Regular Grammars

Course: Formal Languages & Finite Automata

Author: Dimbitchi Sergiu

Theory

1. Context-Free Grammars :
I think of CFGs like a set of rules for building sentences. You have a starting point (like a sentence you want to create), and you follow certain rules to put words together. These rules tell you how to combine different parts (like nouns, verbs, etc.) to make a valid sentence. It's kind of like playing with building blocks, where each block represents a word or a group of words, and the rules show you how to stack them together.

2. Finite Automata:
Imagine you have a little machine that reads symbols (like letters or numbers) one at a time and moves between different states based on what it reads. It's like a mini-game where you move through different levels by pressing buttons according to the symbols you see. The machine has a starting point, some rules for moving between states, and maybe some special places you want to reach (like winning points).

3. Relationship between CFGs and Finite Automata:
These CFGs and finite automata are like two sides of the same coin. CFGs are good at describing how to build sentences or strings, while finite automata are good at recognizing patterns in those strings. It's like having a recipe book (CFG) that tells you how to make different dishes, and a taste tester (finite automaton) that checks if a dish matches a certain recipe. They work together to understand and create languages in a computer-friendly way.

So, in this project, I made some rules for creating arithmetic expressions and then used those rules to make some valid expressions. Converting these rules into a machine that can recognize valid expressions is a bit trickier and involves more advanced concepts, but it's all about understanding how languages are built and understood by computers.

Objectives:

Understand what an automaton is and what it can be used for.

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.

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

a)

```python
# a) Convert FA to a Regular Grammar
def convert_fa_to_rg(states, transitions, accepting_states):
    grammar = {}
    for state in states:
        grammar[state] = []
        for (current_state, symbol), next_state in transitions.items():
            if current_state == state:
                grammar[state].append((symbol, next_state))
        if state in accepting_states:
            grammar[state].append(("ε",))
    return grammar


converted_grammar = convert_fa_to_rg(states, transitions, accepting_states)
readable_grammar = {}
for state, productions in converted_grammar.items():
    readable_grammar[state] = ["".join(prod) for prod in productions]

print("Regular Grammar:")
for state, rules in readable_grammar.items():
    for rule in rules:
        print(f"{state} -> {rule}")
```

The "convert_fa_to_rg" function transforms a finite automaton (FA) into a regular grammar (RG). It starts by creating a dictionary to represent the RG, where each state from the FA becomes a non-terminal symbol with corresponding production rules. The function then iterates through each state and its transitions. For each transition from a state, it adds a production rule to that state's list in the RG, combining the transition symbol and the destination state. If a state is an accepting state, it also adds a rule for producing the empty string, representing the end of a valid string in the language. Finally, the function returns the completed RG, mapping each state to its set of production rules.

The output :

```
Regular Grammar:
q0 -> aq1
q1 -> bq2
q2 -> bq3
q2 -> aq4
q3 -> aq1
q4 -> ε
```

b)

```python
# b) Determine if the FA is deterministic
def is_deterministic(transitions, states, alphabet):
    transition_count = {}
    for state in states:
        for symbol in alphabet:
            transition_count[(state, symbol)] = 0
    for (state, symbol), _ in transitions.items():
        transition_count[(state, symbol)] += 1
    return all(count <= 1 for count in transition_count.values())


is_dfa = is_deterministic(transitions, states, alphabet)

if is_dfa:
    print("\nThe FA is deterministic (DFA).")
else:
    print("\nThe FA is non-deterministic (NFA).")
```

The function takes three arguments: transitions, which is a dictionary mapping state-symbol pairs to states; states, a list of the FA's states; and alphabet, a list of symbols the FA can recognize.It initializes a dictionary transition_count to keep track of how many transitions exist for each state-symbol pair.The function then iterates through each state and symbol pair, initializing their count to zero in the transition_count dictionary, indicating that initially, there are no transitions defined for any state-symbol pair.Next, the function iterates through the transitions dictionary to count the actual transitions for each state-symbol pair, incrementing the relevant count in transition_count for each observed transition.After tallying the transitions, the function checks if the FA is deterministic: it returns True if every state-symbol pair has at most one transition (meaning each count in transition_count is less than or equal to one), indicating a DFA. Otherwise, it returns False, indicating an NFA.Outside the function, the is_deterministic function is called with the appropriate parameters, and the result is stored in the variable is_dfa.Finally, based on the value of is_dfa, the script prints whether the FA is deterministic (DFA) or non-deterministic (NFA).

