Laboratory Work Nr.2

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

Course: Formal Languages & Finite Automata

Author: Tofan Liviu

Group: FAF-223

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.

The Chomsky Hierarchy

The Chomsky hierarchy is a classification of formal languages based on their generative grammars. Developed by Noam Chomsky in the 1950s, this hierarchy divides languages into four levels, each with increasing expressive power. These levels are:

1. Type 0 (Recursively Enumerable Languages): The most general class, generated by unrestricted grammars. These languages can be recognized by a Turing machine, but not all can be decided (meaning that there is not always a Turing machine that can always halt and state whether a given string belongs to the language or not).

2. Type 1 (Context-Sensitive Languages): Generated by context-sensitive grammars, these languages can be recognized by a linear bounded automaton (a Turing machine with a tape length linearly bounded by the input size). Context-sensitive languages are more restricted than recursively enumerable languages and include all the languages that can be generated by grammars where the productions have contexts around the symbols being replaced.

3. Type 2 (Context-Free Languages): These languages are generated by context-free grammars, which are used to describe the syntax of programming languages and can be recognized by pushdown automata. The productions in a context-free grammar allow a single non-terminal symbol to be replaced by a string of terminal and/or non-terminal symbols, without any context considerations.

4. Type 3 (Regular Languages): The simplest class, generated by regular grammars, and recognizable by finite automata. Regular languages are those that can be generated by grammars that have productions with a single non-terminal symbol being replaced by a single terminal symbol, possibly followed by a single non-terminal symbol.

NFAs and DFAs

NFAs (Nondeterministic Finite Automata) and DFAs (Deterministic Finite Automata) are both types of finite automata used in the study of computer science and formal language theory. They are models of computation that define how inputs are processed and accepted or rejected based on the state transitions of the automaton. Despite serving similar purposes, NFAs and DFAs have distinct characteristics:

Deterministic Finite Automata (DFAs)

- *Deterministic:* In a DFA, for a given state and input symbol, the next possible state is uniquely determined. There is exactly one transition for every symbol of the alphabet in each state.

- *Structure:* A DFA consists of a finite set of states, a finite set of input symbols, a transition function, a start state, and a set of accept states.

- *Computation:* The DFA reads an input string symbol by symbol, and based on the current state and the input symbol, it moves to the next state according to its transition function. If the input ends in an accept state, the input is accepted; otherwise, it is rejected.

Nondeterministic Finite Automata (NFAs)

- *Nondeterministic:* In an NFA, for a given state and input symbol, there can be several possible next states. An NFA can choose any of these next moves; in other words, it can "guess" the right path to take. Additionally, NFAs can have ε-transitions.

- *Structure:* Similar to DFAs, NFAs are defined by a finite set of states, a finite set of input symbols, transition functions , a start state, and accept states.

- *Computation:* NFAs can move to several new states from a given state and symbol, or even without consuming a symbol. If any sequence of transitions leads to an accept state by the end of the input, the input is accepted.

- *Equivalence:* Despite their nondeterminism, NFAs are equivalent to DFAs in terms of the languages they can recognize every NFA has an equivalent DFA that recognizes the same language, although the resulting DFA may have exponentially more states.

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 ,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).

