

# COMPUTER SCIENCE AND ENGINEERING PROGRAMME

<u>LIST OF EXPERIMENTS</u>	Marks
1. Write a C program to simulate a Deterministic Finite Automata (DFA) for the given language representing strings that start with a and end with a	
2. Write a C program to simulate a Non-Deterministic Finite Automata (NFA) for the given language representing strings that start with o and end with l	
3. Write a C program to find $\epsilon$ -closure for all the states in a Non-Deterministic Finite Automata (NFA) with $\epsilon$ -moves	
4. To write a C program to check whether a string belongs to the grammar $S \rightarrow 0 A 1$ $A \rightarrow 0 A \mid 1 A \mid \epsilon$	
5. To write a C program to check whether a string belongs to the grammar $S \rightarrow 0 S 0 \mid 1 S 1 \mid 0 \mid 1 \mid \epsilon$	
6. To write a C program to check whether a string belongs to the grammar $S \rightarrow 0 S 0 \mid A A \rightarrow 1 A \mid \epsilon$	
7. Write a C program to check whether a given string belongs to the language defined by a Context Free Grammar (CFG) $S \rightarrow 0 S 1 \mid \epsilon$	
8. Write a C program to check whether a given string belongs to the language defined by a Context Free Grammar (CFG) $S \rightarrow A 1 0 1 A, A \rightarrow 0 A \mid 1 A \mid \epsilon$	
9. Design DFA using simulator to accept the input string "a", "ac", and "bac"	
10. Design a Push Down Automata that accepts the language $L = \{w \mid w \in (a + b)^* \text{ and } n_a(w) = n_b(w)\}$ $n_a(w)$ is the number of a's in w $n_b(w)$ is the number of b's in w	

11. Design PDA using simulator to accept the input string $a^n b^{2n}$ i. $L = \{ a^n b^{2n} \mid w \in (a + b) \}$	
12. Design TM using simulator to accept the input string $a^n b^n$ $L = \{ a^n b^n \mid w \in (a + b) \}$	
13. Design TM using simulator to accept the input string $a^n b^{2n}$	
14. Design TM using simulator to accept the input string Palindrome ababa	
15. Design TM using simulator to accept the input string ww	
16. Design TM using simulator to perform addition of 'aa' and 'aaa'	
17. Design TM using simulator to perform subtraction of aaa-aa	
18. Design DFA using simulator to accept even number of a's	
19. Design DFA using simulator to accept odd number of a's	
20. Design DFA using simulator to accept the string the end with ab over set {a,b} W= aaabab	
21. Design DFA using simulator to accept the string having 'ab' as substring over the set {a,b}	
22. Design DFA using simulator to accept the string start with a or b over the set {a,b}	
23. Design TM using simulator to accept the input string Palindrome bbabb	
24. Design TM using simulator to accept the input string wcw	
25. Design DFA using simulator to accept the string even number of a's and odd number of b's	
26. Design DFA using simulator to accept the input string "bc", "c", and "bcaaa"	
27. Design NFA to accept any number of a's where input={a,b}	
28. Design PDA using simulator to accept the input string $a^n b^n$	
29. Design TM using simulator to perform string comparison where $w = \{aba\}$	
30. Design DFA using simulator to accept the string having 'abc' as substring over the set {a,b,c}	
31. Design DFA using simulator to accept even number of c's over the set {a,b,c}	

32. Design DFA using simulator to accept strings in which a's always appear tripled over input {a,b}	
33. Design NFA using simulator to accept the string the start with a and end with b over set {a,b} and check W= abaab is accepted or not	
34. Design NFA using simulator to accept the string that start and end with different symbols over the input {a,b}	
35. Let L be regular language, L consist set of string over { a,b) number a's minus number b's less than or equal to 2. Design DFA to accept the the language L.	
36. Design DFA using simulator to accept the string the end with abc over set {a,b,c) W= abbaababc	
37. Design NFA to accept any number of b's where input={a,b}	
38. The Automatic Teller Machine (Atm)	
39. Pattern Searching	
40. Vending Machine	
41. Natural Language Processing	
42.construct a Turing Machine to perform the function Multiplication, using Subroutines.	

**EXP NO : 1**

## **DETERMINISTIC FINITE AUTOMATA (DFA)**

**AIM :**

To write a C program to simulate a Deterministic Finite Automata.

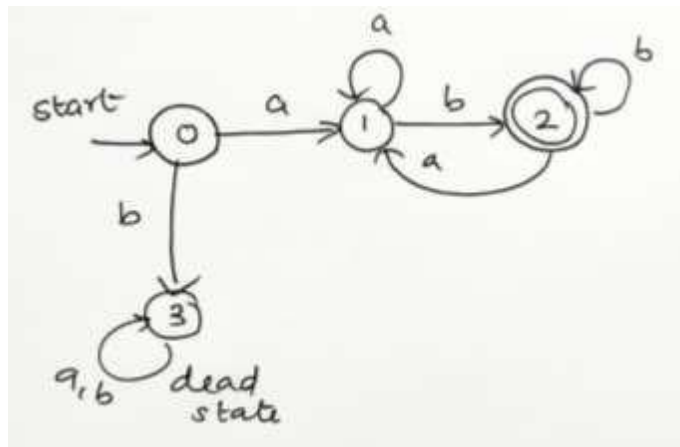
**ALGORITHM :**

1. Draw a DFA for the given language and construct the transition table.
2. Store the transition table in a two-dimensional array.
3. Initialize present\_state, next\_state and final\_state
4. Get the input string from the user.
5. Find the length of the input string.
6. Read the input string character by character.
7. Repeat step 8 for every character
8. Refer the transition table for the entry corresponding to the present state and the current input symbol and update the next state.
9. When we reach the end of the input, if the final state is reached, the input is accepted. Otherwise the input is not accepted.

### **Example:**

Simulate a DFA for the language representing strings over  $\Sigma=\{a,b\}$  that start with a and end with b

### ***Design of the DFA***



### ***Transition Table:***

State / Input	a	b
→ 0	1	3
1	1	2
2	1	2
3	3	3

### **PROGRAM :**

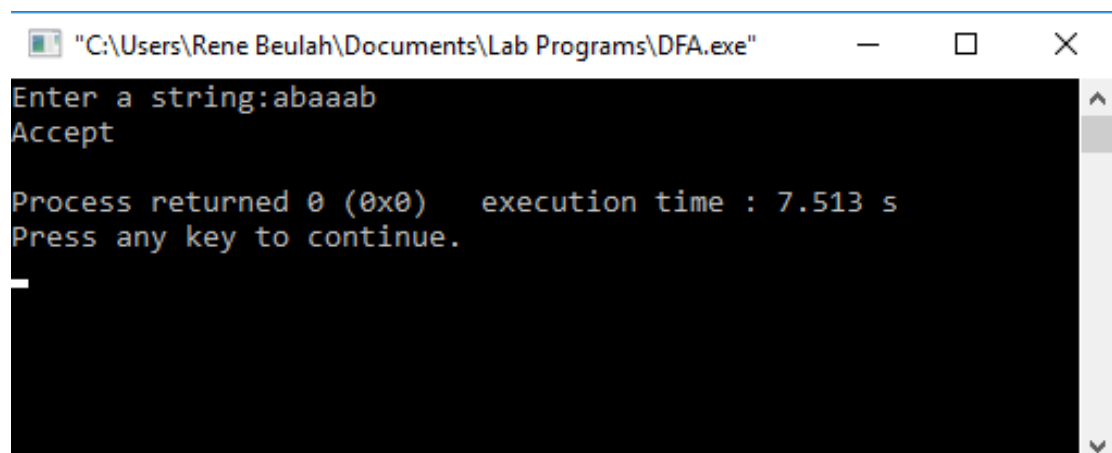
```
#include<stdio.h>
#include<string.h>
#define max 20
int main()
{
    int trans_table[4][2]={{1,3},{1,2},{1,2},{3,3}};
    int final_state=2,i; int
    present_state=0; int
    next_state=0;
    int invalid=0;
    char input_string[max];
```

```

printf("Enter a string:");
scanf("%s",input_string); int
l=strlen(input_string);
for(i=0;i<l;i++)
{
    if(input_string[i]=='a')
        next_state=trans_table[present_state][0];
    else if(input_string[i]=='b')
        next_state=trans_table[present_state][1];
    else
        invalid=1; present_state=next_state;
}
if(invalid==1)
{
    printf("Invalid input");
}
else if(present_state==final_state)
    printf("Accept\n");
else
    printf("Don't Accept\n");
}

```

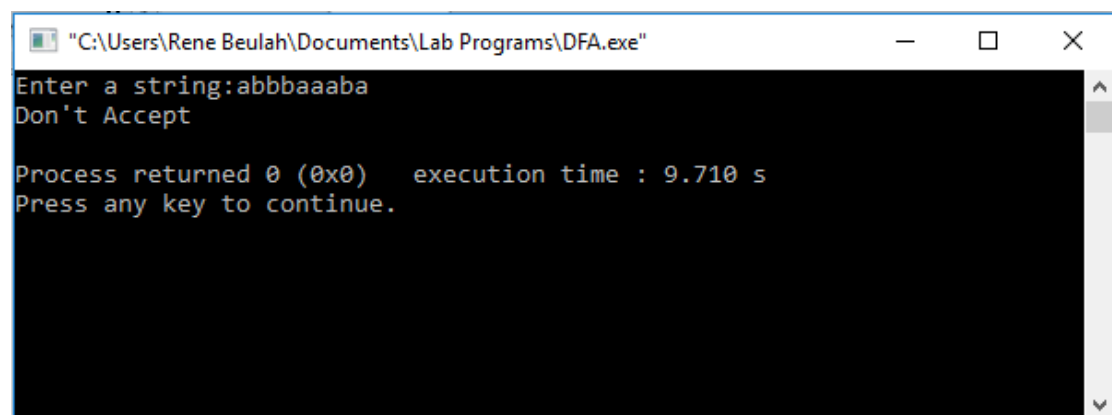
## OUTPUT



```

C:\Users\Rene Beulah\Documents\Lab Programs\DFA.exe
Enter a string:abaaab
Accept
Process returned 0 (0x0)   execution time : 7.513 s
Press any key to continue.

```



```

C:\Users\Rene Beulah\Documents\Lab Programs\DFA.exe
Enter a string:abbbbaaaba
Don't Accept
Process returned 0 (0x0)   execution time : 9.710 s
Press any key to continue.

```

## EXP NO : 2

### NON-DETERMINISTIC FINITE AUTOMATA (NFA)

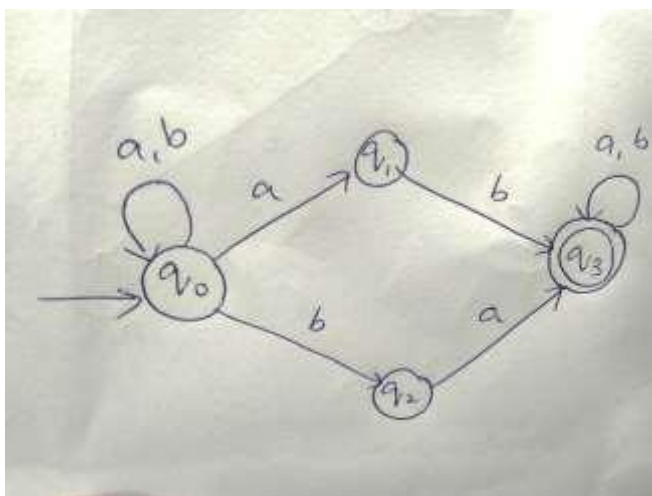
#### AIM :

To write a C program to simulate a Non-Deterministic Finite Automata.

#### ALGORITHM :

1. Get the following as input from the user.
  - i. Number of states in the NFA
  - ii. Number of symbols in the input alphabet and the symbols
  - iii. Number of final states and their names
2. Declare a 3-dimensional matrix to store the transitions and initialize all the entries with -1
3. Get the transitions from every state for every input symbol from the user and store it in the matrix.

For example, consider the NFA shown below.



There are 4 states 0, 1, 2 and 3

There are two input symbols a and b. As the array index always starts with 0, we assume 0<sup>th</sup> symbol is a and 1<sup>st</sup> symbol is b.

