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

Course: Formal Languages & Finite Automata

Author: Durbailo Daniel

Group: FAF-222

Variant: 9

Chisinau 2024

My variant (9):

```
94      Variant 9
95      Q = {q0,q1,q2,q3,q4},
96      Σ = {a,b,c},
97      F = {q4},
98      δ(q0,a) = q1,
99      δ(q1,b) = q2,
100     δ(q2,c) = q0,
101     δ(q1,b) = q3,
102     δ(q3,a) = q4,
103     δ(q3,b) = q0.
```

Theory

Finite automata theory is a branch of theoretical computer science and mathematics that deals with the study of finite-state machines (FSMs) or finite automata. These are abstract machines that can be in only a finite number of states at any given time. Finite automata are widely used in various fields, including computer science, linguistics, and electrical engineering, to model and analyze systems with discrete and finite states.

1.Finite Automaton Structure:

A finite automaton consists of a finite set of states, a set of transitions between these states, an input alphabet, a starting state, and a set of final states. The transitions define how the automaton moves from one state to another upon consuming input symbols from the input alphabet.

2. Determinism vs. Non-determinism:

A deterministic finite automaton (DFA) is one in which, for each state and input symbol, there is exactly one possible transition to the next state. In contrast, a non-deterministic finite automaton (NFA) allows for multiple possible transitions from a given state for a particular input symbol or a set of input symbols. This non-determinism can lead to ambiguity or uncertainty in the behavior of the automaton.

3. Determinism and Predictability:

In systems theory, determinism refers to the predictability of a system's behavior. A deterministic system will always produce the same output for a given input, making its behavior fully predictable. However, in non-deterministic systems such as NFAs, the behavior may not be entirely predictable due to the possibility of multiple transitions for the same input.

4. Achieving Determinism:

While NFAs may exhibit non-determinism, it is often desirable to convert them into equivalent DFAs for easier analysis and implementation. Determinization algorithms, such as the subset construction algorithm, can be applied to convert an NFA into a DFA. These algorithms systematically explore the possible states and

transitions of the NFA to construct an equivalent DFA that eliminates non-determinism.

5. Benefits of Determinism:

Deterministic finite automata are preferred in practical applications due to their simplicity and predictability. DFAs are easier to implement and analyze compared to NFAs. They have efficient algorithms for various tasks, such as pattern matching, lexical analysis, and parsing. Determinism simplifies reasoning about system behavior and facilitates optimization.

In summary, finite automata theory encompasses the study of finite-state machines, including deterministic and non-deterministic variants. While non-determinism may arise in certain automata due to multiple possible transitions, deterministic behavior is often desirable for predictability and ease of analysis. Determinization algorithms enable the conversion of non-deterministic automata into equivalent deterministic forms, offering practical benefits in various applications.

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.

Please consider that all elements of the task 3 can be done manually, writing a detailed report about how you've done the conversion and what changes have you introduced. In case if you'll be able to write a complete program that will take some finite automata and then convert it to the regular grammar - this will be **a good bonus point**.

Implementation:

