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

Course: Formal Languages & Finite Automata

Author: Grosu Renat

Theory:

Chomsky Hierarchy:
The Chomsky Hierarchy is a framework for classifying formal grammars according to their generative power. Developed by Noam Chomsky, it categorizes grammars into four levels:

Type 0 (Recursively Enumerable): The most general class, allowing any production rules, including those where both sides can contain terminals and non-terminals. These grammars generate recursively enumerable languages, which are the languages that a Turing machine can recognize. There are no restrictions on the form of its production rules.

Type 1 (Context-Sensitive Grammars): These grammars generate context-sensitive languages. Their production rules must be in the form $\alpha A \beta \rightarrow \alpha \gamma \beta$, where A is a non-terminal, and α, β, and γ are strings of terminals and non-terminals. The rule ensures that the length of the production's right-hand side is not less than the left-hand side, except for certain conditions allowing the empty string.

Type 2 (Context-Free Grammars): These grammars generate context-free languages and have rules in the form $A \rightarrow \gamma$, where A is a single non-terminal, and γ is a string of terminals and non-terminals. Context-free grammars are crucial in programming language design and parsing algorithms.

Type 3 (Regular Grammars): The least powerful class, generating regular languages. Their rules are restricted further to A → aB or A → a forms, where A and B are non-terminals, and a is a terminal. Regular grammars correspond to finite automata and are used in designing lexical analyzers and simple language tokens.

Regular Grammar:
A regular grammar is a formal grammar that is right-linear or left-linear, meaning its production rules satisfy the conditions for creating regular languages. In practical terms, regular grammars underpin regular expressions, define search patterns, and are foundational to lexical analysis in compilers.

Finite Automata:
Finite Automata (FAs) are abstract mathematical models of computation used to simulate sequential logic and recognize patterns of inputs. There are two main types:

Deterministic Finite Automata (DFA): In DFAs, for every state and input symbol, there is exactly one transition to a new state. DFAs are used to represent and recognize regular languages, showcasing a direct correlation with regular grammars.

Non-Deterministic Finite Automata (NFA): NFAs allow for multiple or zero transitions for a single state and input symbol combination. Despite this apparent flexibility, NFAs are equivalent in language recognition power to DFAs, as any NFA can be converted into a corresponding DFA, albeit potentially with an exponential increase in the number of 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 (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 description:

Chomsky hierarchy classifier

```
def classify(self):
    is_regular = True
    is_context_free = True
    is_context_sensitive = True

    for lhs, rhs_list in self.rules.items():
        for rhs in rhs_list:
            if not (len(rhs) == 1 and rhs in self.terminals) and not \
                    (len(rhs) == 2 and ((rhs[0] in self.terminals and rhs[1] in self.non_terminals) or
                                        (rhs[0] in self.non_terminals and rhs[1] in self.terminals))):
                is_regular = False
            if lhs not in self.non_terminals:
                is_context_free = False
            if len(rhs) < len(lhs):
                is_context_sensitive = False

    if is_regular:
        return "Type 3 (Regular)"
    elif is_context_free:
        return "Type 2 (Context-free)"
    elif is_context_sensitive:
        return "Type 1 (Context-sensitive)"
    else:
        return "Type 0 (Recursively enumerable)"
```

The classify method assesses a grammar's production rules to categorize it within the Chomsky hierarchy by applying specific checks:

Regular Grammar Check: A grammar is considered regular if each production rule transforms a non-terminal into a single terminal, or a single terminal followed by a single non-terminal. The method flags the grammar as non-regular if it encounters any rule that doesn't meet these conditions.

