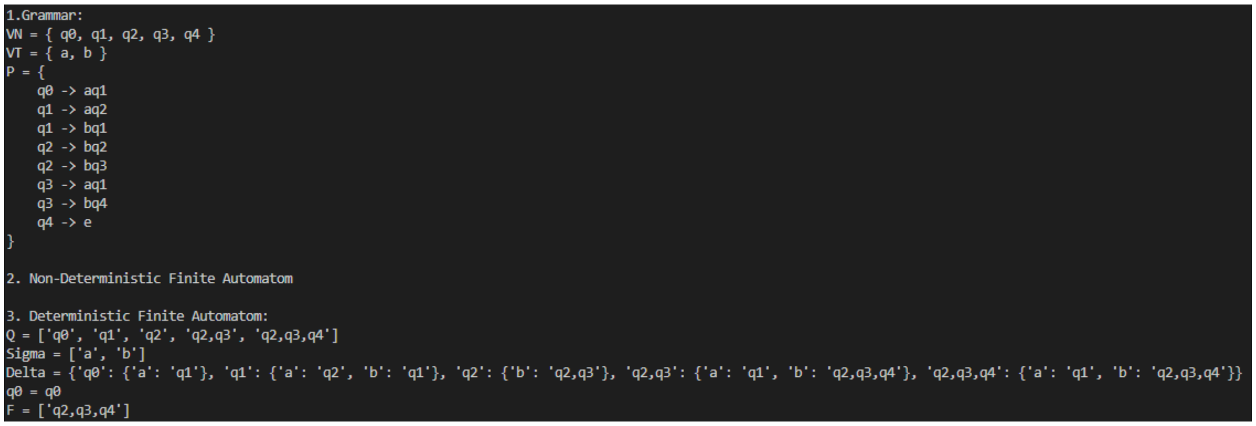
dictionary and identifies final states based on whether they contain any final state of the NFA. Finally, it prints the resulting DFA's states, input symbols, transitions, initial state, and final states. It then creates an instance of the DFA class with these parameters and calls its view method to visualize the DFA.

```
#main
finiteAutomatom = FiniteAutomatom()
grammar = finiteAutomatom.convert_to_grammar()
print("1.Grammar:")
grammar.show_gramamr()

print()

if finiteAutomatom.checkDeterministic()==False:
    print('2. Non-Deterministic Finite Automatom\n')
else:
    print('2. Deterministic Finite Automatom\n')

print("3. Deterministic Finite Automatom:")
finiteAutomatom.NFAtoDFA()

#NFA to compare graphicly with DFA
NFA({'q0','q1','q2','q3','q4'}, {'a','b'},
    {'q0' : {'a' : {'q1'}},
     'q1' : {'b' : {'q1'}, 'a' : {'q2'}},
     'q2' : {'b' : {'q2', 'q3'}},
     'q3' : {'b' : {'q4'}, 'a' : {'q1'}}},
     'q0', {'q4'}).view("NFA")
```

This code appears to be the main part of your program. It performs the following tasks:
It creates an instance of the FiniteAutomaton class named finiteAutomaton.
It converts the finite automaton to a grammar using the convert_to_grammar method and displays the grammar.
It checks if the finite automaton is deterministic using the checkDeterministic method and prints a message accordingly.
It converts the non-deterministic finite automaton to a deterministic finite automaton using the NFAtoDFA method and displays the resulting DFA.

Finally, it creates an instance of the NFA class, representing the original NFA, and visualizes it graphically using the view method.

