**Assignment -8**

**Formal Methods Lab**

1. DFA Implementation

```python

class DFA:

def \_\_init\_\_(self, states, alphabet, transition, start\_state, accept\_states):

"""

Initialize the DFA.

Parameters:

- states: set of states

- alphabet: set of input symbols

- transition: dict of dicts representing transition function

- start\_state: the initial state

- accept\_states: set of accepting states

"""

self.states = states

self.alphabet = alphabet

self.transition = transition

self.start\_state = start\_state

self.accept\_states = accept\_states

def accepts(self, input\_string):

"""Check if the DFA accepts the given input string."""

current\_state = self.start\_state

for symbol in input\_string:

if symbol not in self.alphabet:

return False # Invalid symbol in input

current\_state = self.transition[current\_state][symbol]

return current\_state in self.accept\_states

def is\_language\_empty(self):

"""Check if the language recognized by the DFA is empty."""

# Perform a BFS to see if any accept state is reachable

visited = set()

queue = [self.start\_state]

while queue:

state = queue.pop(0)

if state in self.accept\_states:

return False

visited.add(state)

for symbol in self.alphabet:

next\_state = self.transition[state][symbol]

if next\_state not in visited and next\_state not in queue:

queue.append(next\_state)

return True

def get\_reachable\_states(self):

"""Get all states reachable from the start state."""

reachable = set()

queue = [self.start\_state]

while queue:

state = queue.pop(0)

if state not in reachable:

reachable.add(state)

for symbol in self.alphabet:

next\_state = self.transition[state][symbol]

if next\_state not in reachable:

queue.append(next\_state)

return reachable

# Example usage:

if \_\_name\_\_ == "\_\_main\_\_":

# DFA that accepts strings with an even number of '1's

states = {'q0', 'q1'}

alphabet = {'0', '1'}

transition = {

'q0': {'0': 'q0', '1': 'q1'},

'q1': {'0': 'q1', '1': 'q0'}

}

start\_state = 'q0'

accept\_states = {'q0'}

dfa = DFA(states, alphabet, transition, start\_state, accept\_states)

test\_strings = ['', '0', '1', '00', '01', '10', '11', '000', '001', '010', '100', '110', '101', '011', '111']

for s in test\_strings:

print(f"'{s}': {dfa.accepts(s)}")