The answer :

The FA is non-deterministic (NFA)

c)

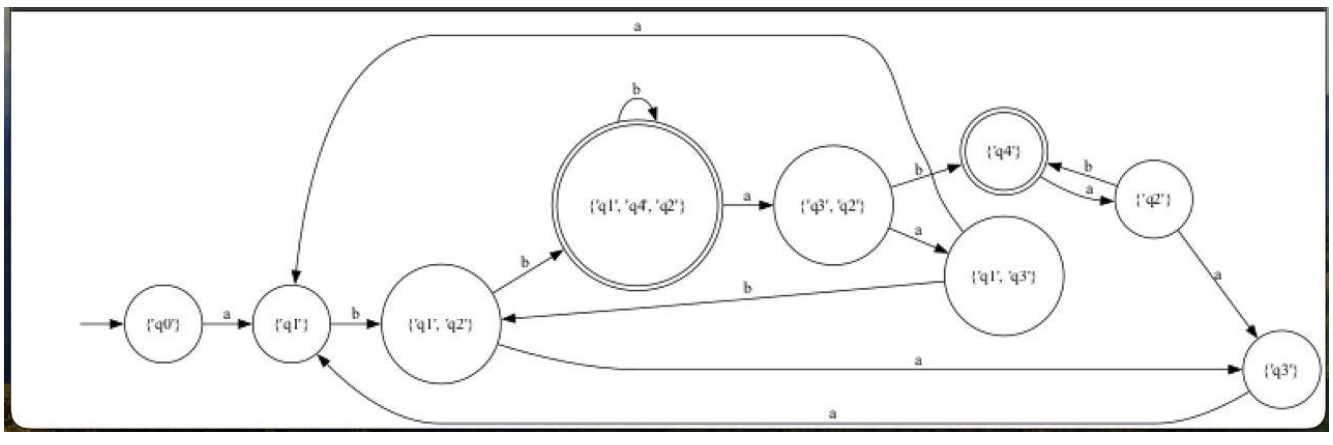
```python
# Exercise c) Convert NFA to DFA
nfa = {
    "states": ["q0", "q1", "q2", "q3", "q4"],
    "alphabet": ["a", "b"],
    "transitions": {
        "q0": {"a": ["q1"]},
        "q1": {"b": ["q1", "q2"]},
        "q2": {"a": ["q3"], "b": ["q4"]},
        "q3": {"a": ["q1"]},
        "q4": {"a": ["q2"]},
    },
    "start_state": "q0",
    "accept_states": ["q4"],
}
```

```python
def nfa_to_dfa(nfa):
    new_states_list = []
    dfa = {
        "states": [frozenset()],
        "alphabet": nfa["alphabet"],
        "transitions": defaultdict(dict),
        "start_state": frozenset([nfa["start_state"]]),
        "accept_states": set(),
    }
    states_queue = [frozenset([nfa["start_state"]])]
    dfa["transitions"][frozenset()][nfa["alphabet"][0]] = frozenset()
    dfa["transitions"][frozenset()][nfa["alphabet"][1]] = frozenset()
    while states_queue:
        current_state = states_queue.pop(0)
        if current_state not in new_states_list:
            new_states_list.append(current_state)
            for symbol in nfa["alphabet"]:
                next_state = frozenset(
                    sum(
                        [
                            nfa["transitions"].get(state, {}).get(symbol, [])
                            for state in current_state
                        ],
                        [],
                    )
                )
                if not next_state:
                    next_state = frozenset()
                dfa["transitions"][current_state][symbol] = next_state
                if next_state not in new_states_list and next_state:
                    states_queue.append(next_state)
        for new_state in new_states_list:
            if any(state in nfa["accept_states"] for state in new_state):
                dfa["accept_states"].add(new_state)
        dfa["states"].extend(new_states_list)
        return dfa


converted_dfa = nfa_to_dfa(nfa)
```

