NFA TO DFA CONVERSION PROJECT

INTRODUCTION

- Automata Theory, also known as the Theory of Computation (TOC) , is a foundational field in computer science and mathematics that studies abstract machines and computational models. It focuses on how problems can be solved using algorithms and how machines (like Finite Automata, Pushdown Automata, and Turing Machines) process inputs and perform computations

- Understanding models such as NFA (Non-deterministic Finite Automata) and DFA (Deterministic Finite Automata) and the conversion between them plays a crucial role in areas like compiler design, lexical analysis, and pattern matching.

PROBLEM

1. Easier to Implement
DFA has exactly one transition for each symbol from a state.

2. Better Performance
DFA takes constant time per input symbol since it doesn’t need to explore multiple paths

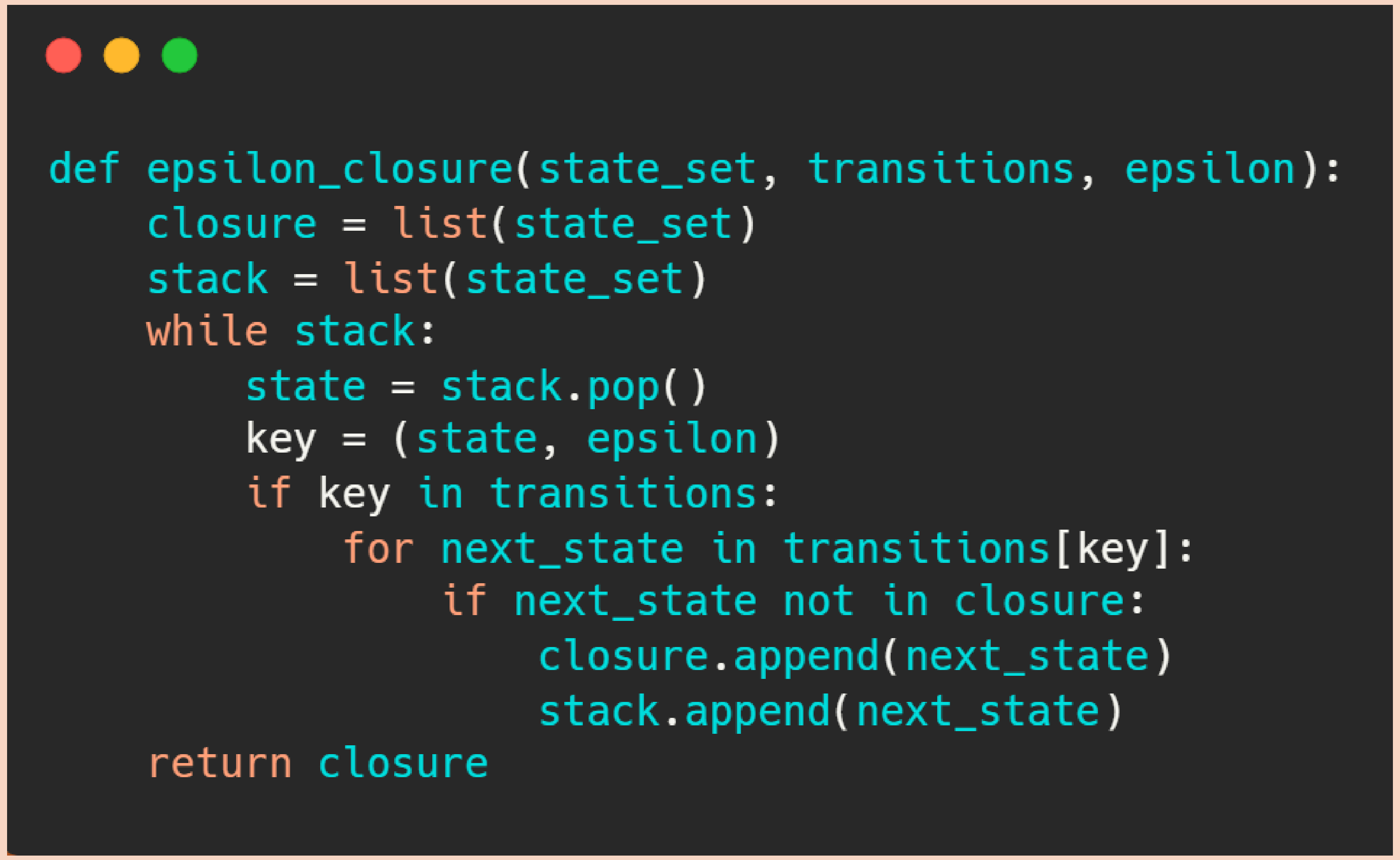
3. Easier to Analyze and Optimize
DFA can be minimized to reduce the number of states

4. No e -Transitions
NFA can include transitions that consume no input (e -transitions), making them harder to simulate

5. Practical Use in Real Systems
Tools like lexical analyzers, pattern matchers, and parsers rely on DFA due to their deterministic nature and reliability in real-time systems

