

PRACTICAL 1 : Study the working and implementation of LEX tool .

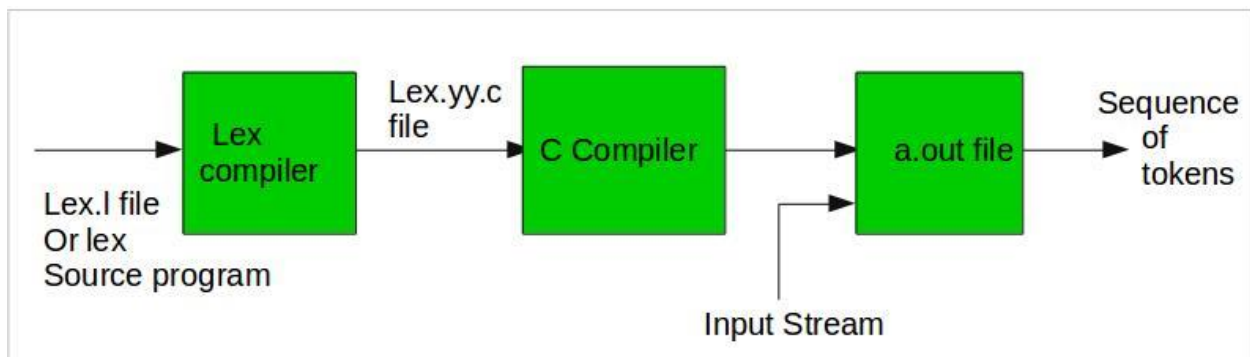
Aim : The aim of this practical is to study the working of FLEX tool which can be used in first phase of the compiler design .

FLEX (fast lexical analyzer generator) is a tool/computer program for generating lexical analyzers (scanners or lexers) written by Vern Paxson in C around 1987. It is used together with Berkeley Yacc parser generator or GNU Bison parser generator. Flex and Bison both are more flexible than Lex and Yacc and produces faster code.

Bison produces parser from the input file provided by the user. The function `yylex()` is automatically generated by the flex when it is provided with a `.l` file and this `yylex()` function is expected by parser to call to retrieve tokens from current/this token stream.

The function `yylex()` is the main flex function that runs the Rule Section and extension (`.l`) is the extension used to save the programs.

Given image describes how the Flex is used:



Step 1: An input file describes the lexical analyzer to be generated named `lex.l` is written in lex language. The lex compiler transforms `lex.l` to C program, in a file that is always named `lex.yy.c`.

Step 2: The C compiler compile `lex.yy.c` file into an executable file called `a.out`.

Step 3: The output file `a.out` take a stream of input characters and produce a stream of tokens.

Program Structure:

In the input file, there are 3 sections:

1. Definition Section: The definition section contains the declaration of variables, regular definitions, manifest constants. In the definition section, text is enclosed in “`%{ %}`” brackets. Anything written in this brackets is copied directly to the file `lex.yy.c`

Syntax:

`%{`

// Definitions

% }

2. Rules Section: The rules section contains a series of rules in the form: *pattern action* and pattern must be unintended and action begin on the same line in { } brackets. The rule section is enclosed in “%% %%”.

Syntax:

%%

pattern action

%%

Examples: Table below shows some of the pattern matches.

Pattern	It can match with
[0-9]	all the digits between 0 and 9
[0+9]	either 0, + or 9
[0, 9]	either 0, ‘, ‘ or 9
[0 9]	either 0, ‘ ‘ or 9
[-09]	either -, 0 or 9
[-0-9]	either – or all digit between 0 and 9
[0-9]+	one or more digit between 0 and 9
[^a]	all the other characters except a
[^A-Z]	all the other characters except the upper case letters
a{2, 4}	either aa, aaa or aaaa
a{2, }	two or more occurrences of a
a{4}	exactly 4 a’s i.e, aaaa

Pattern	It can match with
.	any character except newline
a*	0 or more occurrences of a
a+	1 or more occurrences of a
[a-z]	all lower case letters
[a-zA-Z]	any alphabetic letter
w(x y)z	wxz or wyz

3. User Code Section: This section contains C statements and additional functions. We can also compile these functions separately and load with the lexical analyzer.

Basic Program Structure:

```
% {
// Definitions
% }

%%

Rules

%%
```

User code section.

