MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA

TECHNICAL UNIVERSITY OF MOLDOVA

FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Laboratory work 2:

Determinism in Finite Automata. Conversion from NDFA 2 DFA. Chomsky Hierarchy.

Course: Formal Languages & Finite Automata

Author: Bostan Victor, FAF-222

Chișinău, 2024

THEORY

Overview of Finite Automata:

A finite automaton (FA) is a theoretical model of computation used to simulate sequential logic and represent different types of processes. Similar to a state machine, a finite automaton consists of states, transitions between those states, and a set of inputs that trigger these transitions. The term "finite" reflects the fact that the automaton operates within a limited or bounded context, having a clear starting point and a set of possible end states. Automata are utilized in various fields such as computer science, linguistics, and engineering to model systems and processes ranging from software applications to biological networks.

Finite automata can be deterministic (DFA) or non-deterministic (NFA). In a DFA, for a given state and input, the machine transitions to exactly one next state, making the system's future behavior completely predictable. Conversely, an NFA allows multiple possible next states for a given state and input, introducing a level of unpredictability or non-determinism. Despite this difference, any NFA can be transformed into an equivalent DFA using well-established algorithms, thereby converting a non-deterministic process into a deterministic one, which simplifies analysis and implementation.

Chomsky Hierarchy:

The Chomsky hierarchy classifies grammars into four types based on their generative power and complexity. This hierarchy is foundational in the study of formal languages and automata theory, providing a framework for understanding the capabilities and limitations of different computational models:

1. Type 0 (Recursively Enumerable Grammars): The most general class, capable of expressing any language that can be recognized by a Turing machine. These grammars have no restrictions on their production rules.
2. Type 1 (Context-Sensitive Grammars): These grammars can generate languages that require the context of a symbol for its proper substitution rules. They are more restrictive than Type 0 but can still describe a wide array of languages, including all those that can be recognized by a linear-bounded automaton.
3. Type 2 (Context-Free Grammars): These grammars generate context-free languages, which are pivotal in programming language design and compiler construction. They allow for a non-terminal to be replaced by a string of terminals and non-terminals irrespective of context.
4. Type 3 (Regular Grammars): The most restrictive, these grammars correspond to regular languages, which can be recognized by finite automata. Regular grammars provide the foundation for regular expressions, a powerful tool in pattern matching and text processing.

Understanding and applying the Chomsky hierarchy enables the classification of languages and automata based on their complexity and the context required for their grammar rules, aiding in the systematic study of computational systems and languages.

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.
    a. 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

Building on the first laboratory the classes have remained the same. I have a Grammar and a FiniteAutomaton class. The only change I made was to create „generic" classes for grammar and finite automaton in seperate files, with the constructor and a print function, so I don't have to redefine them for every laboratory:

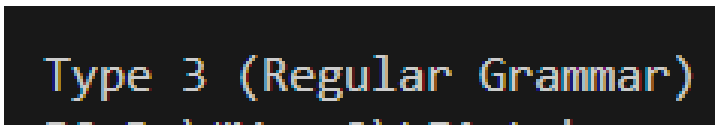
```
3   class GenericGrammar:
4       def __init__(self, vn, vt, p, s):
5           self.Vn = vn
6           self.Vt = vt
7           self.P = p
8           self.S = s
9
10      def print_grammar(self):
11          print('Vn:', self.Vn)
12          print('Vt:', self.Vt)
13          print('P: ', self.P)
14          print('S: ', self.S, '\n')
15
```

```python
class GenericFiniteAutomaton:
    def __init__(self, states, alphabet, transition_function, start, accept):
        self.states = states
        self.alphabet = alphabet
        self.transition_function = transition_function
        self.start = start
        self.accept = accept

    def print_automaton(self):
        print('States:', self.states)
        print('Alphabet:', self.alphabet)
        print('Transition function:', self.transition_function)
        print('Start state:', self.start)
        print('Accept state:', self.accept, '\n')

