DETERMINISTIC FINITE AUTOMATON (DFA) PROJECT

FINAL REPORT

Section 1. SUMMARY OF THE DETERMINISTIC FINITE AUTOMATON

1.1 DESCRIPTION OF THE DFA

My project implements a(DFA) simulator in C++.
It consists of:
- A DFA class that loads DFA specifications from files and simulates string acceptance
- Input validation and error handling
- An interactive command-line simulator for testing DFA behavior

1.2 WHAT DOES IT DO?

1. Reads a DFA specification from a text file containing:
   - A set of states (Q)
   - An alphabet of symbols (Σ)
   - An initial state (q0)
   - A set of accepting/final states (F)
   - A transition function (δ) defined as rules

2. Validates the DFA structure to ensure:
   - The initial state exists in the state set
   - All accepting states exist in the state set
   - All transitions reference valid states and symbols

3. Tests string acceptance by:
   - Starting at the initial state
   - Following transitions for each input symbol
   - Accepting the string if it ends in a final state

4. Provides interactive commands:
   - load <file>: Load a DFA from a file
   - readstring <string>: Test if a string is accepted

- visualize <string>: Display state transitions step-by-step
- display: Show the DFA formal representation
- exit: Exit the simulator

1.3 DATA STRUCTURES USED

The implementation uses the following data structures:

1. vector<string> Q
   - Type: Dynamic array of strings
   - Purpose: Stores all states in the DFA

2. vector<char> sigma
   - Type: Dynamic array of characters
   - Purpose: Stores all symbols in the alphabet

3. string q0
   - Type: String
   - Purpose: Stores the initial/start state

4. vector<string> F
   - Type: Dynamic array of strings
   - Purpose: Stores all accepting/final states

5. map<pair<string,char>, string> delta
   - Type: Hash map with key (state, symbol) and value (next state)
   - Purpose: Stores the transition function (δ)
   - Key structure: pair<string,char> where first is current state, second is symbol
   - Value: Next state to transition to

Section 2. CLASS DOCUMENTATION

2.1 CLASS: DFA

Description:
The DFA class represents a Deterministic Finite Automaton. It inherits from tcp::Object to support string output formatting. The class encapsulates the

five components of a formal DFA: states, alphabet, initial state, accepting states, and transition function.

PUBLIC FIELDS:
None (all members are private)

PRIVATE FIELDS:
- vector<string> Q: Set of states
- vector<char> Sigma: Alphabet of input symbols
- string q0: Initial state
- vector<string> F: Set of accepting/final states
- map<pair<string,char>, string> delta: Transition function

2.2 METHOD DOCUMENTATION

METHOD: DFA()
Return Type: (Constructor)

PRECONDITIONS:
- None

PSEUDOCODE:
```
    FUNCTION DFA()
      Q.clear()
      Sigma.clear()
      q0 = ""
      F.clear()
     delta.clear()
    END FUNCTION
```

DESCRIPTION:
Initializes a new DFA object with empty state set, alphabet, initial state, final states, and transition function. All vectors and maps are cleared to have a clean state.

METHOD: loadFromFile(const string& filename)
Return Type: bool
Parameters: string filename - the path to the DFA specification file

PRECONDITIONS:
- The file must exist and be readable
- The file must contain DFA specification in the correct format:
  Line 1: Space-separated states
  Line 2: Symbol characters (no spaces)
  Line 3: Initial state name

```
  Line 4: Space-separated final states
  Lines 5+: Transitions in format "state symbol nextstate"

PSEUDOCODE:
    FUNCTION loadFromFile(filename: string) RETURNS boolean
        Open file at filename
        IF file cannot be opened THEN
            Print error message
            RETURN false
        END IF

        Clear all DFA components: Q, Sigma, q0, F, delta

        step = 0
        FOR EACH line in file DO
            Remove leading and trailing whitespace
            IF line is empty THEN continue END IF

            IF step == 0 THEN    // Parse states Q
                Extract space-separated states
                FOR EACH state DO
                    Validate: state contains only alphanumeric or underscore
                    Add to unique set
                END FOR
                Q = unique states sorted
                step = 1
            ELSE IF step == 1 THEN    // Parse alphabet Sigma
                Extract alphanumeric characters
                Add unique symbols to Sigma
                step = 2
            ELSE IF step == 2 THEN    // Parse initial state q0
                q0 = line
                step = 3
            ELSE IF step == 3 THEN    // Parse final states F
                Extract space-separated states
                Add to unique set
                F = unique states sorted
                step = 4
            ELSE IF step >= 4 THEN    // Parse transitions delta
                Parse: "start symbol end"
                delta[(start, symbol)] = end
            END IF
        END FOR

        Close file
        RETURN isValid()
    END FUNCTION

DESCRIPTION:
```