The transitions will be stored in the matrix as follows:

From state 0, for input a, there are two transitions to state 0 and 1, which can be stored in the matrix as

$$m[0][0][0]=0$$

$$m[0][0][1]=1$$

Similarly, the other transitions can be stored as follows:  $m[0][1][0]=0$

(From state 0, for input b, one transition is to state 0)  $m[0][1][1]=2$

(From state 0, for input b, next transition is to state 2)  $m[1][1][0]=3$

(From state 1, for input b, move to state 3)  $m[2][0][0]=3$  (From state 2,

for input a, move to state 3)  $m[3][0][0]=3$  (From state 3, for input a,

move to state 3)  $m[3][1][0]=3$  (From state 3, for input b, move to state

3)

All the other entries in the matrix will be -1 indicating no moves

4. Get the input string from the user.
5. Find the length of the input string.
6. Read the input string character by character.
7. Repeat step 8 for every character
8. Refer the transition table for the entry corresponding to the present state and the current input symbol and update the next state. As there can be more than one transition, the next state will be an array.
9. From every state in the next state array, find the list of new transitions and update the next state array.
10. When we reach the end of the input, if at least one of the final states is present in the next state array, it means there is a path to a final state. So the input is accepted. Otherwise the input is not accepted.



## PROGRAM :

```
#include<stdio.h>
#include<string.h>
int main()
{
    int i,j,k,l,m,next_state[20],n,mat[10][10][10],flag,p; int
    num_states,final_state[5],num_symbols,num_final; int
    present_state[20],prev_trans,new_trans;
    char ch,input[20];
    int symbol[5],inp,inp1;
    printf("How many states in the NFA : ");
    scanf("%d",&num_states);
    printf("How many symbols in the input alphabet : ");
    scanf("%d",&num_symbols);
    for(i=0;i<num_symbols;i++)
    {
        printf("Enter the input symbol %d : ",i+1);
        scanf("%d",&symbol[i]);
    }
    printf("How many final states : ");
    scanf("%d",&num_final);
    for(i=0;i<num_final;i++)
    {
        printf("Enter the final state %d : ",i+1);
        scanf("%d",&final_state[i]);
    }
    //Initialize all entries with -1 in Transition table
    for(i=0;i<10;i++)
    {
        for(j=0;j<10;j++)
        {
            for(k=0;k<10;k++)
            {
                mat[i][j][k]=-1;
            }
        }
    }
    //Get input from the user and fill the 3D transition table for(i=0;i<num_states;i++)
    {
        for(j=0;j<num_symbols;j++)
        {
            printf("How many transitions from state %d
            for the input %d :
            ",i,symbol[j]);
            sca
            nf("
            %d
            ",&
            n);
            for(
            k=0
            ;k<
```

n;k  
++)  
{

```

printf("Enter the transition %d from state %d
%d : ",k+1,i,symbol[j]);
                                for the input scanf("%d",&mat[i][j][k]);
                                }
                                }
                                }

printf("The transitions are stored as shown below\n");
for(i=0;i<10;i++)
{
    for(j=0;j<10;j++)
    {
        for(k=0;k<10;k++)
        {
            if(mat[i][j][k]!=-1) printf("mat[%d][%d][%d] =
%d\n",i,j,k,mat[i][j][k]);
        }
    }
}
while(1)
{
    printf("Enter the input string : ");
    scanf("%s",input);
    present_state[0]=0; prev_trans=1;
    l=strlen(input);
    for(i=0;i<l;i++)
    {
        if(input[i]=='0')
            inp1=0;
        else if(input[i]=='1')
            inp1=1;
        else
        {
            printf("Invalid input\n");
            exit(0);
        }
        for(m=0;m<num_symbols;m++)
        {
            if(inp1==symbol[m])
            {
                inp=m;
                break;
            }
        }
        new_trans=0;
        for(j=0;j<prev_trans;j++)
        {
            k=0;
            p=present_state[j];

```

```

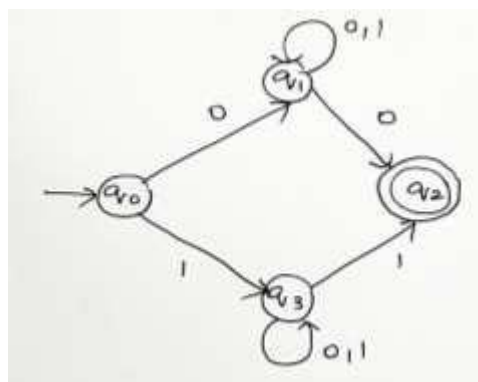
        while(mat[p][inp][k]!=-1)
        {
            next_state[new_trans++]=mat[p][inp][k]; k++;
        }
    }
    for(j=0;j<new_trans;j++)
    {
        present_state[j]=next_state[j];
    }
    prev_trans=new_trans;
}
flag=0;
for(i=0;i<prev_trans;i++)
{
    for(j=0;j<num_final;j++)
    {
        if(present_state[i]==final_state[j])
        {
            flag=1;
            break;
        }
    }
}
if(flag==1)
    printf("Accepted\n");
else
    printf("Not accepted\n");
printf("Try with another input\n");
}
}

```

### **Example:**

Simulate a NFA for the language representing strings over  $\Sigma=\{a,b\}$  that start and end with the same symbol

### ***Design of the NFA***



## Transition Table

State / Input	0	1
→ 0	1	3
1	{1,2}	1
2	-	-
3	3	{2,3}

## OUTPUT:

```
"C:\Users\Rene Beulah\Documents\Lab Programs\NFA_new.exe"
How many states in the NFA : 4
How many symbols in the input alphabet : 2
Enter the input symbol 1 : 0
Enter the input symbol 2 : 1
How many final states : 1
Enter the final state 1 : 2
How many transitions from state 0 for the input 0 : 1
Enter the transition 1 from state 0 for the input 0 : 1
How many transitions from state 0 for the input 1 : 1
Enter the transition 1 from state 0 for the input 1 : 3
How many transitions from state 1 for the input 0 : 2
Enter the transition 1 from state 1 for the input 0 : 1
Enter the transition 2 from state 1 for the input 0 : 2
How many transitions from state 1 for the input 1 : 1
Enter the transition 1 from state 1 for the input 1 : 1
How many transitions from state 2 for the input 0 : 0
How many transitions from state 2 for the input 1 : 0
How many transitions from state 3 for the input 0 : 1
Enter the transition 1 from state 3 for the input 0 : 3
How many transitions from state 3 for the input 1 : 2
Enter the transition 1 from state 3 for the input 1 : 2
Enter the transition 2 from state 3 for the input 1 : 3
The transitions are stored as shown below
mat[0][0][0] = 1
mat[0][1][0] = 3
mat[1][0][0] = 1
mat[1][0][1] = 2
mat[1][1][0] = 1
mat[3][0][0] = 3
mat[3][1][0] = 2
mat[3][1][1] = 3
Enter the input string : 0111010
Accepted
Try with another input
Enter the input string : 10010101
Accepted
Try with another input
Enter the input string : 100100
Not accepted
Try with another input
Enter the input string : 011011
Not accepted
```

## EXP NO : 3

### FINDING $\epsilon$ -CLOSURE FOR NFA WITH $\epsilon$ -MOVES

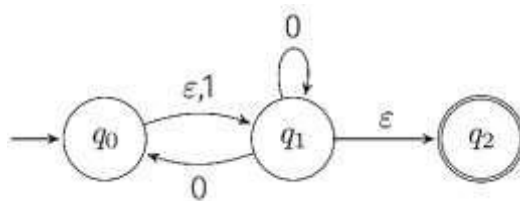
#### AIM :

To write a C program to find  $\epsilon$ -closure of a Non-Deterministic Finite Automata with  $\epsilon$ -moves

#### ALGORITHM :

1. Get the following as input from the user.
  - i. Number of states in the NFA
  - ii. Number of symbols in the input alphabet including  $\epsilon$
  - iii. Input symbols
  - iv. Number of final states and their names
2. Declare a 3-dimensional matrix to store the transitions and initialize all the entries with -1
3. Get the transitions from every state for every input symbol from the user and store it in the matrix.

For example, consider the NFA shown below.



There are 3 states 0, 1, and 2

There are three input symbols  $\epsilon$ , 0 and 1. As the array index always starts with 0, we assume 0<sup>th</sup> symbol is  $\epsilon$ , 1<sup>st</sup> symbol is 0 and 2<sup>nd</sup> symbol is 1.

The transitions will be stored in the matrix as follows:

From state 0, for input  $\epsilon$ , there is one transition to state 1, which can be stored in the matrix as

$$m[0][0][0]=1$$

From state 0, for input 0, there is no transition.

From state 0, for input 1, there is one transition to state 1, which can be stored in the matrix as

$$m[0][2][0]=1$$

Similarly, the other transitions can be stored as follows:  $m[1][0][0]=2$

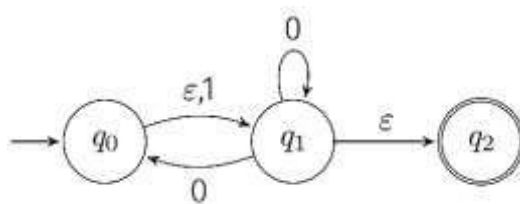
(From state 1, for input  $\epsilon$ , the transition is to state 2)  $m[1][1][0]=1$

(From state 1, for input 0, the transition is to state 1) All the other entries in the matrix will be -1 indicating no moves

4. Initialize a two-dimensional matrix  $e\_closure$  with -1 in all the entries.
5.  $\epsilon$ -closure of a state  $q$  is defined as the set of all states that can be reached from state  $q$  using only  $\epsilon$ -transitions.

Example:

Consider the NFA with  $\epsilon$ -transitions given below:



$$\epsilon\text{-closure}(0)=\{0,1,2\}$$

$$\epsilon\text{-closure}(1)=\{1,2\}$$

$$\epsilon\text{-closure}(2)=\{2\}$$

Here, we see that  $\epsilon$ -closure of every state contains that state first. So initialize the first entry of the array  $e\_closure$  with the same state.  $e\_closure(0,0)=0$ ;

e\_closure(1,0)=1;

e\_closure(2,0)=2;

6. For every state  $i$ , find  $\epsilon$ -closure as follows:

If there is an  $\epsilon$ -transition from state  $i$  to state  $j$ , add  $j$  to the matrix e\_closure[ $i$ ]. Call the recursive function find\_e\_closure( $j$ ) and add the other states that are reachable from  $i$  using  $\epsilon$

7. For every state, print the  $\epsilon$ -closure values

### **The function *find\_e\_closure(i)***

This function finds  $\epsilon$ -closure of a state recursively by tracing all the  $\epsilon$ - transitions

### **PROGRAM :**

```
#include<stdio.h>
#include<string.h>
int trans_table[10][5][3];
char symbol[5],a;
int e_closure[10][10],ptr,state; void
find_e_closure(int x);
int main()
{
    int i,j,k,n,num_states,num_symbols;
    for(i=0;i<10;i++)
    {
        for(j=0;j<5;j++)
        {
            for(k=0;k<3;k++)
            {
                trans_table[i][j][k]=-1;
            }
        }
    }
    printf("How may states in the NFA with e-moves:"); scanf("%d",&num_states);
    printf("How many symbols in the input alphabet including e :");
    scanf("%d",&num_symbols);
    printf("Enter the symbols without space. Give 'e' first:");
    scanf("%s",symbol);
    for(i=0;i<num_states;i++)
    {
        for(j=0;j<num_symbols;j++)
```



```

        {
            printf("How many transitions from state %d for
the input scanf("%d",&n);
for(k=0;k<n;k++)
{
            printf("Enter the transitions %d from state %d
for the input scanf("%d",&trans_table[i][j][k]);

        }
    }
}
for(i=0;i<10;i++)
{
    for(j=0;j<10;j++)
    {
        e_closure[i][j]=-1;
    }
}
for(i=0;i<num_states;i++)
e_closure[i][0]=i;
for(i=0;i<num_states;i++)
{
    if(trans_table[i][0][0]==-1)
        continue;
    else
    {
        state=i;
        ptr=1;
        find_e_closure(i);
    }
}
for(i=0;i<num_states;i++)
{
    printf("e-closure(%d)= {" ,i);
    for(j=0;j<num_states;j++)
    {
        if(e_closure[i][j]!=-1)
        {
            printf("%d, ",e_closure[i][j]);
        }
    }
    printf("}\n");
}
}
void find_e_closure(int x)
{

```

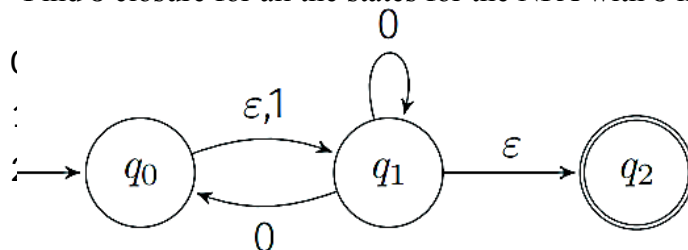
```

int i,j,y[10],num_trans;
i=0;
while(trans_table[x][0][i]!=-1)
{
    y[i]=trans_table[x][0][i];
    i=i+1;
}
num_trans=i;
for(j=0;j<num_trans;j++)
{
    e_closure[state][ptr]=y[j];
    ptr++; find_e_closure(y[j]);
}
}

