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

Course: Formal Languages & Finite Automata

Author: Moraru Gabriel

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.

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**.

Variant 19

Q = {q0,q1,q2},

∑ = {a,b},

F = {q2},

δ(q0,a) = q1,

δ(q0,a) = q0,

δ(q1,b) = q2,

δ(q0,b) = q0,

δ(q1,b) = q1,

δ(q2,b) = q2.

Implementation description:

```
def __init__(self, states, alphabet, transitions, start_state, final_states):
    self.states = states
    self.alphabet = alphabet
    self.transitions = transitions
    self.start_state = start_state
    self.final_states = final_states
```

What It Does:

- This is the **constructor method** for the FiniteAutomaton class.
- It initializes the automaton with:
  - **states** → The set of all states in the automaton.
  - **alphabet** → The set of input symbols (e.g., {a, b}).
  - **transitions** → A dictionary defining state transitions.
  - **start_state** → The initial state.
  - **final_states** → The set of final (accepting) states.
- Stores these as instance attributes to be used in other methods.

```python
def is_deterministic(self):
    for state, transitions in self.transitions.items():
        for symbol, next_states in transitions.items():
            if len(next_states) > 1:
                return False
    return True
```

What It Does:

- Checks whether the automaton is **deterministic** (DFA) or **non-deterministic** (NFA).
- It iterates through the transitions dictionary:
    - If any state has **more than one possible next state for the same input symbol**, it means the automaton is an **NFA**, and the function returns False.
    - If every symbol leads to at most **one** state, it returns True (i.e., it is a DFA).
- **Example:**
    - **NFA:** { "q1": { "b": {"q2", "q3"} } } (two possible next states → non-deterministic)
    - **DFA:** { "q1": { "b": {"q2"} } } (only one next state → deterministic)

```python
def to_dfa(self):
    dfa_states = set()
    dfa_transitions = {}
    state_map = {}

    queue = [frozenset([self.start_state])]
    while queue:
        current = queue.pop(0)
        dfa_states.add(current)
        state_map[current] = "{" + ",".join(current) + "}"

        transitions = {}
        for symbol in self.alphabet:
            next_state = frozenset(itertools.chain.from_iterable(self.transitions.get(state, {}).get(symbol, []) for state in current))
            if next_state:
                transitions[symbol] = next_state
                if next_state not in dfa_states and next_state not in queue:
                    queue.append(next_state)
        dfa_transitions[current] = transitions

    dfa_final_states = {state for state in dfa_states if any(s in self.final_states for s in state)}
    return FiniteAutomaton(dfa_states, self.alphabet, dfa_transitions, frozenset([self.start_state]), dfa_final_states), state_map
```

What It Does:

- Converts a **Non-deterministic Finite Automaton (NFA)** to a **Deterministic Finite Automaton (DFA)**.
- Uses **subset construction (powerset construction)** to handle multiple possible transitions.
- **Steps:**
    1. **Initialize**:

- dfa_states: A set to track new DFA states.
- dfa_transitions: A dictionary to store the DFA transition table.
- state_map: A dictionary mapping DFA states to readable names.
- queue: A list to process new DFA states (starting with {start_state}).

2. **Process each DFA state:**
   - Extracts all possible transitions for each symbol.
   - Groups multiple NFA states into a single DFA state.
   - Adds new states to the queue if they haven't been seen before.
3. **Identify final states:** Any DFA state that contains at least one NFA final state is marked as a **DFA final state**.
4. **Returns:**
   - A new FiniteAutomaton object representing the DFA.
   - A state_map for translating DFA states into readable names.

```
def print_as_regular_grammar(self):
    print("Regular Grammar:")
    print("Non-terminals:", list(self.states))
    print("Terminals:", list(self.alphabet))
    print("Productions:")
    for state, transitions in self.transitions.items():
        for symbol, next_states in transitions.items():
            for next_state in next_states:
                print(f"{state} : {symbol}{next_state}")
        if state in self.final_states:
            print(f"{state} : ε")  # Epsilon transition for final states
```

What It Does:

- Converts the automaton into an **equivalent regular grammar**.
- **Outputs:**
  - **Non-terminals** → The states.
  - **Terminals** → The alphabet symbols.
  - **Productions** (rules):
    - {state} : {symbol}{next_state} for each transition.
    - If a state is final, it gets an **epsilon transition (ε)**, allowing it to end a string.

```python
def print_as_fa(self, state_map):
    print("Converted NFA to DFA:")
    print("States:", [state_map[state] for state in self.states])
    print("Alphabet:", list(self.alphabet))
    print("Transitions:")
    for state, transitions in self.transitions.items():
        for symbol, next_state in transitions.items():
            print(f"{state_map[state]}--{symbol}-->{state_map[next_state]}")
```

What It Does:

- Prints the **converted DFA** with readable state names.
- **Outputs:**
    - **States** in the DFA.
    - **Alphabet** used.
    - **Transitions** in a readable format:

{q0}--a-->{q0}

{q0}--b-->{q1}

- Uses state_map to convert **frozensets (DFA states)** into human-readable names.
- Helps verify whether the **NFA-to-DFA conversion was correct**.

Output example:

```
Regular Grammar:
Non-terminals: ['q1', 'q2', 'q0']
Terminals: ['b', 'a']
Productions:
q0 : aq1
q0 : aq0
q0 : bq0
q1 : bq1
q1 : bq2
q2 : bq2
q2 : ε
Finite Automaton is non-deterministic
Converted NFA to DFA:
States: ['{q1,q0}', '{q1,q2,q0}', '{q0}']
Alphabet: ['b', 'a']
Transitions:
{q0}--b-->{q0}
{q0}--a-->{q1,q0}
{q1,q0}--b-->{q1,q2,q0}
{q1,q0}--a-->{q1,q0}
{q1,q2,q0}--b-->{q1,q2,q0}
{q1,q2,q0}--a-->{q1,q0}
```

Conclusion:

In this lab, we implemented a Finite Automaton class handling both NFAs and DFAs, providing a practical approach to automata theory. We defined an NFA, checked its determinism using `is_deterministic`, and converted it to a DFA using `to_dfa`, applying the subset construction algorithm. The DFA's regular grammar and transition table were displayed for clarity. This work demonstrated key concepts like state transitions, determinism, and NFA-to-DFA conversion, which are essential in compiler design, lexical analysis, and formal language theory. The implementation reinforces the role of automata in computation, pattern recognition, and programming language design.

References:

1. Else Course FAF.LFA21.1
2. Finite Automata https://www.geeksforgeeks.org/finite-automata-algorithm-for-pattern-searching/

   https://stackoverflow.com/questions/35272592/how-are-finite-automata-implemented-in-code