```

2. NFA Simulation and DFA Equivalence

```python

from collections import deque

class NFA:

def \_\_init\_\_(self, states, alphabet, transition, start\_state, accept\_states):

"""

Initialize the NFA.

Parameters:

- states: set of states

- alphabet: set of input symbols

- transition: dict of dicts representing transition function (values are sets)

- start\_state: the initial state

- accept\_states: set of accepting states

"""

self.states = states

self.alphabet = alphabet

self.transition = transition

self.start\_state = start\_state

self.accept\_states = accept\_states

def epsilon\_closure(self, states):

"""Compute the epsilon closure of a set of states."""

closure = set(states)

queue = deque(states)

while queue:

state = queue.popleft()

# Check for epsilon transitions (represented by 'ε' or '')

for next\_state in self.transition.get(state, {}).get('', set()):

if next\_state not in closure:

closure.add(next\_state)

queue.append(next\_state)

return closure

def accepts(self, input\_string):

"""Check if the NFA accepts the given input string."""

current\_states = self.epsilon\_closure({self.start\_state})

for symbol in input\_string:

if symbol not in self.alphabet:

return False

next\_states = set()

for state in current\_states:

next\_states.update(self.transition.get(state, {}).get(symbol, set()))

current\_states = self.epsilon\_closure(next\_states)

if not current\_states:

return False

return any(state in self.accept\_states for state in current\_states)

def to\_dfa(self):

"""Convert the NFA to an equivalent DFA using subset construction."""

from itertools import product

dfa\_states = set()

dfa\_transition = {}

dfa\_start = frozenset(self.epsilon\_closure({self.start\_state}))

dfa\_accept = set()

unprocessed\_states = [dfa\_start]

while unprocessed\_states:

current\_dfa\_state = unprocessed\_states.pop()

dfa\_states.add(current\_dfa\_state)

# Check if this DFA state contains any NFA accept states

if any(state in self.accept\_states for state in current\_dfa\_state):

dfa\_accept.add(current\_dfa\_state)

# Process each symbol in the alphabet

for symbol in self.alphabet:

next\_states = set()

for nfa\_state in current\_dfa\_state:

next\_states.update(self.transition.get(nfa\_state, {}).get(symbol, set()))

epsilon\_next = self.epsilon\_closure(next\_states)

if not epsilon\_next:

continue

dfa\_next\_state = frozenset(epsilon\_next)

if current\_dfa\_state not in dfa\_transition:

dfa\_transition[current\_dfa\_state] = {}

dfa\_transition[current\_dfa\_state][symbol] = dfa\_next\_state

if dfa\_next\_state not in dfa\_states and dfa\_next\_state not in unprocessed\_states:

unprocessed\_states.append(dfa\_next\_state)

# Convert frozenset states to more readable names (q0, q1, etc.)

state\_map = {state: f'q{i}' for i, state in enumerate(dfa\_states)}

mapped\_transition = {}

for state in dfa\_transition:

mapped\_state = state\_map[state]

mapped\_transition[mapped\_state] = {}

for symbol in dfa\_transition[state]:

mapped\_transition[mapped\_state][symbol] = state\_map[dfa\_transition[state][symbol]]

mapped\_start = state\_map[dfa\_start]

mapped\_accept = {state\_map[state] for state in dfa\_accept}

mapped\_states = set(state\_map.values())

return DFA(mapped\_states, self.alphabet, mapped\_transition, mapped\_start, mapped\_accept)

# Example usage:

if \_\_name\_\_ == "\_\_main\_\_":

# NFA that accepts strings ending with '01'

states = {'q0', 'q1', 'q2'}

alphabet = {'0', '1'}

transition = {

'q0': {

'0': {'q0', 'q1'},

'1': {'q0'}

},

'q1': {

'1': {'q2'}

}

}

start\_state = 'q0'

accept\_states = {'q2'}

nfa = NFA(states, alphabet, transition, start\_state, accept\_states)

# Test some strings

test\_strings = ['01', '001', '101', '0001', '1001', '0101', '1010101', '1', '0', '10', '00', '11']

for s in test\_strings:

print(f"'{s}': {nfa.accepts(s)}")

# Convert to DFA and test equivalence

dfa = nfa.to\_dfa()

print("\nDFA states:", dfa.states)

print("DFA accept states:", dfa.accept\_states)

print("DFA transition:", dfa.transition)

# Verify equivalence

print("\nTesting equivalence:")

for s in test\_strings:

nfa\_result = nfa.accepts(s)

dfa\_result = dfa.accepts(s)

print(f"'{s}': NFA={nfa\_result}, DFA={dfa\_result}, {'Match' if nfa\_result == dfa\_result else 'Mismatch'}")