```

For the first task in this laboratory I had to add a method to de Grammar class that would classify the grammar based on Chomsky hierarchy. So, I created the *classify* method that would do just that:

```python
def classify(self):
    is_regular = True
    is_context_free = True
    is_context_sensitive = True

    for lhs, rhs_list in self.P.items():
        for rhs in rhs_list:
            # Check for Type 3 (Regular Grammar)
            if not (len(rhs) == 1 and rhs in self.Vt) and not (len(rhs) == 2 and rhs[0] in self.Vt and rhs[1] in self.Vn):
                is_regular = False
            # Check for Type 2 (Context-Free Grammar)
            if len(lhs) != 1 or not lhs.isupper():
                is_context_free = False
            # Check for Type 1 (Context-Sensitive Grammar)
            if len(rhs) < len(lhs):
                is_context_sensitive = False

    if is_regular:
        return "Type 3 (Regular Grammar)"
    elif is_context_free:
        return "Type 2 (Context-Free Grammar)"
    elif is_context_sensitive:
        return "Type 1 (Context-Sensitive Grammar)"
    else:
        return "Type 0 (Unrestricted Grammar)"
```

The way this works, as you can see in the image, it first checks if the grammar is of Type 3, if not it checks if it's of Type 2, if we again get a negative result we check if it is of Type 1 and finally if it's not even of Type 1 then it is of Type 0. Here is the result I get when testing this method for my grammar from the first laboratory, which was a Regular Grammar:



The next task was to implement the functionality to convert a Finite Automata into it's equivalent Grammar. I implemented the algorithm given at the lectures:

$$G = (V_N, V_T, P, S) \mid FA = (Q, \Sigma, \sigma, q_0, F)$$

1) $V_N = Q$
2) $V_T = \Sigma$
3) If we have $\sigma(q_i, a) = \{q_k\}$, then $q_i \rightarrow aq_k$
4) Final states F. For all final states we add $q_{fin} \rightarrow \varepsilon$

Here is how it looks in code:

```
8       def fa_to_grammar(self):
9           vn = self.states
10          vt = self.alphabet
11          s = self.start
12          p = {state: [] for state in vn}
13
14          for nt, vals in self.transition_function.items():
15              for terminal, vals2 in vals.items():
16                  for i in vals2:
17                      if i == self.accept[0]:
18                          p[nt].append(terminal)
19                      production = terminal + i
20                      p[nt].append(production)
21
22          return GenericGrammar(vn, vt, p, s)
```

Now, for this laboratory we also have variants that we test the methods on. Here is mine, Variant 1:

```
4       Variant 1
5       Q = {q0,q1,q2,q3},
6       ∑ = {a,c,b},
7       F = {q2},
8       δ(q0,a) = q0,
9       δ(q0,a) = q1,
10      δ(q1,c) = q1,
11      δ(q1,b) = q2,
12      δ(q2,b) = q3,
13      δ(q3,a) = q1.
```

Now, what I needed to do was to create the driver code and execute the *fa_to_grammar* method on my finite automata. Note that in my code I made these substitutions: q0 → S, q1 → A, q2 → B, q3 → C; since it is easier to work with characters rather than strings for symbols:

```python
    states = ['S', 'A', 'B', 'C']
    alphabet = ['a', 'c', 'b']
    transition_function = {
        'S': {'a': ['S', 'A']},  # δ(S,a) = S, δ(S,a) = A
        'A': {'c': ['A'], 'b': ['B']},  # δ(A,c) = A, δ(A,b) = B
        'B': {'b': ['C']},  # δ(B,b) = C
        'C': {'a': ['A']}  # δ(C,a) = A
    }
    start = 'S'
    accept = ['B']

    FA = FiniteAutomaton(states, alphabet, transition_function, start, accept)
    FA.fa_to_grammar().print_grammar()
```

Here are the results, which proved to be correct:

```
Vn: ['S', 'A', 'B', 'C']
Vt: ['a', 'c', 'b']
P:  {'S': ['aS', 'aA'], 'A': ['cA', 'b', 'bB'], 'B': ['bC'], 'C': ['aA']}
S:  S
```

The next task was to implement a method that would check if the finite automata is an NFA or a DFA. Here is my solution:

```python
    def check_fa(self):
        is_dfa = True

        for _, transitions in self.transition_function.items():
            for _, list in transitions.items():
                if len(list) > 1:
                    is_dfa = False
                    break

        if is_dfa:
            print("\nThis is a DFA!\n")
        else:
            print("\nThis is a NFA!\n")
```