CODE NFA TO DFA

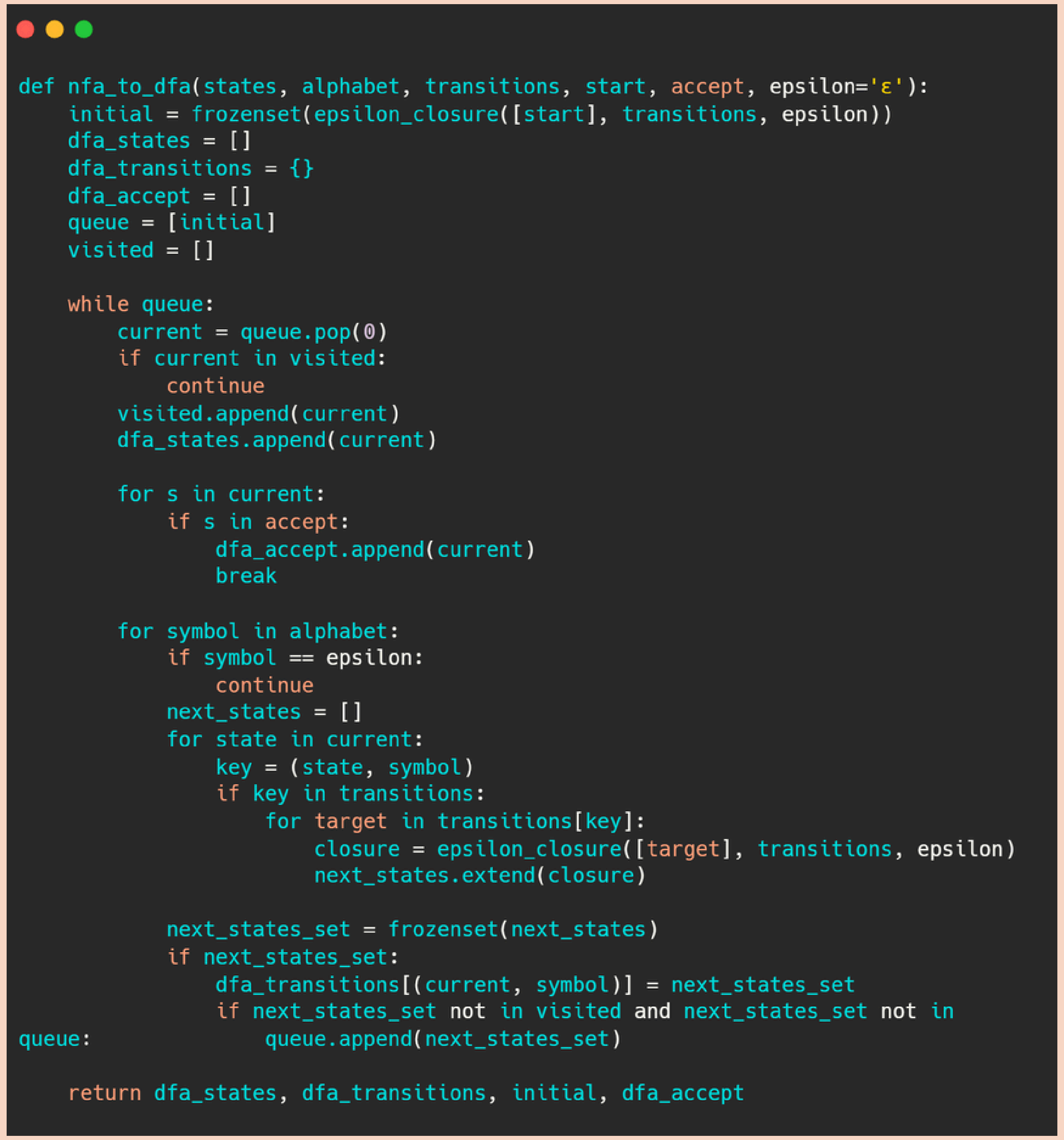
This function computes the ε-closure of a set of NFA states. It returns all states reachable from the given set via ε-transitions only, by recursively exploring ε-moves



```python
def epsilon_closure(state_set, transitions, epsilon):
    closure = list(state_set)
    stack = list(state_set)
    while stack:
        state = stack.pop()
        key = (state, epsilon)
        if key in transitions:
            for next_state in transitions[key]:
                if next_state not in closure:
                    closure.append(next_state)
                    stack.append(next_state)
    return closure
```