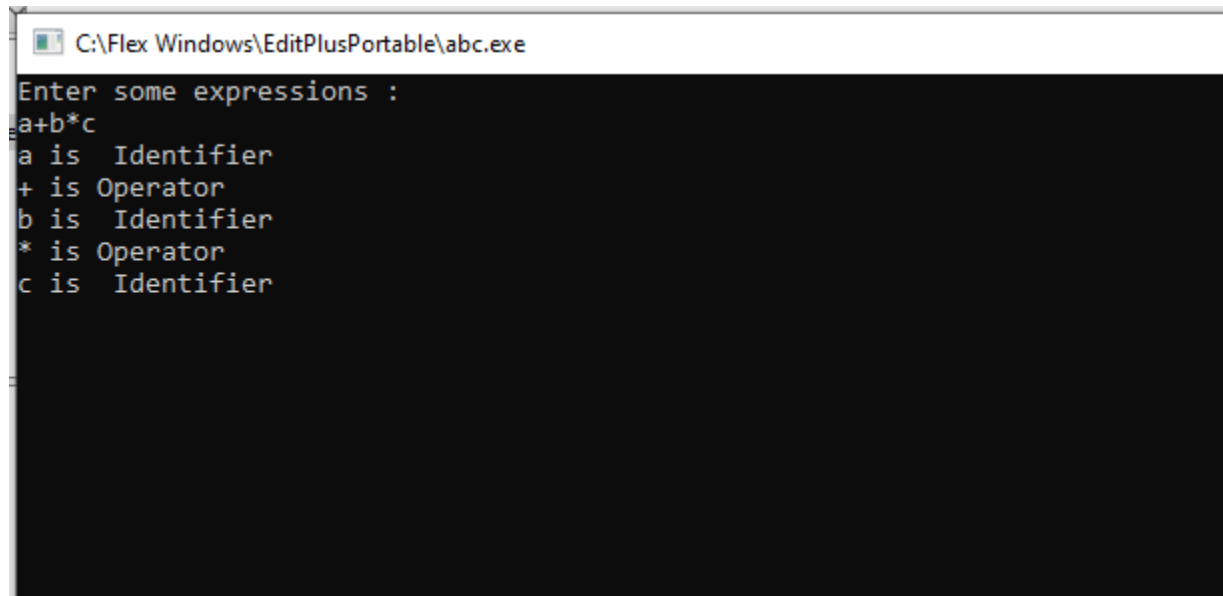
PRACTICAL 2: Write a code in FLEX for the implementation of lexical analysis for any input string

Aim : The aim of this practical is to gain the working knowledge to implement a program in FLEX tool and understand how a string can be tokenized in the first phase of compiler design .

PROGRAM

```
%{  
    #include<stdio.h>  
%}  
  
%%  
int|float|char|if|else|for|while {printf("%s --->Keyword\n",yytext);}   
[a-zA-Z]([a-zA-Z0-9])* { printf("%s is Identifier\n",yytext); }   
([0-9]|([0-9])+"[0-9]) { printf("%s is Number\n",yytext); }   
"+"|"-"|"*"|" "/"|"=" {printf("%s is Operator\n",yytext); }   
[ \t\n]+ /* Eating up space */  
%%  
int yywrap()  
{  
return 1;  
}  
int main()  
{  
    printf("Enter some expressions :\n");  
    yylex();  
    return 0;  
}
```

Output :



```
C:\Flex Windows\EditPlusPortable\abc.exe
Enter some expressions :
a+b*c
a is Identifier
+ is Operator
b is Identifier
* is Operator
c is Identifier
```

The image shows a screenshot of a Windows command prompt window. The title bar at the top reads "C:\Flex Windows\EditPlusPortable\abc.exe". The command prompt displays the prompt "Enter some expressions :" followed by the input "a+b*c". Below the input, the program outputs the following lines: "a is Identifier", "+ is Operator", "b is Identifier", "* is Operator", and "c is Identifier".

PRACTICAL 3: Write a program to Simulate DFA and verify it against various input strings