This particular implementation works for me because of the structure of the transition function in my code. For every state in the FA we have this structure: state: {character from alphabet: [state, state, ...]}. So basically, for every state we have a dictionary with every character that state can use, and for every character we have a list with the states that we can get to from our inital state. The point is that in order to check if our FA is a DFA or an NFA we would just have to check if there exists a list with more than one elements. If we have a list with more than one elements (more than one state that we can get to from our inital state), then our FA is an NFA, since this is the definition of a NFA. Here is the result of the method when calling it for my variant:

```
This is a NFA!
```

We can see that indeed we get the correct answer. The FA from my variant is a NFA since from q0 with a we can get to both q0 and q1, which is non-determinisc. The penultimate task was to implement a method that would convert a NFA to a DFA. I decided to implement the Analytic Algorithm given at the lectures:

1) $q_0 = [q0]$
2) $Q' = \{ [q0] \}$
3) $\sigma'([q0], a), \sigma'([q0], b), \ldots$; if we have a new state, add them to Q'; $\sigma'([q0], a) = [q1q2]$
4) $\sigma'([q1q2], a) = \sigma(q1, a) \cup \sigma(q2, a)$
5) The states that have the original final state are included in F'

Here is the implementation in code:

```
    def nfa_to_dfa(self):
        # Initialize the list of states to process and the set of all created DFA states
        initial_state = frozenset([self.start])
        states_to_process = [initial_state]
        dfa_states = {initial_state}
        # Initialize the DFA transition function and the set of DFA's accept states
        dfa_transitions = {}
        dfa_accept = set()

        while states_to_process:
            current_state = states_to_process.pop()
            dfa_transitions[current_state] = {}  # Create an entry in the DFA transition function for the current state

            for symbol in self.alphabet:
                # Find the union of NFA states reachable from the current NFA states under the current symbol
                next_state = frozenset(
                    sum(
                        [self.transition_function.get(nfa_state, {}).get(symbol, []) for nfa_state in current_state],
                        []
                    )
                )

                # Only add the transition if the next state is not empty
                if next_state:
                    dfa_transitions[current_state][symbol] = next_state

                    # If this set of states is new, add it to the states we need to process
                    if next_state not in dfa_states:
                        dfa_states.add(next_state)
                        states_to_process.append(next_state)

                    # If the new state includes any NFA accept states, add it to the DFA accept states
                    if next_state & set(self.accept):
                        dfa_accept.add(next_state)

        # Convert state sets to strings to make them more readable
        state_names = {state: ''.join(sorted(state)) for state in dfa_states}
        dfa_transitions_named = {
            state_names[state]: {symbol: state_names[next_state] for symbol, next_state in transitions.items()}
            for state, transitions in dfa_transitions.items()
        }
        dfa_states_named = set(state_names.values())
        dfa_accept_named = {state_names[state] for state in dfa_accept}
        dfa_start_named = state_names[initial_state]

        # Make the transitions lists
        for state in dfa_states_named:
            for key, val in dfa_transitions_named[state].items():
                dfa_transitions_named[state][key] = [val]

        # Return the new DFA
        return FiniteAutomaton(dfa_states_named, self.alphabet, dfa_transitions_named, dfa_start_named, dfa_accept_named)
```

Since it is a more extensive implementation I will explain step by step how it works.

Step 1: Initialization

Initial State Creation: The function starts by creating an initial DFA state from the start state of the NFA. This is done by wrapping the NFA's start state in a frozenset, making it immutable and usable as a dictionary key.

States Processing Setup: It initializes states_to_process, a list containing the initial state of the DFA, which will be used to keep track of which states need their transitions defined.

DFA States and Transitions Initialization: Sets dfa_states to hold all the unique states of the DFA (starting with the initial state) and dfa_transitions to hold the transition rules for the DFA.

DFA Accept States Initialization: Initializes dfa_accept, a set that will eventually contain all of the DFA's accept states.

Step 2: Processing Loop

The function enters a loop that continues until there are no more states left to process in states_to_process.

For each state being processed, it:

Initializes its entry in dfa_transitions to prepare for storing its transition rules.

Iterates over each symbol in the NFA's alphabet to determine the transitions from the current DFA state.

Step 3: Transition Calculation

For each symbol, the function computes the union of all NFA states that can be reached from any of the NFA states contained within the current DFA state when the symbol is applied. This is done using list comprehension and set operations.