We load a DFA specification from a file. Parses the five-tuple from the file in the specified format. It then validates state names and removes duplicates then finally calls isValid() to verify the complete DFA structure.

```
METHOD: isValid() const
Return Type: bool

PRECONDITIONS:
- All DFA components must be populated
- This is typically called after loadFromFile()

PSEUDOCODE:
    FUNCTION isValid() RETURNS boolean
         Check 1: Initial state in Q
        IF q0 not in Q THEN
            Print error: "Invalid DFA: q0 not in Q"
            RETURN false
        END IF

         Check 2: All final states in Q
        FOR EACH state f in F DO
            IF f not in Q THEN
                Print error: "Invalid DFA: F contains state not in Q"
                RETURN false
            END IF
        END FOR

         Check 3: All transitions valid
        FOR EACH transition (state, symbol) -> nextstate in delta DO
            IF state not in Q THEN
                Print error: "Invalid DFA: transition from state not in Q"
                RETURN false
            END IF
            IF nextstate not in Q THEN
                Print error: "Invalid DFA: transition to state not in Q"
                RETURN false
            END IF
            IF symbol not in Sigma THEN
                Print error: "Invalid DFA: transition uses symbol not in Sigma"
               RETURN false
            END IF
        END FOR

        RETURN true
    END FUNCTION

DESCRIPTION:
Validates the DFA structure by checking:
```

1. The initial state q0 exists in the state set Q
2. All accepting states in F are members of Q
3. All transitions reference valid states and alphabet symbols

This ensures the DFA follows the formal definition before execution.

METHOD: displayFormal() const
Return Type: void

PRECONDITIONS:
- The DFA must be loaded and valid

PSEUDOCODE:

```
    FUNCTION displayFormal()
        Print "Q: "
        FOR EACH state in Q DO
            Print state + " "
        END FOR
        Print newline

        Print "Σ: "
        FOR EACH symbol in Sigma DO
            Print symbol + " "
        END FOR
        Print newline

        Print "q0: " + q0
        Print newline

        Print "F: "
        FOR EACH state in F DO
            Print state + " "
        END FOR
        Print newline

        Print "δ: "
        Print newline
        FOR EACH transition (state, symbol) -> nextstate in delta DO
            Print state + " " + symbol + " " + nextstate
            Print newline
        END FOR
    END FUNCTION
```

DESCRIPTION:
Displays the formal definition of the DFA in the standard mathematical
notation,
showing all five components in a readable format.

```
METHOD: runString(const string& input) const
Return Type: bool
Parameters: string input - the input string to test

PRECONDITIONS:
- The DFA must be loaded and valid
- All symbols in input must be in Sigma

PSEUDOCODE:
    FUNCTION runString(input: string) RETURNS boolean
        current = q0  // Start at initial state

        FOR EACH character c in input DO
            Look up transition (current, c) in delta
            IF transition does not exist THEN
                RETURN false  // No valid path
            END IF
            current = nextstate from transition
        END FOR

        // Check if we ended in an accepting state
        IF current in F THEN
            RETURN true
        ELSE
            RETURN false
        END IF
    END FUNCTION

DESCRIPTION:
Simulates the DFA by processing each input character in sequence, following
transitions from the delta table. Returns true if the DFA ends in an accepting
state, false otherwise.


METHOD: displayComputation(const string& input) const
Return Type: void
Parameters: string input - the input string to display computation for

PRECONDITIONS:
- The DFA must be loaded and valid
- All symbols in input should be in Sigma

PSEUDOCODE:
    FUNCTION displayComputation(input: string)
        current = q0
        Print "Initial state: " + current
        Print newline
```

```
        FOR i = 0 to length(input)-1 DO
            c = input[i]
            Look up transition (current, c) in delta
            IF transition does not exist THEN
                Print "No transition from state " + current +
                      " on symbol " + c
                Print newline
                RETURN
            END IF
            nextstate = result from delta lookup
            Print "δ(" + current + ", " + c + ") -> " + nextstate
            Print newline
            current = nextstate
        END FOR

        IF current in F THEN
            Print "String accepted. Final state: " + current
        ELSE
            Print "String rejected. Final state: " + current
        END IF
        Print newline
    END FUNCTION
```

