DATE:23/07/2025

EXP:05

PATTERN MATCHING WITH LEX AND DFA CONSTRUCTION

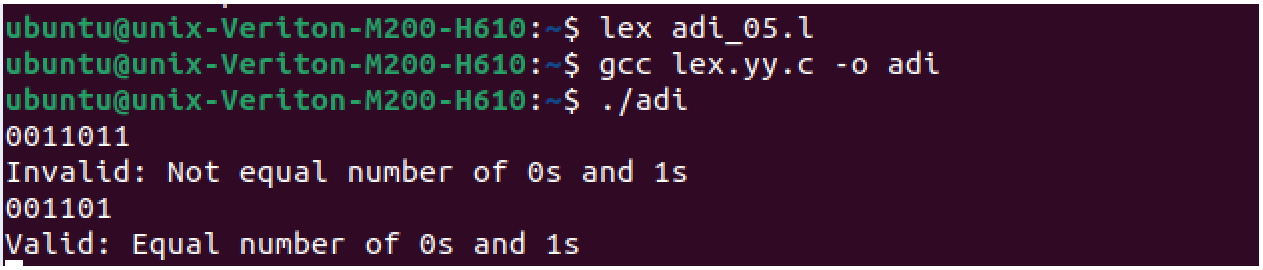
Q1:

Write a lex program to validate a set of strings of 0’s and 1’s with equal number of 1’s and 0’s.

CODE:

```
%{
int zero_count = 0, one_count = 0;
%}

%%
0    { zero_count++; }
1    { one_count++; }
[^01\n]+    { /* Ignore any non-0/1 character except newline */ }
\n   {
        if (zero_count == one_count)
            printf("Valid: Equal number of 0s and 1s\n");
        else
            printf("Invalid: Not equal number of 0s and 1s\n");
        zero_count = one_count = 0; // Reset for next line
    }

%%
int yywrap(void) {
    return 1;
}

int main(void) {
    yylex();
    return 0;

}
```

OUTPUT:



```
ubuntu@unix-Veriton-M200-H610:~$ lex adi_05.l
ubuntu@unix-Veriton-M200-H610:~$ gcc lex.yy.c -o adi
ubuntu@unix-Veriton-M200-H610:~$ ./adi
0011011
Invalid: Not equal number of 0s and 1s
001101
Valid: Equal number of 0s and 1s
```

2. Write a C program to construct DFA from NFA using subset construction algorithm for regular expression (a|b)*.

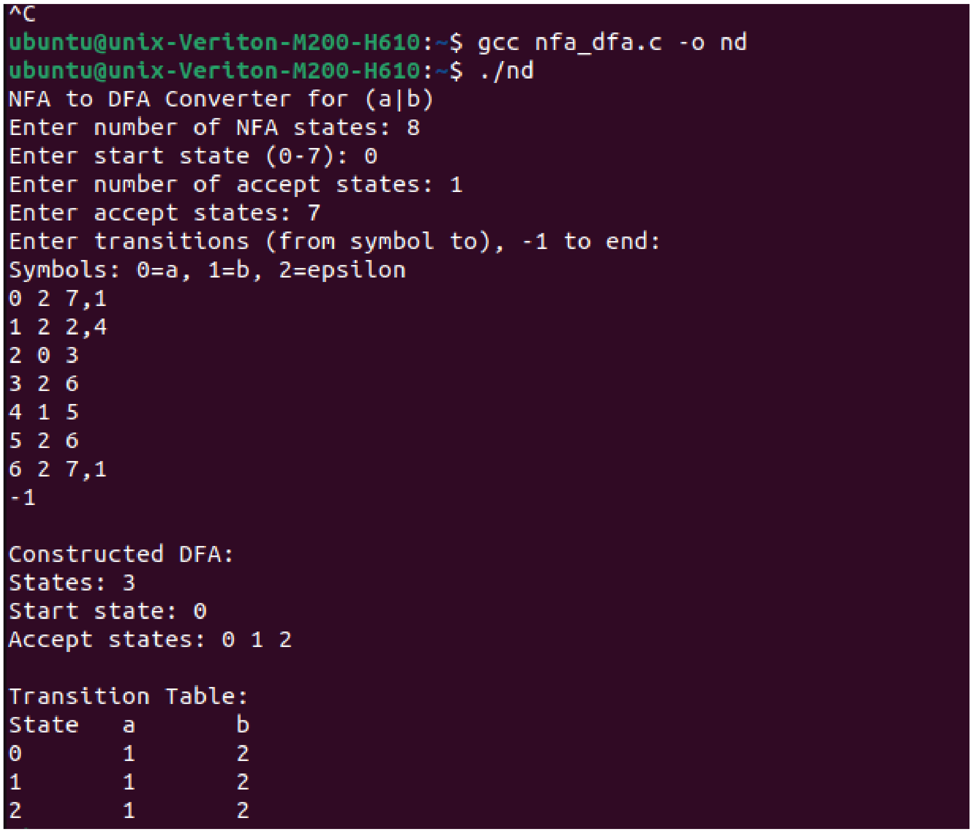
CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <ctype.h>