The function starts by creating an empty DFA structure. It initializes this with an empty set of states, inherits the alphabet from the NFA, and sets up a transitions table using a defaultdict. The start state of the DFA is the frozenset containing the NFA's start state, and it initializes an empty set for the accept states.
The function uses a queue (states_queue) to manage the exploration of new states, starting with the DFA's start state. It then enters a loop where it processes each state in the queue one by one.
For each state dequeued, the function checks if it has been encountered before; if not, it adds the state to the list of new states. Then, for each symbol in the NFA's alphabet, the function calculates the next state based on the transitions of all the NFA states contained in the current DFA state.
The next state for a given symbol is the union (frozenset) of all possible states the NFA could transition to from any of the states currently represented in the DFA state, considering the given symbol.
The new state (if not empty and not already processed) is added to the DFA's transitions table and queued for further exploration. This step effectively builds the DFA's transition function.
After all states have been explored, the function goes through the new states list. If any of these composite states contain an NFA accept state, that new state is marked as an accept state in the DFA.
Finally, the function updates the DFA's state list with all the new states discovered during the conversion and returns the complete DFA.
The conversion is executed by calling nfa_to_dfa with the NFA as the argument, and the resulting DFA is stored in converted_dfa.

The output:

```
DFA:
States: [set(), {'q0'}, {'q1'}, {'q1', 'q2'}, {'q3'}, {'q4', 'q1', 'q2'}, {'q3', 'q2'}, {'q1', 'q3'}, {'q4'}, {'q2'}]
Start State: {'q0'}
Accept States: [{'q4'}, {'q4', 'q1', 'q2'}]
Transitions:
    set() --a--> set()
    set() --b--> set()
    {'q0'} --a--> {'q1'}
    {'q0'} --b--> set()
    {'q1'} --a--> set()
    {'q1'} --b--> {'q1', 'q2'}
    {'q1', 'q2'} --a--> {'q3'}
    {'q1', 'q2'} --b--> {'q4', 'q1', 'q2'}
    {'q3'} --a--> {'q1'}
    {'q3'} --b--> set()
    {'q4', 'q1', 'q2'} --a--> {'q3', 'q2'}
    {'q4', 'q1', 'q2'} --b--> {'q4', 'q1', 'q2'}
    {'q3', 'q2'} --a--> {'q1', 'q3'}
    {'q3', 'q2'} --b--> {'q4'}
    {'q1', 'q3'} --a--> {'q1'}
    {'q1', 'q3'} --b--> {'q1', 'q2'}
    {'q4'} --a--> {'q2'}
    {'q4'} --b--> set()
    {'q2'} --a--> {'q3'}
    {'q2'} --b--> {'q4'}
```