1.BaseGrammar class:

```
class BaseGrammar:
    def __init__(self, non_terminals, terminals,
productions, start_symbol):
        self.NonTerminals = non_terminals
        self.Terminals = terminals
        self.Productions = productions
        self.StartSymbol = start_symbol

    def display_grammar(self):
        print('Non-terminals:', self.NonTerminals)
        print('Terminals:', self.Terminals)
        print('Productions:', self.Productions)
        print('Start Symbol:', self.StartSymbol,
'\n')

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

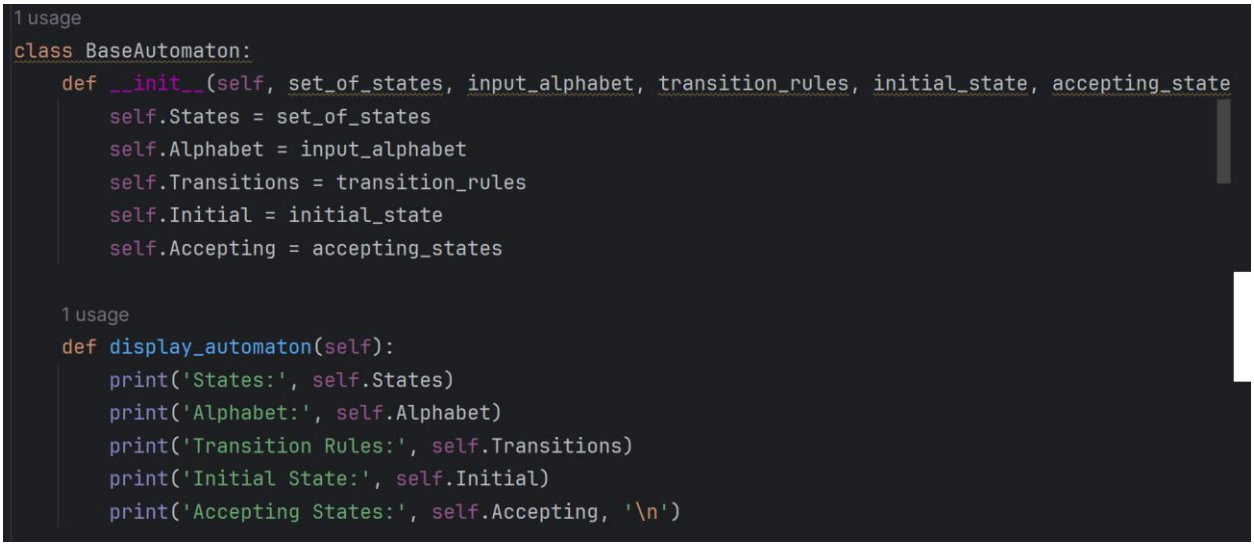
```
        is_context_sensitive = True

        for lhs, rhs_list in
self.Productions.items():
            for rhs in rhs_list:
                # Check for Type 3 (Regular
Grammar)
                if not (len(rhs) == 1 and rhs in
self.Terminals) and not (len(rhs) == 2 and rhs[0]
in self.Terminals and rhs[1] in self.NonTerminals):
                    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)"
```

1)__init__(self, non_terminals, terminals, productions, start_symbol): This constructor initializes a grammar object with its components such as non-terminals, terminals, productions, and the start symbol.

2)display_grammar(self): This method simply prints out the details of the grammar.

3)classify(self): This method classifies the grammar based on the structure of its productions. It iterates through each production rule and checks certain conditions to determine whether the grammar is regular, context-free, context-sensitive, or unrestricted

2.BaseAutomaton Class:



```
1 usage
class BaseAutomaton:
    def __init__(self, set_of_states, input_alphabet, transition_rules, initial_state, accepting_state
        self.States = set_of_states
        self.Alphabet = input_alphabet
        self.Transitions = transition_rules
        self.Initial = initial_state
        self.Accepting = accepting_states

    1 usage
    def display_automaton(self):
        print('States:', self.States)
        print('Alphabet:', self.Alphabet)
        print('Transition Rules:', self.Transitions)
        print('Initial State:', self.Initial)
        print('Accepting States:', self.Accepting, '\n')
```

Figure 1. BaseAutomaton Class:

1)__init__(self, set_of_states, input_alphabet, transition_rules, initial_state, accepting_states): This constructor initializes an automaton object with its components such as states, alphabet, transition rules, initial state, and accepting states.

2)display_automaton(self): This method prints out the details of the automaton.

3.Automaton Class (inherits from Base Automaton):

```python
class Automaton(BaseAutomaton):
    2 usages
    def convert_to_grammar(self):
        non_terminals = self.States
        terminals = self.Alphabet
        start_symbol = self.Initial
        productions = {state: [] for state in non_terminals}

        for state, trans in self.Transitions.items():
            for letter, next_states in trans.items():
                for next_state in next_states:
                    if next_state in self.Accepting:
                        productions[state].append(letter)
                    else:
                        production_rule = letter + next_state
                        productions[state].append(production_rule)

        return BaseGrammar(non_terminals, terminals, productions, start_symbol)
```

Figure 2. The method that converts the automaton to grammar