Screenshots



```
1.Grammar:
VN = { q0, q1, q2, q3, q4 }
VT = { a, b }
P = {
    q0 -> aq1
    q1 -> aq2
    q1 -> bq1
    q2 -> bq2
    q2 -> bq3
    q3 -> aq1
    q3 -> bq4
    q4 -> e
}

2. Non-Deterministic Finite Automatom

3. Deterministic Finite Automatom:
Q = ['q0', 'q1', 'q2', 'q2,q3', 'q2,q3,q4']
Sigma = ['a', 'b']
Delta = {'q0': {'a': 'q1'}, 'q1': {'a': 'q2', 'b': 'q1'}, 'q2': {'b': 'q2,q3'}, 'q2,q3': {'a': 'q1', 'b': 'q2,q3,q4'}, 'q2,q3,q4': {'a': 'q1', 'b': 'q2,q3,q4'}}
q0 = q0
F = ['q2,q3,q4']
```

Figure 1: Results of testing

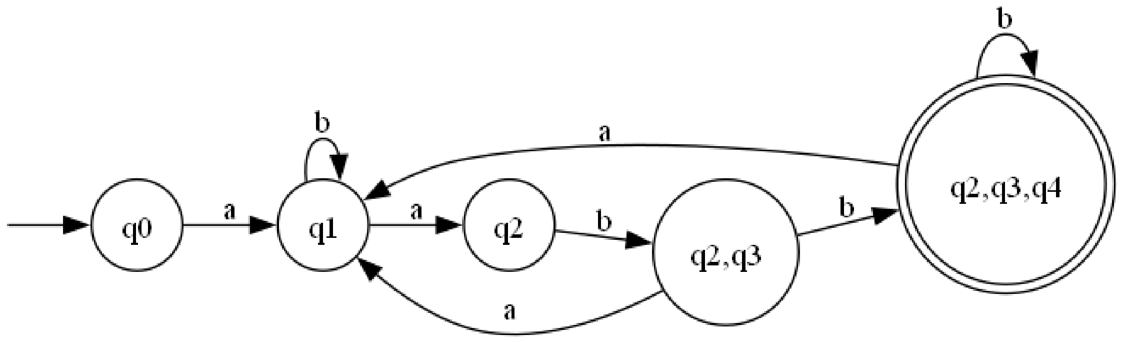


Figure 2: Graph for DFA

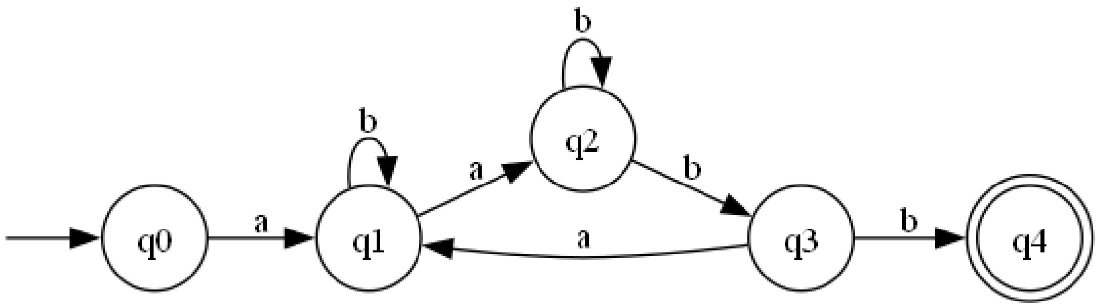


Figure 3: Graph for NFA

Conclusions

In conclusion, this lab has provided a comprehensive exploration of finite automata, grammars, and their interrelationships. We started by defining a finite automaton class and demonstrated how to convert it into a grammar representation. Through this conversion process, we gained insight into the structure and behavior of grammars corresponding to finite automata.

Furthermore, we explored deterministic and non-deterministic finite automata, as well as their transformations. By implementing algorithms for converting non-deterministic finite automata to deterministic finite automata, we illustrated the concept of determinism and its importance in automata theory.

Overall, this lab deepened our understanding of formal language theory and its practical applications. By analyzing and manipulating finite automata and grammars, we gained valuable insights into their theoretical underpinnings and practical implications. These concepts are fundamental in the study of computer science and play a crucial role in various areas such as compiler design, natural language processing, and algorithm development.