Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Formal Languages and Finite Automata

Laboratory work 2:
Finite Automata

Elaborated:

st. gr. FAF-221 Reabciuc Daria-Brianna

Verified:

asist. univ. Cretu Dumitru
prof. univ. Cojuhari Irina

Chisinau 2024

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

```python
class FiniteAutomaton:
    def __init__(self, states, alphabet, transition_function, start_state, accept_states):
        self.states = states
        self.alphabet = alphabet
        self.transition_function = transition_function
        self.start_state = start_state
        self.accept_states = accept_states

    def to_regular_grammar(self):
        grammar = {}
        for state in self.states:
            grammar[state] = set()
            for char in self.alphabet:
                if (state, char) in self.transition_function:
                    target_state = self.transition_function[(state, char)]
                    grammar[state].add(char + target_state)
            if state in self.accept_states:
                grammar[state].add('ε')  # ε represents an empty string (epsilon)
        return grammar
```

This code is about creating a basic computer program that simulates a simple machine called a

finite automaton, which is like a little robot that follows a set of rules to decide whether a string of letters is acceptable or not based on its rules.

The **__init__** part sets up this machine with its rules, starting point, and acceptable end points. The **to_regular_grammar** part turns the rules of this machine into a type of grammar, like converting the machine's behavior into a set of language rules, showing what sequences of letters it can produce.

This is used in computer science to understand how computers process language and symbols, showing that the patterns the machine accepts can be described like the rules of a language.

The **to_regular_grammar** method converts the finite automaton into a regular grammar. This is significant in the theory of computation as it demonstrates the equivalence between regular grammars and finite automatons. In this method:

- A new dictionary named **grammar** is created to store the conversion result.
- For each state in the automaton, the method iterates through each character in the alphabet. If a transition exists for a state-character pair, the method constructs a production rule in the grammar, where the left-hand side is the state and the right-hand side is the concatenation of the character and the target state. This reflects the nature of regular grammars, which generate strings by replacing non-terminal symbols with combinations of terminals and other non-terminals.
- If the current state is an accept state, an epsilon (ε) production is added, indicating that the automaton can accept the string and terminate in this state. This is represented in the grammar as a transition that produces an empty string.

By defining this class and methods, the code bridges the conceptual gap between finite state machines and regular grammars, showing how each state in the machine can correspond to a non-terminal in the grammar, and how transitions in the machine translate into production rules in the grammar.

```
def is_deterministic(fa):
    for state in fa.states:
        seen_transitions = set()
        for char in fa.alphabet:
            if (state, char) in fa.transition_function:
                if char in seen_transitions:
                    return False  # More than one transition for a state and symbol
                seen_transitions.add(char)
            else:
                return False  # No transition for a state and symbol
    return True
```

The **is_deterministic(fa)** function is like a checker tool for understanding if a set of rules (our finite automaton) plays by strict guidelines (deterministic) or has choices (non-deterministic). For each

rule (state) and situation (symbol), it checks if there's only one possible outcome. If every rule and situation leads to a single specific outcome, then our set of rules is called deterministic, meaning it's predictable and follows a clear path. But, if any rule in any situation can lead to more than one outcome, or if there's a situation with no outcome at all, then our set of rules is not playing by the strict "one situation, one outcome" principle, making it non-deterministic. The function walks through all the rules and situations to give us a clear yes or no answer on whether our rules are strict (deterministic) or not.

The function initiates its operation by iterating through each state within the finite automaton's defined set of states. For each state, it maintains a Python set, **seen_transitions**, intended to record the unique transitions (or moves) possible from that state for different input symbols from the automaton's alphabet.

Within the nested loop, the function examines each character (or symbol) in the finite automaton's alphabet. For each character, the function checks if there is an existing transition from the current state using that character. This check is performed by accessing the automaton's **transition_function**, a data structure typically mapping pairs of states and symbols to resultant states.

```
def convert_ndfa_to_dfa(ndfa):
    # Create new DFA
    new_states = set(['q0'])  # Start with the initial state
    new_accept_states = set()
    new_transition_function = {}
    unprocessed_states = [{'q0'}]  # States to process

    while unprocessed_states:
        current_new_state = unprocessed_states.pop()
        for char in ndfa.alphabet:
            next_new_state = set()
            for state in current_new_state:
                if (state, char) in ndfa.transition_function:
                    next_new_state.add(ndfa.transition_function[(state, char)])
            if next_new_state:
                new_state_name = ''.join(sorted(next_new_state))
                new_transition_function[(''.join(sorted(current_new_state)), char)] = new_state_name
                if new_state_name not in new_states:
                    new_states.add(new_state_name)
                    unprocessed_states.append(next_new_state)
                if next_new_state & ndfa.accept_states:
                    new_accept_states.add(new_state_name)

    return FiniteAutomaton(new_states, ndfa.alphabet, new_transition_function, 'q0', new_accept_states)
```

The **convert_ndfa_to_dfa** function is like a recipe for turning a complicated spaghetti-like roadmap (NDFA) into a straightforward, single-path map (DFA). It takes every possible spot and situation from the NDFA and mixes them to make new, simpler spots (states) for the DFA. Each new spot represents a bunch of old spots at once, making sure that no matter where you were on the old map, there's a clear sign pointing where to go next on the new one. This process keeps going until every possible situation from the NDFA is covered, ensuring that the new map (DFA) gets you to your destination just as well as the old one but without any confusion about which way to turn.

The core of the conversion lies in iterating through every possible state configuration generated by the NDFA's transitions. Since NDFAs can transition to multiple states from a single state under the

same input symbol, the resulting DFA will have states that are combinations (or sets) of NDFA states. The function uses **unprocessed_states** to track which new DFA states need their transitions defined. It starts with the initial state and processes each composite state one at a time.

