Experiment 6

Objective: Write a program to find ε-closure of all states of a given NFA with ε transition.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
int eclosure[MAX][MAX], n;
void findClosure(int state, int closure[MAX], int visited[MAX]) {
   closure[state] = 1;
   visited[state] = 1;
   for (int i = 0; i < n; i++) {
      if (eclosure[state][i] && !visited[i]) {
         findClosure(i, closure, visited);
      }
   }
}
int main() {
   int closure[MAX], visited[MAX];
   printf("Enter the number of states: ");
   scanf("%d", &n);
   printf("Enter the epsilon transition matrix (0 or 1):\n");
   printf("(eclosure[i][j] = 1 if there is an epsilon transition from i to j)\n");
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         scanf("%d", &eclosure[i][j]);
      }
   }
```

```
    printf("\nEpsilon Closures:\n");
    for (int i = 0; i < n; i++) {
        for (int k = 0; k < n; k++) {
            closure[k] = 0;
            visited[k] = 0;
        }
        findClosure(i, closure, visited);
        printf("E-closure(%d): { ", i);
        for (int j = 0; j < n; j++) {
            if (closure[j]) {
                printf("%d ", j);
            }
        }
        printf("}\n");
    }
    return 0;
}
```

Output

```
Enter the number of states: 4
Enter the epsilon transition matrix (0 or 1):
(eclosure[i][j] = 1 if there is an epsilon transition from i to j)
0 1 0 0
0 0 1 0
0 0 0 1
0 0 0 0

Epsilon Closures:
E-closure(0): { 0 1 2 3 }
E-closure(1): { 1 2 3 }
E-closure(2): { 2 3 }
E-closure(3): { 3 }


=== Code Execution Successful ===
```

Experiment 7

Objective: Write a program to convert NFA with ε-transitions into NFA without ε-transitions.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
int epsilon[MAX][MAX], nfa[MAX][MAX][MAX], new_nfa[MAX][MAX];
int n, symbols;
void findClosure(int state, int closure[MAX]) {
   closure[state] = 1;
   for (int i = 0; i < n; i++) {
      if (epsilon[state][i] && !closure[i]) {
         findClosure(i, closure);
      }
   }
}
void removeEpsilonTransitions() {
   int closure[MAX];
   for (int state = 0; state < n; state++) {
      for (int i = 0; i < n; i++)
         closure[i] = 0;
      findClosure(state, closure);
      for (int sym = 0; sym < symbols; sym++) {
         for (int i = 0; i < n; i++) {
            if (closure[i]) {
               for (int j = 0; j < n; j++) {
                  if (nfa[i][sym][j]) {
                     new_nfa[state][sym] |= (1 << j);
```

```
                }
            }
        }
    }
  }
}
int main() {
  printf("Enter number of states: ");
  scanf("%d", &n);
  printf("Enter number of input symbols (excluding epsilon): ");
  scanf("%d", &symbols);
  printf("Enter the epsilon transition matrix (0 or 1):\n");
  printf("(epsilon[i][j] = 1 if there is an epsilon transition from i to j)\n");
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      scanf("%d", &epsilon[i][j]);
  printf("Enter the NFA transition table:\n");
  printf("(nfa[i][sym][j] = 1 if there is a transition from state i to j on input sym)\n");
  for (int i = 0; i < n; i++) {
    for (int sym = 0; sym < symbols; sym++) {
      printf("Enter transitions from state %d on symbol %d:\n", i, sym);
      for (int j = 0; j < n; j++) {
        scanf("%d", &nfa[i][sym][j]);
      }
    }
  }

  removeEpsilonTransitions();
  printf("\nNFA without epsilon transitions:\n");
```

```
    for (int i = 0; i < n; i++) {
        for (int sym = 0; sym < symbols; sym++) {
            printf("From state %d on symbol %d -> { ", i, sym);
            for (int j = 0; j < n; j++) {
                if (new_nfa[i][sym] & (1 << j)) {
                    printf("%d ", j);
                }
            }
            printf("}\n");
        }
    }
    return 0;
}
```