```

### Example:

Find  $\epsilon$ -closure for all the states for the NFA with  $\epsilon$ -moves given below:



### TRANSITION TABLE :

State / Input	$\epsilon$	0	1
$\rightarrow$ 0	1	-	1
1	2	{0,1}	-
2	-	-	-

## OUTPUT :

```
"C:\Users\Rene Beulah\Documents\Lab Programs\NFA with e....
How may states in the NFA with e-moves:3
How many symbols in the input alphabet including e :3
Enter the symbols without space. Give 'e' first:e01
How many transitions from state 0 for the input e:1
Enter the transitions 1 from state 0 for the input e :1
How many transitions from state 0 for the input 0:0
How many transitions from state 0 for the input 1:1
Enter the transitions 1 from state 0 for the input 1 :1
How many transitions from state 1 for the input e:1
Enter the transitions 1 from state 1 for the input e :2
How many transitions from state 1 for the input 0:2
Enter the transitions 1 from state 1 for the input 0 :0
Enter the transitions 2 from state 1 for the input 0 :1
How many transitions from state 1 for the input 1:0
How many transitions from state 2 for the input e:0
How many transitions from state 2 for the input 0:0
How many transitions from state 2 for the input 1:0
e-closure(0)= {0, 1, 2, }
e-closure(1)= {1, 2, }
e-closure(2)= {2, }

Process returned 3 (0x3)   execution time : 43.311 s
Press any key to continue.
```

## **EXP NO : 4**

### **CHECKING WHETHER A STRING BELONGS TO A GRAMMAR**

#### **AIM :**

To write a C program to check whether a string belongs to the grammar

$$S \rightarrow 0 A 1$$

$$A \rightarrow 0 A \mid 1 A \mid \varepsilon$$

#### **Language defined by the Grammar:**

Set of all strings over  $\Sigma=\{0,1\}$  that start with 0 and end with 1

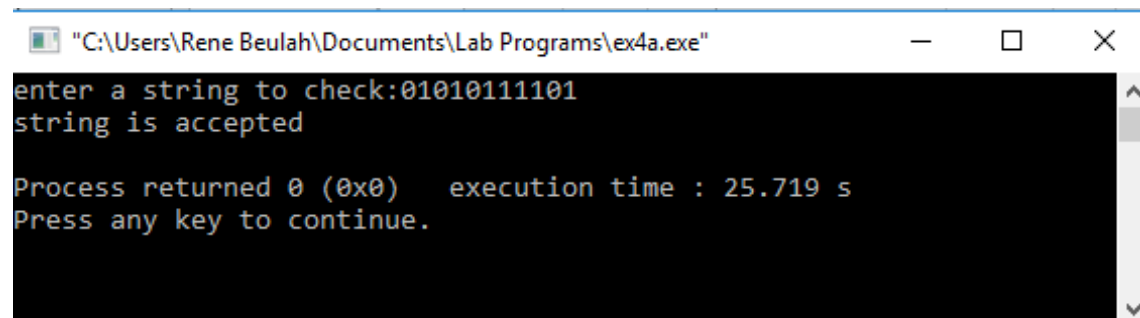
#### **ALGORTIHM :**

1. Get the input string from the user.
2. Find the length of the string.
3. Check whether all the symbols in the input are either 0 or 1. If so, print "String is valid" and go to step 4. Otherwise print "String not valid" and quit the program.
4. If the first symbol is 0 and the last symbol is 1, print "String accepted". Otherwise, print "String not accepted"

## PROGRAM :

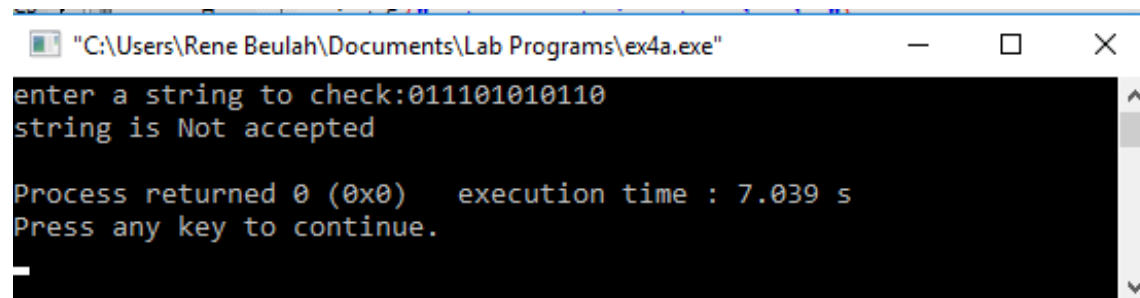
```
#include<stdio.h>
#include<string.h>
int main(){
char s[100];
int i,flag;
int l;
printf("enter a string to check:");
scanf("%s",s);
l=strlen(s);
flag=1;
for(i=0;i<l;i++)
{
    if(s[i]!='0' && s[i]!='1')
    {
        flag=0;
    }
}
if(flag!=1)
    printf("string is Not Valid\n"); if(flag==1)
{
    if (s[0]=='0'&&s[l-1]=='1')
        printf("string is accepted\n");
    else
        printf("string is Not accepted\n");
}
}
```

## OUTPUT :



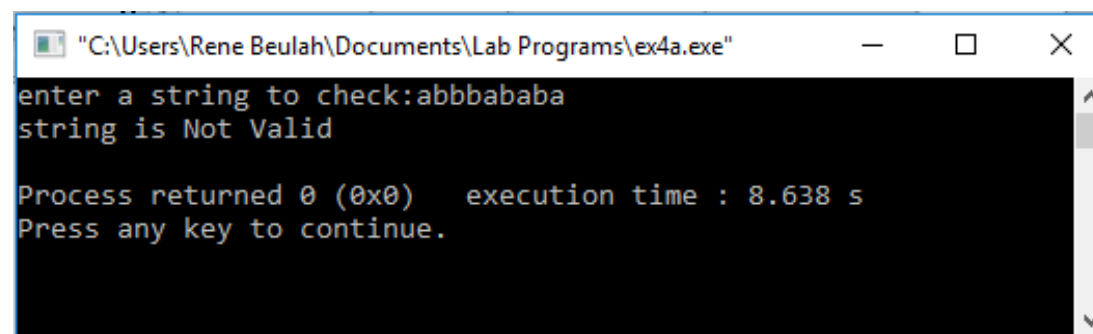
```
"C:\Users\Rene Beulah\Documents\Lab Programs\ex4a.exe"
enter a string to check:01010111101
string is accepted

Process returned 0 (0x0)   execution time : 25.719 s
Press any key to continue.
```



```
"C:\Users\Rene Beulah\Documents\Lab Programs\ex4a.exe"
enter a string to check:011101010110
string is Not accepted

Process returned 0 (0x0)   execution time : 7.039 s
Press any key to continue.
```



```
"C:\Users\Rene Beulah\Documents\Lab Programs\ex4a.exe"
enter a string to check:abbbababa
string is Not Valid

Process returned 0 (0x0)   execution time : 8.638 s
Press any key to continue.
```

## **EXP 5**

### **CHECKING WHETHER A STRING BELONGS TO A GRAMMAR**

#### **AIM :**

To write a C program to check whether a string belongs to the grammar  $S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1 \mid \epsilon$

#### **Language defined by the Grammar**

Set of all strings over  $\Sigma = \{0,1\}$  that are palindrome

#### **ALGORITHM :**

1. Get the input string from the user.
2. Find the length of the string. Let it be  $n$ .
3. Check whether all the symbols in the input are either 0 or 1. If so, print "String is valid" and go to step 4. Otherwise print "String not valid" and quit the program.
4. If the 1<sup>st</sup> symbol and  $n^{\text{th}}$  symbol are the same, 2<sup>nd</sup> symbol and  $(n-1)^{\text{th}}$  symbol are the same and so on, then the given string is palindrome. So, print "String accepted". Otherwise, print "String not accepted"

## PROGRAM :

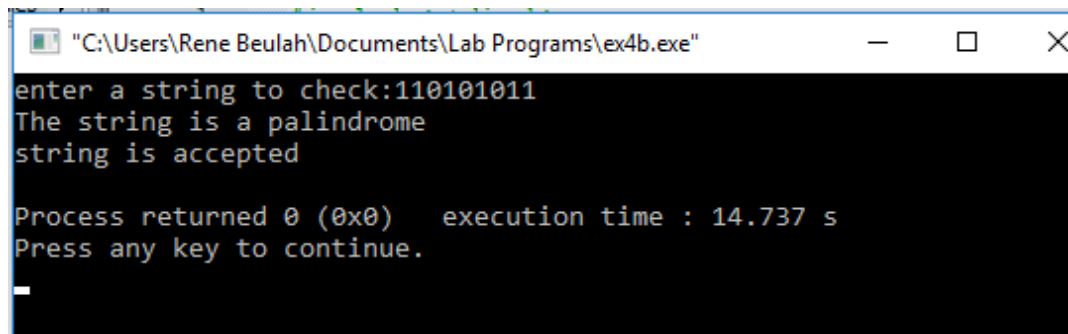
```
#include<stdio.h>
#include<string.h>
void main()
{
    char s[100];
    int i,flag,flag1,a,b;
    int l;
    printf("enter a string to check:");
    scanf("%s",s);
    l=strlen(s);
    flag=1;
    for(i=0;i<l;i++)
    {
        if(s[i]!='0' && s[i]!='1')
        {
            flag=0;
        }
    }
    if(flag!=1)
        printf("string is Not Valid\n");
    if(flag==1)
    {
        flag1=1;
        a=0;b=l-1;
        while(a!=(l/2))
        {
            if(s[a]!=s[b])
            {
                flag1=0;
            }
            a=a+1;
            b=b-1;
        }
        if (flag1==1)
        {
            printf("The string is a palindrome\n");
        }
        else
        {
            printf("string is not a palindrome\n");
        }
    }
}
```

printf("The string is not a palindrome\n");  
printf("string is Not



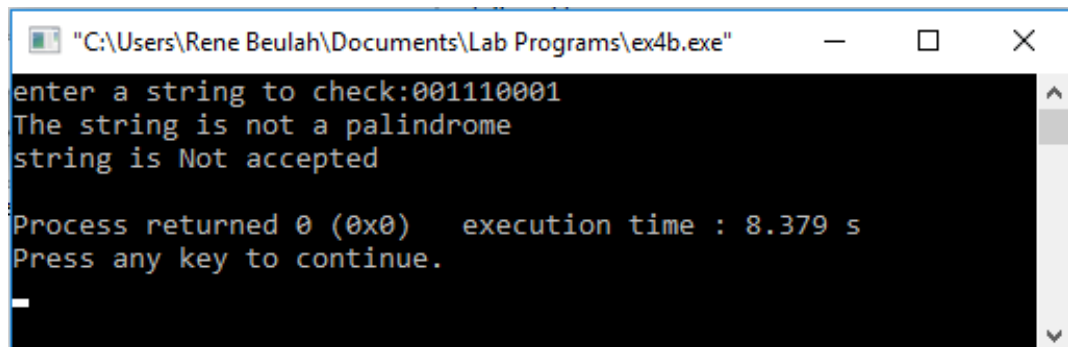
accepted\n");

## OUTPUT :



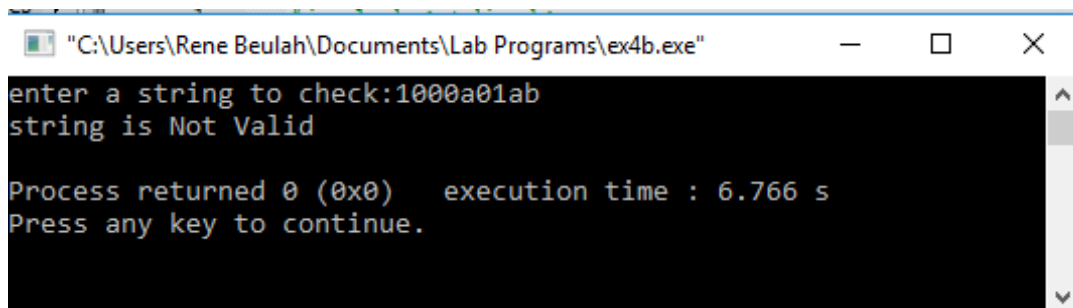
```
"C:\Users\Rene Beulah\Documents\Lab Programs\ex4b.exe"
enter a string to check:110101011
The string is a palindrome
string is accepted

Process returned 0 (0x0)   execution time : 14.737 s
Press any key to continue.
_
```



```
"C:\Users\Rene Beulah\Documents\Lab Programs\ex4b.exe"
enter a string to check:001110001
The string is not a palindrome
string is Not accepted

Process returned 0 (0x0)   execution time : 8.379 s
Press any key to continue.
_
```



```
"C:\Users\Rene Beulah\Documents\Lab Programs\ex4b.exe"
enter a string to check:1000a01ab
string is Not Valid

Process returned 0 (0x0)   execution time : 6.766 s
Press any key to continue.
_
```