For each composite state, the function examines every possible input symbol from the NDFA's alphabet. It then determines the set of NDFA states that can be reached from any state in the composite under each input symbol. This new set of states forms a single state in the DFA. If this new state hasn't been seen before, it is added to the DFA's state set and queued for further processing.

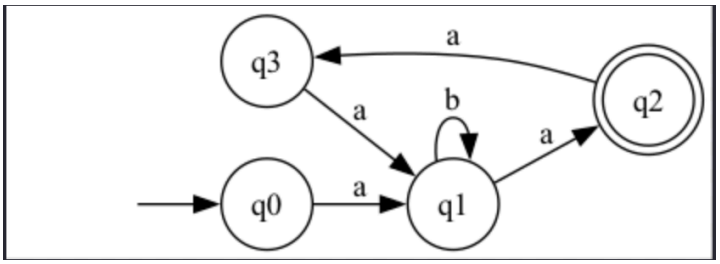
```
def visualize(self):
    dot = Digraph()
    dot.attr(rankdir='LR', size='8,5')

    # Non-accept states
    for state in self.states - self.accept_states:
        dot.node(state, shape='circle')
    # Accept states
    for state in self.accept_states:
        dot.node(state, shape='doublecircle')

    # Invisible start node
    dot.node('', shape='none')
    dot.edge('', self.start_state)

    # Transitions
    for (src, symbol), dst in self.transition_function.items():
        dot.edge(src, dst, label=symbol)

    return dot
```

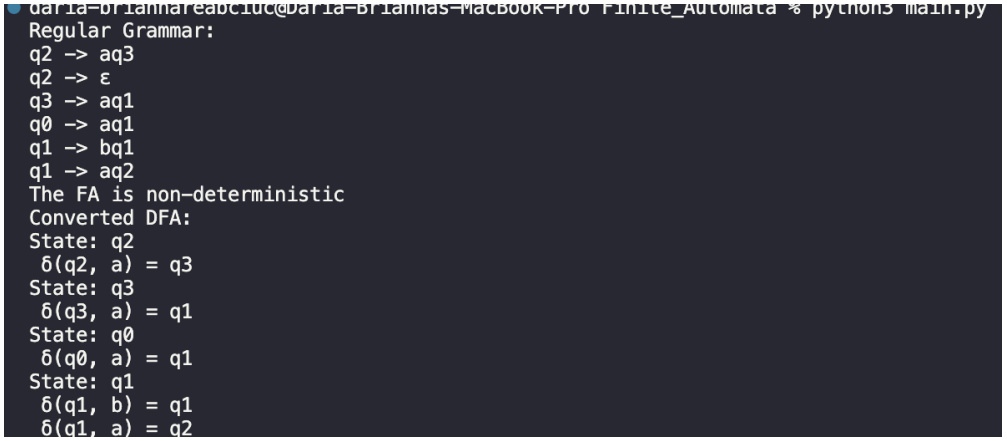


The **visualize** function is like a blueprint creator for our automaton. It takes the abstract idea of states and transitions and turns them into a clear map. Just like in a city map, where different locations are connected by roads, in our automaton map, the circles (states) are connected by arrows (transitions). The big circles (double circles) are special places (accept states) where we want to end up, and the other circles are regular stops along the way. The map starts from a specific place (the start state), shown by an arrow pointing to our first stop. By looking at the drawn map, we can easily see how to move from one state to another using different inputs (like 'a' or 'b'), just like following directions to get to a destination.

The method concludes by returning the **dot** object, which contains the complete graph representation. This object can be rendered into various formats for display or analysis, offering a

visual understanding of the FA's structure and behavior.

Results



First, it turns a set of rules (the FA) into a grammar, kind of like turning action instructions into a recipe. Each line tells us where to go next depending on the 'ingredient' (a letter) we have.

Then, the program checks if the machine is straightforward (deterministic) or if it can do different things in the same situation (non-deterministic). Here, it found that our machine can be uncertain, meaning it's non-deterministic.

Lastly, because computers prefer certainty, the program converts our wishy-washy machine into a strict one (DFA) where every step is clear and predictable. The final part shows the new, no-nonsense rules after conversion. This step makes sure our machine is ready for real-world use, like in a text editor or web browser, where clear and predictable behavior is crucial.

Conclusion

The exploration of finite automata (FA), their conversion to regular grammars, and the transformation from non-deterministic finite automata (NDFA) to deterministic finite automata (DFA) represents a significant journey into the fundamentals of computational theory and its practical implications in computer science.

Starting with the conversion of a finite automaton into a regular grammar, we witnessed how the structural behaviors of computational models are translated into syntactic rules, enabling the representation of state transitions in a formal linguistic format. This conversion is not merely academic but foundational for understanding how programming languages are parsed and interpreted.

The assessment of determinism within a finite automaton brought to light the complexities and nuances of computational processes. Determinism, as a property, dictates the predictability and straightforwardness of computational models, which is essential for the efficiency and reliability of algorithms, especially in areas like pattern recognition and text processing.

Transitioning from non-deterministic to deterministic finite automata encapsulates the essence of computational optimization and clarity. The process of converting an NDFA to a DFA, while theoretically ensuring the same language acceptance, practically simplifies the computational model, making it more transparent and easier to implement in software and hardware solutions.

In conclusion, the intertwining of theoretical concepts and practical implementations seen in finite automata and their transformations underpins much of modern computing. Understanding these concepts is crucial for anyone delving into the fields of computer science, especially in areas related to compilers, programming languages, and automata theory. The ability to abstract from specific instances to general rules, and to ensure deterministic behavior in computational models, is foundational to the development and operation of reliable and efficient computational systems.

https://github.com/dariabrianna/DSL_personal_laboratories/blob/main/Finite_Automata/main.py