DESCRIPTION:
Displays the step by step computation of processing an input string through the DFA. Shows each state transition and whether the final state is accepting or rejecting. Useful for debugging and understanding DFA behavior.

METHOD: toString() const override
Return Type: string

PRECONDITIONS:
- The DFA must be loaded

PSEUDOCODE:

```
    FUNCTION toString() RETURNS string
        output = ""

        output += "DFA Formal Representation:\n"
        output += "Q: "
        FOR EACH state in Q DO
            output += state + " "
        END FOR
        output += "\n"

        output += "Σ: "
        FOR EACH symbol in Sigma DO
            output += symbol + " "
```

```
        END FOR
        output += "\n"

        output += "q0: " + q0 + "\n"

        output += "F: "
        FOR EACH state in F DO
            output += state + " "
        END FOR
        output += "\n"

        output += "δ:\n"
        FOR EACH transition (state, symbol) -> nextstate in delta DO
             output += state + " " + symbol + " " + nextstate + "\n"
         END FOR

         RETURN output
    END FUNCTION
```

DESCRIPTION:
It converts the DFA to a string representation and overrides the parent class method then returns the formal DFA representation suitable for printing.

Section 3. HOW TO RUN THE PROGRAM

STEP 1: COMPILE THE PROGRAM
Open a terminal and navigate to the project directory, then run:

```
    clang++ -std=c++17 main.cpp DFA.cpp -o main
```

Alternative compilers:

```
    g++ -std=c++17 main.cpp DFA.cpp -o main
```

STEP 2: RUN THE PROGRAM
Execute the compiled program:

```
    ./main
```

STEP 3: INTERACTIVE COMMANDS
The program displays the prompt "> " and waits for commands. Type the

following commands:

A. Load a DFA from file:
   Command: load <filename>
   Example: load test_dfa_valid.txt
   Output: "DFA loaded successfully from test_dfa_valid.txt"

B. Test string acceptance:
   Command: readstring <string>
   Example: readstring abba
   Output: "String accepted!" or "String rejected."

C. Display step-by-step computation:
   Command: visualize <string>
   Example: visualize abba
   Output: Shows each state transition:
           Initial state: q0
           δ(q0, a) -> q1
           δ(q1, b) -> q1
           δ(q1, b) -> q1
           δ(q1, a) -> q2
           String accepted. Final state: q2

D. Display DFA definition:
   Command: display
   Output: Shows complete DFA formal representation

E. Exit the program:
   Command: exit

Execution Flow

Command	Execution Steps
LOAD <file>	1. Open specified file 2. Parse DFA components 3. Validate the DFA structure 4. Store in memory 5. Return to the command prompt
READSTRING <string>	1. Start at initial state q0 2. For each input symbol: - Look up transition in delta - Move to next state - If no transition state exists, we will reject it.

	3. Check if final state is in F 4. We print string accepted or rejected based on outcome 5. Return to command prompt
DISPLAY	1. Print out formal DFA representation (Q, Σ, q0, F, δ) 2. Return to command prompt
VISUALIZE <string>	1. Start at q0 2. For each symbol - Look up transition in delta - Print the step: δ(state, symbol) -> nextstate - Move to next state - If no transition exists, prints an error and stops. 3. Print the final result (accepted or rejected) 4. Returns back to command prompt
EXIT	1. Terminates the program and exits. 2. Return control to OS

Section 4. TESTING AND VALIDATION

My project includes two test files for validation:

test_dfa_valid.txt:
    Represents a valid DFA that accepts strings ending with 'a'
    following a transition through 'b'

test_dfa_invalid.txt:
    Represents an invalid DFA (final state q2 not in state set Q)
    Tests error handling and validation logic