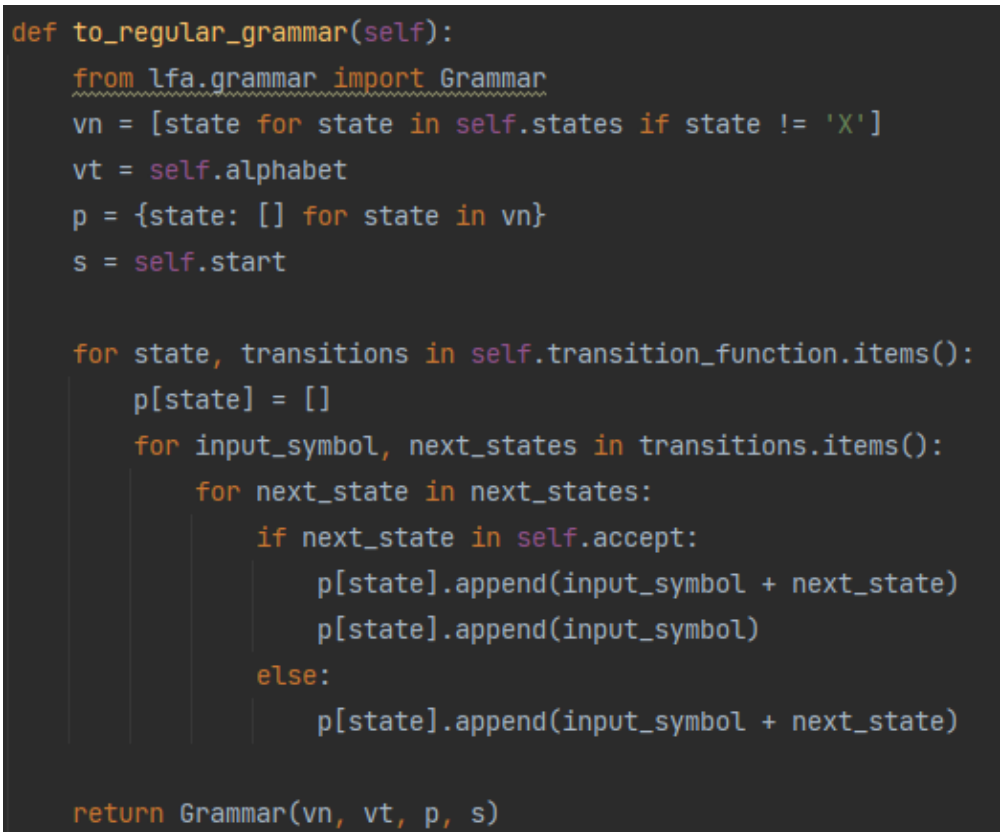
Context-Free Grammar Check: The grammar remains classified as context-free if all production rules have a single non-terminal on the left-hand side. If any rule deviates from this format the grammar is flagged as

not context-free.

Context-Sensitive Grammar Check: For a grammar to be context-sensitive, the right-hand side should not be shorter than the left-hand side. The method flags the grammar as not context-sensitive if any rule violates this principle.

After iterating through all rules and applying these checks, the method classifies the grammar based on which criteria it satisfies, from the most restrictive (Type 3 - Regular) to the least restrictive (Type 0 - Recursively Enumerable).

FA to Regular Grammar Conversion



```
def to_regular_grammar(self):
    from lfa.grammar import Grammar
    vn = [state for state in self.states if state != 'X']
    vt = self.alphabet
    p = {state: [] for state in vn}
    s = self.start

    for state, transitions in self.transition_function.items():
        p[state] = []
        for input_symbol, next_states in transitions.items():
            for next_state in next_states:
                if next_state in self.accept:
                    p[state].append(input_symbol + next_state)
                    p[state].append(input_symbol)
                else:
                    p[state].append(input_symbol + next_state)

    return Grammar(vn, vt, p, s)
```

The to_regular_grammar method converts a finite automaton into its equivalent regular grammar:

1. Initialization: The method initializes several components for the new grammar:
    - vn collects all state names from the automaton, excluding a specific state labeled 'X', which usually represents a dead or error state.

- vt is set to the automaton's alphabet.
- p creates an empty production rule list for each state.
- s sets the start symbol of the grammar to the start state of the automaton.

2. Creating Production Rules:

   The method iterates over each state and its transitions in the automaton.

   For each transition it creates one or two new production rules:

   - If the next state is an accept state, two rules are added: one transitioning to the next state appended by the input symbol, and another for the input symbol leading directly to an empty string.
   - If the next state is not an accept state, a single rule is added linking the current state to the next state with the input symbol.

3. Returning the New Grammar: After constructing the set of production rules, the method wraps up by returning a new Grammar object constructed with the states (non-terminals), alphabet (terminals), production rules, and start symbol defined through the previous steps.

Is FA deterministic

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

The is_deterministic method determines if a finite automaton is deterministic by iterating through each state's transitions. If any state has multiple possible next states for a single input symbol, the automaton is non-deterministic, and the method returns False. If all transitions adhere to determinism, meaning each input symbol from each state leads to exactly one next state, the method returns True.

