Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Course: Formal Languages & Finite Automata

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

Elaborated:

st. gr. FAF-221 Lupan Lucian

Verified:

asist. univ. Dumitru Crețu

Chişinău – 2024

Theory

A finite automaton is a simple machine that can recognize patterns. It can be used to define a regular language (i.e., a language with regular grammar). It takes a string of symbols as input characters and changes its state accordingly, and when it finds the required symbol, it transitions to a new state. It has 2 states – accept and reject – when a string is processed successfully in its final phase, it's accepted, otherwise the finite automaton rejects it.

A finite automaton is a collection of 5-tuple (Q, ∑, δ, q0, F), where:

- Q: finite set of states
- ∑: finite set of the input symbol
- q0: initial state
- F: final state
- δ: Transition function

Grammar is used to create patterns that a finite automaton may recognize. Grammar consists of:

- Non-terminals – symbols that signify intermediate transitions.
- Terminals – symbols that are used to form the words of the language.
- Production – set of rules that define possible transitions.

According to Chomsky hierarchy, grammar is divided into 4 types as follows:

- Type 0 is known as unrestricted grammar.
- Type 1 is known as context-sensitive grammar.
- Type 2 is known as context-free grammar.
- Type 3 Regular Grammar.

Finite automatons diverge into 2 types: DFAs (Deterministic Finite Automata) and NFAS (Non-deterministic Finite Automata). The difference between the 2 is in the way transitions are defined – NFAs can transition to 2 different states with the same input; whereas DFAs only have 1 transition defined for each unique input in a set state.

Converting a NFA to a DFA can be done in 6 steps:

- Step 1: Convert the given NFA to its equivalent transition table.
- Step 2: Create the DFA's start state.
- Step 3: Create the DFA's transition table.
- Step 4: Create the DFA's final states.
- Step 5: Simplify the DFA.
- Step 6: Repeat steps 3-5 until no further simplification is possible.

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.

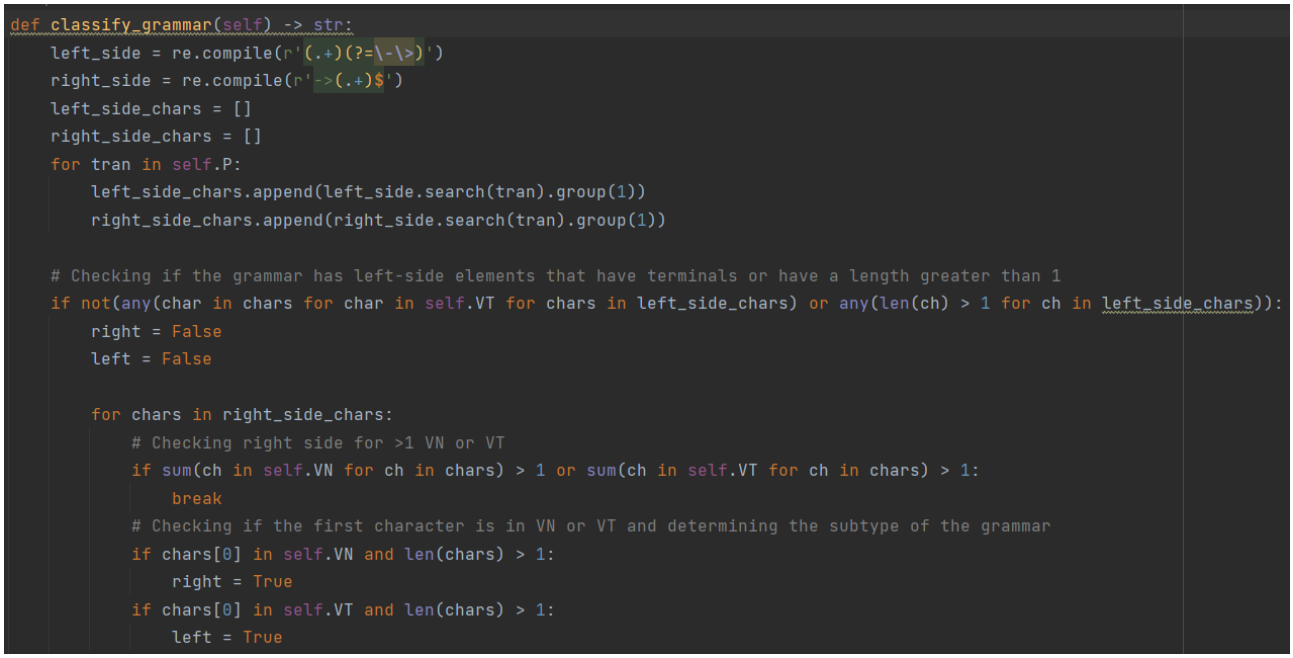
c. Implement some functionality that would convert an NDFA to a DFA.

