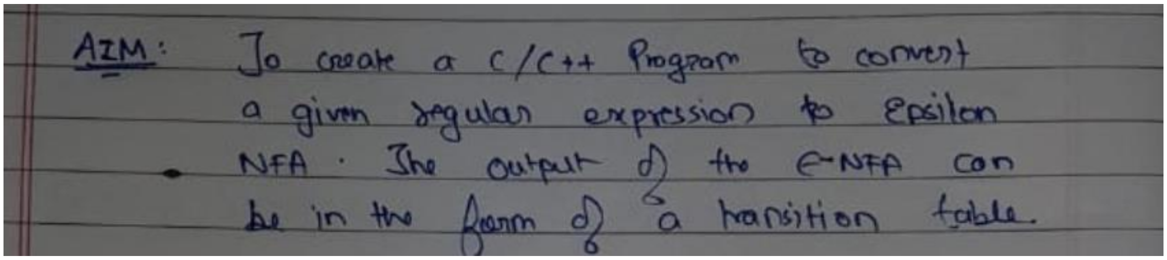
Regular Expression to Epsilon NFA
Compiler Design – Lab 3

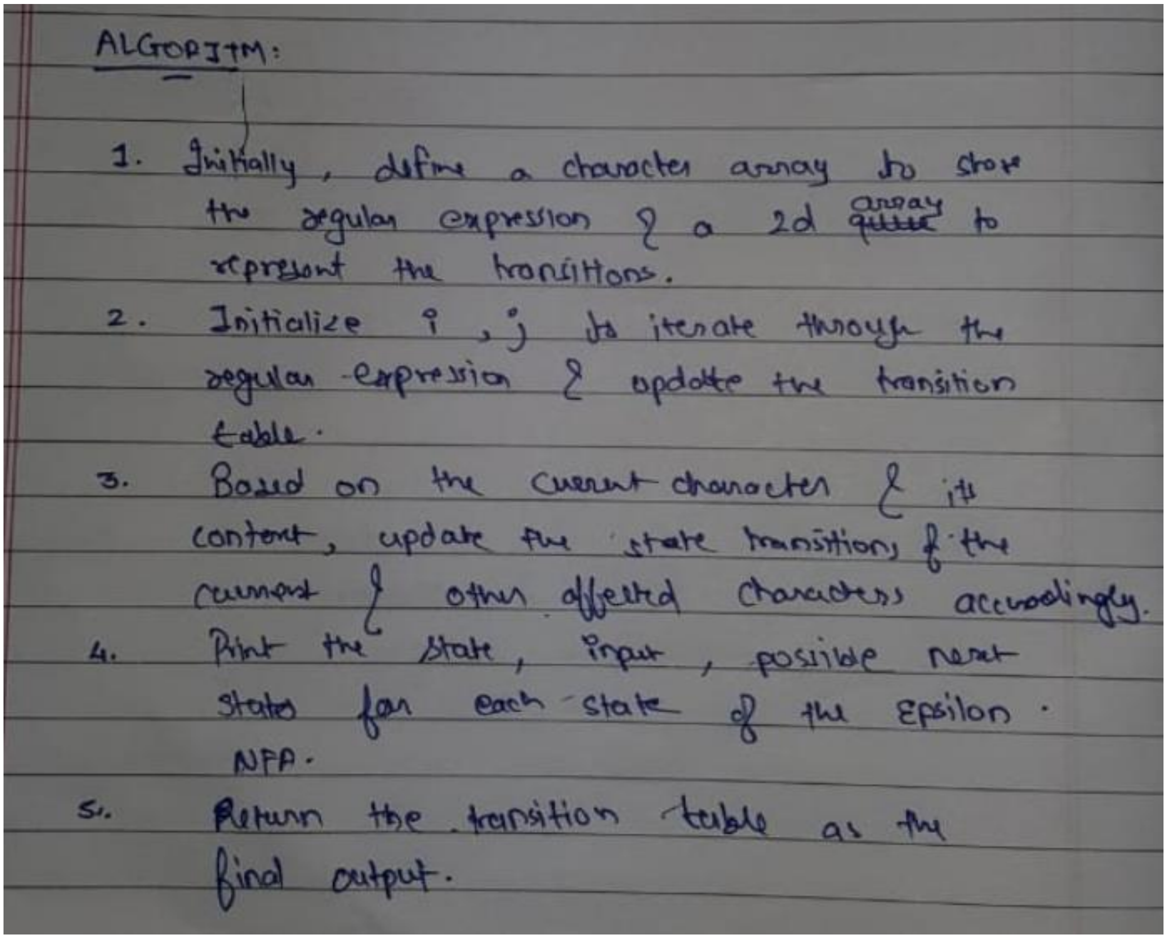
Name: Ojas Patil
Reg No: 21BAI1106

AIM



AIM: To create a C/C++ Program to convert a given regular expression to Epsilon NFA. The output of the ∈-NFA can be in the form of a transition table.