#define MAX_STATES 20
#define MAX_SYMBOLS 2  // Only 'a' (0) and 'b' (1)

typedef struct {
    int transitions[MAX_STATES][MAX_SYMBOLS][MAX_STATES];
    int epsilon[MAX_STATES][MAX_STATES];
    int stateCount;
    int startState;
    int acceptStates[MAX_STATES];
} NFA;

typedef struct {
    int transitions[MAX_STATES][MAX_SYMBOLS];
    int stateCount;
    int startState;
    int acceptStates[MAX_STATES];
} DFA;

NFA nfa;
DFA dfa;

void initialize() {
    memset(&nfa, 0, sizeof(nfa));
    memset(&dfa, 0, sizeof(dfa));
    for (int i = 0; i < MAX_STATES; i++) {
        for (int j = 0; j < MAX_SYMBOLS; j++) {
            for (int k = 0; k < MAX_STATES; k++) {
                nfa.transitions[i][j][k] = -1;
            }
        }
    }
}

void epsilonClosure(int state, bool closure[]) {
    if (state < 0 || state >= MAX_STATES) return;
    if (closure[state]) return;

    closure[state] = true;
    for (int i = 0; i < MAX_STATES; i++) {
        if (nfa.epsilon[state][i]) {
            epsilonClosure(i, closure);
        }
    }
}
```

```
void subsetConstruction() {
    bool dfaStates[MAX_STATES][MAX_STATES] = {false};
    int dfaStateCount = 0;

    // Compute initial state (epsilon closure of NFA start state)
    bool initialClosure[MAX_STATES] = {false};
    epsilonClosure(nfa.startState, initialClosure);

    // Initialize DFA
    dfa.startState = 0;
    memcpy(dfaStates[0], initialClosure, MAX_STATES * sizeof(bool));
    dfaStateCount++;

    for (int i = 0; i < dfaStateCount; i++) {
        for (int sym = 0; sym < MAX_SYMBOLS; sym++) {
            bool newState[MAX_STATES] = {false};

            // Compute move for current symbol
            for (int j = 0; j < nfa.stateCount; j++) {
                if (dfaStates[i][j]) {
                    for (int k = 0; k < MAX_STATES && nfa.transitions[j][sym][k] != -1; k++) {
                        int toState = nfa.transitions[j][sym][k];
                        bool tempClosure[MAX_STATES] = {false};
                        epsilonClosure(toState, tempClosure);
                        for (int m = 0; m < MAX_STATES; m++) {
                            if (tempClosure[m]) newState[m] = true;
                        }
                    }
                }
        }