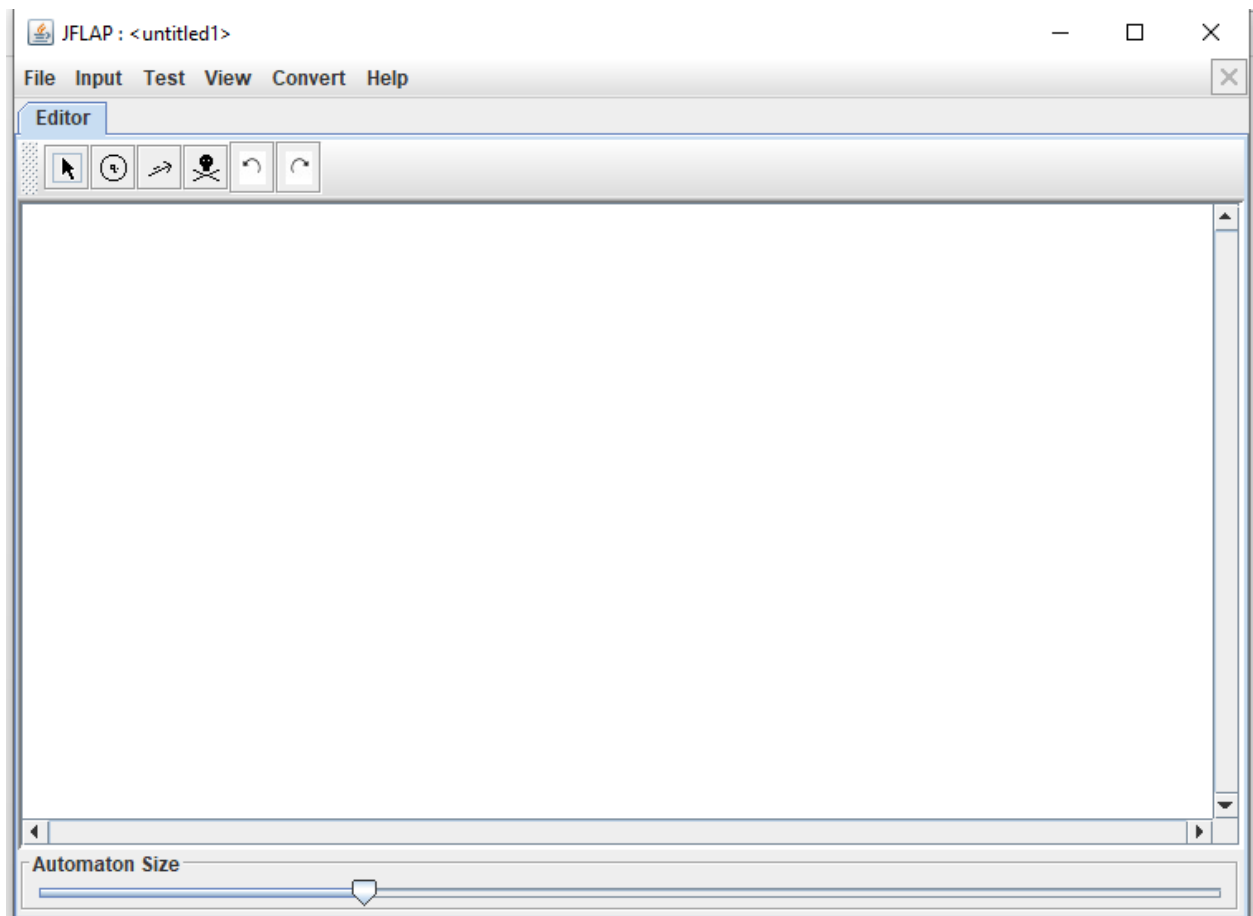
AIM: The aim of this practical is to simulate DFA through any simulation tool such as JFLAP and verify the DFA by passing various valid input strings in it.

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



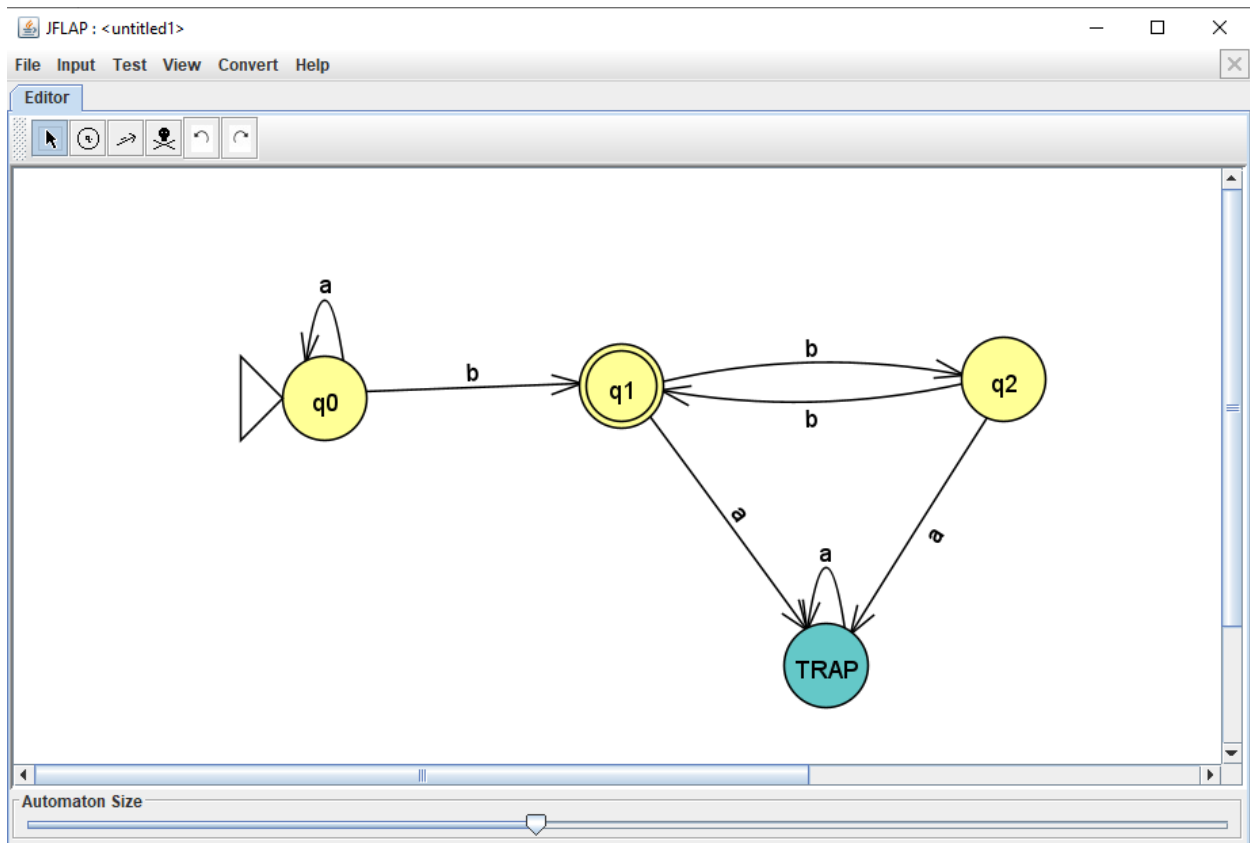
Step 2: Click on the Finite Automaton for DFA simulation