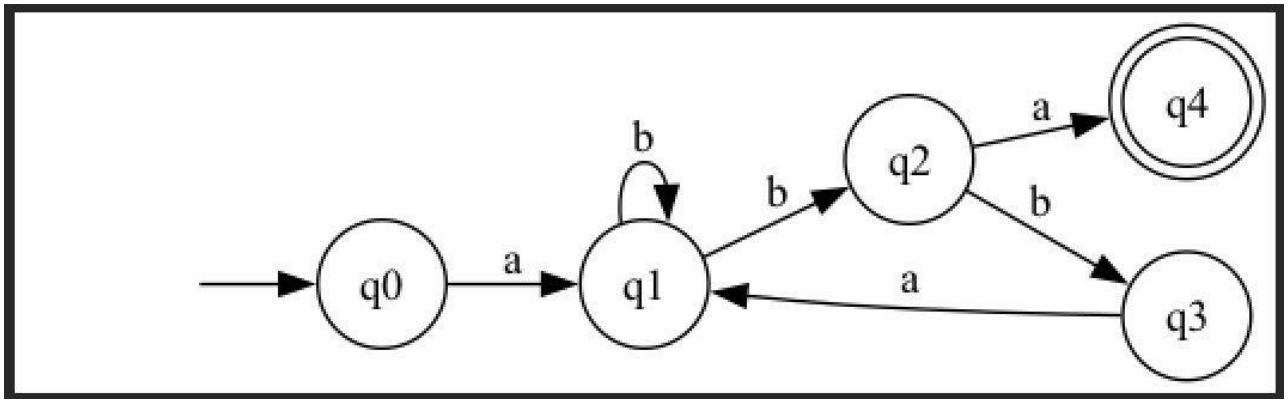


d)

```
# Exercise d) Visualize the FA using Graphviz
fa = Digraph("finite_automaton")
fa.attr(rankdir="LR")

# Add states to the graph
states = ["q0", "q1", "q2", "q3", "q4"]
for state in states:
    if state == "q4":
        fa.node(state, shape="doublecircle")
    else:
        fa.node(state, shape="circle")

fa.node("", shape="none")
fa.edge("", "q0")

transitions = [
    ("q0", "q1", "a"),
    ("q1", "q1", "b"),
    ("q1", "q2", "b"),
    ("q2", "q3", "b"),
    ("q2", "q4", "a"),
    ("q3", "q1", "a"),
]

for src, dst, label in transitions:
    fa.edge(src, dst, label=label)

fa.render("/Users/sergiu_sd/Desktop/finite_automaton", view=True, format="png")
```

The script begins by creating a new directed graph fa for the finite automaton using the Digraph class from Graphviz. It sets the direction of the graph from left to right.
It defines the states of the FA and adds them to the graph. The accepting state 'q4' is distinguished with a double circle by setting the shape attribute to "doublecircle", while all other states are represented as simple circles.
An invisible node (with shape set to "none") and an edge from this invisible node to the start state 'q0' are added. This is a common way to represent the starting point in state diagrams.
The transitions for the FA are defined in a list of tuples, where each tuple contains the source state, destination state, and the symbol that triggers the transition.
The script iterates through the list of transitions, adding edges to the graph for each one, labeling them with the respective symbols.
Finally, the script renders the graph to a PNG file and optionally opens it for viewing. The path where the graph is saved is specified, and the format is set to PNG.

The output:



Conclucions :

The laboratory work undertaken represents a comprehensive exploration into the realm of finite automata (FAs), crucial for understanding computational theories and applications in computer science. Initially, the work begins with an in-depth approach to converting finite automata (FA) into regular grammars (RG), showcasing the fundamental relationship between state machines and language generation, highlighting the practical applications in compiling and pattern recognition.

Through the development of a Python function, the lab delves into the structural nuances between deterministic (DFA) and non-deterministic finite automata (NFA), providing essential insights into their operational mechanisms. The function created to determine whether an FA is deterministic or not is pivotal, as it lays the groundwork for understanding the complexities and efficiencies in computational processes, reflecting on the deterministic nature of most computational systems and the conversion requirements for NFAs.

Moreover, the transformation of an NFA to a DFA, detailed through a well-structured algorithm, stands out as a critical learning exercise. This conversion is vital as DFAs are easier to implement and understand compared to NFAs, and this transformation process is instrumental in the fields of text processing and lexical analysis, where deterministic models are preferred for their simplicity and speed.

In addition, the work incorporates the Graphviz tool for visualization, enhancing comprehension and interpretation of abstract automaton concepts. This visualization is not just a supplementary element; it serves as a fundamental aspect of understanding and debugging automata, providing a clear and intuitive means of representing state transitions and automaton structure.

The lab also emphasizes the importance of the alphabet in automata theory, illustrating how variations in input symbols can lead to different states and highlighting the versatility and adaptability of finite automata in processing different languages. This aspect underlines the importance of automata in designing language parsers and interpreters.

Furthermore, the practical coding exercises supplemented theoretical knowledge with hands-on experience, bridging the gap between abstract concepts and real-world applications. The iterative development process, from defining states and transitions to applying algorithms for conversion and determinism checks, fosters a deeper understanding of the underlying principles of automata theory.

Overall, the laboratory work successfully encapsulates the core principles of finite automata, from their theoretical foundations to practical applications. It provides a solid framework for understanding how automata can be used to model computational processes and languages. The exercises conducted not only reinforce the theoretical underpinnings of computer science but also equip students with the necessary skills to apply these concepts in various computational contexts.

By completing this lab, there is a significant enhancement in understanding the operational mechanisms of finite automata, the process of converting between different types of automata, and the practical implications in computational theory and applications. This comprehensive approach ensures a well-rounded educational experience, preparing students for further studies in computer science and related fields.

References

1. https://www.geeksforgeeks.org/introduction-of-finite-automata/

2. https://www.youtube.com/watch?v=58N2N7zJGrQ&list=PLBlnK6fEyqRgp46KUv4ZY69yXmpwKOIev

3. http://www.gcekjr.ac.in/pdf/lectures/2020/3174I_5th%20Semester_Computer%20Science%20And%20Engineering.pdf

4. Theory lessons from else