MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII

AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

Copta Adrian | FAF-223

Report

Laboratory work n.2

of Formal Languages & Finite Automata

Checked by:

Cretu Dumitru, *university assistant*

DISA, FCIM, UTM

Chișinău – 2024

1. Theory:

Chomsky Hierarchy - A hierarchy of classes of formal grammars named after the scientist Noam Chomsky. The hierarchy is composed of 4 types of formal grammars, lower levels including the higher ones:

Type 0 - Unrestricted grammar: Only 1 restriction on the production rules of this grammar: no empty string on the left side of the production.

Type 1 - Context-Sensitive grammar: The production rules of this grammar must all follow the format xAy = xby, where x,y are a string of any terminals or non-terminals, b is at least 1 terminal or non-terminal character, and A is a non-terminal character. x and y are called the "context" of A. If a non-terminal character has a transition to the empty string, it may not be on the left side of any other transition.

Type 2 - Context-Free grammar: Very similar to Type 1 grammars, as the main difference is that x and y (the "context") is no longer included: they are always empty strings. Production rules follow the format A->b, where A is a non-terminal and b can be the empty string or a string of terminal and/or non-terminals.

Type 3 - Regular grammar: The lowest in the hierarchy, it is the most restrictive type of grammar, where all production rules must either have the format A->Bb(or A->bB, but never both) or A->b, where B is a single non-terminal and b is a single terminal character. Rules of the format A->B and A-><empty string> are allowed if there are rules of the prior format existent.
Type 3 grammars can be classified into 2 types: "Left-regular" and "Right-regular", depending on the side of the nonterminal character relative to the terminal one within the production rule. Type 3 grammars can be converted into Finite Automata and vice versa.

Deterministic Finite Automata - Finite Automata where every transition may be uniquely determined by the input character and the next state. Cannot contain transitions with the empty string as input.

Non-deterministic Finite Automata - Finite Automata where there are multiple transitions with the same input characters corresponding to the same state.

2. Purpose of the task work:

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.

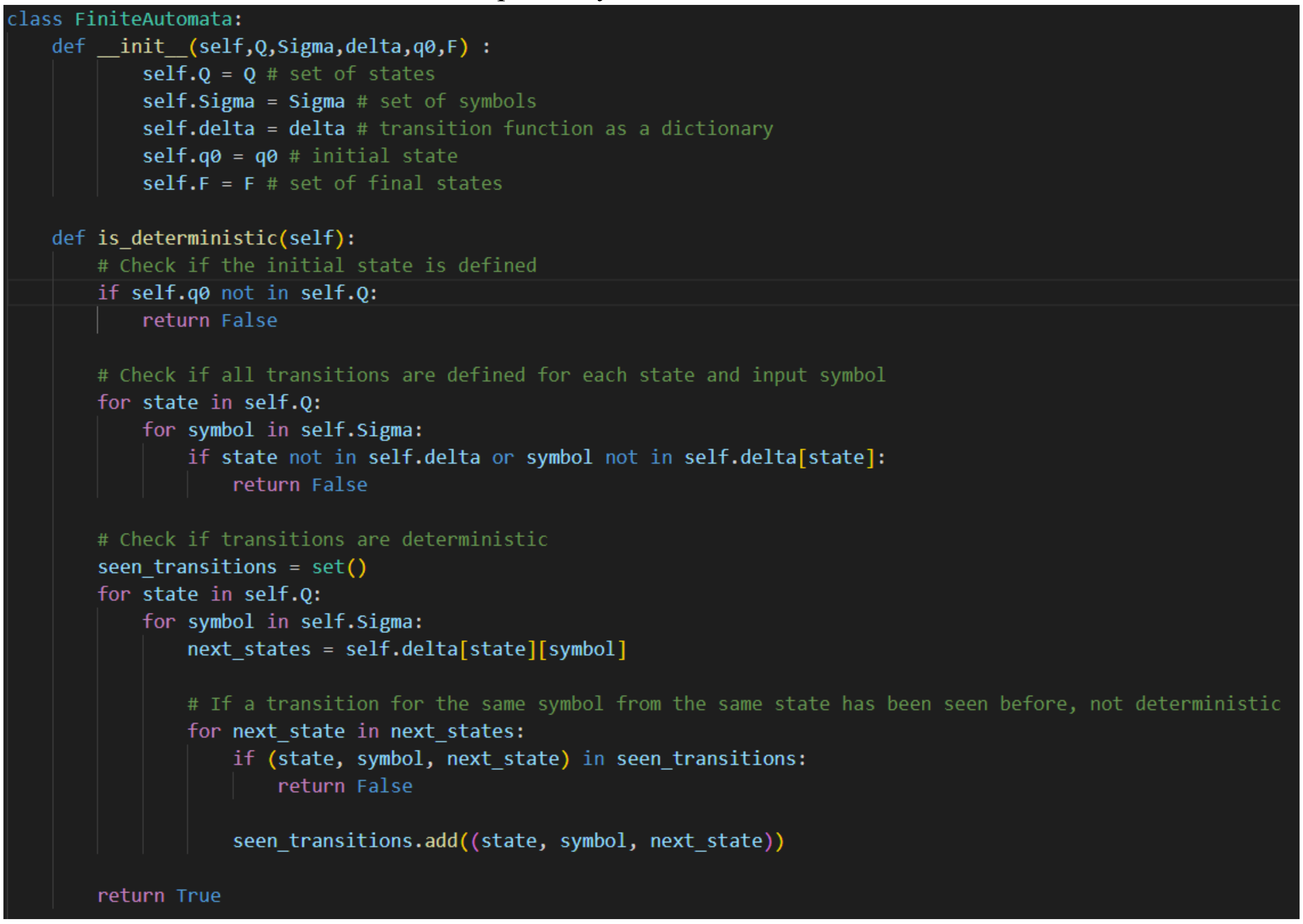
c. Implement some functionality that would convert an NFA to a DFA.