```

3. Regular Expression to Automaton

```python

class RegexToAutomaton:

"""Convert regular expressions to equivalent NFAs using Thompson's construction."""

def \_\_init\_\_(self):

self.state\_counter = 0

def new\_state(self):

"""Generate a new state name."""

state = f'q{self.state\_counter}'

self.state\_counter += 1

return state

def basic\_nfa(self, symbol):

"""Create a basic NFA for a single symbol."""

start = self.new\_state()

accept = self.new\_state()

transition = {

start: {symbol: {accept}}

}

return {

'states': {start, accept},

'alphabet': {symbol},

'transition': transition,

'start\_state': start,

'accept\_states': {accept}

}

def union\_nfa(self, nfa1, nfa2):

"""Create the union of two NFAs (a|b)."""

start = self.new\_state()

accept = self.new\_state()

# Combine all components

transition = {\*\*nfa1['transition'], \*\*nfa2['transition']}

transition[start] = {'': {nfa1['start\_state'], nfa2['start\_state']}}

# Add epsilon transitions from old accept states to new accept state

for state in nfa1['accept\_states']:

if state not in transition:

transition[state] = {}

transition[state][''] = transition[state].get('', set()) | {accept}

for state in nfa2['accept\_states']:

if state not in transition:

transition[state] = {}

transition[state][''] = transition[state].get('', set()) | {accept}

return {

'states': {start, accept} | nfa1['states'] | nfa2['states'],

'alphabet': nfa1['alphabet'] | nfa2['alphabet'],

'transition': transition,

'start\_state': start,

'accept\_states': {accept}

}

def concat\_nfa(self, nfa1, nfa2):

"""Create the concatenation of two NFAs (ab)."""

transition = {\*\*nfa1['transition'], \*\*nfa2['transition']}

# Connect accept states of nfa1 to start state of nfa2 with epsilon transitions

for state in nfa1['accept\_states']:

if state not in transition:

transition[state] = {}

transition[state][''] = transition[state].get('', set()) | {nfa2['start\_state']}

return {

'states': nfa1['states'] | nfa2['states'],

'alphabet': nfa1['alphabet'] | nfa2['alphabet'],

'transition': transition,

'start\_state': nfa1['start\_state'],

'accept\_states': nfa2['accept\_states']

}

def star\_nfa(self, nfa):

"""Create the Kleene star of an NFA (a\*)."""

start = self.new\_state()

accept = self.new\_state()

transition = nfa['transition'].copy()

transition[start] = {'': {nfa['start\_state'], accept}}

# Add epsilon transitions from old accept states to new accept state and start state

for state in nfa['accept\_states']:

if state not in transition:

transition[state] = {}

transition[state][''] = transition[state].get('', set()) | {accept, nfa['start\_state']}

return {

'states': {start, accept} | nfa['states'],

'alphabet': nfa['alphabet'],

'transition': transition,

'start\_state': start,

'accept\_states': {accept}

}

def parse\_regex(self, regex):

"""Parse a regular expression and build the equivalent NFA."""

import re