Implementation

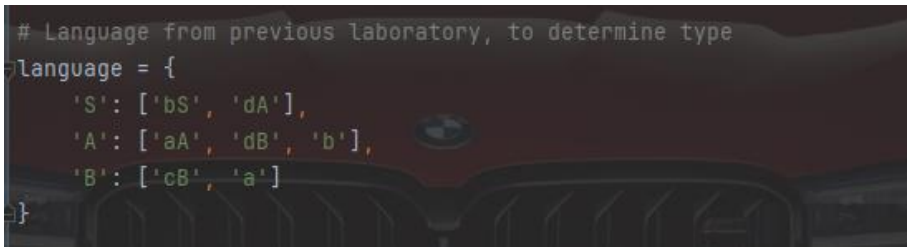
2. Incorporating the variant from the previous laboratory session, I have implemented the following function to classify grammars according to the Chomsky hierarchy:

```
class Grammar:
    new *
    def __init__(self, productions):
        self.P = productions

    1 usage  new *
    def classify_grammar(self):
        if self.is_regular():
            return "Type 3 (Regular)"
        elif self.is_context_free():
            return "Type 2 (Context-Free)"
        elif self.is_context_sensitive():
            return "Type 1 (Context-Sensitive)"
        else:
            return "Type 0 (Unrestricted)"
```

```
def is_regular(self):
    for lhs, productions in self.P.items():
        if len(lhs) > 1:
            return False
        for production in productions:
            if len(production) == 1 and production.islower():  # Terminal symbol
                continue
            elif len(production) == 2 and production[0].islower() and production[1].isupper():
                continue
            elif len(production) == 2 and production[1].islower() and production[0].isupper():
                continue
            else:
                return False
    return True

1 usage  new *
def is_context_free(self):
    for lhs, productions in self.P.items():
        if len(lhs) != 1 or not lhs.isupper():
            return False
    return True
```

```
1 usage  new *
def is_context_sensitive(self):
    s_on_right = False
    for left, productions in self.P.items():
        if 'S' in left:
            s_on_right = True
        for production in productions:
            # Check for S -> epsilon
            if left == 'S' and production == '' and not s_on_right:
                continue
            if len(left) > len(production):
                return False
    return True
```

Figure 1. Grammar Type according to the Chomsky hierarchy.



```
# Language from previous laboratory, to determine type
language = {
    'S': ['bS', 'dA'],
    'A': ['aA', 'dB', 'b'],
    'B': ['cB', 'a']
}
```

Figure 2. Variant from previous lab work.

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

Figure 3. Result type of grammar.

In this Python implementation, a Grammar class is introduced to facilitate the classification of grammars according to the Chomsky hierarchy. The class is initialized with a dictionary of productions, where keys represent non-terminal symbols and values are lists of production rules. The classify_grammar method serves as the core of the classification process. It iterates through different methods (is_regular, is_context_free, is_context_sensitive) to determine the type of grammar. The is_regular method checks if the grammar adheres to the rules of regular grammars (Type 3), ensuring that each production consists of either a single terminal symbol or a non-terminal followed by a single terminal. Next, the is_context_free method verifies if the grammar conforms to the structure of context-free grammars (Type 2), ensuring that each production rule has a single non-terminal symbol on the left-hand side. Lastly, the is_context_sensitive method examines whether the grammar meets the criteria of context-sensitive grammars (Type 1). It evaluates productions to ensure that the left-hand side is not shorter than the right-hand side, except in cases where the start symbol generates an empty string ('epsilon'). Overall, this Grammar class provides a systematic approach to determine the type of grammar based on the principles outlined by the Chomsky hierarchy.

3. a. Conversion of a finite automaton to a regular grammar:

```
def fa_to_regular_grammar(variant):
    grammar = {}

    states = variant["Q"]
    transitions = variant["delta"]
    alphabet = variant["Sigma"]

    for state in states:
        rules = []
        for symbol in alphabet:
            for transition in transitions:
                if state in transition and symbol in transition[state]:
                    destination = transition[state][symbol]
                    rules.append(f'{symbol}{destination}')
        grammar[state] = rules

    return grammar
```

Figure 4. Conversion of Finite Automaton to Regular Grammar.

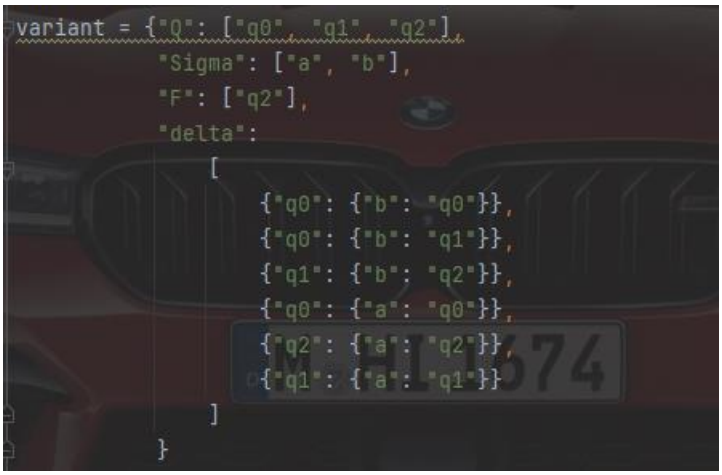


Figure 5. Variant 24.