d. Represent the finite automaton graphically

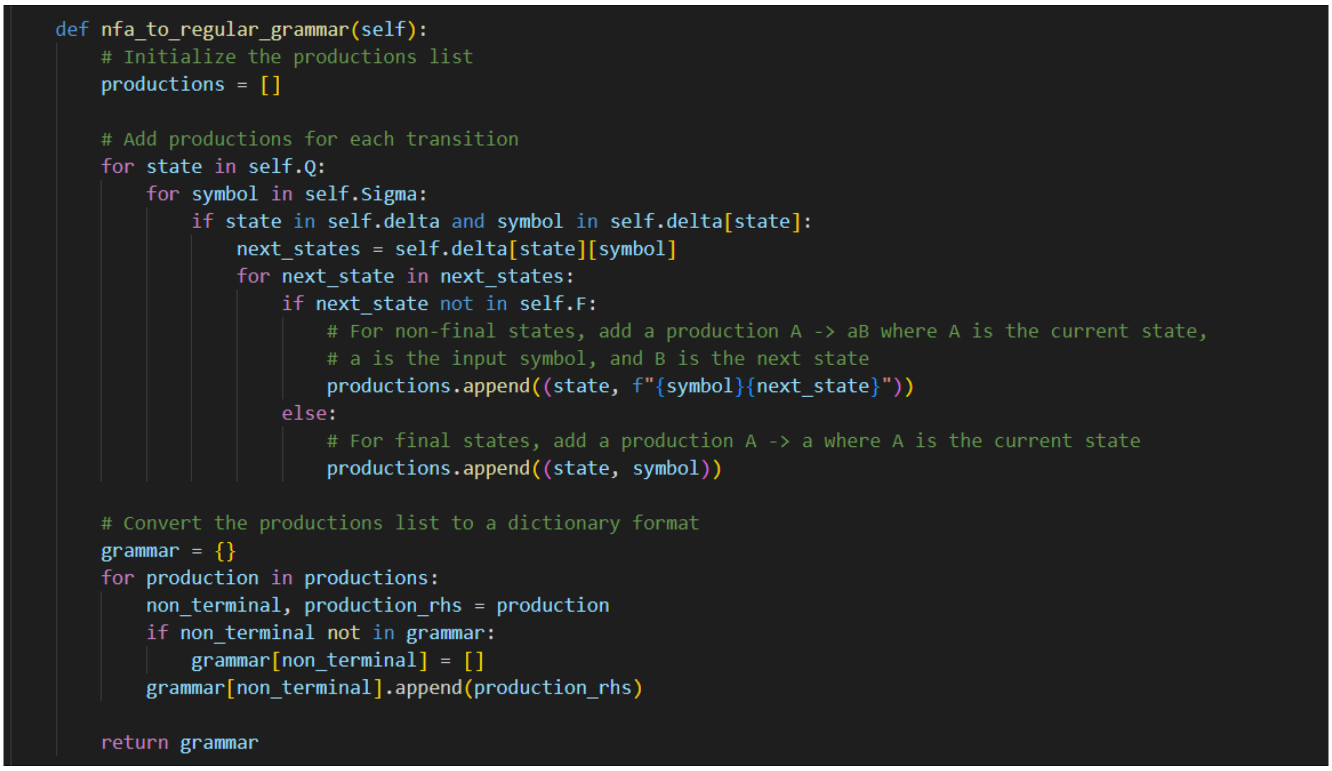
3. Implementation description:

Finite Automata class:

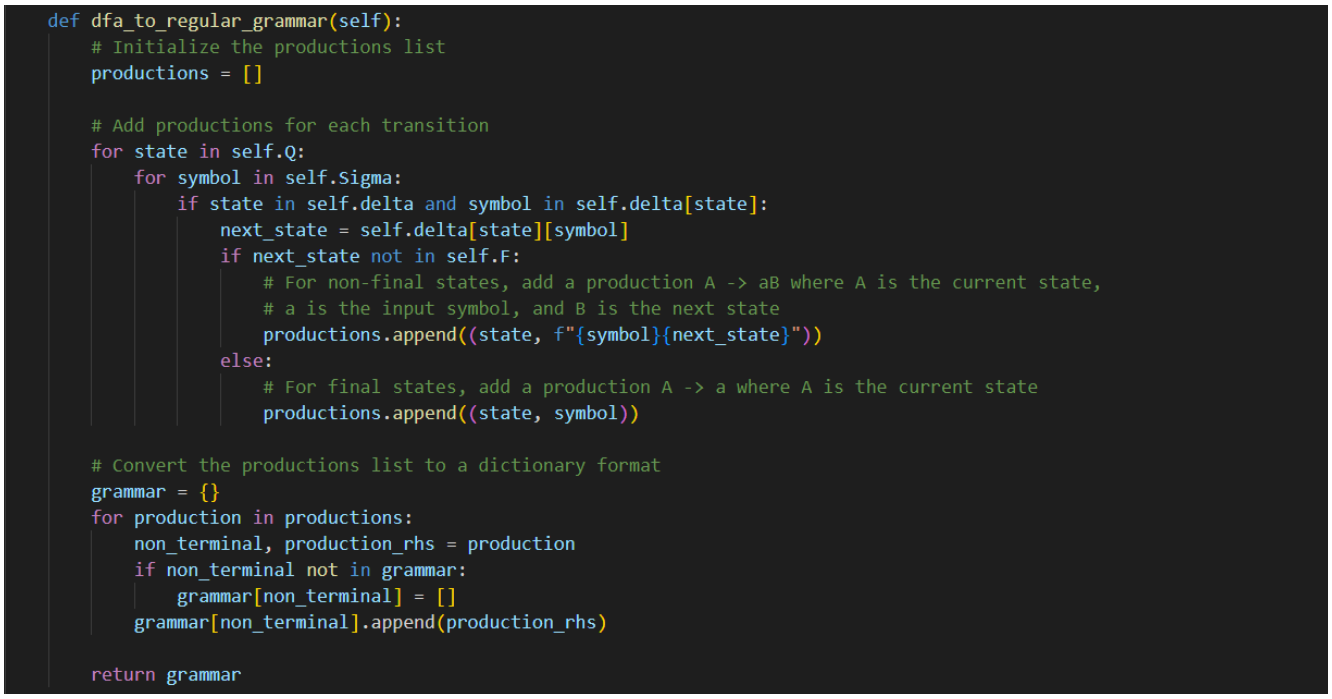
The FiniteAutomata class represents a finite automaton with states, symbols, transitions, an initial state, and a set of final states. It includes methods for checking whether the automaton is deterministic (is_deterministic), converting the NFA to a regular grammar (nfa_to_regular_grammar), converting the DFA to a regular grammar (dfa_to_regular_grammar), and generating the powerset of a set of states (powerset). Additionally, it provides a method nfa_to_dfa for converting an NFA to a DFA. The methods nfa_to_regular_grammar and dfa_to_regular_grammar produce regular grammars based on the transitions of the NFA and DFA, respectively.