Step 3 : Draw DFA using pick and drop

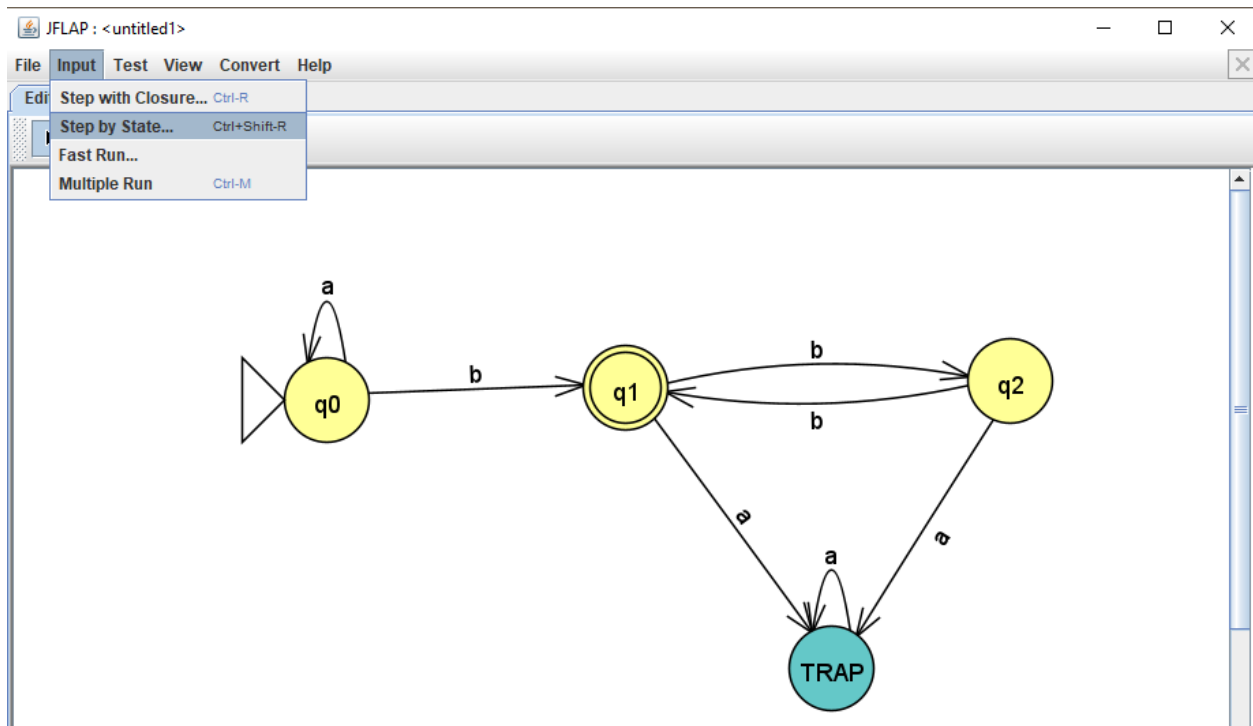
- State creator
- Transition creator and
- Attribute editor

Step 4 :