NFA to DFA conversion

```
def to_dfa(self):
    initial_dfa_state = frozenset([self.start])
    dfa_transitions = defaultdict(dict)
    dfa_states = {initial_dfa_state}
    states_to_process = [initial_dfa_state]
    dfa_accept_states = set()

    state_names = {initial_dfa_state: '_'.join(sorted(initial_dfa_state)) or 'empty'}
```

```
while states_to_process:
    current_dfa_state = states_to_process.pop()
    for symbol in self.alphabet:
        next_states_set = frozenset(
            sum([self.transition_function.get(nfa_state, {}).get(symbol, []) for nfa_state in
                 current_dfa_state], [])
        )
        if next_states_set:
            if next_states_set not in state_names:
                state_names[next_states_set] = '_'.join(sorted(next_states_set)) or 'empty'
            dfa_transitions[current_dfa_state][symbol] = next_states_set
            if next_states_set not in dfa_states:
                dfa_states.add(next_states_set)
                states_to_process.append(next_states_set)
            if any(state in self.accept for state in next_states_set):
                dfa_accept_states.add(next_states_set)

dfa_start = state_names[initial_dfa_state]
dfa_states_named = [state_names[state] for state in dfa_states]
dfa_accept_named = [state_names[state] for state in dfa_accept_states]

dfa_transitions_named = {}
for state, transitions in dfa_transitions.items():
    transition_dict = {}
    for symbol, next_state_set in transitions.items():
        transition_dict[symbol] = [state_names[next_state_set]]
    dfa_transitions_named[state_names[state]] = transition_dict

return FiniteAutomaton(dfa_states_named, self.alphabet, dfa_transitions_named, dfa_start, dfa_accept_named)
```

The to_dfa method converts a non-deterministic finite automaton (NFA) into a deterministic finite automaton (DFA) using the subset construction method implemented in the following steps:

1. Initialization:

- initial_dfa_state is the initial state of the DFA, based on the NFA's start state, represented as a frozenset.
- dfa_transitions uses a defaultdict to store transitions of the DFA.
- dfa_states keeps track of all the states in the DFA.
- states_to_process is a queue of DFA states that need to be analyzed.
- dfa_accept_states identifies which states in the DFA are accept states.
- state_names maps each set of NFA states (each DFA state) to a string name, initially just naming the initial DFA state.

2. Processing States:

The method enters a loop that continues until there are no more DFA states to process.

It pops a state from states_to_process to analyze its possible transitions based on the NFA's transition function and the current symbol of the alphabet.

3. Creating New States and Transitions:

For each symbol, the method calculates the set of all possible next states (from the NFA) that can be reached from the current DFA state. This is stored as a frozenset to ensure uniqueness and immutability.

If this set of next states forms a new DFA state (not already in dfa_states), it's added to dfa_states and states_to_process for further exploration.

If this set of states includes any of the NFA’s accept states, the new DFA state is marked as an accept state by adding it to dfa_accept_states.

4. Naming States and Building Transitions:

New DFA states (sets of NFA states) are assigned human-readable names (combining the names of NFA states contained within each new DFA state).

Transitions in the DFA are updated based on these sets of next states. Each entry in dfa_transitions maps from the current DFA state under analysis to these new sets, thereby constructing the DFA's transition function.

5. Finalizing the DFA:

Once all states have been processed, the method finalizes the naming for DFA states and accept states based on the mapping established in state_names.

It constructs a named transition dictionary for the DFA where transitions reference the named states rather than sets of NFA states.

The method concludes by creating and returning a new FiniteAutomaton object, signifying the complete DFA, with its states, alphabet, transition function, start state, and accept states properly defined.

FA graphical representation

```
def visualize(self, filename='finite_automaton'):
    dot = Digraph()

    for state in self.states:
        if state in self.accept:
            dot.node(state, shape='doublecircle')
        else:
            dot.node(state, shape='circle')

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

    for state, transitions in self.transition_function.items():
        for symbol, next_states in transitions.items():
            for next_state in next_states:
                dot.edge(state, next_state, label=symbol)
    dot.render(filename, format='png', cleanup=True)
```

The visualize method graphically represents a finite automaton using the Graphviz library by doing the following steps:

Graph Initialization: It creates a new directed graph object.

State Nodes Creation: For each state in the automaton, it adds a node to the graph. If the state is an accepting state, it is represented with a double circle; otherwise, it's represented with a single circle.