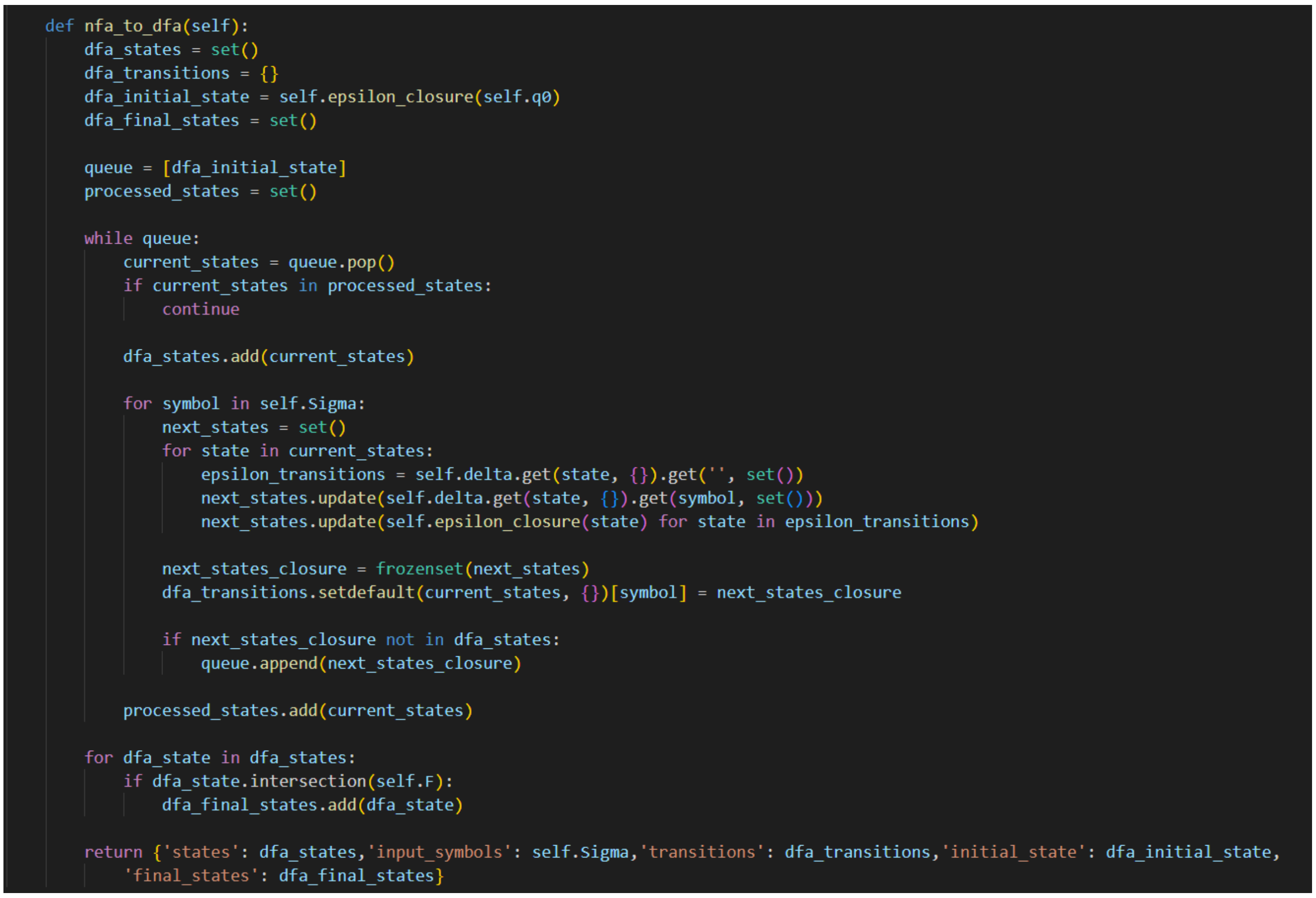
```python
class FiniteAutomata:
    def __init__(self,Q,Sigma,delta,q0,F) :
            self.Q = Q # set of states
            self.Sigma = Sigma # set of symbols
            self.delta = delta # transition function as a dictionary
            self.q0 = q0 # initial state
            self.F = F # set of final states

    def is_deterministic(self):
        # Check if the initial state is defined
        if self.q0 not in self.Q:
            return False

        # Check if all transitions are defined for each state and input symbol
        for state in self.Q:
            for symbol in self.Sigma:
                if state not in self.delta or symbol not in self.delta[state]:
                    return False

        # Check if transitions are deterministic
        seen_transitions = set()
        for state in self.Q:
            for symbol in self.Sigma:
                next_states = self.delta[state][symbol]

                # If a transition for the same symbol from the same state has been seen before, not deterministic
                for next_state in next_states:
                    if (state, symbol, next_state) in seen_transitions:
                        return False

                    seen_transitions.add((state, symbol, next_state))

        return True
```



```python
def nfa_to_regular_grammar(self):
    # Initialize the productions list
    productions = []

    # Add productions for each transition
    for state in self.Q:
        for symbol in self.Sigma:
            if state in self.delta and symbol in self.delta[state]:
                next_states = self.delta[state][symbol]
                for next_state in next_states:
                    if next_state not in self.F:
                        # For non-final states, add a production A -> aB where A is the current state,
                        # a is the input symbol, and B is the next state
                        productions.append((state, f"{symbol}{next_state}"))
                    else:
                        # For final states, add a production A -> a where A is the current state
                        productions.append((state, symbol))

    # Convert the productions list to a dictionary format
    grammar = {}
    for production in productions:
        non_terminal, production_rhs = production
        if non_terminal not in grammar:
            grammar[non_terminal] = []
        grammar[non_terminal].append(production_rhs)

    return grammar
```



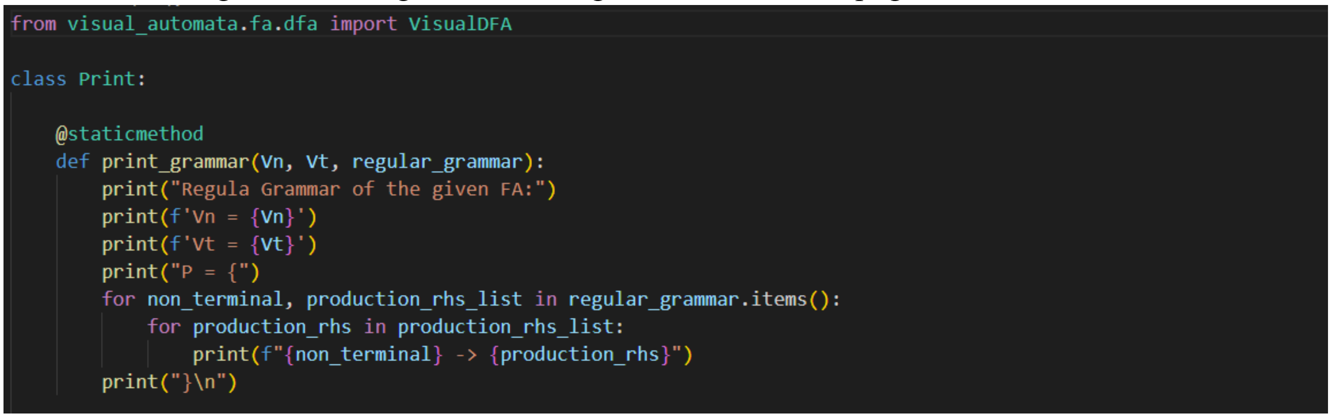
```python
def dfa_to_regular_grammar(self):
    # Initialize the productions list
    productions = []

    # Add productions for each transition
    for state in self.Q:
        for symbol in self.Sigma:
            if state in self.delta and symbol in self.delta[state]:
                next_state = self.delta[state][symbol]
                if next_state not in self.F:
                    # For non-final states, add a production A -> aB where A is the current state,
                    # a is the input symbol, and B is the next state
                    productions.append((state, f"{symbol}{next_state}"))
                else:
                    # For final states, add a production A -> a where A is the current state
                    productions.append((state, symbol))

    # Convert the productions list to a dictionary format
    grammar = {}
    for production in productions:
        non_terminal, production_rhs = production
        if non_terminal not in grammar:
            grammar[non_terminal] = []
        grammar[non_terminal].append(production_rhs)

    return grammar
```