## EXP 6

### CHECKING WHETHER A STRING BELONGS TO A GRAMMAR

#### AIM :

To write a C program to check whether a string belongs to the grammar

$$\begin{aligned} S &\rightarrow 0 S 0 \mid \\ A A &\rightarrow 1 A \mid \\ &\epsilon \end{aligned}$$

#### Language defined by the Grammar

Set of all strings over  $\Sigma=\{0,1\}$  satisfying  $0^n 1^m 0^n$

#### ALGORITHM :

1. Get the input string from the user.
2. Find the length of the string.
3. Check whether all the symbols in the input are either 0 or 1. If so, print “String is valid” and go to step 4. Otherwise print “String not valid” and quit the program.
4. Read the input string character by character
5. Count the number of 0's in the front and store it in the variable *count1*
6. Skip all 1's
7. Count the number of 0's in the end and store it in the variable *count2*
8. If *count1*==*count2*, print “String Accepted”. Otherwise print “String Not Accepted”

## PROGRAM :

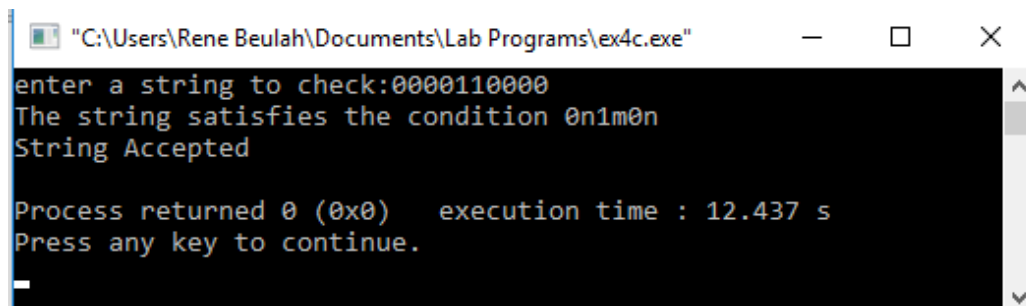
```
#include<stdio.h>
#include<string.h>
void main()
{
    char s[100];
    int i,flag,flag1,a,b;
    int l,count1,count2;
    printf("enter a string to check:");
    scanf("%s",s);
    l=strlen(s);
    flag=1;
    for(i=0;i<l;i++)
    {
        if(s[i]!='0' && s[i]!='1')
        {
            flag=0;
        }
    }
    if(flag!=1)
        printf("string is Not Valid\n");
    if(flag==1)
    {
        i=0;count1=0;
        while(s[i]=='0') // Count the no of 0s in the front
        {
            count1++;
            i++;
        }
        while(s[i]=='1')
        {
            i++; // Skip all 1s
        }
        flag1=1;
        count2=0;
        while(i<l)
        {
            if(s[i]=='0')// Count the no of 0s at the end
            {
                count2++;
            }
            else
            {
                flag1=0;
            } i++;
        }
    }
```

```

        if(flag1==1)
        {
            if(count1==count2)
            {
                printf("The string satisfies the condition\n");
                printf("String Accepted\n");
            }
            else
            {
                printf("The string does not satisfy the condition\n");
                printf("String Not Accepted\n");
            }
        }
    }
}

```

## OUTPUT :

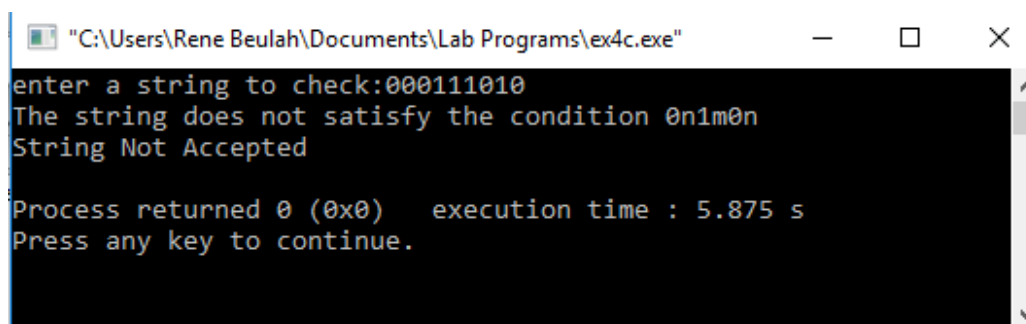


```

"C:\Users\Rene Beulah\Documents\Lab Programs\ex4c.exe"
enter a string to check:0000110000
The string satisfies the condition
String Accepted

Process returned 0 (0x0)   execution time : 12.437 s
Press any key to continue.

```

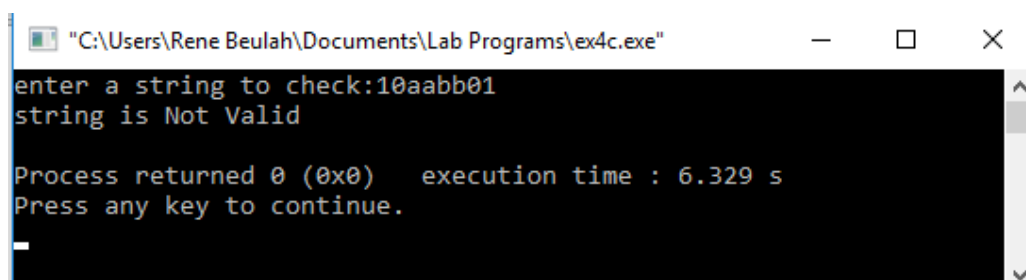


```

"C:\Users\Rene Beulah\Documents\Lab Programs\ex4c.exe"
enter a string to check:000111010
The string does not satisfy the condition
String Not Accepted

Process returned 0 (0x0)   execution time : 5.875 s
Press any key to continue.

```



```

"C:\Users\Rene Beulah\Documents\Lab Programs\ex4c.exe"
enter a string to check:10aabb01
string is Not Valid

Process returned 0 (0x0)   execution time : 6.329 s
Press any key to continue.

```

## **EXP 7**

### **CHECKING WHETHER A STRING BELONGS TO A GRAMMAR**

#### **AIM :**

To write a C program to check whether a string belongs to the grammar

$$S \rightarrow 0 S 1 \mid \varepsilon$$

#### **Language defined by the Grammar**

Set of all strings over  $\Sigma = \{0,1\}$  satisfying  $0^n 1^n$

#### **ALGORITHM :**

1. Get the input string from the user.
2. Find the length of the string.
3. Check whether all the symbols in the input are either 0 or 1. If so, print "String is valid" and go to step 4. Otherwise print "String not valid" and quit the program.
4. Find the length of the string. If the length is odd, then print "String not accepted" and quit the program. If the length is even, then go to step 5.
5. Divide the string into two halves.
6. If the first half contains only 0s and the second half contains only 1s then print "String Accepted". Otherwise print "String Not Accepted"

## PROGRAM :

```
#include<stdio.h>
#include<string.h>
void main()
{
    char s[100];
    int i,flag,flag1,flag2;
    int l;
    printf("enter a string to check:");
    scanf("%s",s);
    l=strlen(s);
    flag=1;
    for(i=0;i<l;i++)
    {
        if(s[i]!='0' && s[i]!='1')
        {
            flag=0;
        }
    }
    if(flag!=1)
        printf("string is Not Valid\n");
    if(flag==1)
    {
        if(l%2!=0) // If string length is odd
        {
            printf("The string does not satisfy the
condition 0n1n\n"); printf("String Not
Accepted\n");
        }
        else
        {
            // To
            chec
            k
            first
            half
            conta
            ins
            0s
            flag1
            =1;
            for(i=0;i<(l/2);i++)
            {
                if(s[i]!='0')
                {
                    flag1=0;
                }
            }
            // To check second half contains 1s
            flag2=1;
            for(i=l/2;i<l;i++)
            {
```

```
        if(s[i]!='1')
        {
            flag2=0;
        }
    }
```

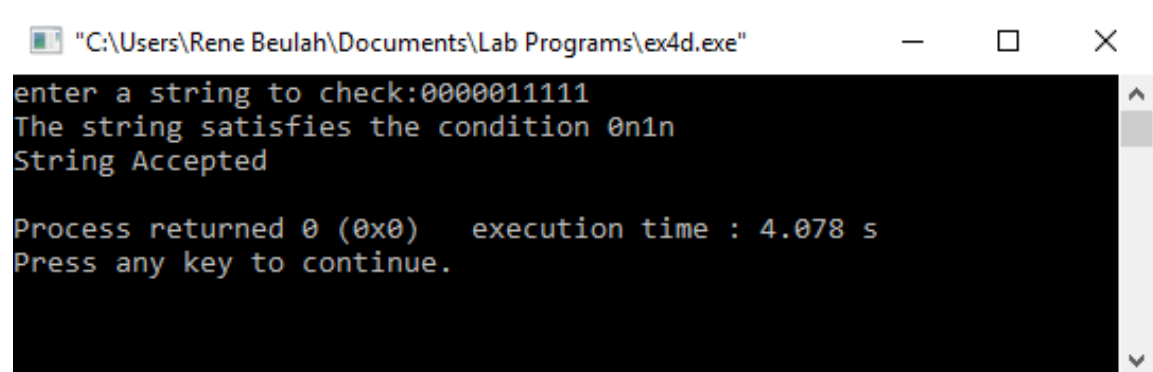


```

        if(flag1==1 && flag2==1)
        {
            printf("The string satisfies the condition
01n\n"); printf("String Accepted\n");
        }
        else
        {
            printf("The string does not satisfy the
condition 01n\n"); printf("String Not
Accepted\n");
        }
    }
}

```

## OUTPUT :

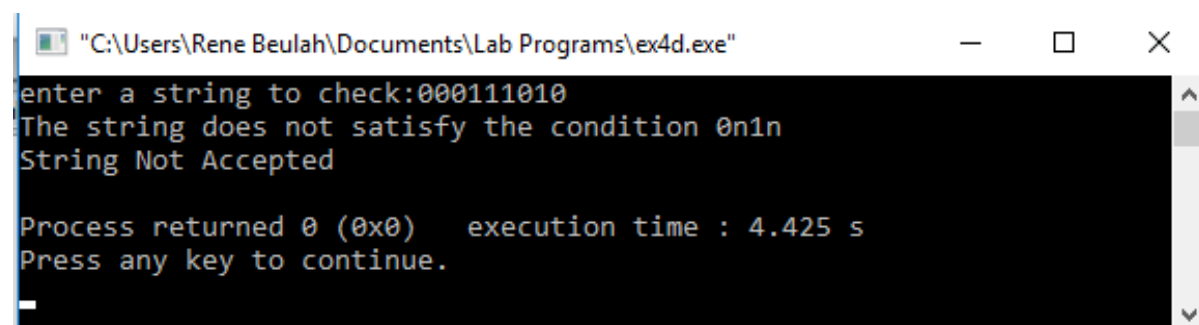


```

"C:\Users\Rene Beulah\Documents\Lab Programs\ex4d.exe"
enter a string to check:0000011111
The string satisfies the condition 01n\n
String Accepted

Process returned 0 (0x0)   execution time : 4.078 s
Press any key to continue.

```

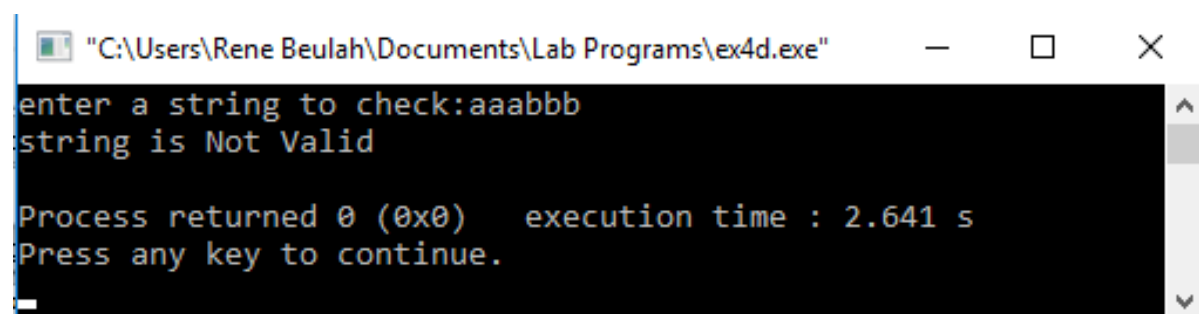


```

"C:\Users\Rene Beulah\Documents\Lab Programs\ex4d.exe"
enter a string to check:000111010
The string does not satisfy the condition 01n\n
String Not Accepted

Process returned 0 (0x0)   execution time : 4.425 s
Press any key to continue.