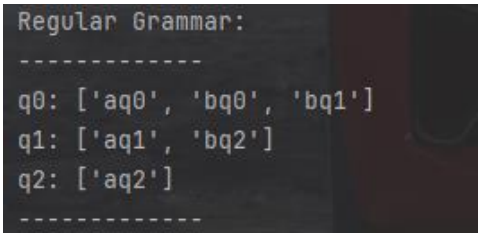


Figure 6. Result, Regular Grammar.

This function takes a finite automaton (FA) variant as input. It then iterates through each state of the automaton and generates production rules for a regular grammar based on transitions from each state to other states via symbols from the alphabet. For each state, it creates rules where symbols from the alphabet are followed by the destination states reachable from that state via transitions. Finally, it returns the generated grammar.

b. Determine whether your FA is deterministic or non-deterministic.

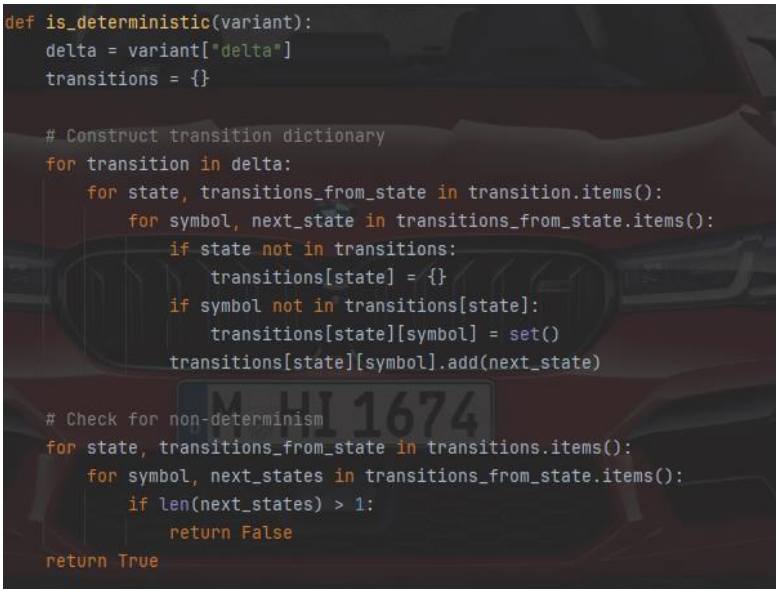


Figure 7. Function to check if FA is deterministic or not.

Figure 8. Result non-deterministic FA.

This is_deterministic() function is designed to determine whether a given finite automaton (FA), represented by a variant, is deterministic. It achieves this by analyzing the transitions of the automaton. First, it constructs a transition dictionary from the automaton's transition function, organizing transitions by states and symbols. Then, it iterates through this dictionary to check if any state has multiple transitions for the same input symbol, which would indicate non-determinism. If such a case is found, the function returns False, indicating that the automaton is non-deterministic. Otherwise, if all transitions are deterministic, it returns True.

c. Convert an NDFA to a DFA.

```
def ndfa_to_dfa(variant):
    dfa = {}

    dfa["input_symbols"] = variant["Sigma"]
    unprocessed_states = [frozenset([variant["Q"][0]])]
    dfa["start_state"] = frozenset([variant["Q"][0]])
    dfa["states"] = set()
    dfa["transitions"] = {}
    dfa["accept_states"] = set()

    while unprocessed_states:
        current_state = unprocessed_states.pop()
        if current_state not in dfa["states"]:
            dfa["states"].add(current_state)
            for input_symbol in dfa["input_symbols"]:
                next_state = frozenset(
                    [s for state in current_state for s in
                     [t[state][input_symbol] for t in variant["delta"] if state in t and input_symbol in t[state]]])
                if next_state:
                    dfa["transitions"][current_state, input_symbol] = next_state
                    unprocessed_states.append(next_state)

    dfa["accept_states"] = {state for state in dfa["states"] if variant["F"][0] in state}

    return dfa
```

Figure 9. Function to convert NDFA to DFA.