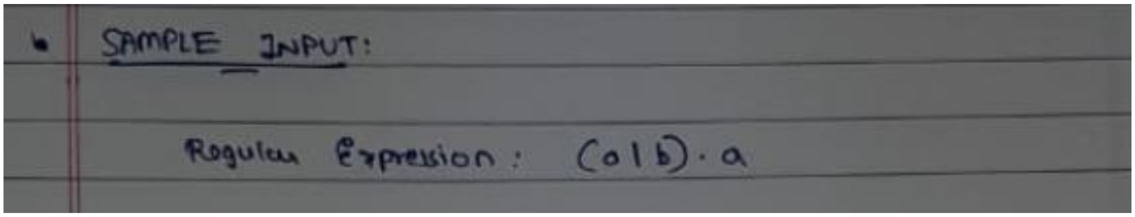
ALGORITHM



ALGORITHM:

1. Initially, define a character array to store the regular expression & a 2d ~~queue~~ array to represent the transitions.
2. Initialize i, j to iterate through the regular expression & update the transition table.
3. Based on the current character & its context, update the state transitions of the current & other affected characters accordingly.
4. Print the state, input, possible next states for each state of the Epsilon NFA.
5. Return the transition table as the final output.

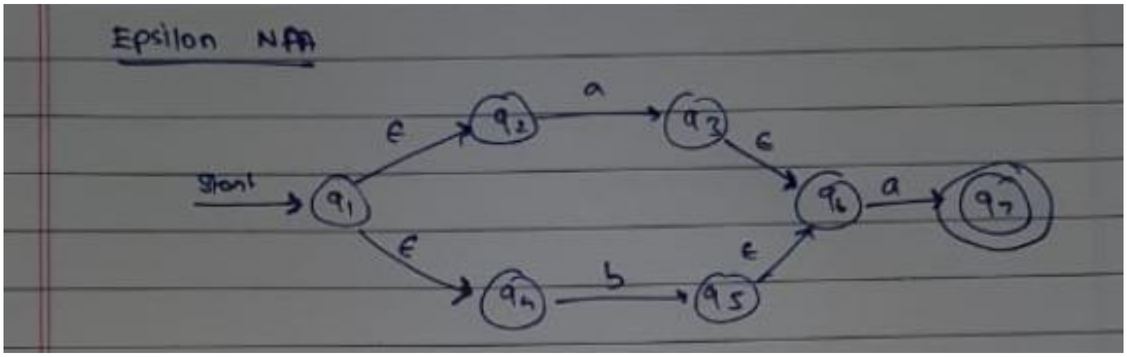
Sample Input



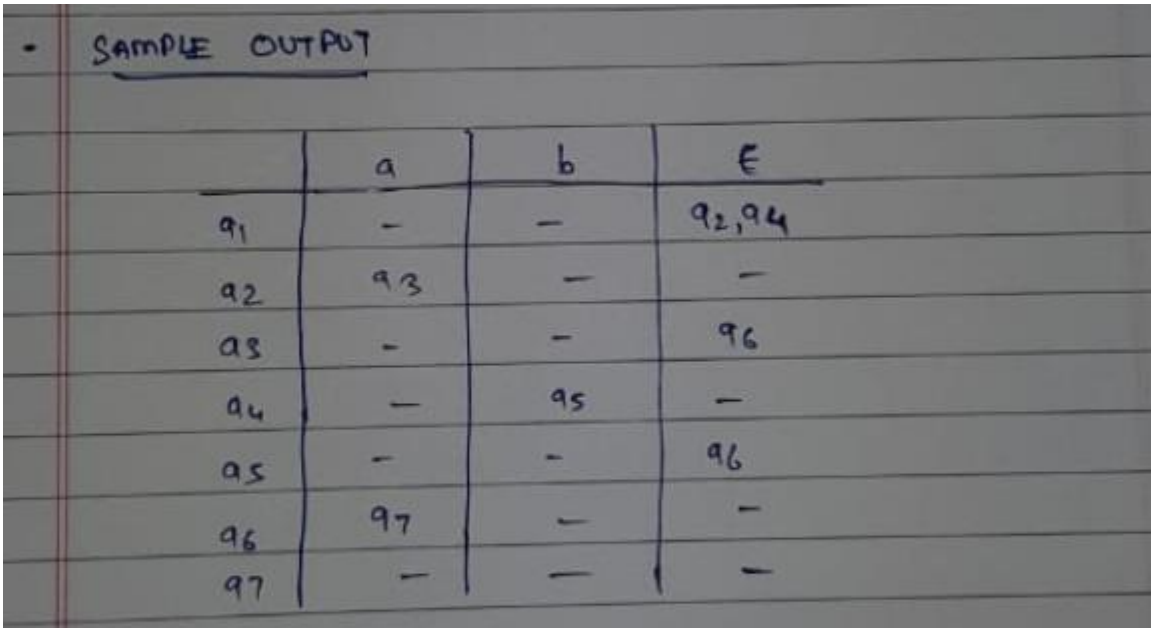
SAMPLE INPUT:

Regular Expression : (a|b)·a

Corresponding E-NFA



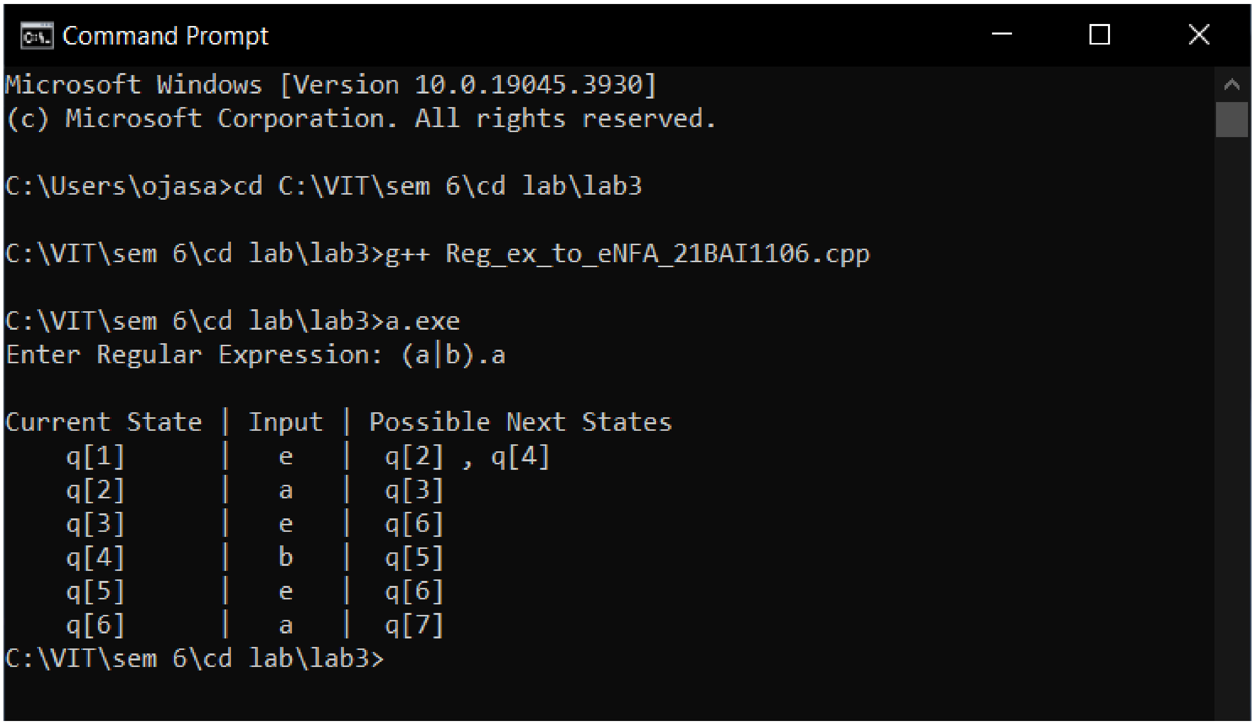
Sample Output



SAMPLE OUTPUT

	a	b	ε
q1	–	–	q2,q4
q2	q3	–	–
q3	–	–	q6
q4	–	q5	–
q5	–	–	q6
q6	q7	–	–
q7	–	–	–

Output Snapshot



```
Microsoft Windows [Version 10.0.19045.3930]
(c) Microsoft Corporation. All rights reserved.

C:\Users\ojasa>cd C:\VIT\sem 6\cd lab\lab3

C:\VIT\sem 6\cd lab\lab3>g++ Reg_ex_to_eNFA_21BAI1106.cpp

C:\VIT\sem 6\cd lab\lab3>a.exe
Enter Regular Expression: (a|b).a

Current State | Input | Possible Next States
    q[1]      |   e   |  q[2] , q[4]
    q[2]      |   a   |  q[3]
    q[3]      |   e   |  q[6]
    q[4]      |   b   |  q[5]
    q[5]      |   e   |  q[6]
    q[6]      |   a   |  q[7]
C:\VIT\sem 6\cd lab\lab3>
```

Program Explanation

This is a CPP Program which performs the computation of converting a regular expression to an Epsilon NFA and gives its transition table as output. Here is a breakdown of its working:

1. **Variable and Array Initialization:**
    - Initialize a character array reg to store the regular expression and a 2D integer array q to represent the state transitions. Initialize variables i and j for iteration. Set all elements of array q to 0.
2. **Input Regular Expression:**
    - Prompt the user to enter a regular expression and read it into the reg array.
3. **Parsing Regular Expression:**
    - Calculate the length of the regular expression and Iterate through the characters of the regular expression.

4. **State Transition Updates:**
   - Update the state transitions in array q based on the current character and its context.
5. **Output State Transitions:**
   - Print the header for the state transitions table. Then, iterate through the state transitions array q and print the current state, input, and possible next states.