CODE NFA TO DFA

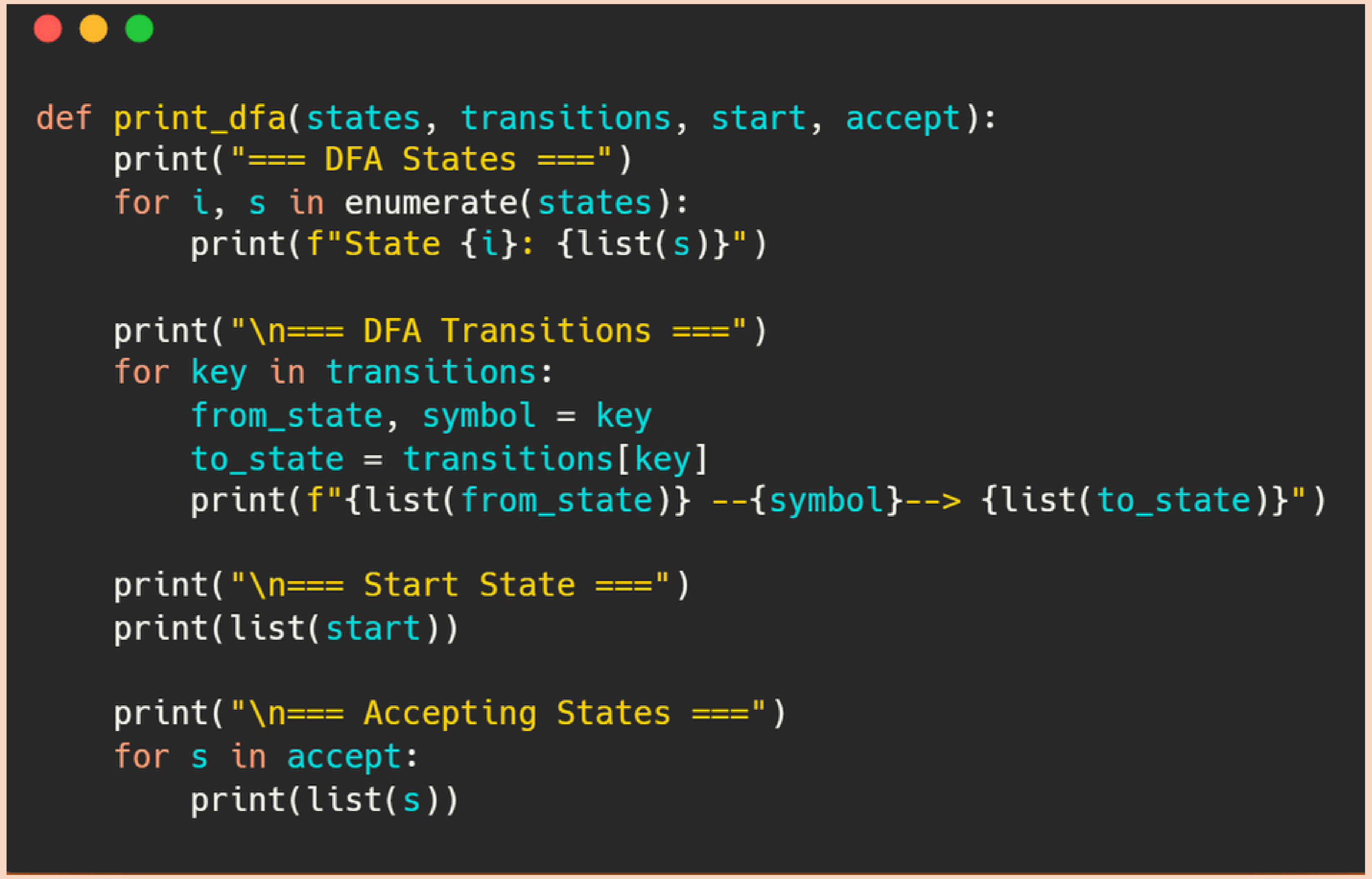
This function converts an NFA (with ε-transitions) to an equivalent DFA using the subset construction method. It builds new DFA states from ε-closures of NFA state sets and defines deterministic transitions accordingly



```
def nfa_to_dfa(states, alphabet, transitions, start, accept, epsilon='ε'):
    initial = frozenset(epsilon_closure([start], transitions, epsilon))
    dfa_states = []
    dfa_transitions = {}
    dfa_accept = []
    queue = [initial]
    visited = []

    while queue:
        current = queue.pop(0)
        if current in visited:
            continue
        visited.append(current)
        dfa_states.append(current)

        for s in current:
            if s in accept:
                dfa_accept.append(current)
                break

        for symbol in alphabet:
            if symbol == epsilon:
                continue
            next_states = []
            for state in current:
                key = (state, symbol)
                if key in transitions:
                    for target in transitions[key]:
                        closure = epsilon_closure([target], transitions, epsilon)
                        next_states.extend(closure)

            next_states_set = frozenset(next_states)
            if next_states_set:
                dfa_transitions[(current, symbol)] = next_states_set
                if next_states_set not in visited and next_states_set not in
queue:              queue.append(next_states_set)

    return dfa_states, dfa_transitions, initial, dfa_accept
```

CODE NFA TO DFA

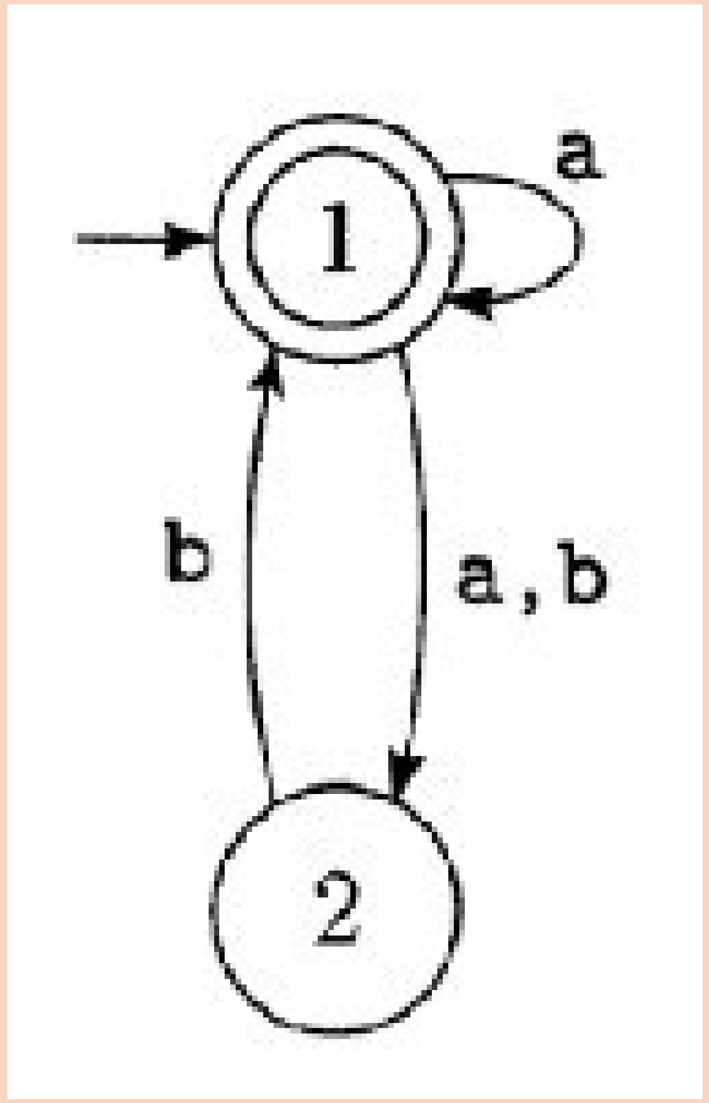
This function prints the components of a DFA, including its states, transitions, start state, and accepting states. It provides a readable summary of the resulting DFA structure