Output

```
Enter number of states: 3
Enter number of input symbols (excluding epsilon): 2
Enter the epsilon transition matrix (0 or 1):
(epsilon[i][j] = 1 if there is an epsilon transition from i to j)
0 1 0
0 0 1
0 0 0

Enter the NFA transition table:
(nfa[i][sym][j] = 1 if there is a transition from state i to j on input sym)
Enter transitions from state 0 on symbol 0:
0 0 0
Enter transitions from state 0 on symbol 1:
0 0 0
Enter transitions from state 1 on symbol 0:
0 0 1
Enter transitions from state 1 on symbol 1:
1 0 0
Enter transitions from state 2 on symbol 0:
0 0 0
Enter transitions from state 2 on symbol 1:
0 1 0

NFA without epsilon transitions:
From state 0 on symbol 0 -> { 2 }
From state 0 on symbol 1 -> { 0 1 }
From state 1 on symbol 0 -> { 2 }
From state 1 on symbol 1 -> { 0 1 }
From state 2 on symbol 0 -> { }
From state 2 on symbol 1 -> { 1 }
```

EXPERIMENT - 9

Objective: Write a program to minimize any given DFA .

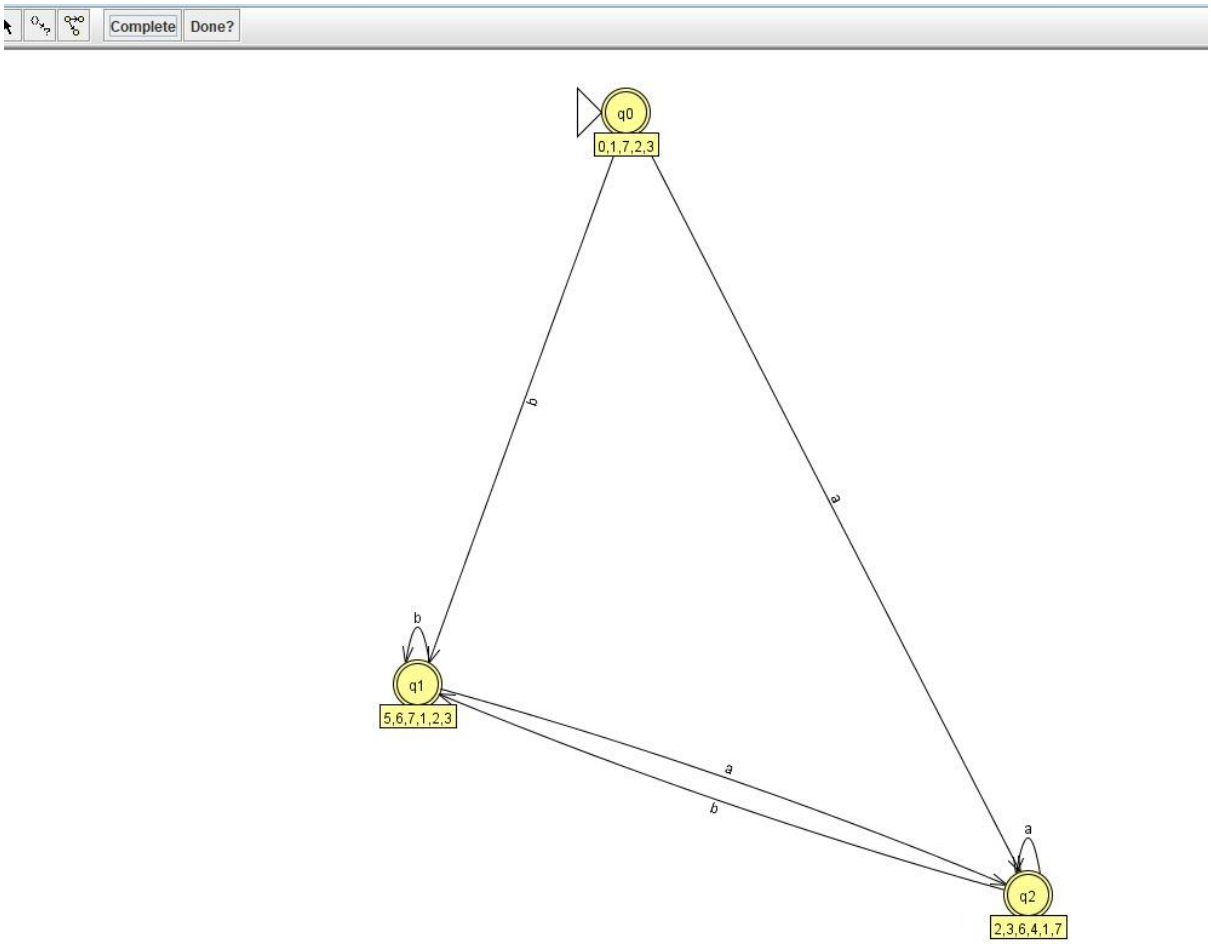
AIM: The aim of this practical is to minimize a given DFA using JFLAP and verify that the minimized DFA accepts the same language as the original DFA.