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 this laboratory work, like the previous one, I used Python. Starting off with the 2nd objective, I decided to implement the grammar classification function inside my previous lab work files. To do this, we simply implement a new method that will take no input, but instead use the attributes of the current object.



```
def classify_grammar(self) -> str:
    left_side = re.compile(r'(.+)(?=\-\>)')
    right_side = re.compile(r'->(.+)$')
    left_side_chars = []
    right_side_chars = []
    for tran in self.P:
        left_side_chars.append(left_side.search(tran).group(1))
        right_side_chars.append(right_side.search(tran).group(1))

    # Checking if the grammar has left-side elements that have terminals or have a length greater than 1
    if not(any(char in chars for char in self.VT for chars in left_side_chars) or any(len(ch) > 1 for ch in left_side_chars)):
        right = False
        left = False

        for chars in right_side_chars:
            # Checking right side for >1 VN or VT
            if sum(ch in self.VN for ch in chars) > 1 or sum(ch in self.VT for ch in chars) > 1:
                break
            # Checking if the first character is in VN or VT and determining the subtype of the grammar
            if chars[0] in self.VN and len(chars) > 1:
                right = True
            if chars[0] in self.VT and len(chars) > 1:
                left = True
```

To start off, we define some regex and get the left side and right side characters for the transitions. Afterwards, we check if any characters inside the left_side_chars, representing the characters that hold the non-terminals we transition from, have a length greater than 1. If that's not the case, we start by checking the right_side_chars, which represent the characters we're transitioning into. If we transition into more than 1 non-terminal or terminal, we break out, as this would be neither a right or left regular grammar. After that, we check if the first character is a non-terminal, meaning we have a right regular grammar, otherwise if it's a terminal, then it's left regular grammar.

```
    # XOR to determine if we have left or right grammar
    if left ^ right:
        return("Type 3")
# If not, then it's a combination of the two, meaning it's type two grammar
    else:
        return("Type 2")
# Type 1 grammar cannot have terminal symbols on the left and can't have an empty string on the right
elif " " not in right_side_chars and not(any(char in chars for char in self.VT for chars in left_side_chars)):
    return("Type 1")
# Otherwise, it's just type 0
else:
    return("Type 0")
```

After finishing the loop, we check if we either have a left regular grammar or a right regular grammar using a XOR, and if it returns true – we have regular grammar. Otherwise, it's type 2. If we didn't enter the initial loop, we check for empty transitions in right_side_chars and terminal characters transitions on the left_side_chars, as type 1 grammar doesn't allow them, and if the logic statement returns true, then it's a type 1 grammar, otherwise – it's type 0.

```
def is_deterministic(self) -> bool:
    for state, transitions in self.Delta.items():
        symbols = []

        for transition in transitions:
            symbols.append(transition[0])
        if len(symbols) != len(set(symbols)):
            return False

    return True

 Lupan
def grammar_conversion(self) -> Grammar:
    grammar = Grammar()

    grammar.VN = self.Q
    grammar.VT = self.Sigma
    grammar.P = self.Delta

    return grammar
```

For the third objective, I created a new class that had similar functionality to the previous laboratory work. Starting off, we made a simple grammar conversion, which return a grammar with

newly defined attributes. The second function determined whether the current object of Finite Automaton class is deterministic. This is easily achieved by checking every transition – if a symbol is seen at least 2 times, meaning the length of the symbols is different from the set of symbols in the transitions, then it's not deterministic, otherwise, if all transitions passed, it's a DFA.

