Aim: - Write a program to construct dfa for the given regular expression.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_STATES 100

#define MAX\_TRANSITIONS 128

// Define structure for NFA state

typedef struct {

    int transitions[MAX\_TRANSITIONS][MAX\_STATES]; // State transitions

    int epsilon[MAX\_STATES]; // Epsilon transitions

    int epsilon\_count;

    int transition\_count[MAX\_TRANSITIONS];

} NFAState;

// Define structure for DFA state

typedef struct {

    int nfa\_states[MAX\_STATES]; // Set of NFA states represented by this DFA state

    int nfa\_state\_count;

} DFAState;

int state\_counter = 0;

NFAState nfa[MAX\_STATES];

int dfa[MAX\_STATES][MAX\_TRANSITIONS];

int visited[MAX\_STATES];

// Function to initialize an NFA state

void initNFAState(NFAState \*state) {

    for (int i = 0; i < MAX\_TRANSITIONS; i++) {

        state->transition\_count[i] = 0;

    }

    state->epsilon\_count = 0;

}

// Function to add a transition to an NFA state

void addTransition(NFAState \*state, int input, int to\_state) {

    state->transitions[input][state->transition\_count[input]++] = to\_state;

}

// Function to add an epsilon transition to an NFA state

void addEpsilonTransition(NFAState \*state, int to\_state) {

    state->epsilon[state->epsilon\_count++] = to\_state;

}

// Function to perform epsilon closure on a set of states

void epsilonClosure(int nfa\_states[], int \*nfa\_state\_count) {

    int stack[MAX\_STATES];

    int top = -1;

    int visited[MAX\_STATES] = {0};

    // Push all initial states into the stack

    for (int i = 0; i < \*nfa\_state\_count; i++) {

        stack[++top] = nfa\_states[i];

    }

    while (top != -1) {

        int current = stack[top--];

        visited[current] = 1;

        // Add epsilon transitions

        for (int i = 0; i < nfa[current].epsilon\_count; i++) {

            int next = nfa[current].epsilon[i];

            if (!visited[next]) {

                stack[++top] = next;

                nfa\_states[(\*nfa\_state\_count)++] = next;

            }

        }

    }

}

// Check if two DFA states (sets of NFA states) are equivalent

int areSameStates(DFAState \*s1, DFAState \*s2) {

    if (s1->nfa\_state\_count != s2->nfa\_state\_count)

        return 0;

    for (int i = 0; i < s1->nfa\_state\_count; i++) {

        if (s1->nfa\_states[i] != s2->nfa\_states[i])

            return 0;

    }

    return 1;

}

// Function to check if a DFA state already exists

int stateExists(DFAState dfa\_states[], int count, DFAState \*state) {

    for (int i = 0; i < count; i++) {

        if (areSameStates(&dfa\_states[i], state))

            return i;

    }

    return -1;

}

// Function to convert NFA to DFA using subset construction

void convertNFAToDFA(int start\_state, int num\_inputs) {

    DFAState dfa\_states[MAX\_STATES]; // DFA states

    int dfa\_state\_count = 0;

    // DFA transition table

    memset(dfa, -1, sizeof(dfa));

    // Initially, epsilon closure of the NFA start state is the DFA start state

    DFAState dfa\_queue[MAX\_STATES];

    int queue\_front = 0, queue\_back = 0;

    // Create the start state for DFA

    dfa\_queue[queue\_back].nfa\_states[0] = start\_state;

    dfa\_queue[queue\_back].nfa\_state\_count = 1;

    epsilonClosure(dfa\_queue[queue\_back].nfa\_states, &dfa\_queue[queue\_back].nfa\_state\_count);

    dfa\_states[dfa\_state\_count++] = dfa\_queue[queue\_back]; // DFA start state corresponds to this set of NFA start states

    queue\_back++;

    // Process each DFA state

    while (queue\_front < queue\_back) {

        DFAState current\_dfa\_state = dfa\_queue[queue\_front++]; // Get the next DFA state to process

        // Process transitions for each input symbol

        for (int input = 0; input < num\_inputs; input++) {

            DFAState new\_dfa\_state;

            new\_dfa\_state.nfa\_state\_count = 0;

            // Compute the set of NFA states reachable on this input

            for (int i = 0; i < current\_dfa\_state.nfa\_state\_count; i++) {

                int nfa\_state = current\_dfa\_state.nfa\_states[i];

                for (int j = 0; j < nfa[nfa\_state].transition\_count[input]; j++) {

                    new\_dfa\_state.nfa\_states[new\_dfa\_state.nfa\_state\_count++] = nfa[nfa\_state].transitions[input][j];

                }

            }

            // Compute epsilon closure of the new DFA state

            epsilonClosure(new\_dfa\_state.nfa\_states, &new\_dfa\_state.nfa\_state\_count);

            // Check if the new DFA state already exists

            int existing\_state = stateExists(dfa\_states, dfa\_state\_count, &new\_dfa\_state);

            if (existing\_state == -1) {

                // New DFA state, add it

                dfa\_states[dfa\_state\_count++] = new\_dfa\_state;

                dfa\_queue[queue\_back++] = new\_dfa\_state;

                existing\_state = dfa\_state\_count - 1;

            }

            // Update the DFA transition table

            dfa[queue\_front - 1][input] = existing\_state;

        }

    }

    // Print the DFA

    printf("\nDFA State Transitions:\n");

    for (int i = 0; i < dfa\_state\_count; i++) {

        printf("DFA State %d:\n", i);

        for (int input = 0; input < num\_inputs; input++) {

            if (dfa[i][input] != -1) {

                printf("  On input %d -> DFA State %d\n", input, dfa[i][input]);

            }

        }

    }

}

int main() {

    char regex[100];

    int num\_inputs = 2; // Assuming binary input (0 or 1 for simplicity)

    // Define NFA (in this case, manually constructing one for demo)

    // In practice, you'd first convert a regex to NFA

    initNFAState(&nfa[0]);

    addTransition(&nfa[0], 0, 1);

    addEpsilonTransition(&nfa[0], 2);

    initNFAState(&nfa[1]);

    addTransition(&nfa[1], 1, 3);

    initNFAState(&nfa[2]);

    addTransition(&nfa[2], 1, 3);

    initNFAState(&nfa[3]); // Accept state

    // Convert the NFA to DFA

    convertNFAToDFA(0, num\_inputs);

    return 0;

}

Output: -