THEORY: DFA minimization is the process of reducing the number of states in a DFA while preserving the language it recognizes. Two states are equivalent if they have the same behavior for all possible input strings. The minimization algorithm identifies these equivalent states and merges them.

Steps to Minimize a DFA:

Step 1: Launch JFLAP and create/load the DFA you want to minimize.

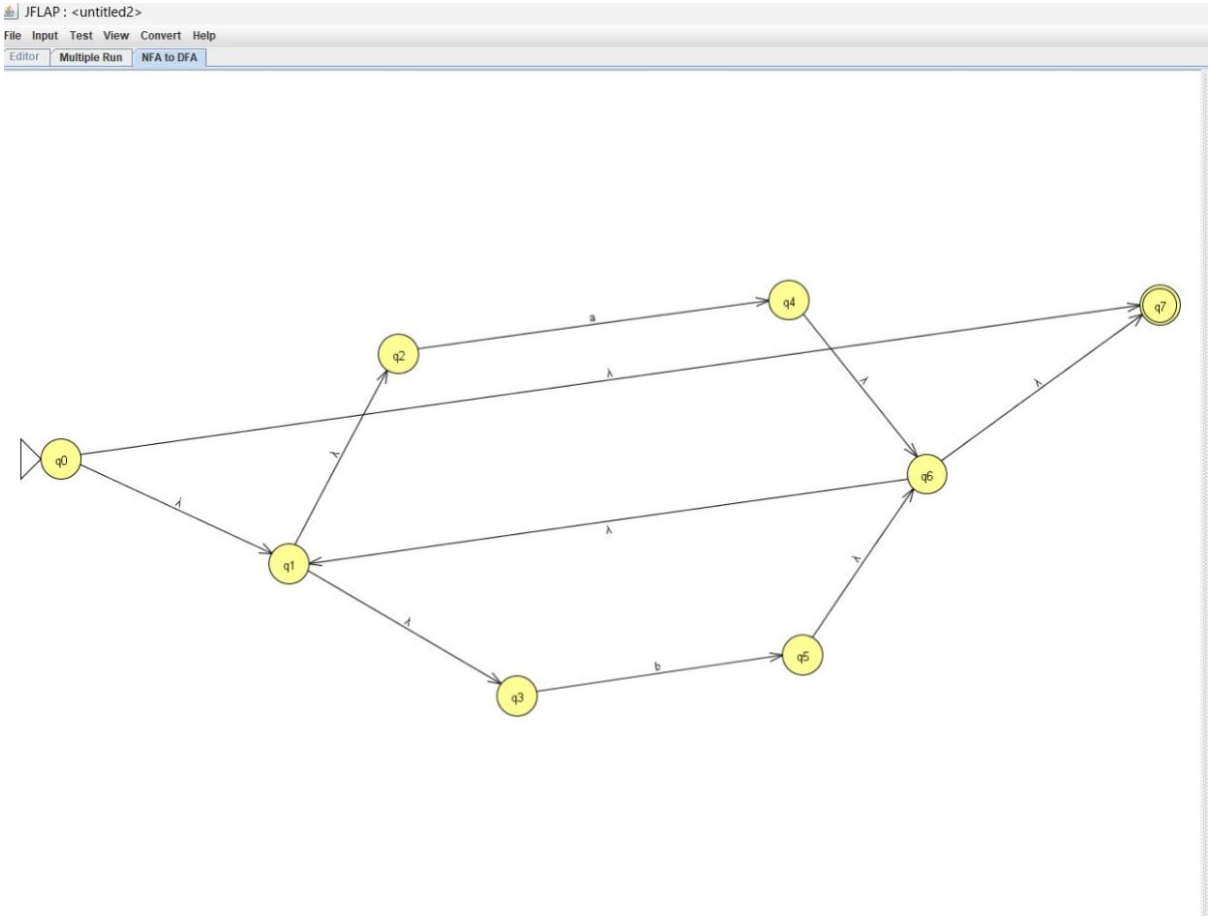
Step 2: Create your initial DFA with states and transitions. You can see an example DFA in the image below with states q0, q1, q2, q3, q4, q5, q6, and q7. Note that q0 is the initial state (marked with an arrow).