# Remove whitespace

regex = re.sub(r'\s', '', regex)

# Handle parentheses and operators in correct precedence

return self.\_parse\_expression(regex)

def \_parse\_expression(self, s):

"""Parse a full expression (handles | operator with lowest precedence)."""

nfa = self.\_parse\_sequence(s)

# Look for union operators

while True:

match = re.search(r'\|', s)

if not match:

break

pos = match.start()

left = s[:pos]

right = s[pos+1:]

left\_nfa = self.\_parse\_sequence(left)

right\_nfa = self.\_parse\_expression(right)

nfa = self.union\_nfa(left\_nfa, right\_nfa)

s = right

return nfa

def \_parse\_sequence(self, s):

"""Parse a sequence of concatenated items."""

if not s:

return self.basic\_nfa('') # Empty string

# Find the first atom

first, remaining = self.\_parse\_atom(s)

nfa = first

# Parse remaining atoms in sequence

while remaining:

next\_nfa, remaining = self.\_parse\_atom(remaining)

nfa = self.concat\_nfa(nfa, next\_nfa)

return nfa

def \_parse\_atom(self, s):

"""Parse an atom (character, parenthesized expression, or starred expression)."""

if not s:

return None, ''

if s[0] == '(':

# Find matching parenthesis

depth = 1

for i in range(1, len(s)):

if s[i] == '(':

depth += 1

elif s[i] == ')':

depth -= 1

if depth == 0:

inner = s[1:i]

remaining = s[i+1:]

nfa = self.\_parse\_expression(inner)

# Check for star operator

if remaining and remaining[0] == '\*':

nfa = self.star\_nfa(nfa)

remaining = remaining[1:]

return nfa, remaining

raise ValueError("Unmatched parentheses")

else:

# Single character

char = s[0]

remaining = s[1:]

# Check for star operator

if remaining and remaining[0] == '\*':

nfa = self.star\_nfa(self.basic\_nfa(char))

return nfa, remaining[1:]

else:

return self.basic\_nfa(char), remaining

# Example usage:

if \_\_name\_\_ == "\_\_main\_\_":

converter = RegexToAutomaton()

# Test with various regular expressions

test\_regexes = [

'a', # single character

'ab', # concatenation

'a|b', # union

'a\*', # Kleene star

'(a|b)\*', # combination

'a(b|c)\*d', # more complex

]

for regex in test\_regexes:

print(f"\nRegular expression: {regex}")

nfa\_dict = converter.parse\_regex(regex)

nfa = NFA(

states=nfa\_dict['states'],

alphabet=nfa\_dict['alphabet'],

transition=nfa\_dict['transition'],

start\_state=nfa\_dict['start\_state'],

accept\_states=nfa\_dict['accept\_states']

)

# Convert to DFA for easier testing

dfa = nfa.to\_dfa()

# Test some strings

test\_strings = {

'a': ['', 'a', 'aa', 'b', 'ab', 'ba'],

'ab': ['', 'a', 'b', 'ab', 'ba', 'aba', 'abb'],

'a|b': ['', 'a', 'b', 'aa', 'bb', 'ab'],

'a\*': ['', 'a', 'aa', 'aaa', 'b', 'ab'],

'(a|b)\*': ['', 'a', 'b', 'aa', 'bb', 'ab', 'ba', 'aab', 'aba'],

'a(b|c)\*d': ['ad', 'abd', 'acd', 'abbd', 'accd', 'abcd', 'a', 'ab', 'ac', 'abc']

}

if regex in test\_strings:

print(f"Testing strings for {regex}:")

for s in test\_strings[regex]:

print(f" '{s}': NFA={nfa.accepts(s)}, DFA={dfa.accepts(s)}")