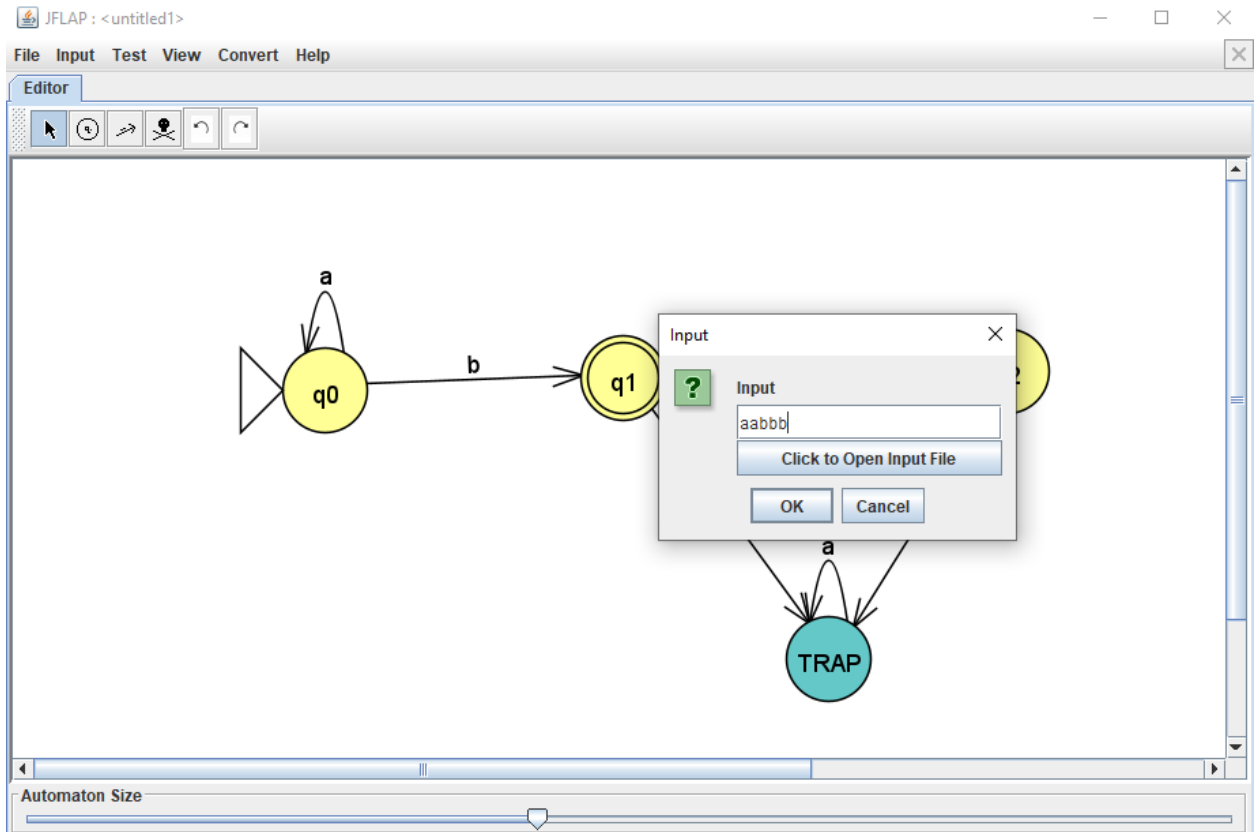


Step 5:

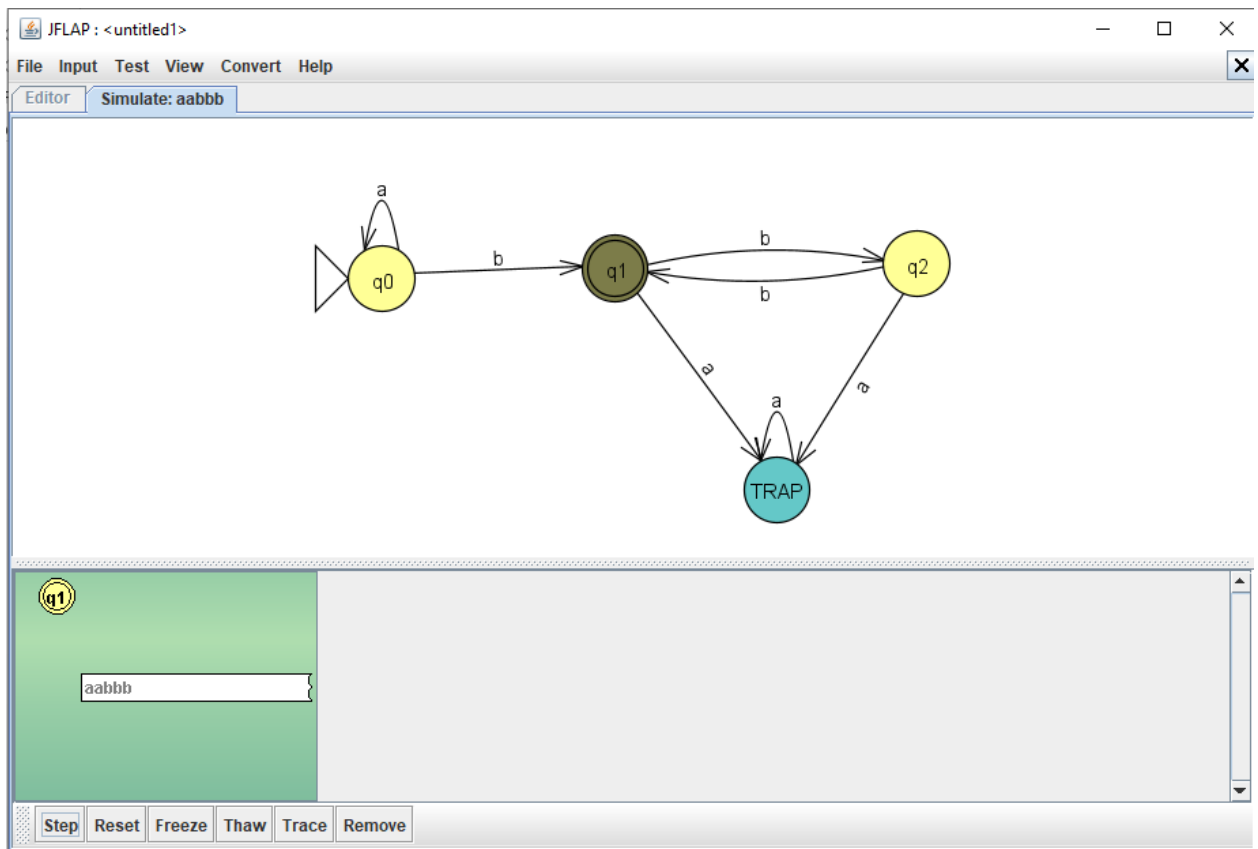
In Menu Bar , click on Input and then on step by state