Step 3: Once you have created your DFA, navigate to the "Convert" menu and select "Minimize DFA".

Step 4: JFLAP will now start the minimization process. In the first step, JFLAP partitions the states into accepting and non-accepting states.

Step 5: You will see a new window showing the initial partitioning. States may be grouped in yellow boxes with numbers, indicating which partition they belong to.



Step 6: The image shows the groups created during the minimization process. You can see:

- State q0 with a partition label [0,1,7,2,3]
- State q1 with a partition label [5,6,7,1,2,3]
- State q2 with a partition label [2,3,6,4,1,7]

These labels represent the equivalence classes the states belong to through iterations of the minimization algorithm.

Step 7: Click "Complete" or "Do Step" to proceed with the minimization algorithm. JFLAP will refine the partitions based on the transitions from each state.

Step 8: For each step, JFLAP checks if states in the same partition lead to the same partition for each input symbol. If not, it refines the partition by splitting it.

Step 9: Continue clicking "Do Step" until JFLAP indicates that the minimization is complete (when no further refinement is possible).

Step 10: When the minimization is complete, JFLAP will display the minimized DFA, where each state represents an equivalence class from the original DFA.

Step 11: You can now test the minimized DFA by inputting strings and checking if they're accepted, just as you would with the original DFA.

Experiment 10

Objective: Develop an operator precedence parser for a given language.

Code:

```
#include <stdio.h>
#include <string.h>
#define MAX 10
char precedence[MAX][MAX];
char terminals[MAX];
int n;
int getIndex(char c) {
    for (int i = 0; i < n; i++) {
        if (terminals[i] == c)
            return i;
    }
    return -1;
}
void parseString(char str[]) {
    char stack[MAX];
    int top = 0;
    stack[top] = '$';
    int i = 0;
    printf("\nStack\tInput\tAction\n");
    while (1) {
        printf("%s\t%s\t", stack, str + i);
        int stackTopIndex = getIndex(stack[top]);
        int inputIndex = getIndex(str[i]);
        if (stack[top] == '$' && str[i] == '$') {
            printf("Accept\n");
```

```
            break;
        }
        if (precedence[stackTopIndex][inputIndex] == '<' ||
precedence[stackTopIndex][inputIndex] == '=') {
            top++;
            stack[top] = str[i];
            printf("Shift %c\n", str[i]);
            i++;
        } else if (precedence[stackTopIndex][inputIndex] == '>') {
            printf("Reduce\n");
            top--;
        } else {
            printf("Error\n");
            break;
        }
    }
}
int main() {
    printf("Enter number of terminals: ");
    scanf("%d", &n);
    printf("Enter terminals: ");
    for (int i = 0; i < n; i++) {
        scanf(" %c", &terminals[i]);
    }
    printf("Enter precedence table:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf(" %c", &precedence[i][j]);
```

```
    }
  }
  char input[MAX];
  printf("Enter input string (end with $): ");
  scanf("%s", input);
  parseString(input);
  return 0;
}
```

Output

```
Enter number of terminals: 3
Enter terminals: + i $

Enter precedence table:
> < >
> > >
< < =
Enter input string (end with $): i+i$

Stack   Input   Action
$   i+i$    Shift i
$i◆g◆◆◆     +i$ Reduce
$i◆g◆◆◆     +i$ Shift +
$+◆g◆◆◆     i$  Shift i
$+ig◆◆◆     $   Reduce
$+ig◆◆◆     $   Reduce
$+ig◆◆◆     $   Accept


=== Code Execution Successful ===
```

Experiment - 11

OBJECTIVE: Write a ‘c’ program to find the first function of given grammar.