        // Find existing state or create new one
        int existingState = -1;
        for (int j = 0; j < dfaStateCount; j++) {
          bool match = true;
          for (int k = 0; k < MAX_STATES; k++) {
            if (dfaStates[j][k] != newState[k]) {
              match = false;
              break;
            }
          }
          if (match) {
            existingState = j;
            break;
          }
        }

        if (existingState == -1) {
          memcpy(dfaStates[dfaStateCount], newState, MAX_STATES * sizeof(bool));
          existingState = dfaStateCount;
          dfaStateCount++;
        }

        dfa.transitions[i][sym] = existingState;
      }
    }

    // Set DFA state count and accept states
    dfa.stateCount = dfaStateCount;
    for (int i = 0; i < dfaStateCount; i++) {
        dfa.acceptStates[i] = 0;
        for (int j = 0; j < nfa.stateCount; j++) {
            if (dfaStates[i][j] && nfa.acceptStates[j]) {
                dfa.acceptStates[i] = 1;
                break;
            }
        }
    }
}

void printDFA() {
    printf("\nConstructed DFA:\n");
    printf("States: %d\n", dfa.stateCount);
    printf("Start state: %d\n", dfa.startState);
    printf("Accept states: ");
    for (int i = 0; i < dfa.stateCount; i++) {
        if (dfa.acceptStates[i]) printf("%d ", i);
    }
    printf("\n\nTransition Table:\n");
    printf("State\ta\tb\n");
    for (int i = 0; i < dfa.stateCount; i++) {
        printf("%d\t%d\t%d\n", i, dfa.transitions[i][0], dfa.transitions[i][1]);
    }
}

void addTransition(int from, int symbol, int to) {
    if (symbol == 2) { // epsilon
        nfa.epsilon[from][to] = 1;
    } else if (symbol >= 0 && symbol < MAX_SYMBOLS) {
        int idx = 0;
        while (idx < MAX_STATES - 1 && nfa.transitions[from][symbol][idx] != -1) {
            idx++;
        }
        nfa.transitions[from][symbol][idx] = to;
    }
}

int main() {
    initialize();

    printf("NFA to DFA Converter for (a|b)\n");
    printf("Enter number of NFA states: ");
    scanf("%d", &nfa.stateCount);

    printf("Enter start state (0-%d): ", nfa.stateCount - 1);
    scanf("%d", &nfa.startState);

    printf("Enter number of accept states: ");
    int numAccept;
    scanf("%d", &numAccept);
    printf("Enter accept states: ");
    for (int i = 0; i < numAccept; i++) {
        int state;
        scanf("%d", &state);
        nfa.acceptStates[state] = 1;
    }

    printf("Enter transitions (from symbol to), -1 to end:\n");
    printf("Symbols: 0=a, 1=b, 2=epsilon\n");
```

```
while (1) {
    int from, symbol;
    char input[100];

    scanf("%s", input); // First try to read the from state
    if (strcmp(input, "-1") == 0) break;
    from = atoi(input);

    scanf("%d", &symbol); // Read the symbol

    // Read all destination states in the comma-separated list
    char destinations[100];
    scanf("%s", destinations);

    char *token = strtok(destinations, ",");
    while (token != NULL) {
        int to = atoi(token);
        addTransition(from, symbol, to);
        token = strtok(NULL, ",");
    }
}

// Add any missing transitions to -1
for (int i = 0; i < MAX_STATES; i++) {
    for (int j = 0; j < MAX_SYMBOLS; j++) {
        for (int k = 0; k < MAX_STATES; k++) {
            if (nfa.transitions[i][j][k] == 0 && k > 0) {
                nfa.transitions[i][j][k] = -1;
            }
        }
        }
    }
    subsetConstruction();
    printDFA();
    return 0;
}
```

OUTPUT:



```
^C
ubuntu@unix-Veriton-M200-H610:~$ gcc nfa_dfa.c -o nd
ubuntu@unix-Veriton-M200-H610:~$ ./nd
NFA to DFA Converter for (a|b)
Enter number of NFA states: 8
Enter start state (0-7): 0
Enter number of accept states: 1
Enter accept states: 7
Enter transitions (from symbol to), -1 to end:
Symbols: 0=a, 1=b, 2=epsilon
0 2 7,1
1 2 2,4
2 0 3
3 2 6
4 1 5
5 2 6
6 2 7,1
-1

Constructed DFA:
States: 3
Start state: 0
Accept states: 0 1 2

Transition Table:
State   a       b
0       1       2
1       1       2
2       1       2
```

RESULT:

The programs have been completed and the outputs have been verified.