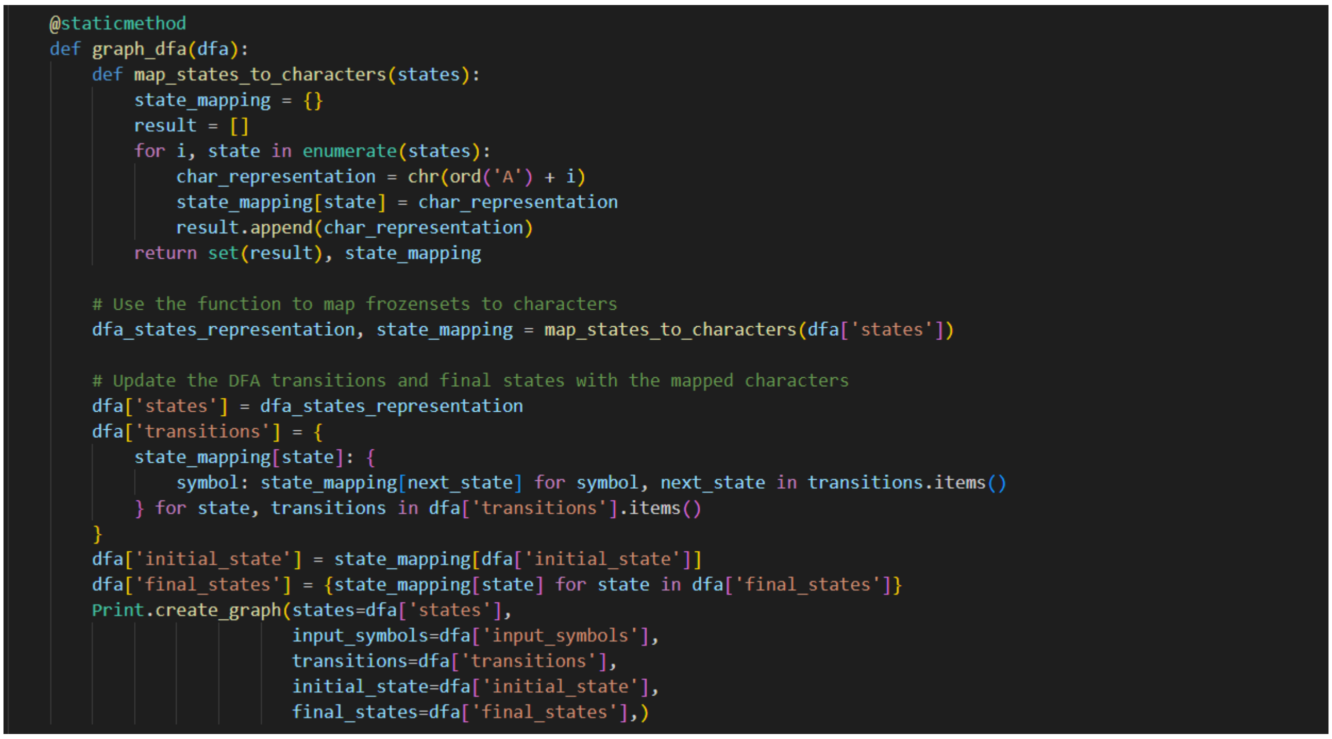
```python
    def nfa_to_dfa(self):
        dfa_states = set()
        dfa_transitions = {}
        dfa_initial_state = self.epsilon_closure(self.q0)
        dfa_final_states = set()

        queue = [dfa_initial_state]
        processed_states = set()

        while queue:
            current_states = queue.pop()
            if current_states in processed_states:
                continue

            dfa_states.add(current_states)

            for symbol in self.Sigma:
                next_states = set()
                for state in current_states:
                    epsilon_transitions = self.delta.get(state, {}).get('', set())
                    next_states.update(self.delta.get(state, {}).get(symbol, set()))
                    next_states.update(self.epsilon_closure(state) for state in epsilon_transitions)

                next_states_closure = frozenset(next_states)
                dfa_transitions.setdefault(current_states, {})[symbol] = next_states_closure

                if next_states_closure not in dfa_states:
                    queue.append(next_states_closure)

            processed_states.add(current_states)

        for dfa_state in dfa_states:
            if dfa_state.intersection(self.F):
                dfa_final_states.add(dfa_state)

        return {'states': dfa_states,'input_symbols': self.Sigma,'transitions': dfa_transitions,'initial_state': dfa_initial_state,
            'final_states': dfa_final_states}
```

Print Class:

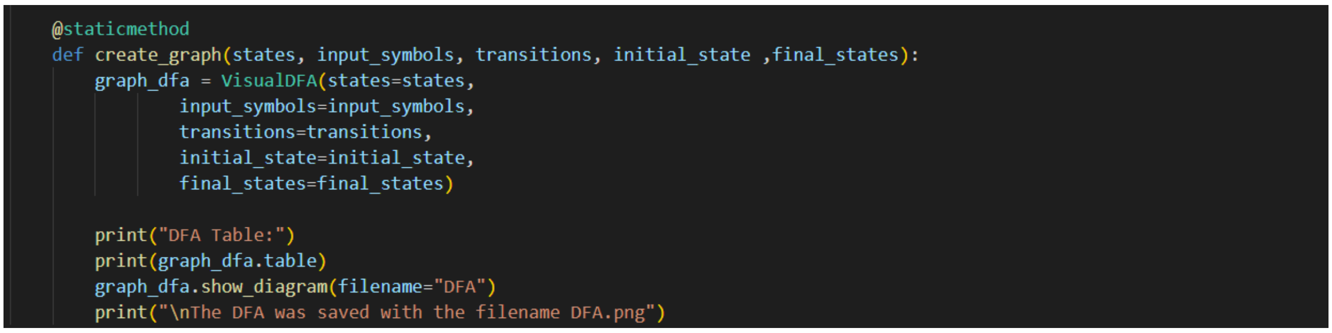
The Print class provides static methods for printing the regular grammar of a given finite automaton (print_grammar), graphing a DFA using the VisualDFA class (graph_dfa), and creating a graph of the DFA (create_graph). The print_grammar method outputs the non-terminals, terminals, and productions of the regular grammar. The graph_dfa method converts the DFA states and transitions into a format suitable for the VisualDFA class and prints the DFA table while also generating and displaying a visual representation of the DFA. The create_graph method initializes a VisualDFA instance and displays its table while saving the DFA diagram as an image file named "DFA.png."



```python
from visual_automata.fa.dfa import VisualDFA

class Print:

    @staticmethod
    def print_grammar(Vn, Vt, regular_grammar):
        print("Regula Grammar of the given FA:")
        print(f'Vn = {Vn}')
        print(f'Vt = {Vt}')
        print("P = {")
        for non_terminal, production_rhs_list in regular_grammar.items():
            for production_rhs in production_rhs_list:
                print(f"{non_terminal} -> {production_rhs}")
        print("}\n")
```



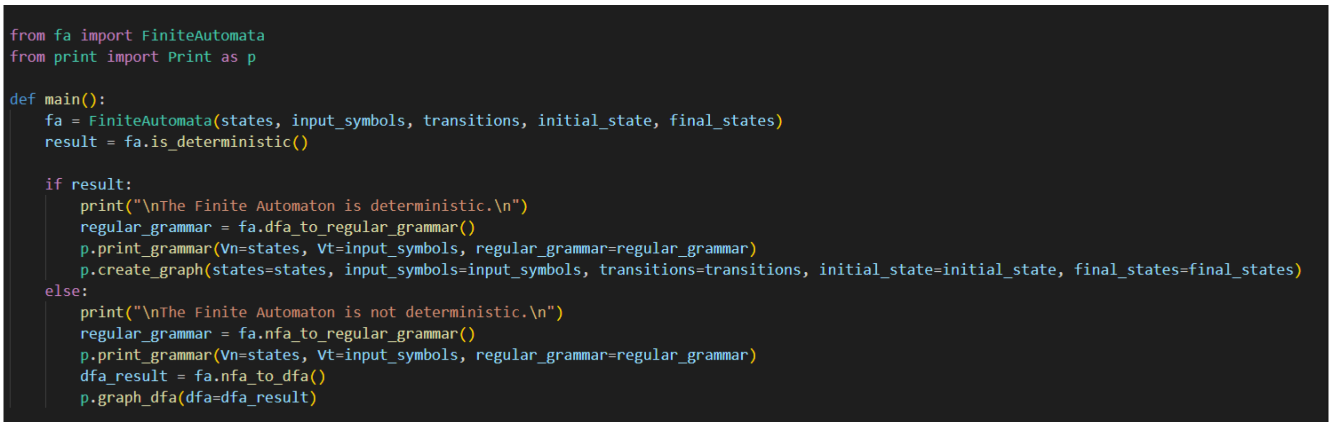
```python
    @staticmethod
    def graph_dfa(dfa):
        def map_states_to_characters(states):
            state_mapping = {}
            result = []
            for i, state in enumerate(states):
                char_representation = chr(ord('A') + i)
                state_mapping[state] = char_representation
                result.append(char_representation)
            return set(result), state_mapping

        # Use the function to map frozensets to characters
        dfa_states_representation, state_mapping = map_states_to_characters(dfa['states'])

        # Update the DFA transitions and final states with the mapped characters
        dfa['states'] = dfa_states_representation
        dfa['transitions'] = {
            state_mapping[state]: {
                symbol: state_mapping[next_state] for symbol, next_state in transitions.items()
            } for state, transitions in dfa['transitions'].items()
        }
        dfa['initial_state'] = state_mapping[dfa['initial_state']]
        dfa['final_states'] = {state_mapping[state] for state in dfa['final_states']}
        Print.create_graph(states=dfa['states'],
                           input_symbols=dfa['input_symbols'],
                           transitions=dfa['transitions'],
                           initial_state=dfa['initial_state'],
                           final_states=dfa['final_states'],)
```