Code:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX 10
char productions[MAX][10];
char first[26][10]; // 26 for A-Z non-terminals
int numProductions;
int visited[26] = {0}; // To avoid infinite recursion
int isNonTerminal(char c) {
 return isupper(c);
}
void addToFirst(char nonTerminal, char symbol) {
 int idx = nonTerminal - 'A';
 for (int i = 0; first[idx][i] != '\0'; i++) {
 if (first[idx][i] == symbol)
 return;
 }
 int len = strlen(first[idx]);
 first[idx][len] = symbol;
 first[idx][len + 1] = '\0';
}
void computeFirst(char nonTerminal) {
 int idx = nonTerminal - 'A';
 if (visited[idx]) return;
 visited[idx] = 1;
 for (int i = 0; i < numProductions; i++) {
```

```
if (productions[i][0] == nonTerminal) {
int j = 2;
while (productions[i][j] != '\0') {
char symbol = productions[i][j];
if (!isNonTerminal(symbol)) {
addToFirst(nonTerminal, symbol);
break;
} else {
computeFirst(symbol);
int symIdx = symbol - 'A';
for (int k = 0; first[symIdx][k] != '\0'; k++) {
if (first[symIdx][k] != '#')
addToFirst(nonTerminal, first[symIdx][k]);
}
// If epsilon is not in FIRST(symbol), stop
if (strchr(first[symIdx], '#') == NULL)
break;
else
j++; // Continue to next symbol
}
}
if (productions[i][j] == '\0')
addToFirst(nonTerminal, '#');
}
}
}
int main() {
printf("Enter number of productions: ");
scanf("%d", &numProductions);
printf("Enter productions (e.g., E=TX or X=#):\n");
for (int i = 0; i < numProductions; i++) {
```

```
scanf("%s", productions[i]);
}
printf("\nFIRST sets:\n");
for (int i = 0; i < numProductions; i++) {
char nonTerminal = productions[i][0];
if (!visited[nonTerminal - 'A']) {
computeFirst(nonTerminal);
printf("FIRST(%c) = { ", nonTerminal);
for (int j = 0; first[nonTerminal - 'A'][j] != '\0'; j++)
printf("%c ", first[nonTerminal - 'A'][j]);
printf("}\n");
}
}
return 0;
}
```

Output

```
Enter number of productions: 4
Enter productions (e.g., E=TX or X=#):
S=(Ea)
E=E+T
T=*F
F=#

FIRST sets:
FIRST(S) = { ( }
FIRST(E) = { }
FIRST(T) = { * }
FIRST(F) = { # }


=== Code Execution Successful ===
```

Experiment 12

Objective: Write a program to find the FOLLOW function of a grammar.

Code:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
char production[10][10];
char follow[10];
int n, m = 0;
void findFollow(char c);
void calculateFollow(char c) {
   if (production[0][0] == c)
      follow[m++] = '$';
   for (int i = 0; i < n; i++) {
      for (int j = 2; production[i][j] != '\0'; j++) {
         if (production[i][j] == c) {
            if (production[i][j + 1] != '\0') {
               if (islower(production[i][j + 1]))
                  follow[m++] = production[i][j + 1];
               else
                  findFollow(production[i][j + 1]);
            }
            if (production[i][j + 1] == '\0' && c != production[i][0])
               findFollow(production[i][0]);
         }
      }
   }
}
void findFollow(char c) {
```

```
    calculateFollow(c);
}
int main() {
    int i;
    char c, ch;
    printf("Enter number of productions: ");
    scanf("%d", &n);
    printf("Enter the productions:\n");
    for (i = 0; i < n; i++) {
        scanf("%s", production[i]);
    }
    printf("Enter the non-terminal to find FOLLOW: ");
    scanf(" %c", &c);
    findFollow(c);
    printf("FOLLOW(%c) = { ", c);
    for (i = 0; i < m; i++) {
        printf("%c ", follow[i]);
    }
    printf("}\n");
    return 0;
}
```

Output

```
Enter number of productions: 3
Enter the productions:
S=AB
A=a
B=b
Enter the non-terminal to find FOLLOW: A
FOLLOW(A) = { $ }


=== Code Execution Successful ===
```

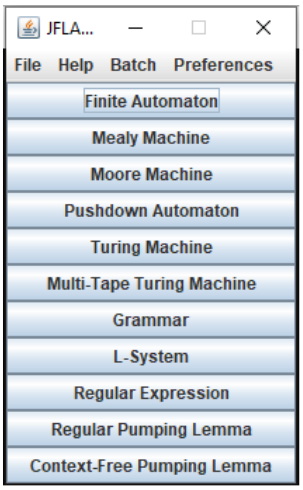
EXPERIMENT - 8

Objective : Write a program to convert NFA to DFA.

