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

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Define structure for a state

typedef struct State {

    int state\_num;

    struct State\* on\_a; // Transition for 'a'

    struct State\* on\_b; // Transition for 'b'

    struct State\* epsilon1; // Epsilon transition 1

    struct State\* epsilon2; // Epsilon transition 2

} State;

// Define structure for an NFA

typedef struct NFA {

    State\* start\_state;

    State\* accept\_state;

} NFA;

int state\_counter = 0; // Global state counter

// Function to create a new state

State\* createState() {

    State\* state = (State\*)malloc(sizeof(State));

    state->state\_num = state\_counter++;

    state->on\_a = NULL;

    state->on\_b = NULL;

    state->epsilon1 = NULL;

    state->epsilon2 = NULL;

    return state;

}

// Function to create an NFA for a single character

NFA\* createCharNFA(char c) {

    NFA\* nfa = (NFA\*)malloc(sizeof(NFA));

    State\* start = createState();

    State\* accept = createState();

    if (c == 'a') {

        start->on\_a = accept; // Transition on 'a'

    } else if (c == 'b') {

        start->on\_b = accept; // Transition on 'b'

    }

    nfa->start\_state = start;

    nfa->accept\_state = accept;

    return nfa;

}

// Function to combine two NFAs using concatenation

NFA\* concatenateNFA(NFA\* nfa1, NFA\* nfa2) {

    NFA\* nfa = (NFA\*)malloc(sizeof(NFA));

    nfa1->accept\_state->epsilon1 = nfa2->start\_state; // Epsilon transition from nfa1's accept to nfa2's start

    nfa->start\_state = nfa1->start\_state;

    nfa->accept\_state = nfa2->accept\_state;

    return nfa;

}

// Function to combine two NFAs using union (a | b)

NFA\* unionNFA(NFA\* nfa1, NFA\* nfa2) {

    NFA\* nfa = (NFA\*)malloc(sizeof(NFA));

    State\* start = createState(); // New start state

    State\* accept = createState(); // New accept state

    // Epsilon transitions from new start state to the start states of nfa1 and nfa2

    start->epsilon1 = nfa1->start\_state;

    start->epsilon2 = nfa2->start\_state;

    // Epsilon transitions from the accept states of nfa1 and nfa2 to the new accept state

    nfa1->accept\_state->epsilon1 = accept;

    nfa2->accept\_state->epsilon1 = accept;

    nfa->start\_state = start;

    nfa->accept\_state = accept;

    return nfa;

}

// Function to apply Kleene star to an NFA (a\*)

NFA\* kleeneStarNFA(NFA\* nfa) {

    NFA\* new\_nfa = (NFA\*)malloc(sizeof(NFA));

    State\* start = createState(); // New start state

    State\* accept = createState(); // New accept state

    // Epsilon transitions

    start->epsilon1 = nfa->start\_state; // From new start to old start

    start->epsilon2 = accept;           // From new start to new accept (for zero repetitions)

    nfa->accept\_state->epsilon1 = nfa->start\_state; // From old accept to old start (loop)

    nfa->accept\_state->epsilon2 = accept;           // From old accept to new accept

    new\_nfa->start\_state = start;

    new\_nfa->accept\_state = accept;

    return new\_nfa;

}

// Function to print the NFA transitions

void printNFA(State\* state, int visited[]) {

    if (visited[state->state\_num]) return;

    visited[state->state\_num] = 1;

    if (state->on\_a) {

        printf("State %d --a--> State %d\n", state->state\_num, state->on\_a->state\_num);

        printNFA(state->on\_a, visited);

    }

    if (state->on\_b) {

        printf("State %d --b--> State %d\n", state->state\_num, state->on\_b->state\_num);

        printNFA(state->on\_b, visited);

    }

    if (state->epsilon1) {

        printf("State %d --ε--> State %d\n", state->state\_num, state->epsilon1->state\_num);

        printNFA(state->epsilon1, visited);

    }

    if (state->epsilon2) {

        printf("State %d --ε--> State %d\n", state->state\_num, state->epsilon2->state\_num);

        printNFA(state->epsilon2, visited);

    }

}

// Function to construct NFA from regular expression

NFA\* constructNFA(char\* regex) {

    NFA\* stack[100]; // Stack for NFAs

    int top = -1;

    for (int i = 0; i < strlen(regex); i++) {

        if (regex[i] == 'a' || regex[i] == 'b') {

            // Create NFA for character

            stack[++top] = createCharNFA(regex[i]);

        } else if (regex[i] == '.') {

            // Concatenation

            NFA\* nfa2 = stack[top--];

            NFA\* nfa1 = stack[top--];

            stack[++top] = concatenateNFA(nfa1, nfa2);

        } else if (regex[i] == '|') {

            // Union

            NFA\* nfa2 = stack[top--];

            NFA\* nfa1 = stack[top--];

            stack[++top] = unionNFA(nfa1, nfa2);

        } else if (regex[i] == '\*') {

            // Kleene star

            NFA\* nfa = stack[top--];

            stack[++top] = kleeneStarNFA(nfa);

        }

    }

    return stack[top]; // Final NFA

}

int main() {

    char regex[100];

    // Input regular expression in postfix notation

    printf("Enter the regular expression in postfix notation (e.g., ab.a.): ");

    scanf("%s", regex);

    // Construct the NFA

    NFA\* nfa = constructNFA(regex);

    // Print the NFA

    int visited[100] = {0};

    printf("NFA transitions:\n");

    printNFA(nfa->start\_state, visited);

    return 0;

}

Output: -