```

4. Text Parser with Formal Grammar

```python

class GrammarParser:

"""A simple parser based on formal grammar rules."""

def \_\_init\_\_(self, grammar):

"""

Initialize with a grammar.

Grammar should be a dict with:

- keys: non-terminal symbols

- values: list of possible productions (each production is a list of symbols)

"""

self.grammar = grammar

self.start\_symbol = next(iter(grammar)) # First non-terminal is start symbol

def parse(self, input\_tokens):

"""Parse the input tokens using the grammar."""

from collections import deque

# Initialize parse stack with start symbol

stack = [self.start\_symbol]

input\_queue = deque(input\_tokens)

derivation = []

while stack and input\_queue:

top = stack[-1]

if top in self.grammar:

# Non-terminal - expand using grammar

stack.pop()

production = self.grammar[top][0] # Simple: always use first production

stack.extend(reversed(production)) # Push in reverse order

derivation.append(f"{top} -> {' '.join(production)}")

else:

# Terminal - match with input

if top == input\_queue[0]:

stack.pop()

input\_queue.popleft()

else:

return False, derivation # Mismatch

# Check if both stack and input are empty

success = not stack and not input\_queue

return success, derivation

def to\_pda(self):

"""Convert the grammar to an equivalent Pushdown Automaton (PDA)."""

# This is a simplified conversion for CFGs

states = {'q0', 'q1', 'q2'}

alphabet = set()

stack\_alphabet = set()

transition = {}

start\_state = 'q0'

accept\_states = {'q2'}

# Collect all terminals and non-terminals

for nt in self.grammar:

stack\_alphabet.add(nt)

for production in self.grammar[nt]:

for symbol in production:

if symbol not in self.grammar: # Terminal

alphabet.add(symbol)

stack\_alphabet.add(symbol)

# Define transitions

# Initial transition: push start symbol and go to q1

transition[('q0', '', '')] = [('q1', self.start\_symbol)]

# Processing transitions

for nt in self.grammar:

for production in self.grammar[nt]:

# For each production A -> α, add (q1, ε, A) → (q1, α)

transition[('q1', '', nt)] = transition.get(('q1', '', nt), []) + [('q1', production)]

# Terminal matching transitions

for t in alphabet:

transition[('q1', t, t)] = [('q1', '')]

# Accepting transition

transition[('q1', '', '')] = [('q2', '')]

return {

'states': states,

'alphabet': alphabet,

'stack\_alphabet': stack\_alphabet,

'transition': transition,

'start\_state': start\_state,

'accept\_states': accept\_states

}

# Example usage:

if \_\_name\_\_ == "\_\_main\_\_":

# Simple arithmetic expression grammar

grammar = {

'E': [['T', 'E\'']],

'E\'': [['+', 'T', 'E\''], ['']],

'T': [['F', 'T\'']],

'T\'': [['\*', 'F', 'T\''], ['']],

'F': [['(', 'E', ')'], ['id']]

}

parser = GrammarParser(grammar)

# Test parsing

test\_cases = [

['id'],

['id', '+', 'id'],

['id', '\*', 'id'],

['(', 'id', '+', 'id', ')', '\*', 'id'],

['id', '+', 'id', '\*', 'id']

]

for tokens in test\_cases:

success, derivation = parser.parse(tokens)

print(f"\nInput: {' '.join(tokens)}")

print("Success:", success)

print("Derivation:")

for step in derivation:

print(" ", step)

# Convert to PDA

pda = parser.to\_pda()

print("\nPDA representation:")

print("States:", pda['states'])

print("Alphabet:", pda['alphabet'])

print("Stack Alphabet:", pda['stack\_alphabet'])

print("Start State:", pda['start\_state'])

print("Accept States:", pda['accept\_states'])

print("Transitions:")

for key, value in pda['transition'].items():

print(f" {key}: {value}")