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 .

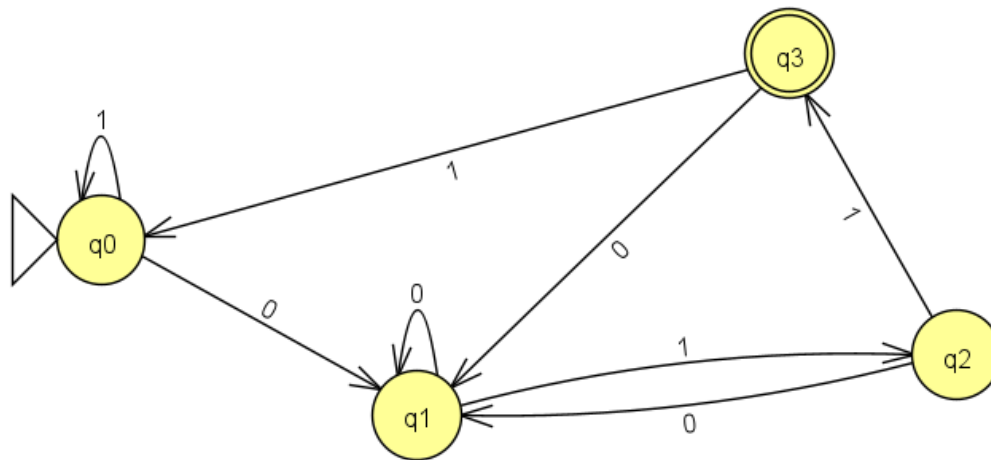


PRACTICAL 4: Write a program to Simulate any FA and generate regular expression for the given condition

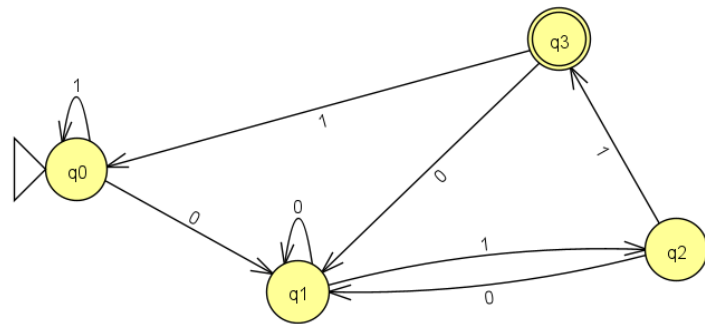
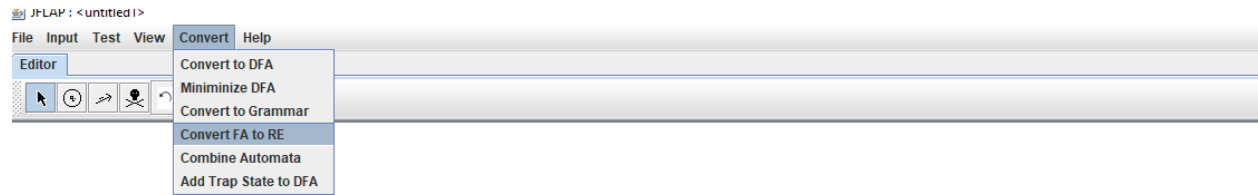
AIM: The aim of this practical is to simulate any FA through any simulation tool such as JFLAP and generate the possible regular expression which shall be valid for every possible accepted string.