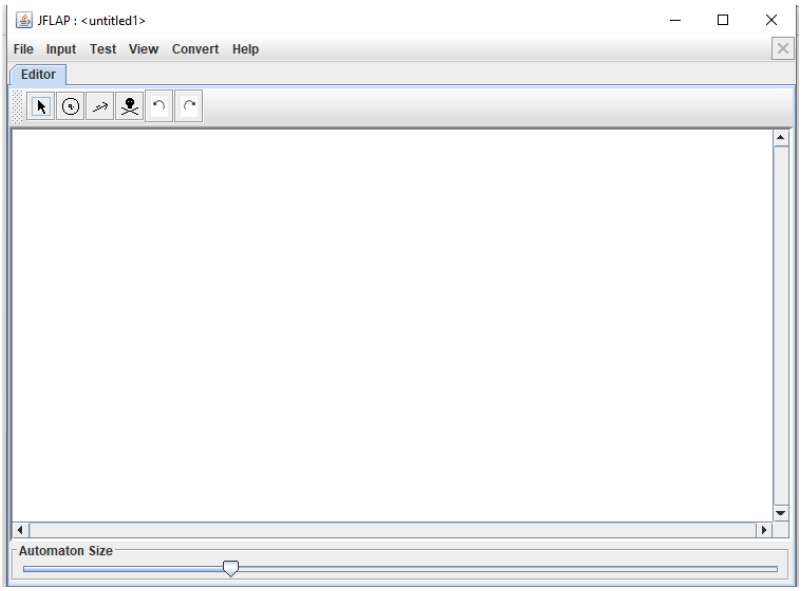
AIM: The aim of this practical is to simulate DFA through any simulation tool such as JFLAP and verify the DFA by passing various valid input strings in it.