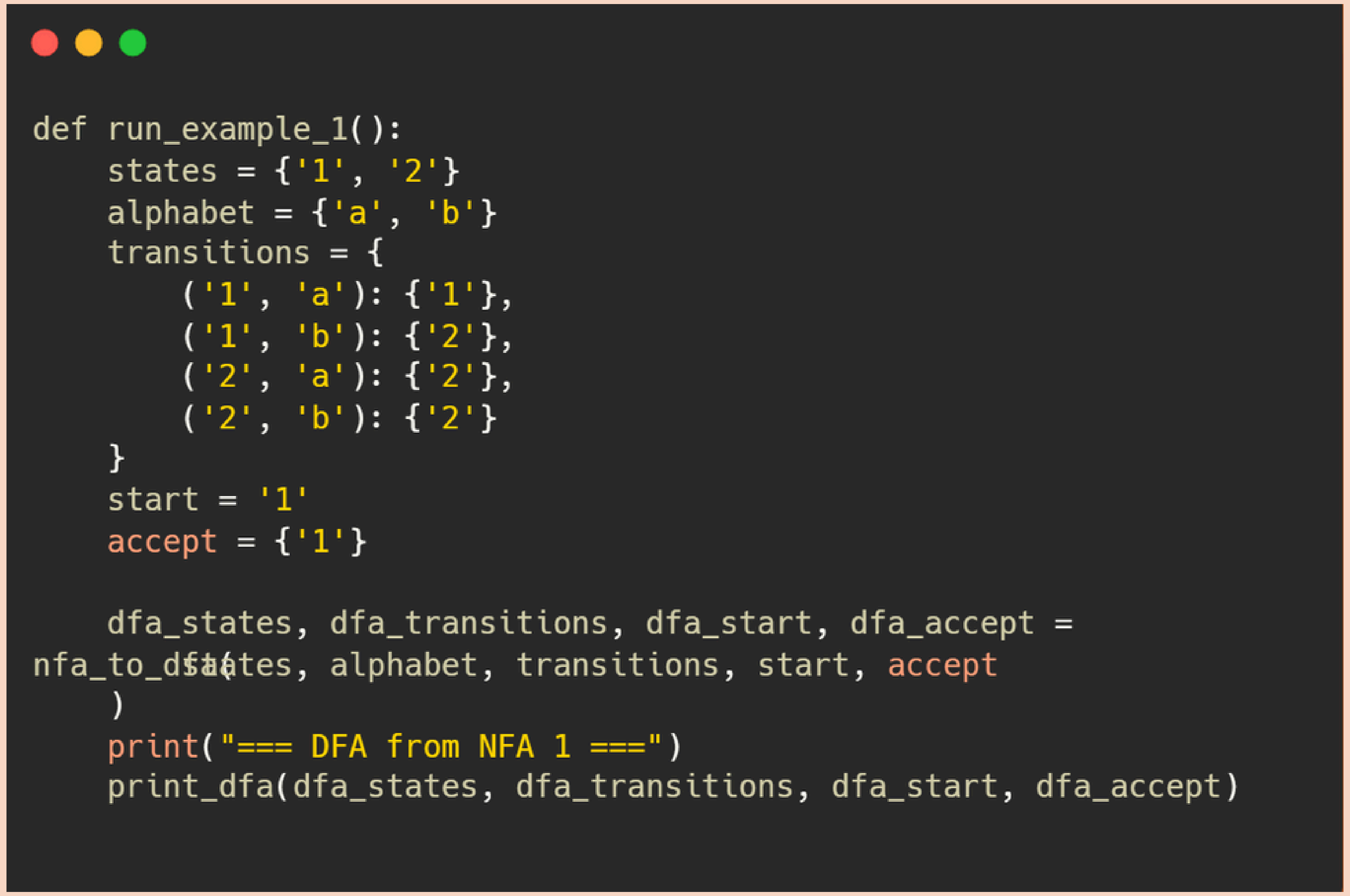


```python
def print_dfa(states, transitions, start, accept):
    print("=== DFA States ===")
    for i, s in enumerate(states):
        print(f"State {i}: {list(s)}")

    print("\n=== DFA Transitions ===")
    for key in transitions:
        from_state, symbol = key
        to_state = transitions[key]
        print(f"{list(from_state)} --{symbol}--> {list(to_state)}")

    print("\n=== Start State ===")
    print(list(start))

    print("\n=== Accepting States ===")
    for s in accept:
        print(list(s))
```