```python
def transform_nfa_to_dfa(self):
    initial = frozenset([self.Initial])
    unprocessed = [initial]
    dfa_states = {initial}
    dfa_transitions = {}
    dfa_finals = set()

    while unprocessed:
        current = unprocessed.pop()
        dfa_transitions[current] = {}

        for char in self.Alphabet:
            new_state = frozenset(
                sum([self.Transitions.get(state, {}).get(char, []) for state in current], [])
            )
            if new_state:
                dfa_transitions[current][char] = new_state
                if new_state not in dfa_states:
                    dfa_states.add(new_state)
                    unprocessed.append(new_state)
                if new_state & set(self.Accepting):
                    dfa_finals.add(new_state)
```

Figure 3. The method that converts the NFA to DFA

1)convert_to_grammar(self): This method converts the automaton into an equivalent grammar. It creates productions for each state based on transition rules. If a state leads to an accepting state, it generates a production that corresponds to the input symbol.

2)transform_nfa_to_dfa(self): This method transforms a non-deterministic finite automaton (NFA) into a deterministic finite automaton (DFA) using the subset construction algorithm. It iterates through each state of the NFA and computes the set of states reachable from it for each input symbol. These sets become the states of the DFA.

3)verify_automaton_type(self): This method verifies whether the automaton is deterministic (DFA) or non-deterministic (NFA) based on the transition rules. If any transition rule leads to multiple states for a given input symbol, the automaton is classified as non-deterministic.

4)visualize(self): This method visualizes the automaton using NetworkX and Matplotlib libraries. It creates a graphical representation of states, transitions, and the input alphabet.

3.Visual Results

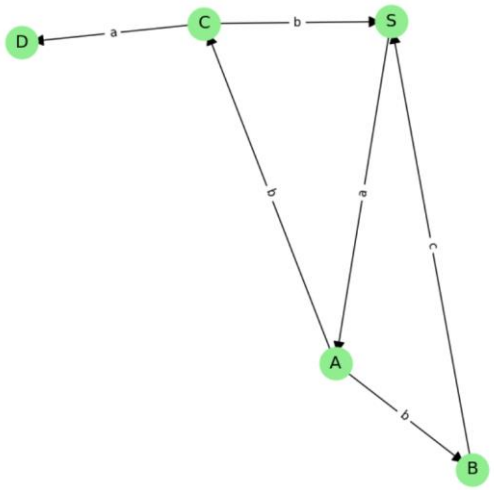


Figure 4. The diagram of the NFA

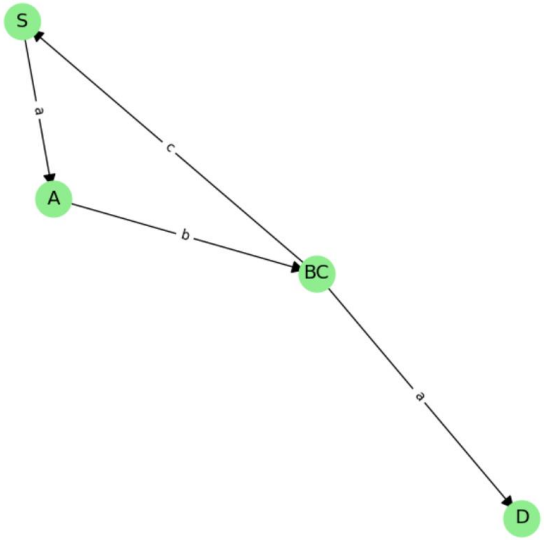


Figure 5. The DFA obtained from the NFA