Source Code

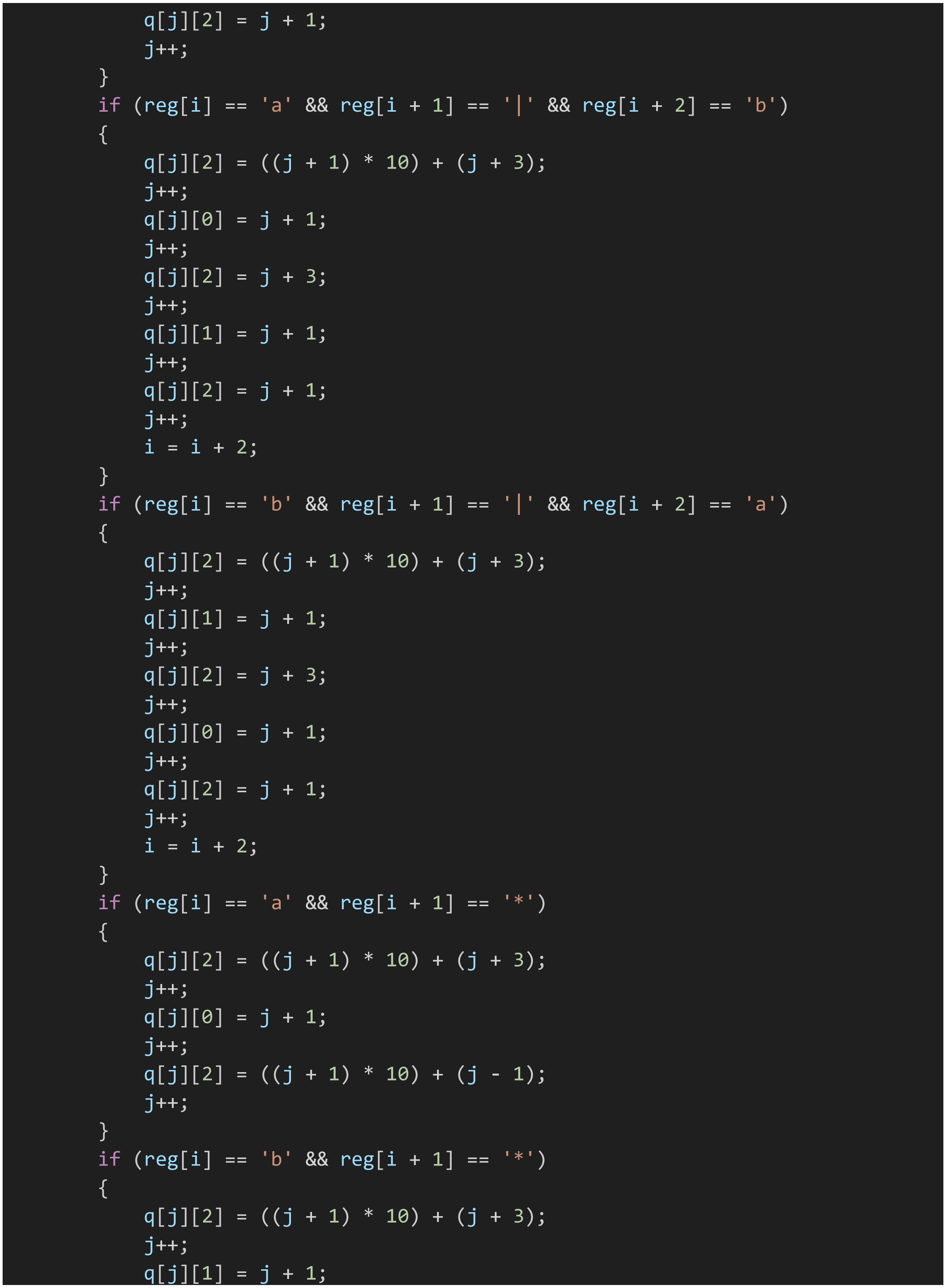
```c
#include <stdio.h>
#include <string.h>

int main()
{
    char reg[20];
    int q[20][3], i = 0, j = 1, len, a, b;

    for (a = 0; a < 20; a++)
        for (b = 0; b < 3; b++)
            q[a][b] = 0;

    printf("Enter Regular Expression: ");
    scanf("%s", reg);
    printf("\n");

    len = strlen(reg);
    while (i < len)
    {
        if (reg[i] == 'a' && reg[i + 1] != '|' && reg[i + 1] != '*')
        {
            q[j][0] = j + 1;
            j++;
        }
        if (reg[i] == 'b' && reg[i + 1] != '|' && reg[i + 1] != '*')
        {
            q[j][1] = j + 1;
            j++;
        }
        if (reg[i] == 'e' && reg[i + 1] != '|' && reg[i + 1] != '*')
        {
```



```
            q[j][2] = j + 1;
            j++;
        }
        if (reg[i] == 'a' && reg[i + 1] == '|' && reg[i + 2] == 'b')
        {
            q[j][2] = ((j + 1) * 10) + (j + 3);
            j++;
            q[j][0] = j + 1;
            j++;
            q[j][2] = j + 3;
            j++;
            q[j][1] = j + 1;
            j++;
            q[j][2] = j + 1;
            j++;
            i = i + 2;
        }
        if (reg[i] == 'b' && reg[i + 1] == '|' && reg[i + 2] == 'a')
        {
            q[j][2] = ((j + 1) * 10) + (j + 3);
            j++;
            q[j][1] = j + 1;
            j++;
            q[j][2] = j + 3;
            j++;
            q[j][0] = j + 1;
            j++;
            q[j][2] = j + 1;
            j++;