CODE NFA TO DFA

Example 1

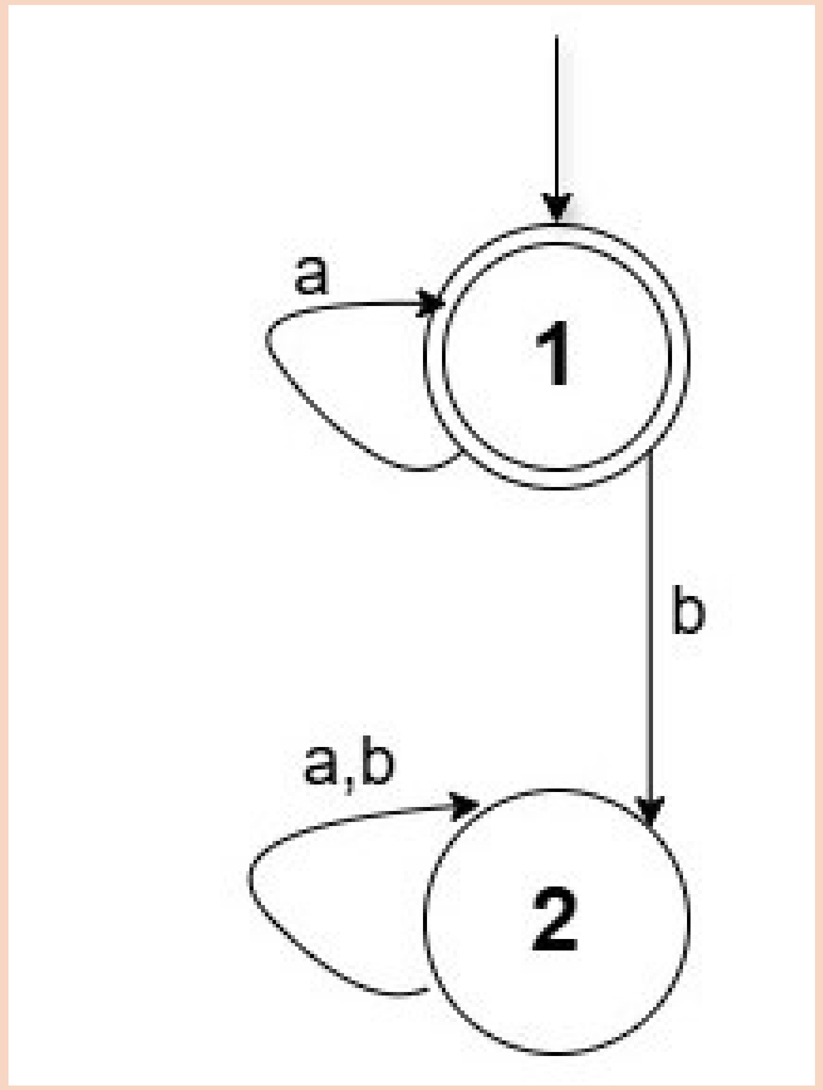


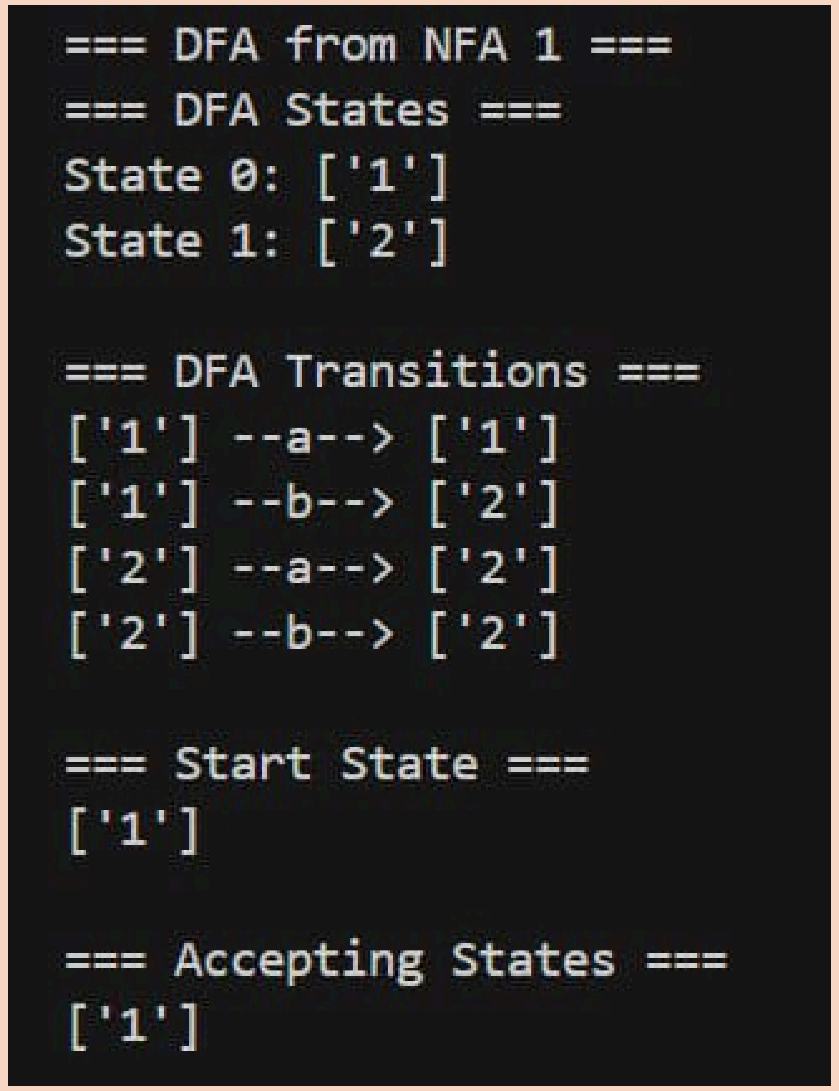


```
def run_example_1():
    states = {'1', '2'}
    alphabet = {'a', 'b'}
    transitions = {
        ('1', 'a'): {'1'},
        ('1', 'b'): {'2'},
        ('2', 'a'): {'2'},
        ('2', 'b'): {'2'}
    }
    start = '1'
    accept = {'1'}

    dfa_states, dfa_transitions, dfa_start, dfa_accept =
nfa_to_dfa(states, alphabet, transitions, start, accept
    )
    print("=== DFA from NFA 1 ===")
    print_dfa(dfa_states, dfa_transitions, dfa_start, dfa_accept)
```