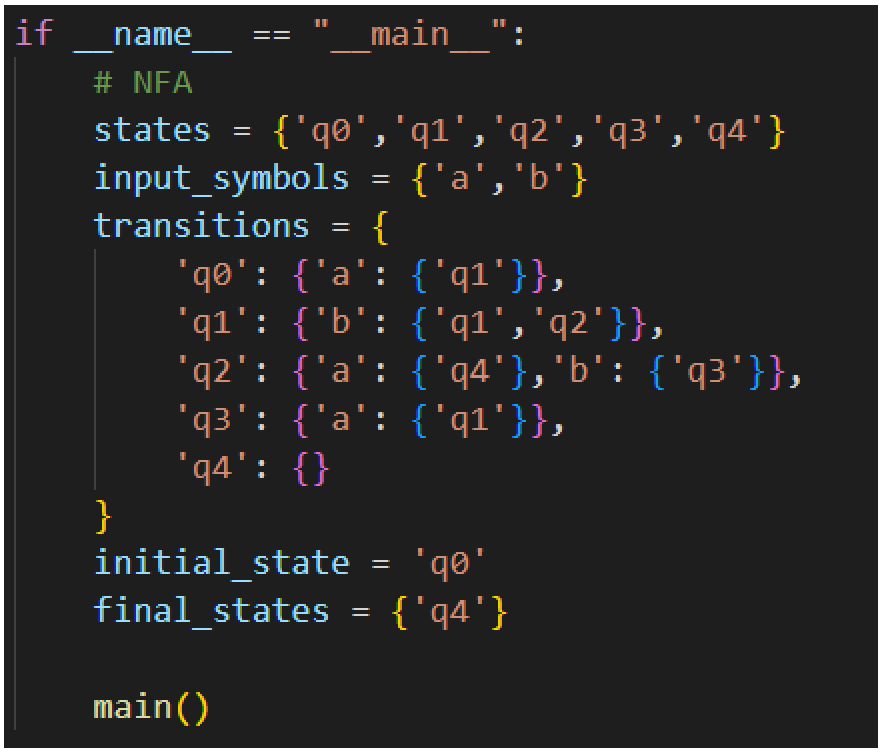
```python
    @staticmethod
    def create_graph(states, input_symbols, transitions, initial_state ,final_states):
        graph_dfa = VisualDFA(states=states,
                input_symbols=input_symbols,
                transitions=transitions,
                initial_state=initial_state,
                final_states=final_states)

        print("DFA Table:")
        print(graph_dfa.table)
        graph_dfa.show_diagram(filename="DFA")
        print("\nThe DFA was saved with the filename DFA.png")
```

The main:

The main function initializes a Finite Automaton (FA) based on the provided NFA specifications. It then checks whether the FA is deterministic using the is_deterministic method and proceeds to print the regular grammar and create a visual representation of the DFA. If the FA is not deterministic, it prints the regular grammar of the NFA and generates and displays a visual representation of the converted DFA.

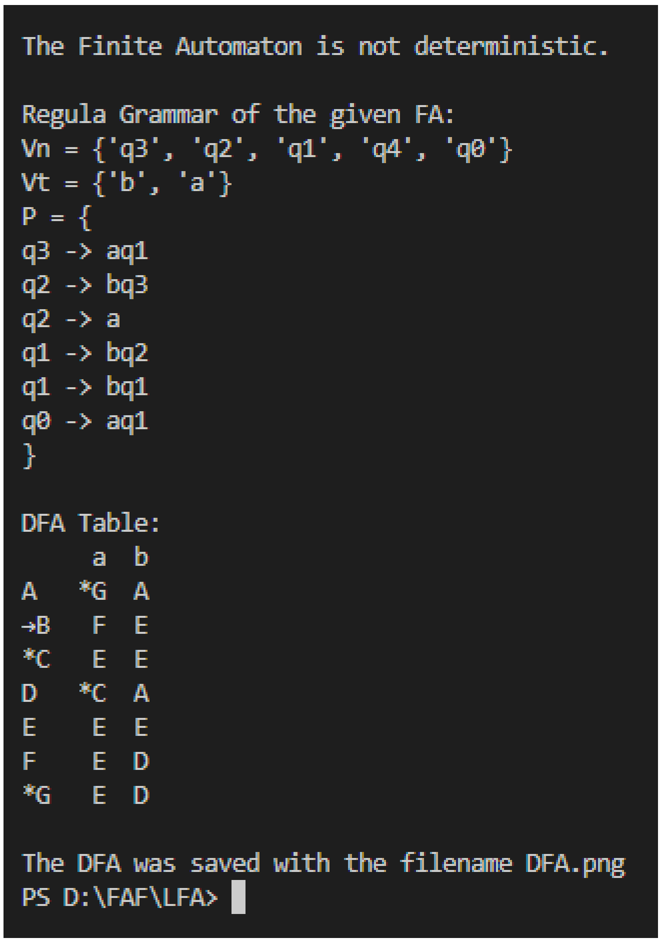


```python
from fa import FiniteAutomata
from print import Print as p

def main():
    fa = FiniteAutomata(states, input_symbols, transitions, initial_state, final_states)
    result = fa.is_deterministic()

    if result:
        print("\nThe Finite Automaton is deterministic.\n")
        regular_grammar = fa.dfa_to_regular_grammar()
        p.print_grammar(Vn=states, Vt=input_symbols, regular_grammar=regular_grammar)
        p.create_graph(states=states, input_symbols=input_symbols, transitions=transitions, initial_state=initial_state, final_states=final_states)
    else:
        print("\nThe Finite Automaton is not deterministic.\n")
        regular_grammar = fa.nfa_to_regular_grammar()
        p.print_grammar(Vn=states, Vt=input_symbols, regular_grammar=regular_grammar)
        dfa_result = fa.nfa_to_dfa()
        p.graph_dfa(dfa=dfa_result)
```