Draw FA and give the RE for the language accepting strings ending with '011' over input alphabets $\Sigma = \{ 0, 1 \}$

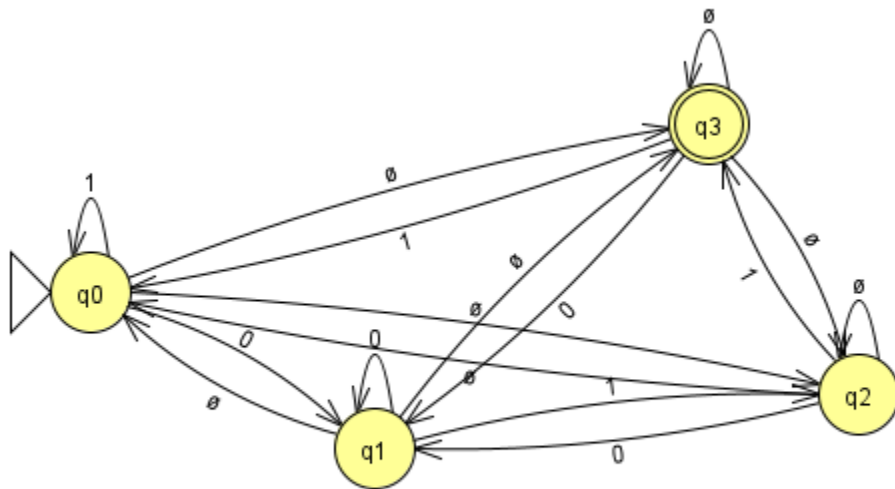
Step 1 . Draw the FA using JFLAP

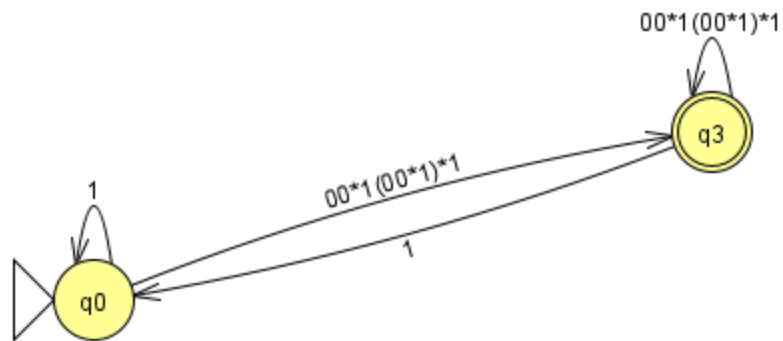


Step 2. Goto Convert -> Convert FA to RE from Menu bar

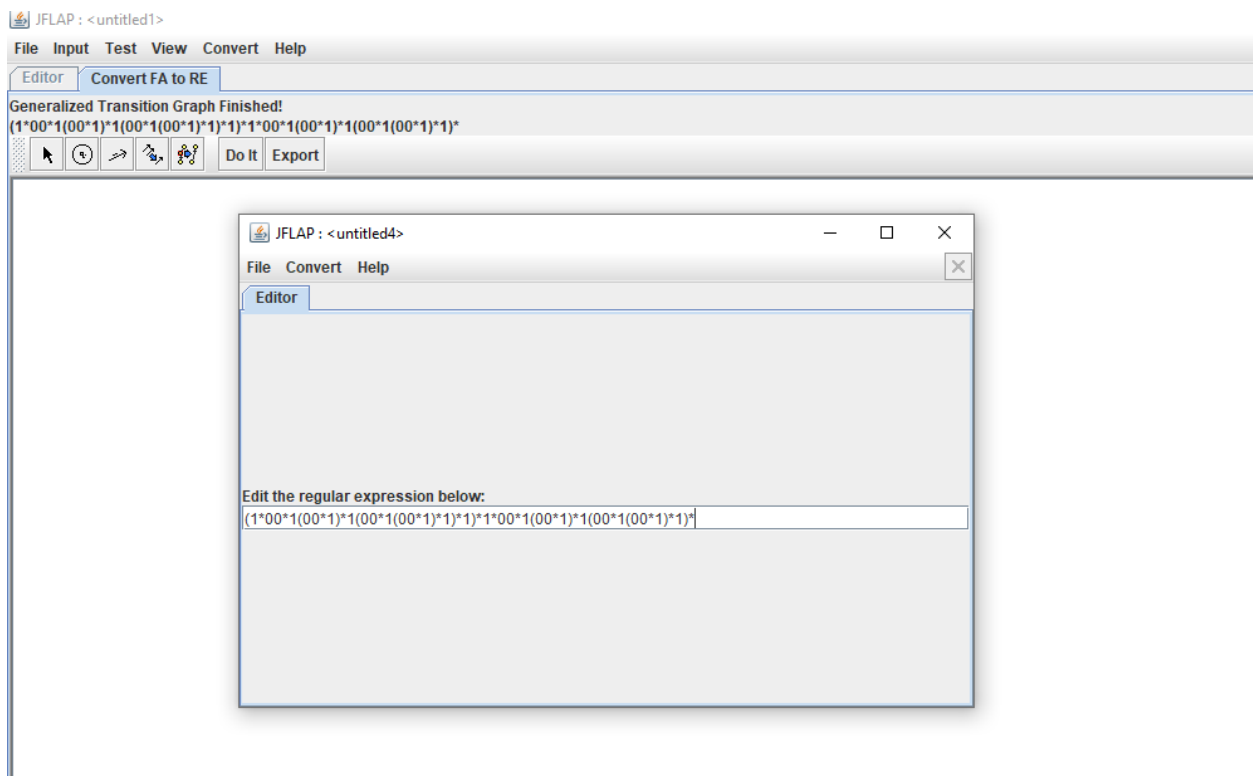


Step 3 . Click on Do it for minimized version of FA





Step 3: Click on export to check the RE generated



PRACTICAL 5: Write a program to generate First() for any symbol present in the given grammar

To compute FIRST(X) for all grammar symbols x, apply the following rules until no more terminals can be added to any FIRST set.