If this union (the next_state) is not empty, it adds a transition from the current state to this new state under the symbol in dfa_transitions.

If this next_state hasn't been seen before (i.e., it's not in dfa_states), it adds it to both dfa_states and states_to_process for future processing.

Step 4: Accept States Identification

After determining next_state, the function checks if this set intersects with the NFA's set of accept states. If so, the corresponding DFA state is marked as an accept state by adding it to dfa_accept.

Step 5: State and Transition Naming

After processing all states, the function assigns each DFA state a unique name by joining sorted characters of each state's set (for readability and consistent naming).

It then updates dfa_transitions to reflect these new names, creating dfa_transitions_named with DFA states (now strings) as keys and updates the transitions to use these new names.

Step 6: Final Adjustments for DFA Transitions

The function iterates through all states and transitions in dfa_transitions_named, converting each transition mapping from a single state to a list containing that state. This is to maintain the format expected for DFA transitions, even though each list will only contain one state (since DFA transitions are deterministic).

Step 7: Return the New DFA

Finally, the function constructs and returns a new FiniteAutomaton instance representing the DFA, using the named states, original alphabet, renamed transitions, named start state, and determined accept states.

```
144     DFA = FA.nfa_to_dfa()
145     DFA.print_automaton()
146     DFA.check_fa()
```

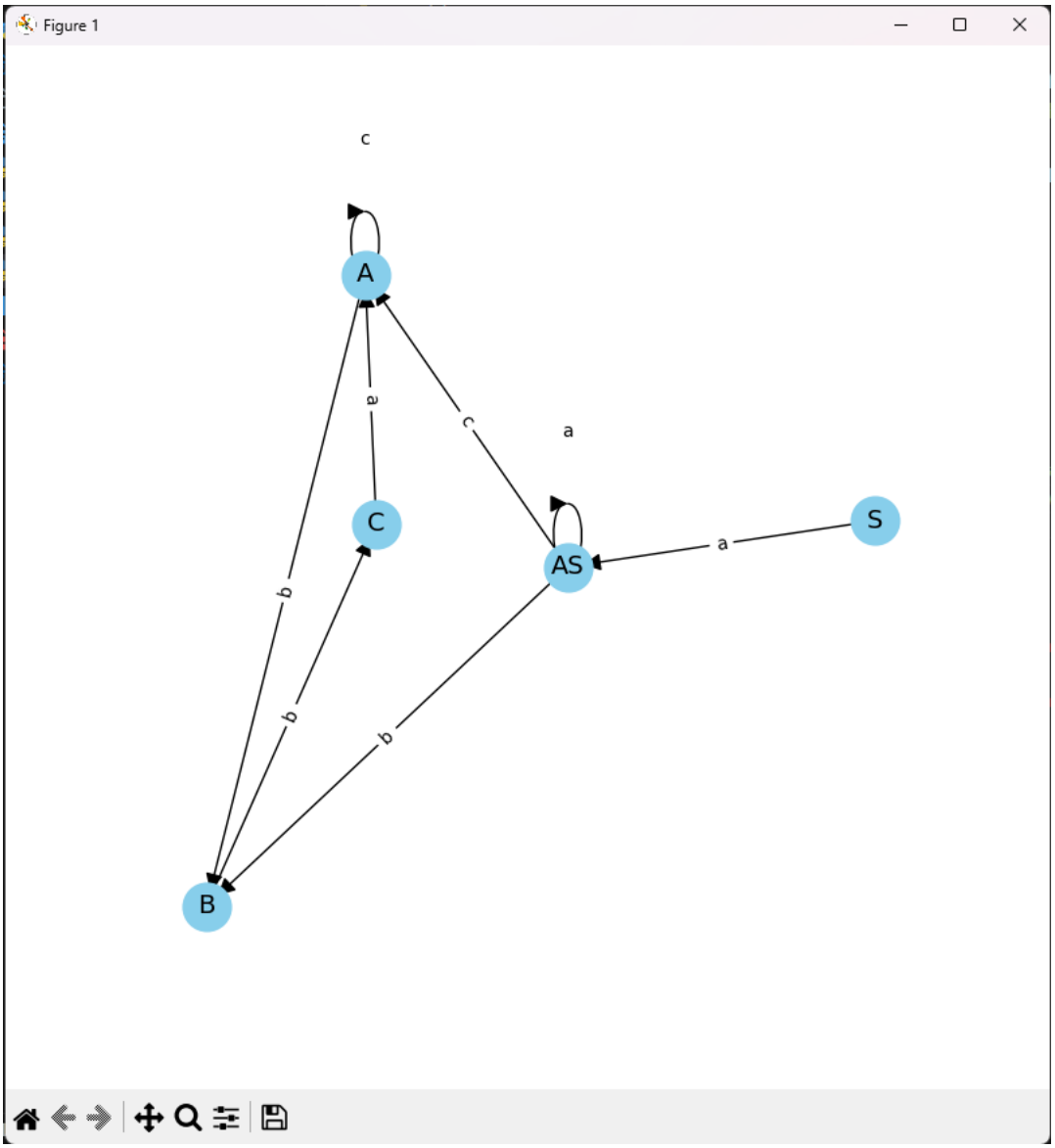
Here is the output I get when converting the NFA from variant 1 to a DFA. I also called the *check_fa* method described earlier to check if it is indeed a DFA:

```
States: {'A', 'AS', 'C', 'S', 'B'}
Alphabet: ['a', 'c', 'b']
Transition function: {'S': {'a': ['AS']}, 'AS': {'a': ['AS'], 'c': ['A'], 'b': ['B']}, 'B': {'b': ['C']
}, 'C': {'a': ['A']}, 'A': {'c': ['A'], 'b': ['B']}}
Start state: S
Accept state: {'B'}


This is a DFA!
```