            i = i + 2;
        }
        if (reg[i] == 'a' && reg[i + 1] == '*')
        {
            q[j][2] = ((j + 1) * 10) + (j + 3);
            j++;
            q[j][0] = j + 1;
            j++;
            q[j][2] = ((j + 1) * 10) + (j - 1);
            j++;
        }
        if (reg[i] == 'b' && reg[i + 1] == '*')
        {
            q[j][2] = ((j + 1) * 10) + (j + 3);
            j++;
            q[j][1] = j + 1;
            j++;
            q[j][2] = ((j + 1) * 10) + (j - 1);
            j++;
        }
        if (reg[i] == ')' && reg[i + 1] == '*')
        {
            q[0][2] = ((j + 1) * 10) + 1;
            q[j][2] = ((j + 1) * 10) + 1;
            j++;
        }
        i++;
    }

    printf("Current State | Input | Possible Next States");

    for (i = 0; i <= j; i++)
    {
        if (q[i][0] != 0)
            printf("\n    q[%d]      |   a   |  q[%d]", i, q[i][0]);
        if (q[i][1] != 0)
            printf("\n    q[%d]      |   b   |  q[%d]", i, q[i][1]);
        if (q[i][2] != 0)
        {
            if (q[i][2] < 10)
                printf("\n    q[%d]      |   e   |  q[%d]", i, q[i][2]);
            else
                printf("\n    q[%d]      |   e   |  q[%d] , q[%d]", i, q[i][2] /
10, q[i][2] % 10);
        }
    }

    return 0;
}
```

Conclusion

Thus, we have studied and created a CPP program which converts a given Regular Expression to a Epsilon NFA and returns its transition table as its output while covering all the required operations.