1. if X is terminal, then FIRST(X) is {X}.
2. if X is nonterminal and $X \rightarrow a\alpha$ is a production, then add a to FIRST(X). if $X \rightarrow \epsilon$ to FIRST(X)
3. if $\rightarrow Y_1 Y_2 \dots Y_k$ is a production, then for all i such that all of Y_1, \dots, Y_{i-1} are nonterminals and FIRST(Y_j) contains ϵ for $j=1, 2, \dots, i-1$, add every non- ϵ symbol in FIRST(Y_i) to FIRST(x). if V is in FIRST(Y_j) for $j=1, 2, \dots, k$, then add ϵ to FIRST(X).

```
#include<stdio.h>
#include<ctype.h>
void FIRST(char[],char );
void addToResultSet(char[],char);
int numOfProductions;
char productionSet[10][10];
main()
{
    int i;
    char choice;
    char c;
    char result[20];
    printf("How many number of productions ? :");
    scanf(" %d",&numOfProductions);
    for(i=0;i<numOfProductions;i++)//read production string eg: E=E+T
    {
        printf("Enter productions Number %d : ",i+1);
        scanf(" %s",productionSet[i]);
    }
    do
    {
        printf("\n Find the FIRST of :");
        scanf(" %c",&c);
        FIRST(result,c); //Compute FIRST; Get Answer in 'result' array
        printf("\n FIRST(%c)= { ",c);
```

```

        for(i=0;result[i]!='\0';i++)
            printf(" %c ",result[i]);    //Display result
        printf("}\n");
        printf("press 'y' to continue : ");
        scanf(" %c",&choice);
    }
    while(choice=='y'||choice=='Y');
}
/*
*Function FIRST:
*Compute the elements in FIRST(c) and write them
*in Result Array.
*/
void FIRST(char* Result,char c)
{
    int i,j,k;
    char subResult[20];
    int foundEpsilon;
    subResult[0]='\0';
    Result[0]='\0';
    //If X is terminal, FIRST(X) = {X}.
    if(!(isupper(c)))
    {
        addToResultSet(Result,c);
        return ;
    }
    //If X is non terminal
    //Read each production
    for(i=0;i<numOfProductions;i++)
    {
//Find production with X as LHS
        if(productionSet[i][0]==c)
        {
//If  $X \rightarrow \epsilon$  is a production, then add  $\epsilon$  to FIRST(X).
            if(productionSet[i][2]=='$') addToResultSet(Result,'$');
            //If X is a non-terminal, and  $X \rightarrow Y_1 Y_2 \dots Y_k$ 
            //is a production, then add a to FIRST(X)
            //if for some i, a is in FIRST( $Y_i$ ),
            //and  $\epsilon$  is in all of FIRST( $Y_1$ ), ..., FIRST( $Y_{i-1}$ ).
            else
            {
                j=2;
                while(productionSet[i][j]!='\0')

```



```

        {
        foundEpsilon=0;
        FIRST(subResult,productionSet[i][j]);
        for(k=0;subResult[k]!='\0';k++)
            addToResultSet(Result,subResult[k]);
        for(k=0;subResult[k]!='\0';k++)
            if(subResult[k]=='$')
            {
                foundEpsilon=1;
                break;
            }
        //No  $\epsilon$  found, no need to check next element
        if(!foundEpsilon)
            break;
        j++;
    }
}
}
return ;
}
/* addToResultSet adds the computed
*element to result set.
*This code avoids multiple inclusion of elements
*/
void addToResultSet(char Result[],char val)
{
    int k;
    for(k=0 ;Result[k]!='\0';k++)
        if(Result[k]==val)
            return;
    Result[k]=val;
    Result[k+1]='\0';
}

```

Ouput

```
How many number of productions ? :8
Enter productions Number 1 : E=TD
Enter productions Number 2 : D=+TD
Enter productions Number 3 : D=$
Enter productions Number 4 : T=FS
Enter productions Number 5 : S=*FS
Enter productions Number 6 : S=$
Enter productions Number 7 : F=(E)
Enter productions Number 8 : F=a

Find the FIRST of :E
FIRST(E)= { ( a )
press 'y' to continue : Y

Find the FIRST of :D
FIRST(D)= { + $ }
press 'y' to continue : Y

Find the FIRST of :S
FIRST(S)= { * $ }
press 'y' to continue : Y

Find the FIRST of :a
FIRST(a)= { a }
press 'y' to continue :
```