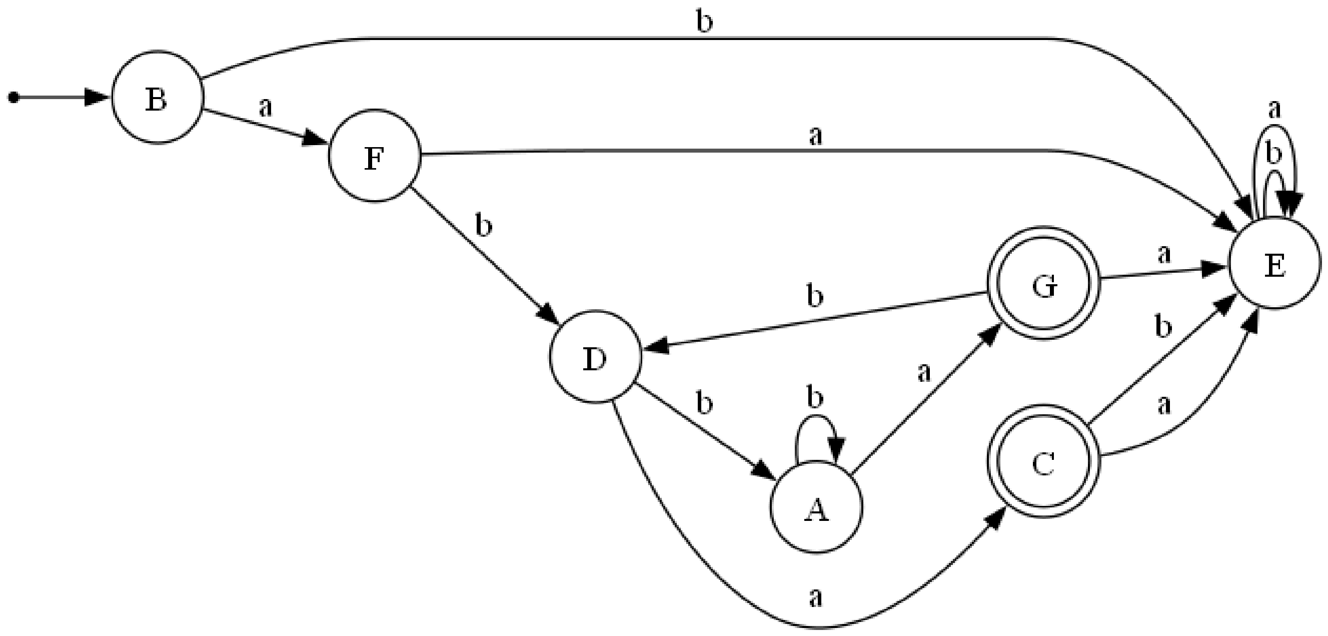


```python
if __name__ == "__main__":
    # NFA
    states = {'q0','q1','q2','q3','q4'}
    input_symbols = {'a','b'}
    transitions = {
        'q0': {'a': {'q1'}},
        'q1': {'b': {'q1','q2'}},
        'q2': {'a': {'q4'},'b': {'q3'}},
        'q3': {'a': {'q1'}},
        'q4': {}
    }
    initial_state = 'q0'
    final_states = {'q4'}

    main()
```

4. Program execution:



```
The Finite Automaton is not deterministic.

Regula Grammar of the given FA:
Vn = {'q3', 'q2', 'q1', 'q4', 'q0'}
Vt = {'b', 'a'}
P = {
q3 -> aq1
q2 -> bq3
q2 -> a
q1 -> bq2
q1 -> bq1
q0 -> aq1
}

DFA Table:
     a  b
A   *G  A
→B   F  E
*C   E  E
D   *C  A
E    E  E
F    E  D
*G   E  D

The DFA was saved with the filename DFA.png
PS D:\FAF\LFA>
```



DFA.png

5. Conclusion:

In conclusion, the laboratory work was a comprehensive exploration into finite automata (FA) and their properties using Python. The program developed during the lab has the capability to analyze an automaton, determine whether it is deterministic or not, and perform conversions between different representations. The key functionalities include checking determinism (is_deterministic method), converting FA to regular grammar (nfa_to_regular_grammar and dfa_to_regular_grammar methods), converting NFA to DFA (nfa_to_dfa method), printing FA tables, and generating visual graphs using the VisualDFA class.

This practical application allowed for a deeper understanding of automata theory, grammar, and the Python programming language. The ability to visualize and analyze automata through graphical representation is a valuable tool for both educational and practical purposes. The modular structure of the code, encapsulated in the FiniteAutomata class, promotes code readability and reusability, contributing to the efficiency of the implementation.

Overall, the laboratory work successfully combined theoretical concepts with practical implementation, providing hands-on experience in automata theory and algorithm development. The Python program developed in this lab is a versatile tool for automata analysis and visualization, with potential applications in various fields, including formal language processing and compiler design.