Starting Node: An invisible node is created, which acts as a starting point, and an edge is drawn from this node to the start state of the automaton.

Transition Edges: For each state, the method looks through all possible transitions (symbol-next states pairs). It adds an edge for each transition from the current state to the next state, labeled with the symbol triggering the transition.

Rendering: The graph is rendered to a PNG file with the given filename, and temporary files are cleaned up after rendering.

Conclusions/Screenshots/Results:

My grammar definition variant:

```
Variant 10:
VN={S, B, L},
VT={a, b, c},
P={
    S → aB
    B → bB
    B → cL
    L → cL
    L → aS
    L → b
}
```

Grammar classifier results:

```
Type 3 (Regular)
```

Grammar to automaton results:

```
Automaton states: ['S', 'L', 'B', 'X']
Automaton alphabet: ['a', 'b', 'c']
Automaton transition function: {'X': {}, 'S': {'a': 'B'}, 'L': {'c': 'L', 'a': 'S', 'b': 'X'}, 'B': {'b': 'B', 'c': 'L'}}
Automaton start: S
Automaton accept: ['X']
```

My FA variant:

```
Variant 10
Q = {q0,q1,q2,q3},
Σ = {a,b,c},
F = {q3},
δ(q0,a) = q1,
δ(q1,b) = q2,
δ(q2,c) = q3,
δ(q3,a) = q1,
δ(q1,b) = q1,
δ(q0,b) = q2.
```

Is FA deterministic results:

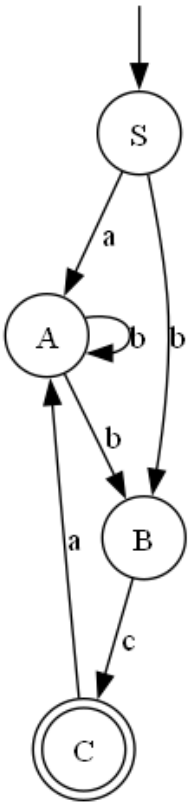
```
Is the FA deterministic ? False
```

NFA to DFA conversion results:

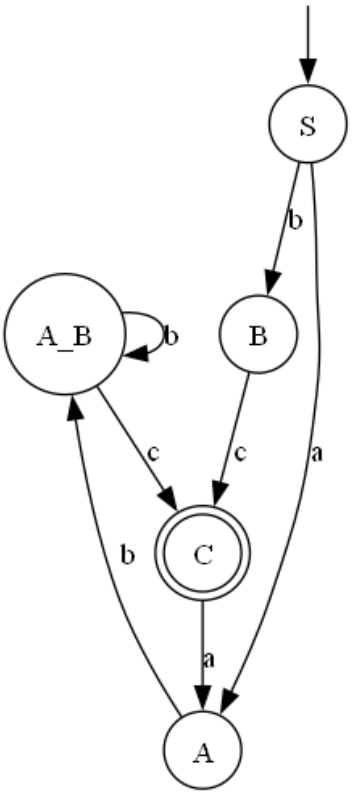
```
DFA states: ['A', 'S', 'A_B', 'C', 'B']
DFA alphabet: ['a', 'b', 'c']
DFA transition function: {'S': {'a': ['A'], 'b': ['B']}, 'B': {'c': ['C']}, 'C': {'a': ['A']}, 'A': {'b': ['A_B']}, 'A_B': {'b': ['A_B'], 'c': ['C']}}
DFA start: S
DFA accept: ['C']
```

FA graphical representation results:

NFA:



DFA:



Through these exercises, I got a better grip on how finite automata work and how they're used to understand different language patterns. I learned how to classify grammars using the Chomsky hierarchy, which shows the different levels of complexity languages can have. I also tackled converting a finite automaton into a regular grammar, which showed me how different formats can express the same ideas.

I figured out how to check if an automaton is deterministic or not. This was important because deterministic automata are simpler and more straightforward to use than non-deterministic ones. Plus, I learned to convert a non-deterministic automaton (NFA) into a deterministic one (DFA), which is useful for making automata easier to deal with.

The option to visually represent automata was also important as it helped me see how these abstract concepts actually play out visually. All in all, this laboratory work helped me better understand concepts like finite automatons and regular grammars and how they are similar.

References:

https://introcs.cs.princeton.edu/java/51language/

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