1. Draw the DFA for a string that begins with zero or more number of ‘a’ and followed by odd number of ‘b’

Step 1 : RUN JFLAP



Step 2: Click on the Finite Automaton for DFA simulation



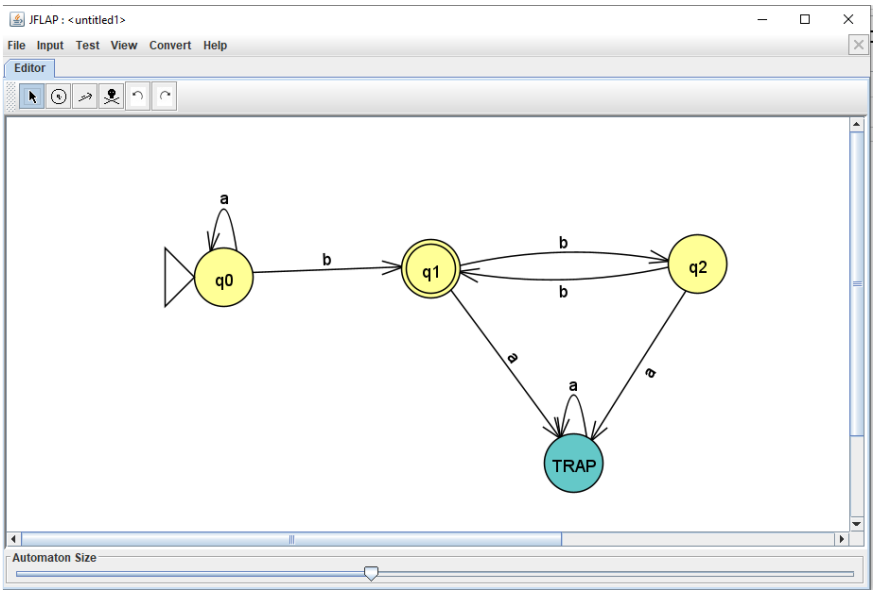
Step 3 : Draw DFA using pick and drop

- State creator

- Transition creator and

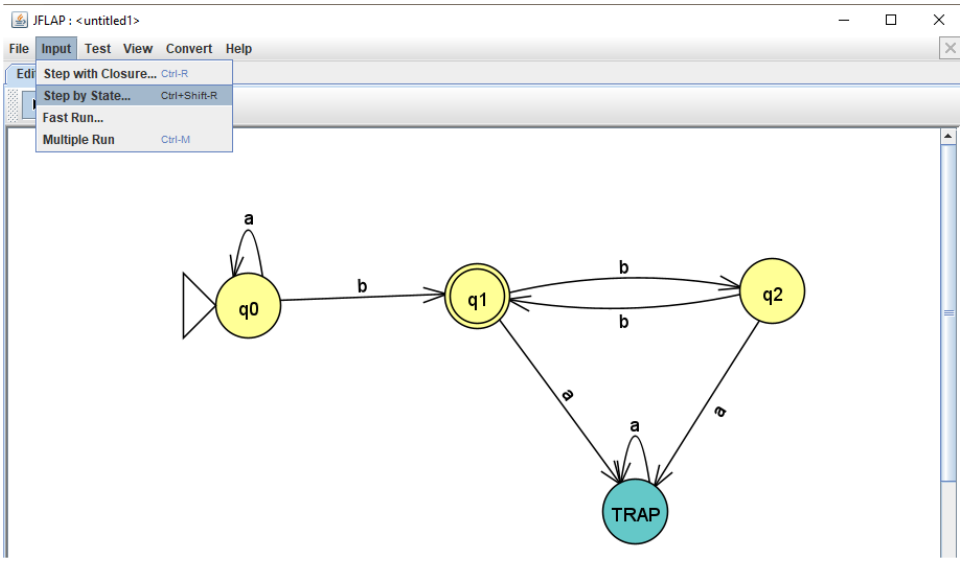
- Attribute editor

Step 4 :

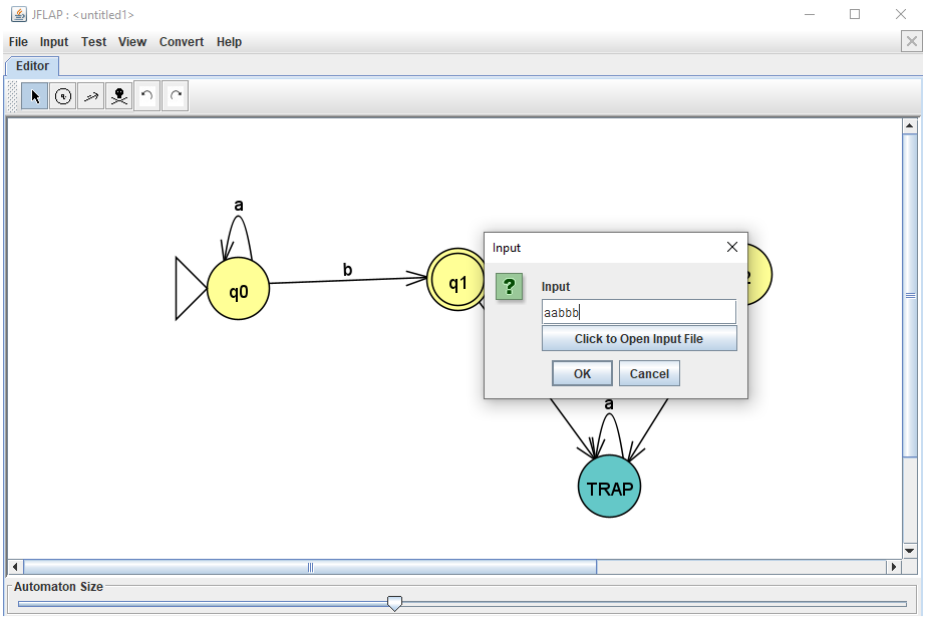


Step 5:

In Menu Bar , click on Input and then on step by state



Step 6 : pass the test input string to test for eg aabbb



Step 7 : keep pressing step for n number of times where n represents the length of the string . at last after passing the last input symbol , final state should be reachable