Finally, the bonus task states that we have to create a method that would represent the FA graphically. Here is my solution:

```python
    def create_graph(self):
        G = nx.DiGraph()  # Directed graph for finite automaton

        # Add states as nodes
        for state in self.states:
            G.add_node(state)

        # Add transitions as edges
        for state, transitions in self.transition_function.items():
            for symbol, next_states in transitions.items():
                for next_state in next_states:
                    G.add_edge(state, next_state, label=symbol)

        # Graph layout
        pos = nx.spring_layout(G)  # positions for all nodes, spring layout

        # Drawing
        plt.figure(figsize=(8, 8))  # Increase figure size for better visibility
        nx.draw_networkx_nodes(G, pos, node_size=700, node_color='skyblue')
        nx.draw_networkx_edges(G, pos, arrowstyle='-|>', arrowsize=20)
        nx.draw_networkx_labels(G, pos, font_size=14)

        edge_labels = nx.get_edge_attributes(G, 'label')
        nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)

        # Manually adjust and draw labels for self-loops
        for node, (x, y) in pos.items():
            for neighbor in G.neighbors(node):
                if node == neighbor:  # This means we have a self-loop
                    label = edge_labels[(node, neighbor)]
                    # Adjust these values to move the label around the self-loop as needed
                    loop_label_pos = (x, y + 0.2)
                    plt.text(loop_label_pos[0], loop_label_pos[1], label, size=10, ha='center', va='center')

        plt.axis('off')  # Hide axes
        plt.show()  # Display the graph
```

For doing this I used two python libraries (networkx and matplotlib). Here networkx is used to construct and manage the structure of the finite automaton as a directed graph, including states (nodes) and transitions (edges). matplotlib, on the other hand, is employed to visually render this graph, providing a graphical representation of the automaton with customized layout, node and edge styles, and labels. Here is how the converted DFA looks when calling this method:



CONCLUSION

In conclusion, this laboratory exercise provided a comprehensive exploration into the theoretical and practical aspects of finite automata, highlighting the fundamental differences between deterministic and non-deterministic models. By implementing conversion algorithms from NFA to DFA and from finite automata to regular grammars, I gained a deeper understanding of the underlying mechanics and theoretical principles that govern these computational models. The use of Python libraries such as NetworkX and Matplotlib enabled me to visualize and better comprehend the structure and transitions of automata, thereby enhancing my ability to analyze and interpret their behavior. Overall, this laboratory has reinforced the importance of finite automata in the study of computer science, particularly in the contexts of language processing, compiler design, and algorithmic logic, offering a solid foundation for further exploration in the field of theoretical computer science.