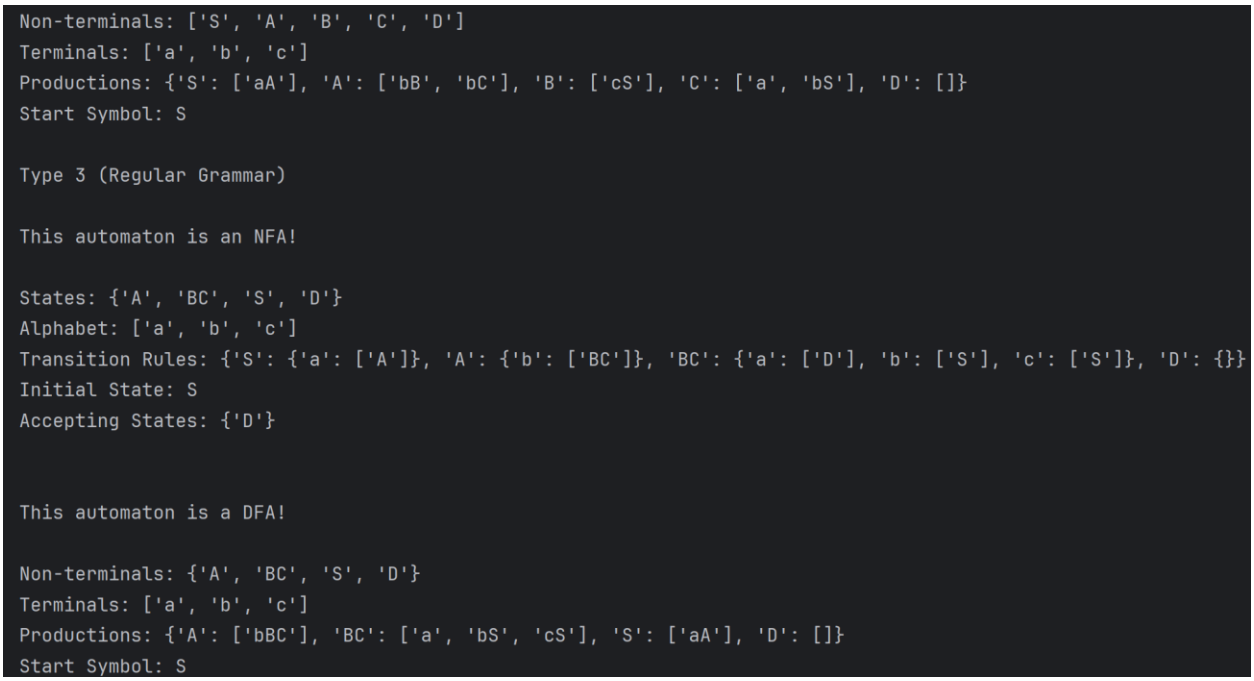


Figure 6. Information about the Grammar and Automaton

4.Conclusions:

Upon completing the laboratory work on finite automata and grammars, I've gained a deeper understanding of these fundamental concepts in computer science and mathematics. Through hands-on experience, I now comprehend the intricacies of finite automata and grammars, including their components, properties, and practical applications in modeling computational processes.

Implementing and manipulating finite automata and grammars programmatically has been an enriching experience. I've learned how to represent these abstract concepts in code and apply various algorithms to analyze and transform them. This practical aspect of the laboratory work has reinforced my understanding and enabled me to apply theoretical concepts in real-world scenarios.

One of the highlights of the laboratory work was learning conversion and transformation techniques. By converting NFAs to DFAs and classifying grammars based on their production rules, I've gained insights into the computational complexity and expressive power of different automata and grammars. Understanding the underlying algorithms involved in these processes has deepened my knowledge and enhanced my problem-solving skills.

Visualizing finite automata using NetworkX and Matplotlib libraries has been particularly enlightening. Seeing the structure and behavior of automata graphically has helped me analyze and communicate their properties more effectively. Visualization has proven to be a valuable tool in understanding complex concepts and reinforcing learning.

Overall, the laboratory work has fostered critical thinking and problem-solving skills. I've encountered challenges in understanding theoretical concepts, implementing algorithms, and interpreting results, which have contributed to my growth as a computer scientist. This experience has prepared me for further study and application in areas such as automata theory, formal languages, compiler design, and computational linguistics.