RESULT NFA TO DFA

Example 1

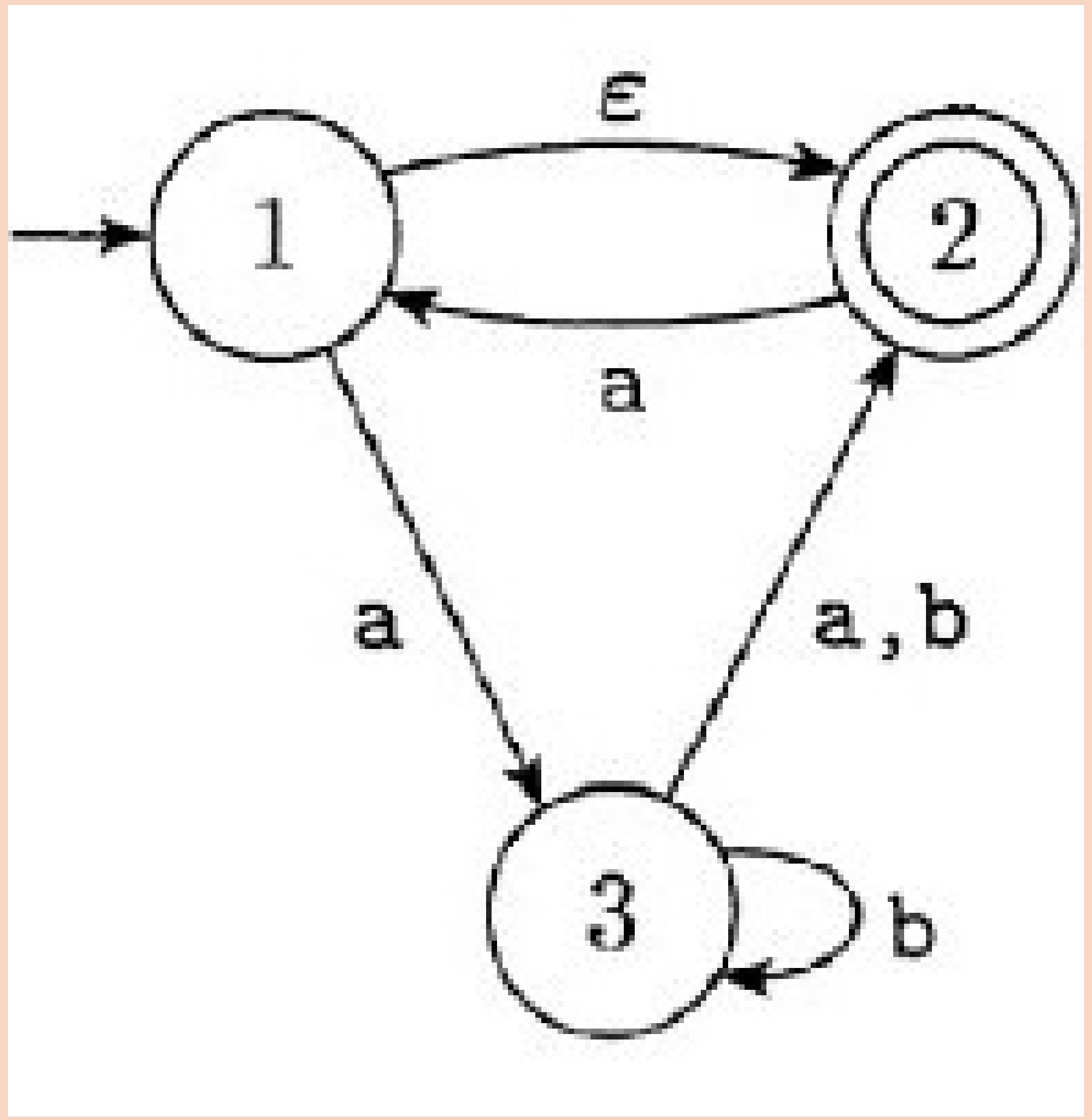


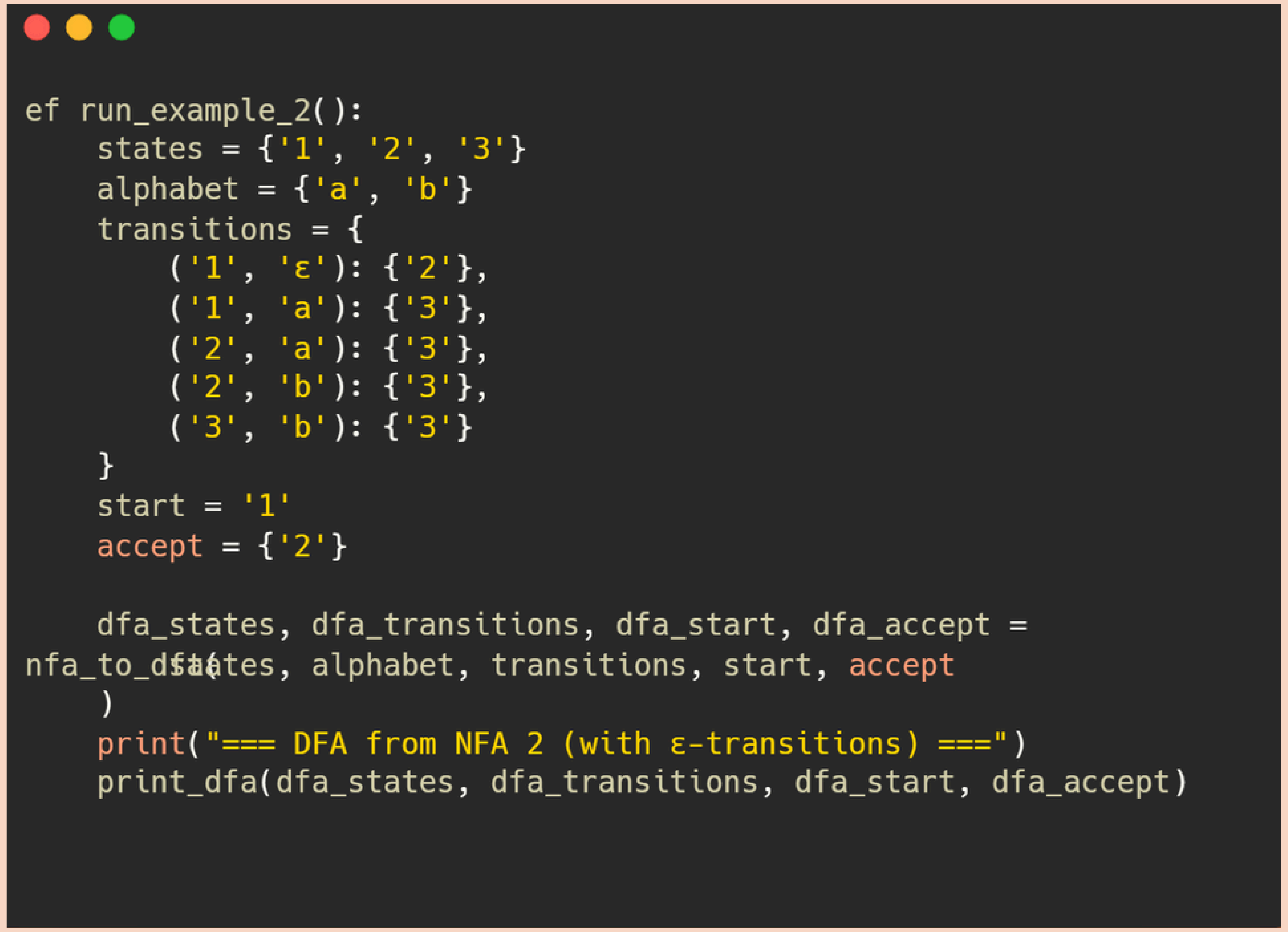


```
=== DFA from NFA 1 ===
=== DFA States ===
State 0: ['1']
State 1: ['2']

=== DFA Transitions ===
['1'] --a--> ['1']
['1'] --b--> ['2']
['2'] --a--> ['2']
['2'] --b--> ['2']

=== Start State ===
['1']

=== Accepting States ===
['1']
```