For the sake of keeping the report short (and not having to add another 4 images whole images solely describing the ndfa to dfa conversion), I will only summarize and not go in depth about the ndfa conversion. If you require more details, visit the Github files and read the code comments.

```
def NFA_to_DFA(self):
    if self.is_deterministic():
        return
    dfa = FiniteAutomaton()

    transition_chars_pattern = re.compile(r'^(.*?)\s')
    transition_states_pattern = re.compile(r'\s(.*)$')

    transitions = {self.q0: self.group_states(self.Delta[self.q0])}

    finished_states = []
    last_state = ""

    while not(self.has_all_states(transitions)):

        transitions_copy = copy.deepcopy(transitions)
        for value in finished_states:
            del transitions_copy[value]

        for key, values in transitions_copy.items():
            symbols = []
            states = []
            # Getting all the symbols and transition states for the current state
            for value in values:
                symbols.append(transition_chars_pattern.search(value).group(1))
                states.append(transition_states_pattern.search(value).group(1))
```

We start off by checking if the current finite automaton is deterministic, and if it is – we simply return. Otherwise, we create an object of finite automaton and define regex to get the transition symbols and states. As our first state may be ambiguous, we start off by grouping the states (which checks for similar transition symbols and groups together all those states) and create a new dictionary with our initial state and those states. We also create a finished_states array, which is used to keep track of all the states and optimize the code, so we don't iterate through the states we've already added to our dictionary. We also keep track of the last state, to append it to the finished_states when we end the loop.
Beginning the loop, we check if the current dictionary has all the states that it has defined inside of its dictionary. If it's missing a state, we analyze those transitions. Thus, we start by creating a copy of the transition dictionary (as to avoid errors since we're modifying the transition dictionary) and get the symbols of the current state and transition in the dictionary, which are very important later on in determining the new states and new transitions for those states.

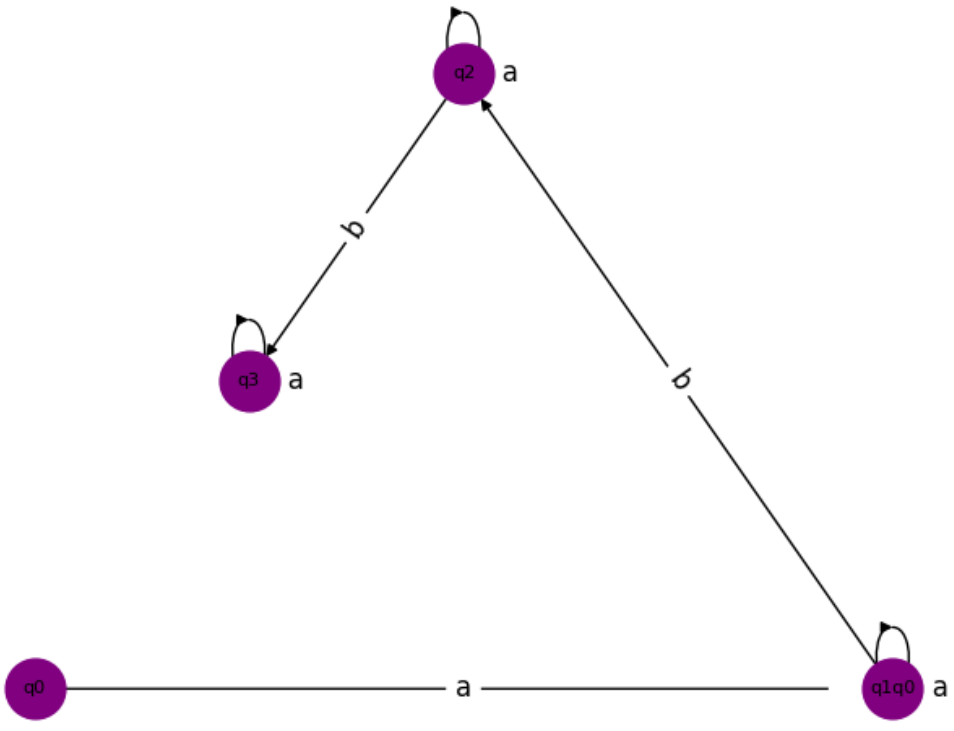
As mentioned before, I will not be going in depth about the code and will jump to the last part, which is the graphical representation.

```
def draw_automaton(self):
    graph = nx.DiGraph()

    # Setting the nodes and edges using our finite automaton's transition dictionary
    for node, edges in self.Delta.items():
        for edge in edges:
            label, target = edge.split(' ', 1)
            # Adding edges and labels
            graph.add_edge(node, target, label=label)
            # Modifying the label for the self loop by adding padding so that it's not drawn inside the node
            if node == target:
                label = "           " + label
                graph.add_edge(node, target, label=label)

    pos = nx.planar_layout(graph)
    labels = nx.get_edge_attributes(graph, 'label')
    # Drawing the graph
    nx.draw(graph, pos, with_labels=True, node_size=700, node_color="purple", font_size=8)
    nx.draw_networkx_edge_labels(graph, pos, edge_labels=labels, font_color='black', font_size=12)
    plt.show()
```