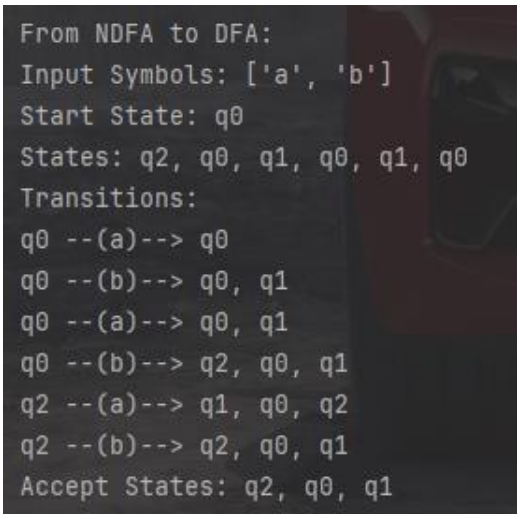


Figure 10. DFA representation.

The ndfa_to_dfa() function is designed to convert a non-deterministic finite automaton (NFA), represented by a given variant, into an equivalent deterministic finite automaton (DFA). It initializes a DFA structure and processes states iteratively to determine transitions. The function begins by setting up the DFA with input symbols, start state, and empty sets for states, transitions, and accept states. It then iterates through unprocessed states, constructing DFA transitions based on NFA transitions for each input symbol. The resulting DFA includes states reachable from the start state through deterministic transitions. The function continues until all states are processed, ensuring all possible state combinations are explored. Accept states in the DFA are identified based on whether they contain any of the NFA's accept states. Finally, the function returns the constructed DFA. In essence, this function provides a systematic approach to converting an NFA to an equivalent DFA, facilitating deterministic behavior for automata processing.

d. Represent the finite automaton graphically

Figure 11. Function to represent graphically FA.

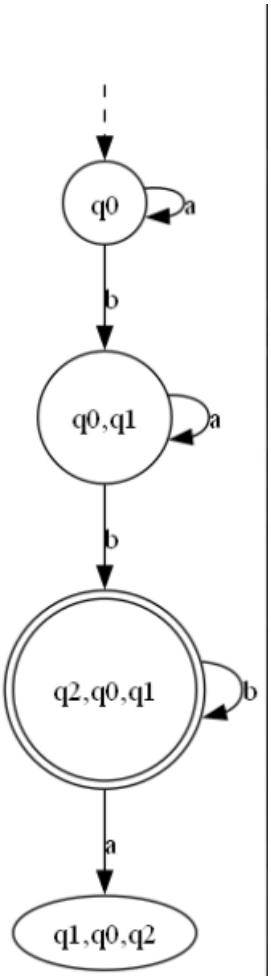


Figure 12. FA graphically representation using graphviz.

The draw_dfa function utilizes the Graphviz library to generate a graphical representation (diagram) of a deterministic finite automaton (DFA) provided as input. It constructs a directed graph where nodes represent states of the DFA and edges represent transitions between states based on input symbols. The function iterates through the DFA's states and transitions, adding nodes with appropriate shapes (circles for regular states and double circles for accept states) and edges with labels indicating input symbols. Additionally, it handles the start state by adding an invisible node and a dashed edge to denote the start state visually.

Conclusion

Through this lab, I delved into the foundational concepts of the Chomsky hierarchy and explored the intricacies of converting nondeterministic finite automata (NFAs) to deterministic finite automata (DFAs). This journey provided valuable insights into the structural disparities and computational implications inherent in different classes of formal languages and automata within the Chomsky hierarchy. The transition from nondeterminism to determinism, particularly in the context of finite automata, emerged as a focal point of my study. I underscored the practical significance of regular languages and finite automata, elucidating their pivotal roles in various computational processes and algorithms. The conversion of NFAs to DFAs emerged as a crucial aspect of computational theory and applications. While NFAs offer flexibility and intuitive design, DFAs are indispensable for implementation due to their deterministic nature, ensuring a unique computation path for any input string. The subset construction algorithm, exemplified in this lab, showcased a systematic approach to this conversion, despite the potential exponential increase in state count to maintain determinism. In essence, this exploration of the Chomsky hierarchy and the NFA to DFA conversion process shed light on the delicate balance between expressiveness and computational efficiency in computational model design. This lab not only deepened my understanding of the theoretical underpinnings of computer science but also furnished me with practical skills and insights applicable across diverse domains, from software development to the analysis of intricate systems.