```

5. FSM Minimization and Equivalence Checking

```python

class FSMMinimizer:

"""Implement minimization of finite state machines and equivalence checking."""

@staticmethod

def minimize\_dfa(dfa):

"""Minimize a DFA using Hopcroft's algorithm."""

# Initial partition: accept and non-accept states

P = [set(dfa.accept\_states), set(dfa.states) - set(dfa.accept\_states)]

W = [set(dfa.accept\_states), set(dfa.states) - set(dfa.accept\_states)]

while W:

A = W.pop(0)

for c in dfa.alphabet:

# Find all states that transition to A on c

X = set()

for state in dfa.states:

if dfa.transition[state][c] in A:

X.add(state)

if not X:

continue

# Refine each partition Y in P

new\_P = []

for Y in P:

intersect = Y & X

difference = Y - X

if intersect and difference:

new\_P.append(intersect)

new\_P.append(difference)

if Y in W:

W.remove(Y)

W.append(intersect)

W.append(difference)

else:

if len(intersect) <= len(difference):

W.append(intersect)

else:

W.append(difference)

else:

new\_P.append(Y)

P = new\_P

# Create new states based on partitions

state\_map = {}

new\_states = set()

new\_accept\_states = set()

new\_transition = {}

for i, partition in enumerate(P):

new\_state = f'q{i}'

for state in partition:

state\_map[state] = new\_state

new\_states.add(new\_state)

# Check if this partition contains the start state

if dfa.start\_state in partition:

new\_start\_state = new\_state

# Check if this partition contains any accept states

if partition & set(dfa.accept\_states):

new\_accept\_states.add(new\_state)

# Build new transition function

for new\_state in new\_states:

# Pick any state from the original partition

for old\_state, mapped\_state in state\_map.items():

if mapped\_state == new\_state:

representative = old\_state

break

new\_transition[new\_state] = {}

for c in dfa.alphabet:

old\_target = dfa.transition[representative][c]

new\_target = state\_map[old\_target]

new\_transition[new\_state][c] = new\_target

return DFA(new\_states, dfa.alphabet, new\_transition, new\_start\_state, new\_accept\_states)

@staticmethod

def are\_equivalent(dfa1, dfa2):

"""Check if two DFAs are equivalent using product construction."""

# First check if alphabets are the same

if dfa1.alphabet != dfa2.alphabet:

return False

# Create product automaton

from itertools import product

product\_states = set(product(dfa1.states, dfa2.states))

product\_transition = {}

product\_start = (dfa1.start\_state, dfa2.start\_state)

# Check if initial states have different acceptance

if (dfa1.start\_state in dfa1.accept\_states) != (dfa2.start\_state in dfa2.accept\_states):

return False

# Build transition function

for (state1, state2) in product\_states:

product\_transition[(state1, state2)] = {}

for c in dfa1.alphabet:

next1 = dfa1.transition[state1][c]

next2 = dfa2.transition[state2][c]

product\_transition[(state1, state2)][c] = (next1, next2)

# Perform BFS to see if we can reach a state where one is accepting and the other is not

visited = set()

queue = [product\_start]

while queue:

current = queue.pop(0)

if current in visited:

continue

visited.add(current)

state1, state2 = current

# Check if one is accepting and the other is not

if (state1 in dfa1.accept\_states) != (state2 in dfa2.accept\_states):

return False

for c in dfa1.alphabet:

next\_state = product\_transition[current][c]

if next\_state not in visited:

queue.append(next\_state)

return True

# Example usage:

if \_\_name\_\_ == "\_\_main\_\_":

# Create two equivalent DFAs (minimized and non-minimized versions)

# DFA that accepts strings with an even number of '1's

states1 = {'q0', 'q1'}

alphabet1 = {'0', '1'}

transition1 = {

'q0': {'0': 'q0', '1': 'q1'},

'q1': {'0': 'q1', '1': 'q0'}

}

start\_state1 = 'q0'

accept\_states1 = {'q0'}

dfa1 = DFA(states1, alphabet1, transition1, start\_state1, accept\_states1)

# A non-minimized version of the same DFA

states2 = {'q0', 'q1', 'q2', 'q3'}

alphabet2 = {'0', '1'}

transition2 = {

'q0': {'0': 'q0', '1': 'q1'},

'q1': {'0': 'q1', '1': 'q2'},

'q2': {'0': 'q2', '1': 'q3'},

'q3': {'0': 'q3', '1': 'q0'}

}

start\_state2 = 'q0'

accept\_states2 = {'q0', 'q3'}

dfa2 = DFA(states2, alphabet2, transition2, start\_state2, accept\_states2)

# Minimize the second DFA

minimizer = FSMMinimizer()

minimized\_dfa2 = minimizer.minimize\_dfa(dfa2)

print("Original DFA1 states:", dfa1.states)

print("Original DFA1 accept states:", dfa1.accept\_states)

print("Original DFA1 transition:", dfa1.transition)

print("\nOriginal DFA2 states:", dfa2.states)

print("Original DFA2 accept states:", dfa2.accept\_states)

print("Original DFA2 transition:", dfa2.transition)

print("\nMinimized DFA2 states:", minimized\_dfa2.states)

print("Minimized DFA2 accept states:", minimized\_dfa2.accept\_states)

print("Minimized DFA2 transition:", minimized\_dfa2.transition)

# Check equivalence

print("\nAre DFA1 and DFA2 equivalent?", minimizer.are\_equivalent(dfa1, dfa2))

print("Are DFA1 and minimized DFA2 equivalent?", minimizer.are\_equivalent(dfa1, minimized\_dfa2))

# Test with some strings

test\_strings = ['', '0', '1', '00', '01', '10', '11', '000', '001', '010', '100', '110', '101', '011', '111']

print("\nTesting strings:")

for s in test\_strings:

r1 = dfa1.accepts(s)

r2 = dfa2.accepts(s)

r3 = minimized\_dfa2.accepts(s)

print(f"'{s}': DFA1={r1}, DFA2={r2}, MinDFA2={r3}")

```.