This is another method inside the finite automaton class, which utilizes the network and matplotlib.pyplot libraries to draw and display the graph. We start off by creating a digraph and defining edges for this graph. Using our newly defined transition inside the DFA object, we add the transition characters (label) and transition states (target) and add an edge. Also, if we have the case where the node is the target, then that means we're having a self-transition. As the label is displayed inside of the node for the self-transition, we have to add a buffer set of spaces to move it outside of the node. Afterwards, by using the draw function, we create the graph and add the labels to the edges. Below, you can visualize the final result:



Results

Input:

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

Output functions:

```
finite_automaton = FiniteAutomaton()
dfa = finite_automaton.NFA_to_DFA()
print(dfa.Delta)
print(dfa.F)
print(dfa.Q)
print(dfa.is_deterministic())
```

Output:

```
{'q0': ['a q1q0'], 'q1q0': ['a q1q0', 'b q2'], 'q2': ['a q2', 'b q3'], 'q3': ['a q3']}
['q3']
['q0', 'q1q0', 'q2', 'q3']
True
```

Conclusions

Much like the previous laboratory work, this lab taught me a lot about working with Python. Apart from learning about Python classes, I learnt how to work with dictionaries, lists, list comprehension, various functions, and even a new library – NetworkX.

Starting off, I didn't encounter any issues when dealing with the grammar classification, finite automaton to grammar conversion and finding out about whether the current finite automaton is deterministic or not. The 4th task, however, proved to be more challenging and time consuming. I had to go back and forth, trying out different approaches before the program finally worked for my variant. I (unfortunately) decided that it was not enough and wanted to make my program work for every (meaning 3 test cases because I hate testing) NFA. To achieve this, I once again had to modify my function, fighting against infinite loops using print statements until I finally found what caused my program to not behave the way it had to.

After I was finally done with the NFA conversion, I only had to graphically represent the NFA, which was an absolute breeze compared to the previous task. The only problem I encountered during this step was representing the labels for the self-transitions, but I managed to solve this problem by adding a buffer to the strings, offsetting them outside of the nodes.

Although this laboratory work was rather time-consuming, I didn't find it stressful, but rather enjoyable. Having to use logic for problem solving, aka finding a solution to the problem before you is much more interesting than simply writing code for a problem whose solution is obvious from the start.

References:

1. Chomsky Hierarchy in terms of computation. [online]. [accessed 03.03.2024]. Available: https://www.geeksforgeeks.org/chomsky-hierarchy-in-theory-of-computation/
2. Conversion from NFA to DFA. [online]. [accessed 03.03.2024]. Available: https://www.geeksforgeeks.org/conversion-from-nfa-to-dfa/