CODE NFA TO DFA

Example 1

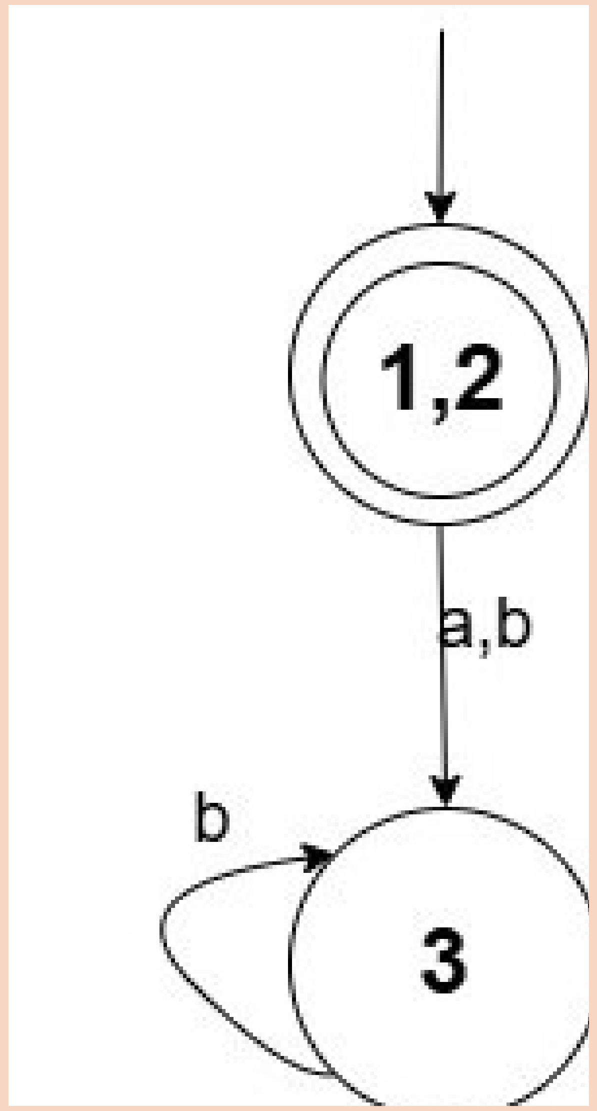


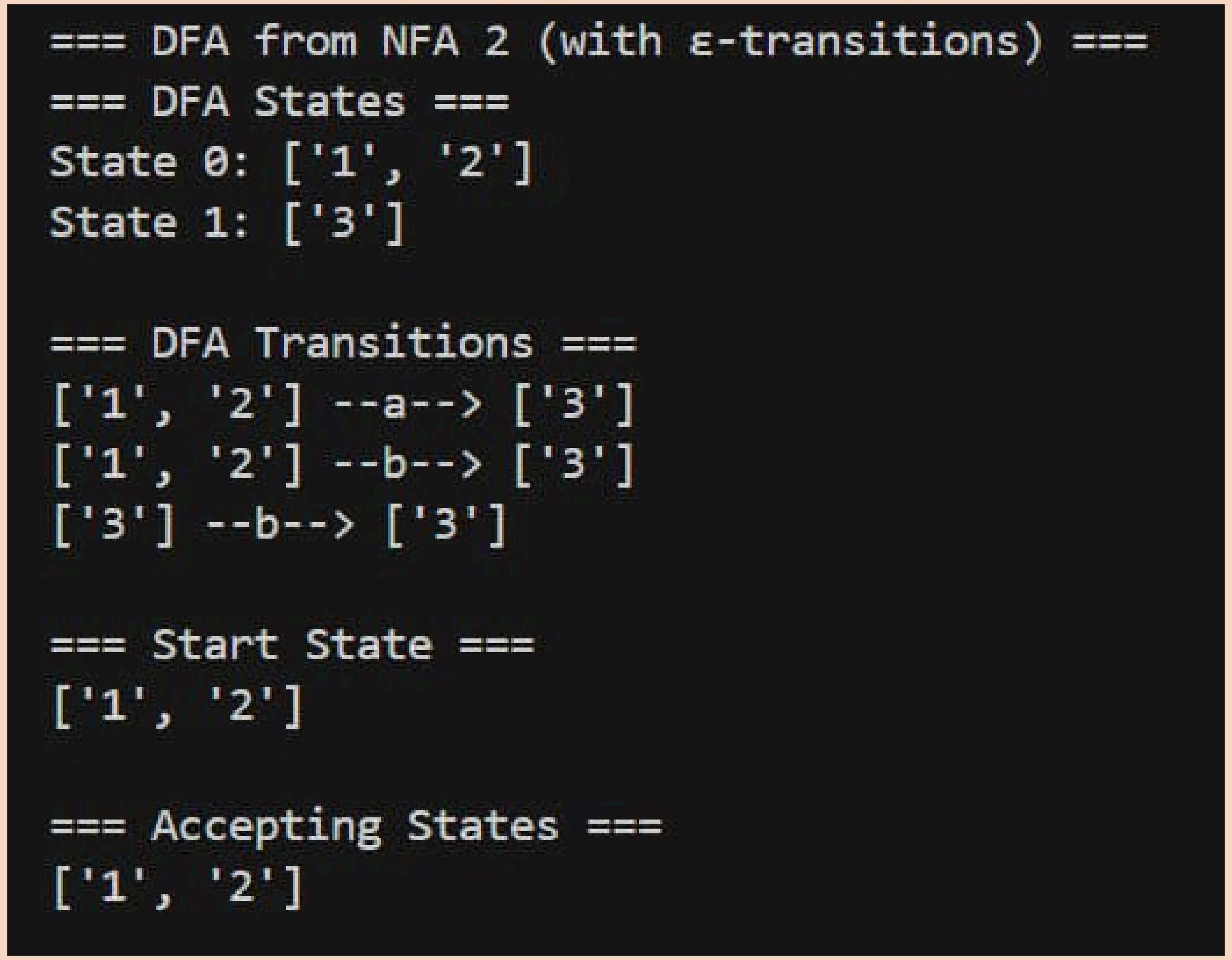


```
ef run_example_2():
    states = {'1', '2', '3'}
    alphabet = {'a', 'b'}
    transitions = {
        ('1', 'ε'): {'2'},
        ('1', 'a'): {'3'},
        ('2', 'a'): {'3'},
        ('2', 'b'): {'3'},
        ('3', 'b'): {'3'}
    }
    start = '1'
    accept = {'2'}

    dfa_states, dfa_transitions, dfa_start, dfa_accept =
nfa_to_dfa(states, alphabet, transitions, start, accept
    )
    print("=== DFA from NFA 2 (with ε-transitions) ===")
    print_dfa(dfa_states, dfa_transitions, dfa_start, dfa_accept)
```

RESULT NFA TO DFA

Example 2





```
=== DFA from NFA 2 (with ε-transitions) ===
=== DFA States ===
State 0: ['1', '2']
State 1: ['3']

=== DFA Transitions ===
['1', '2'] --a--> ['3']
['1', '2'] --b--> ['3']
['3'] --b--> ['3']

=== Start State ===
['1', '2']

=== Accepting States ===
['1', '2']
```

REAL-WORLD APPLICATION (1)

ATM Password Verification

When you insert your card, the system starts in the initial state (q0)
If the password is correct, it moves to an accepting state (q2)
If the password is wrong, it goes to a reject state (q1) and allows another try

When do we convert NFA to DFA?

We convert to DFA when implementing the ATM logic to ensure there is only one clear path for each input ,No ambiguity or guessing allowed

REAL-WORLD APPLICATION (2)

Lexical Analysis in Compilers

Keywords like if, for, and while are defined using regular expressions
These regex patterns are first converted to an NFA for simplicity
Then the NFA is converted into a DFA for faster and more efficient scanning

When do we convert NFA to DFA?

During the compiler implementation, after defining patterns with regex → we convert to DFA for performance and determinism in scanning source code

THANK YOU

For your attention

For Code : https://github.com/OmarAhmedWahby/NFA-to-DFA-Conversion