```



```

"C:\Users\Rene Beulah\Documents\Lab Programs\ex4d.exe"
enter a string to check:aaabbb
string is Not Valid

Process returned 0 (0x0)   execution time : 2.641 s
Press any key to continue.

```

## **EXP 8**

### **CHECKING WHETHER A STRING BELONGS TO A GRAMMAR**

#### **AIM :**

To write a C program to check whether a string belongs to the grammar  $S \rightarrow$

$A 1 0 1 A$

$A \rightarrow 0 A \mid 1 A \mid \epsilon$

#### **Language defined by the Grammar**

Set of all strings over  $\Sigma=\{0,1\}$  having 101 as a substring

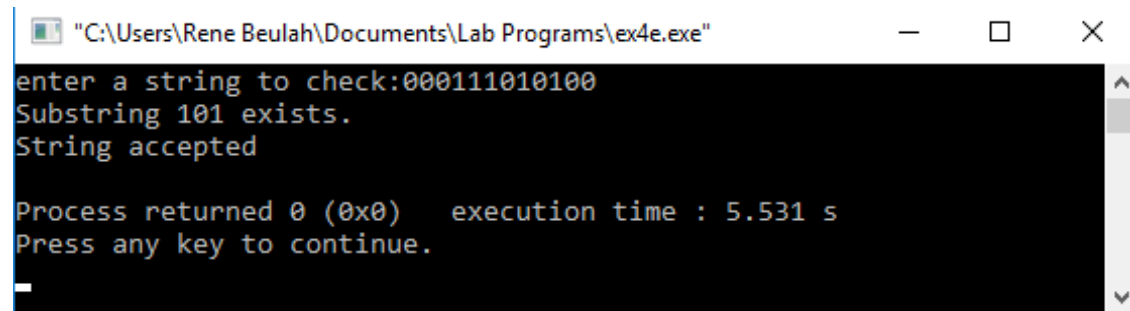
#### **ALGORITHM :**

1. Get the input string from the user.
2. Find the length of the string.
3. Check whether all the symbols in the input are either 0 or 1. If so, print “String is valid” and go to step 4. Otherwise print “String not valid” and quit the program.
4. Read the input string character by character
5. If the  $i^{\text{th}}$  input symbol is 1, check whether  $(i+1)^{\text{th}}$  symbol is 0 and  $(i+2)^{\text{th}}$  symbol is 1. If so, the string has the substring 101. So print “String Accepted”. Otherwise, print “String Not Accepted”

## PROGRAM :

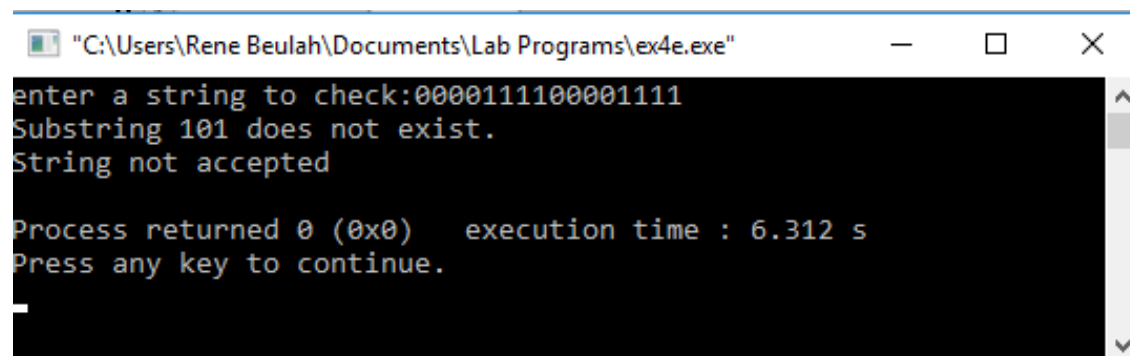
```
#include<stdio.h>
#include<string.h>
int main()
{
    char s[100];
    int i,flag,flag1;
    int l;
    printf("enter a string to check:");
    scanf("%s",s);
    l=strlen(s);
    flag=1;
    for(i=0;i<l;i++)
    {
        if(s[i]!='0' && s[i]!='1')
        {
            flag=0;
        }
    }
    if(flag==1)
        printf("string is Valid\n");
    else
        printf("string is Not Valid\n");
    if(flag==1)
    {
        flag1=0;
        for(i=0;i<l-2;i++)
        {
            if(s[i]=='1')
            {
                if(s[i+1]=='0' && s[i+2]=='1')
                {
                    flag1=1;
                    printf("Substring 101 exists. String accepted\n");
                    break;
                }
            }
        }
    }
    if(flag1==0)
        printf("Substring 101 does not exist. String not accepted\n");
}
}
```

## OUTPUT :



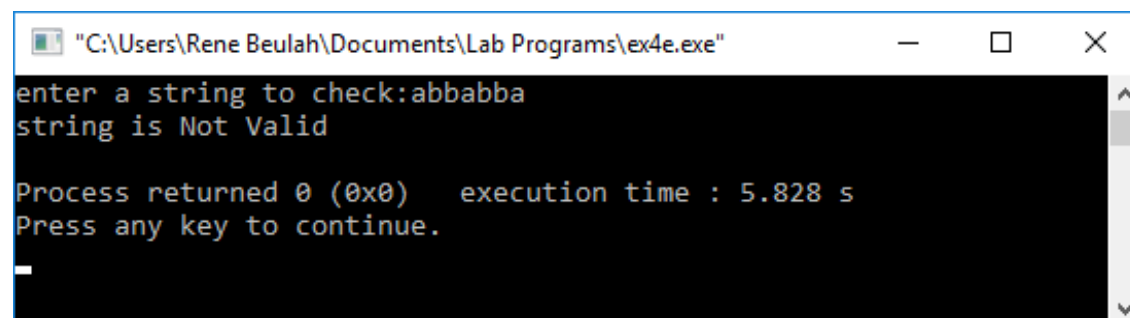
```
"C:\Users\Rene Beulah\Documents\Lab Programs\ex4e.exe"
enter a string to check:000111010100
Substring 101 exists.
String accepted

Process returned 0 (0x0)   execution time : 5.531 s
Press any key to continue.
_
```



```
"C:\Users\Rene Beulah\Documents\Lab Programs\ex4e.exe"
enter a string to check:0000111100001111
Substring 101 does not exist.
String not accepted

Process returned 0 (0x0)   execution time : 6.312 s
Press any key to continue.
_
```



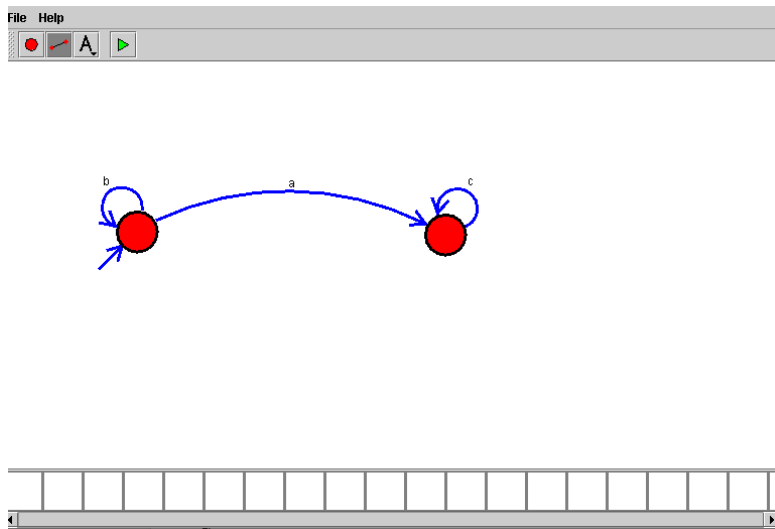
```
"C:\Users\Rene Beulah\Documents\Lab Programs\ex4e.exe"
enter a string to check:abbabba
string is Not Valid

Process returned 0 (0x0)   execution time : 5.828 s
Press any key to continue.
_
```

## EXP :9

**AIM:**Design DFA using simulator to accept the input string "a", "ac", and "bac"

Simulaton



## EXP :10

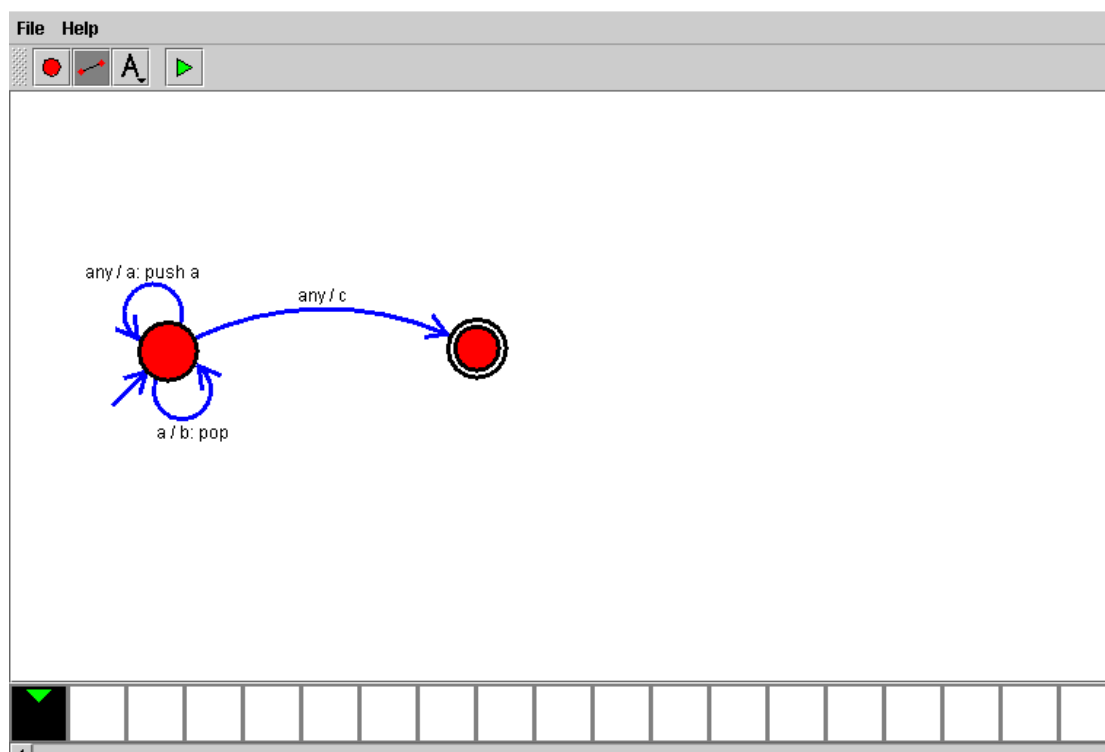
**AIM:**Design a Push Down Automata that accepts the language

$$L = \{w \mid w \in (a + b)^* \text{ and } n_a(w) = n_b(w)\}$$

$n_a(w)$  is the number of a's in w

$n_b(w)$  is the number of b's in w

Simulation

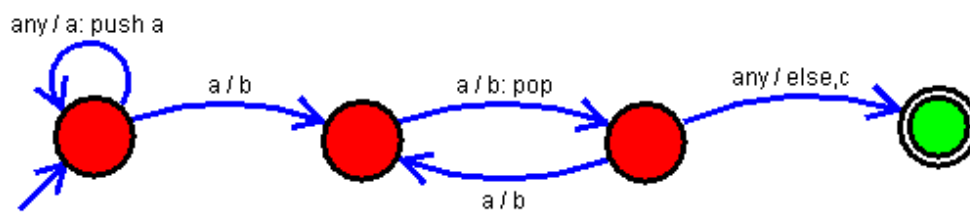
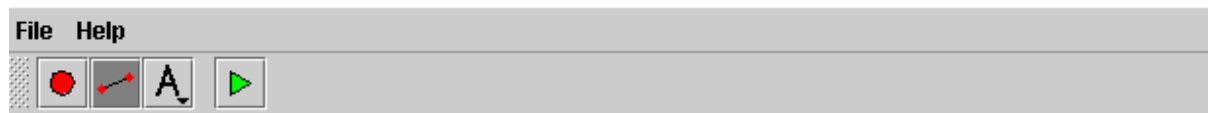


## EXP :11

**AIM:**Design PDA using simulator to accept the input string  $a^n b^{2n}$

$$L = \{ a^n b^{2n} \mid w \in (a + b) \}$$

Simulation:

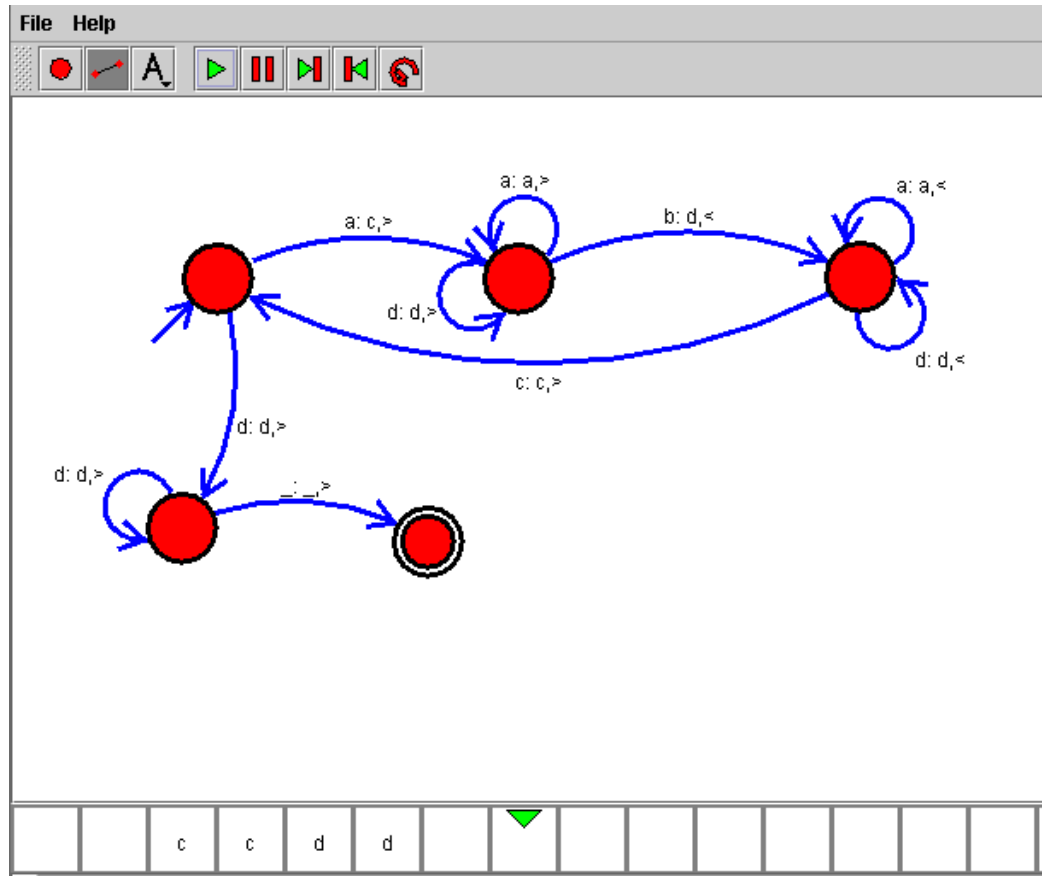


## EXP :12

**AIM:**Design TM using simulator to accept the input string  $a^n b^n$

$$L = \{ a^n b^n \mid w \in (a + b)^* \}$$

Simulation

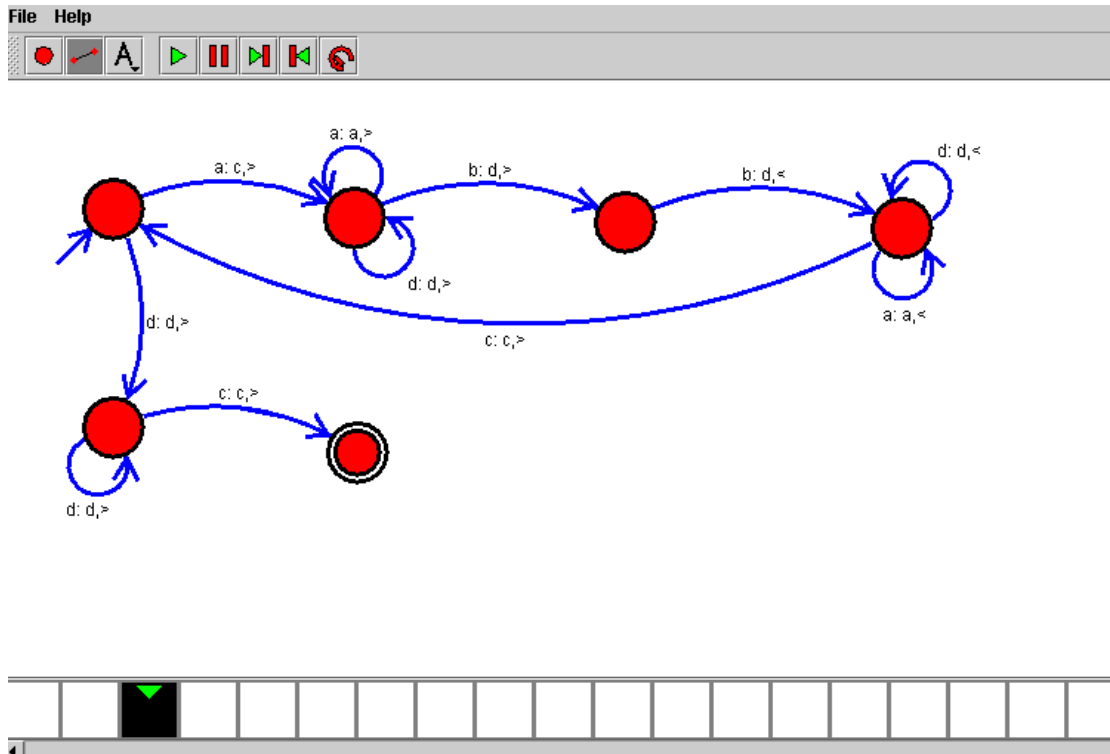




## EXP :13

**AIM:**Design TM using simulator to accept the input string  $a^n b^{2n}$

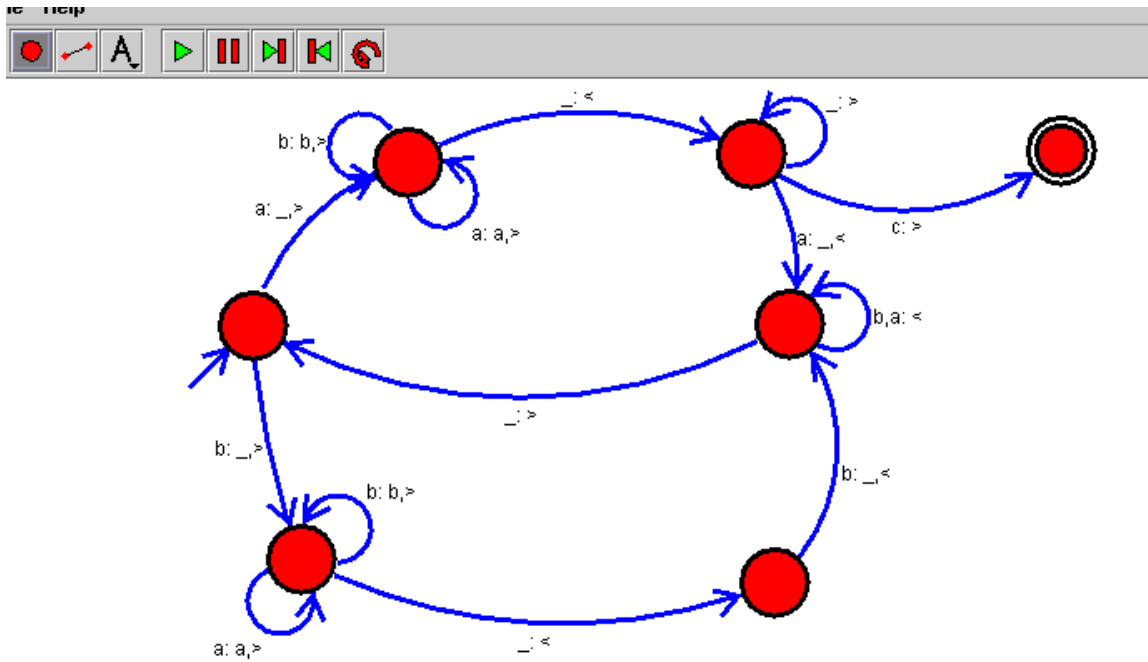
Simulation:



## EXP :14

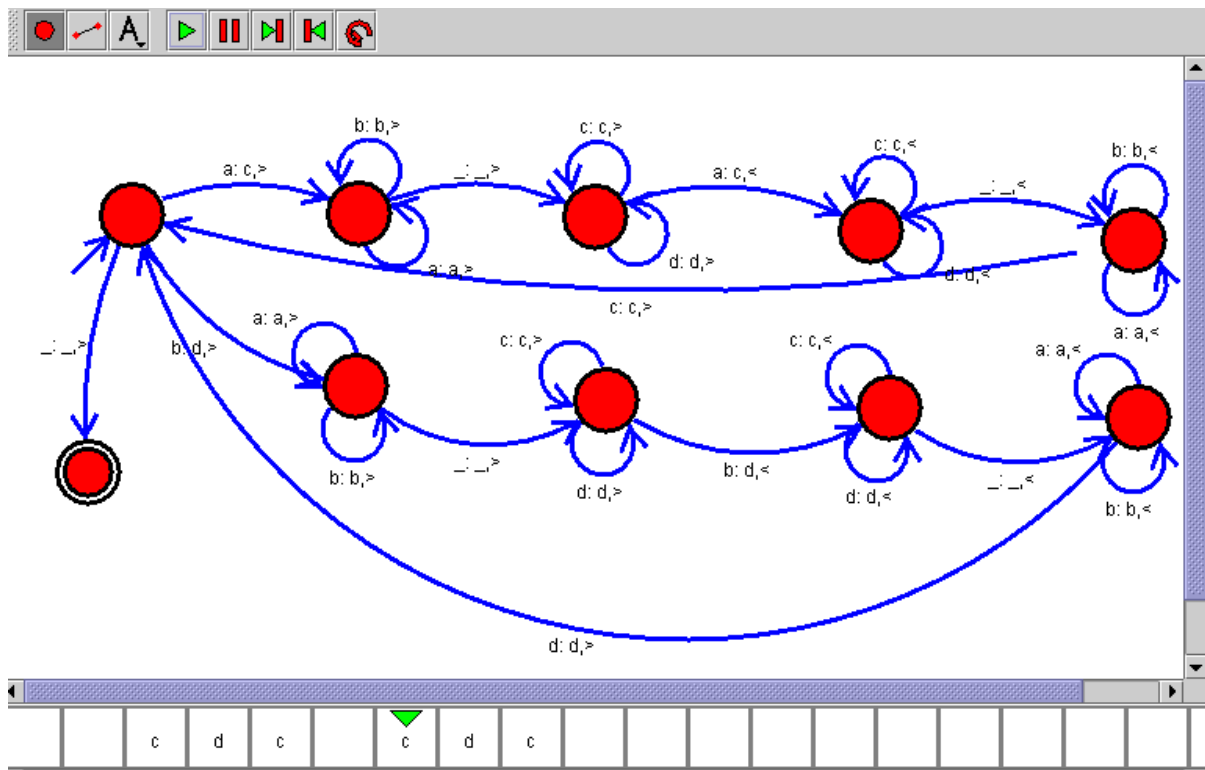
**AIM:**Design TM using simulator to accept the input string Palindrome ababa

Simulation:



## EXP :15

**AIM:** Design TM using simulator to accept the input string ww



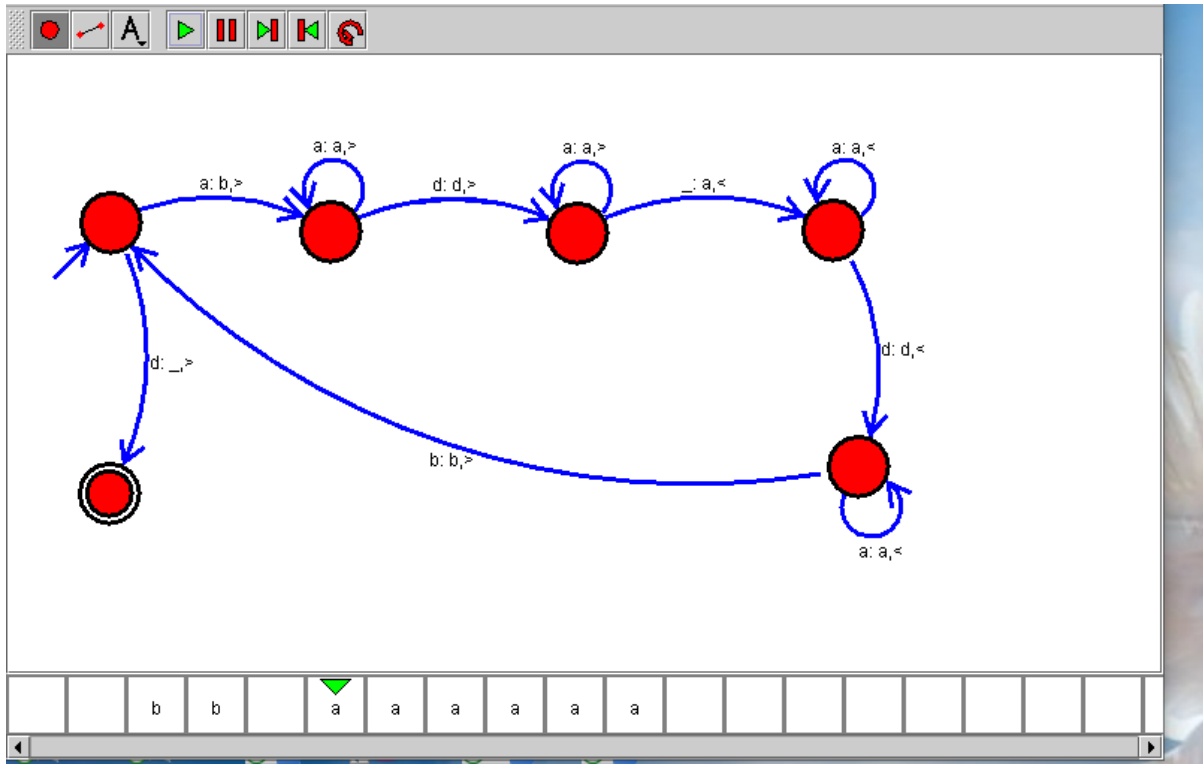
## EXP :16

**AIM:**Design TM using simulator to perform addition of ‘aa’ and ‘aaa’

$$W = aa + aaaa$$

After Addition of a's = aaaaaa

Simulation:



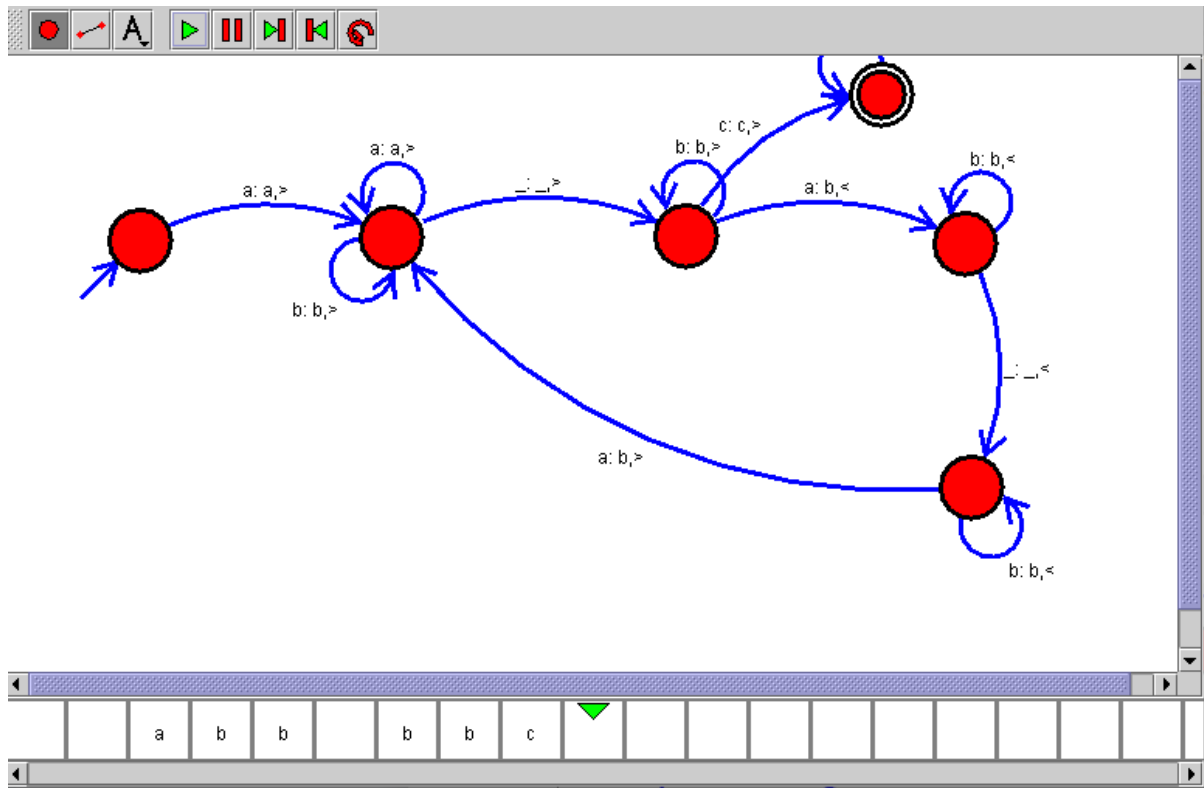
## EXP :17

**AIM:**Design TM using simulator to perform subtraction of aaa-aa

### Logic

W= aaa-aa

The Result of Subtraction is = a

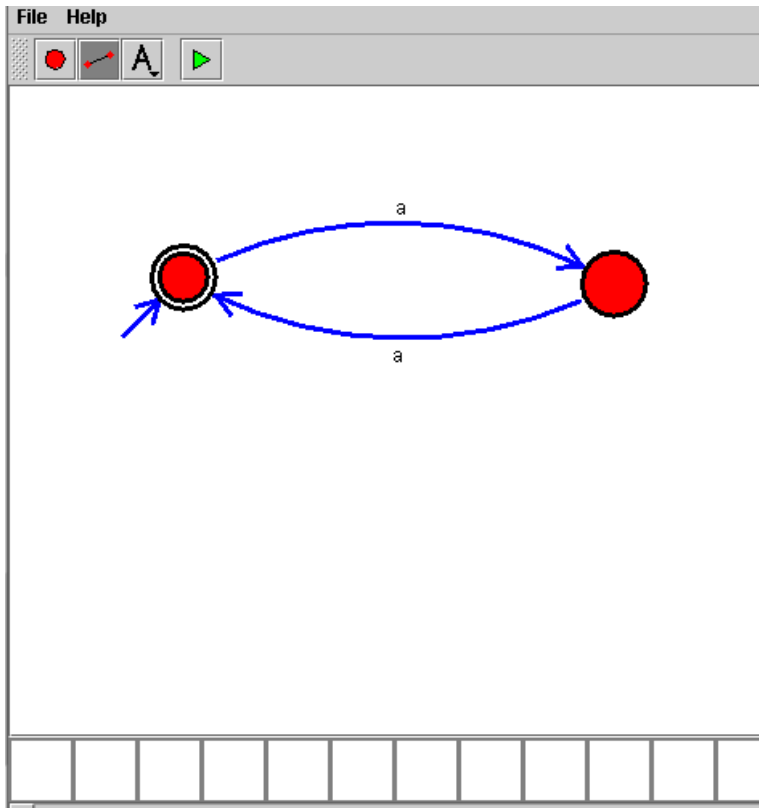


## EXP :18

**AIM:**Design DFA using simulator to accept even number of a's

$W \{ aa, aaaa, aaaaaa \}$

Simulation

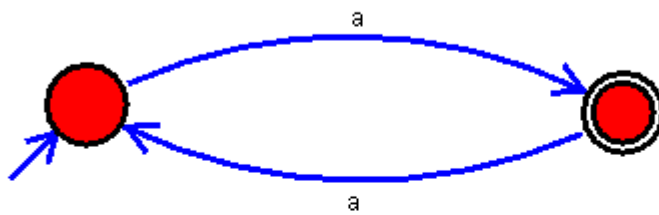


## EXP :19

**AIM:**Design DFA using simulator to accept odd number of a's

$W\{a, aaa, aaaaa\}$

Simulation

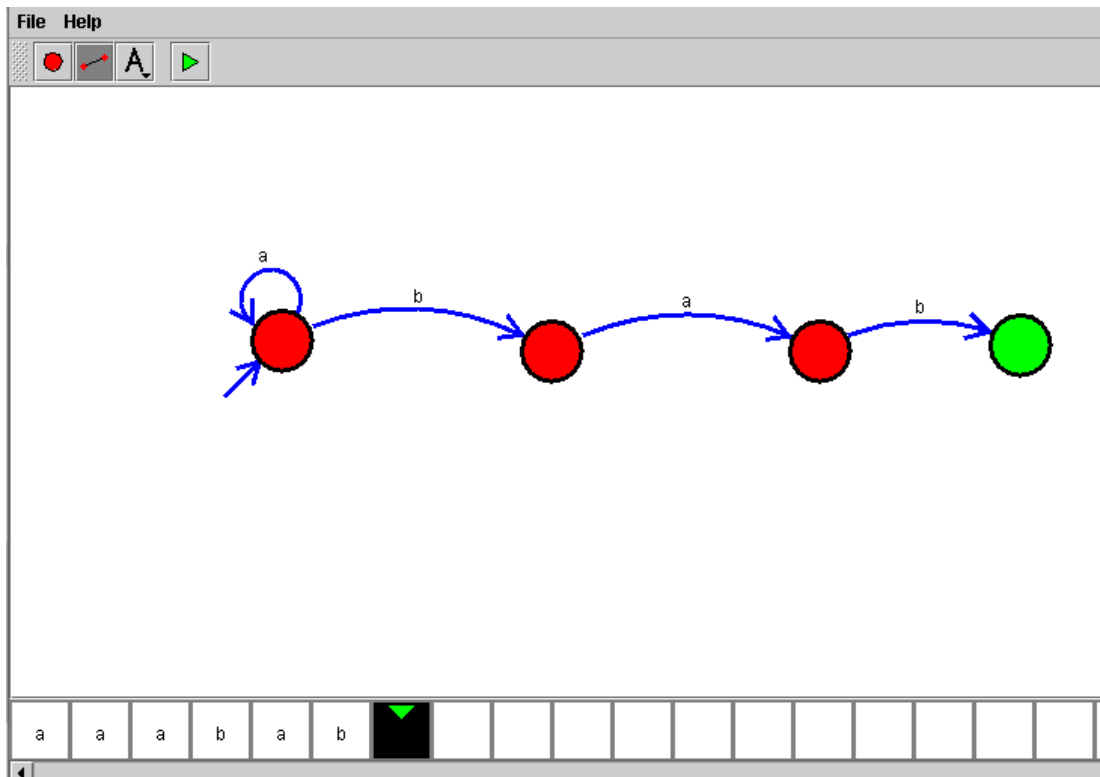


## EXP :20

**AIM:** Design DFA using simulator to accept the string the end with ab over set {a,b}

W=

aaabab

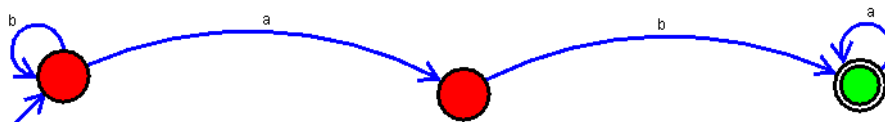




## EXP :21

**AIM:** Design DFA using simulator to accept the string having 'ab' as substring over the set {a,b}

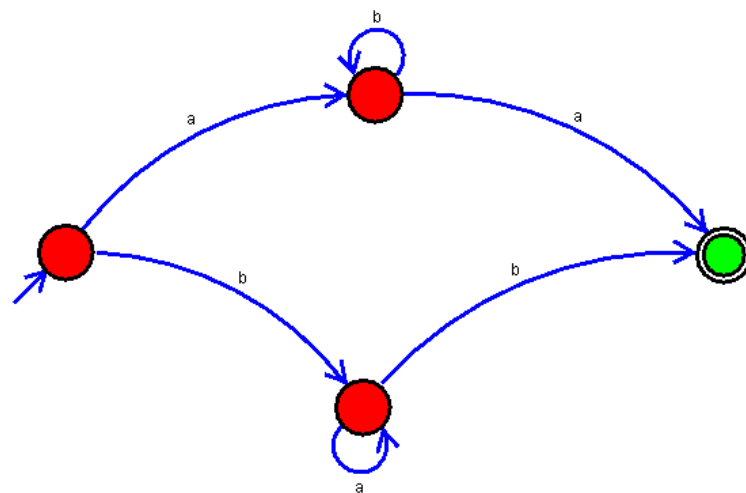
W= babaaaaa



## EXP :22

**AIM:**Design DFA using simulator to accept the string start with a or b over the set {a,b}

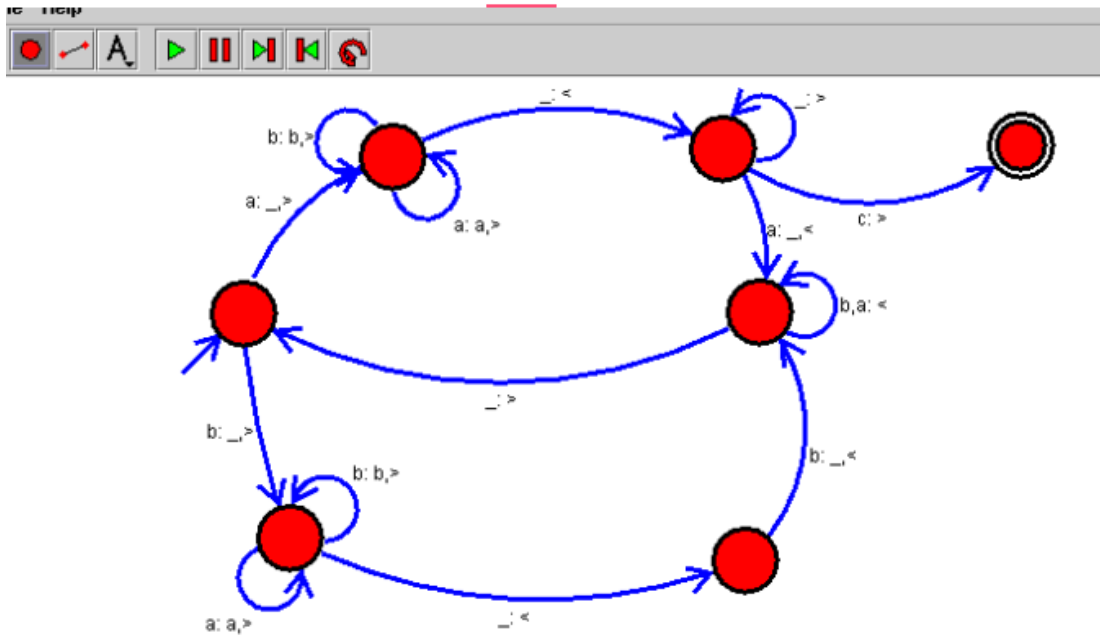
$W = \{ \text{abbbbba}, \text{baaaaab} \}$



## EXP :23

**AIM:**Design TM using simulator to accept the input string Palindrome bbabb

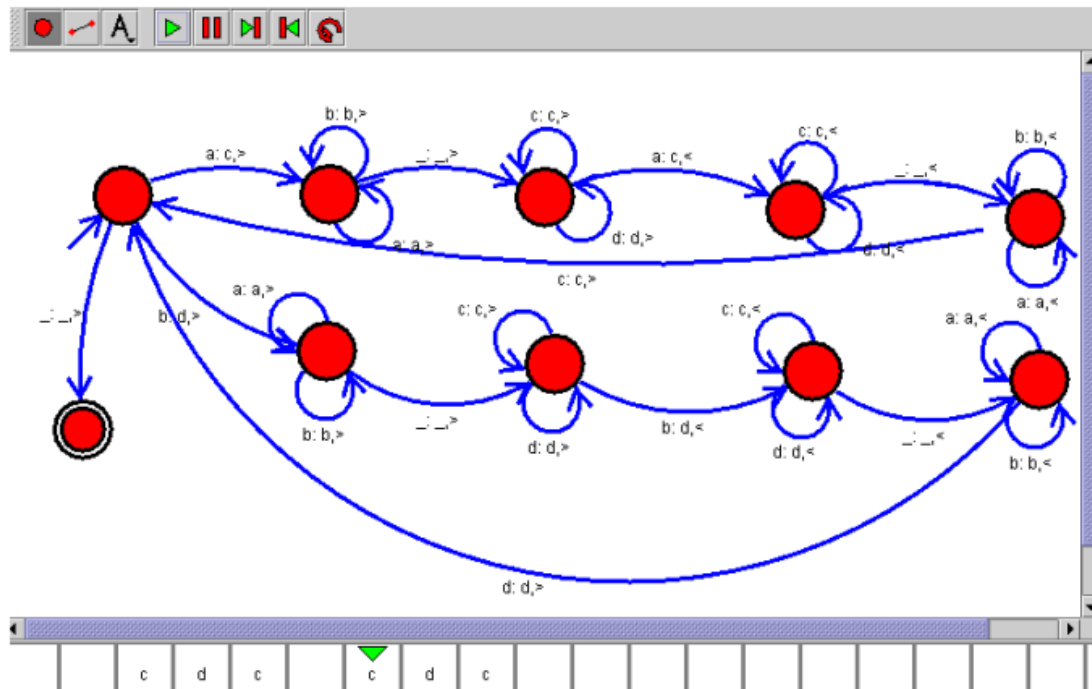
$W=\{bbabb\}$



## EXP :24

**AIM:**Design TM using simulator to accept the input string wcw

$W = \{ aa\ aa, bb\ bb, ab\ ab \}$

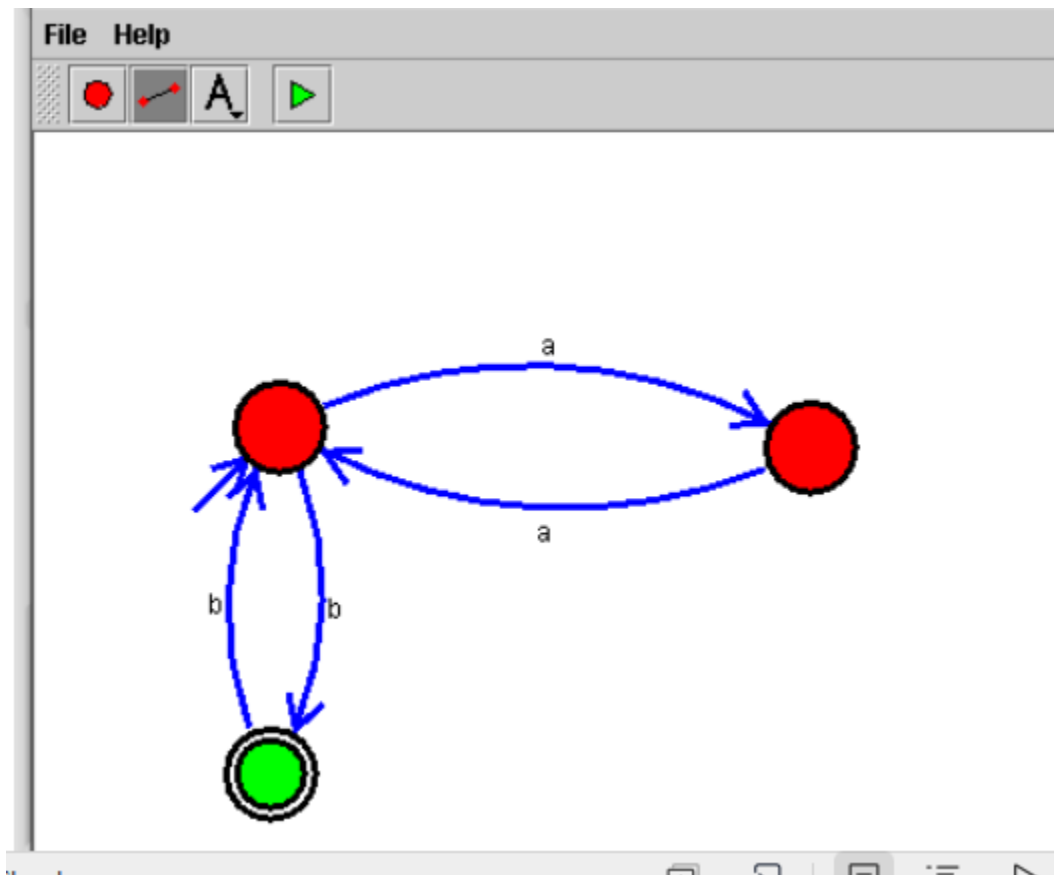


## EXP :25

**AIM:**Design DFA using simulator to accept the string even number of a's and odd number of b's

**W={aab, bbaab}**

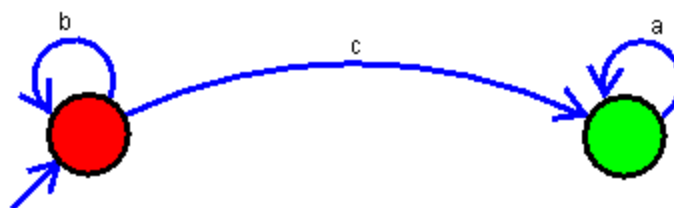
**Simulation**



## EXP :26

**AIM:**Design DFA using simulator to accept the input string "bc", "c", and "bcaaa"

Simulation

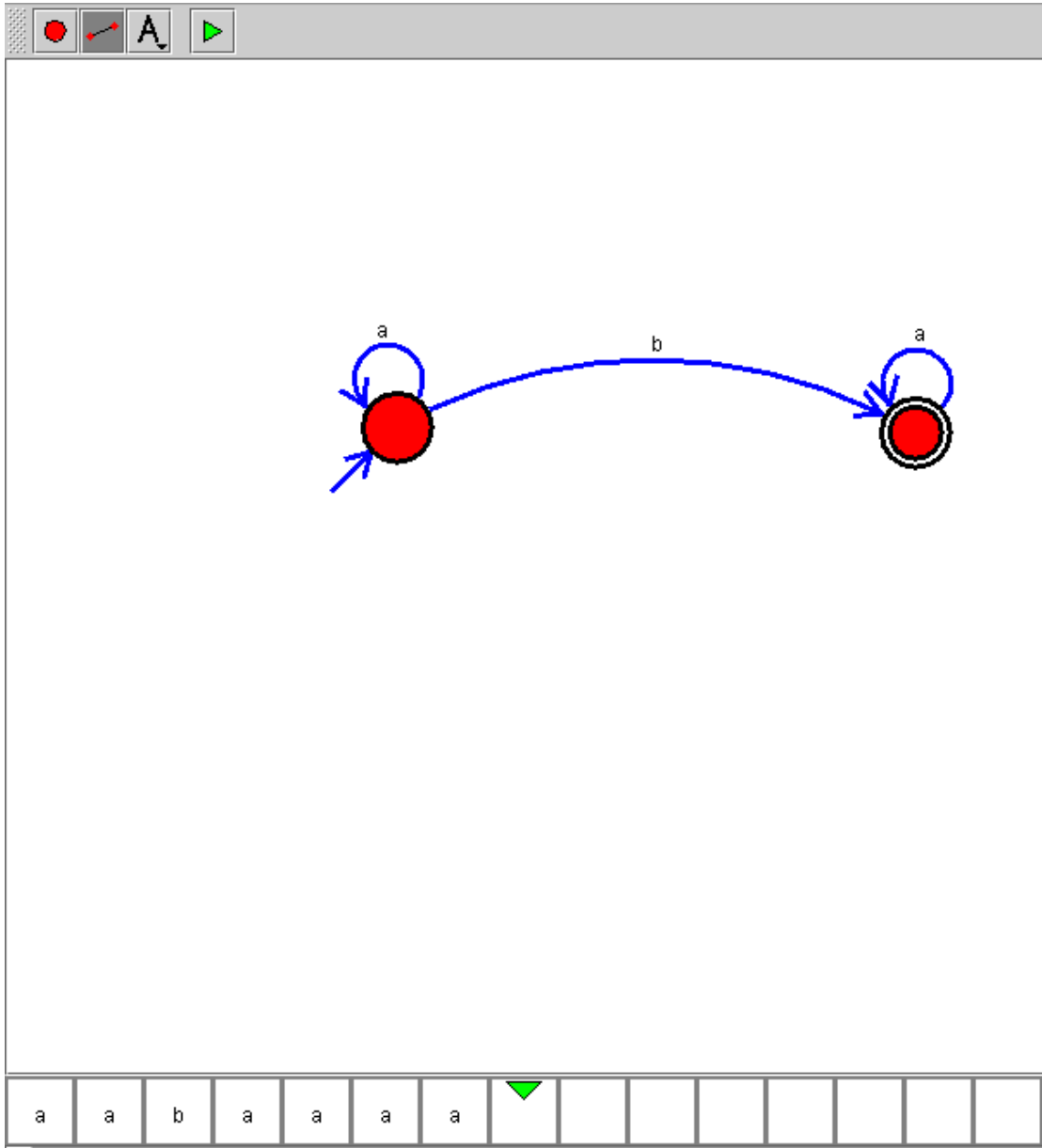


b	b	b	c	a	a								
---	---	---	---	---	---	--	--	--	--	--	--	--	--

## EXP :27

**AIM:**Design NFA to accept any number of a's where input={a,b}

W={ aaaab, baaaaaa }



## EXP :28

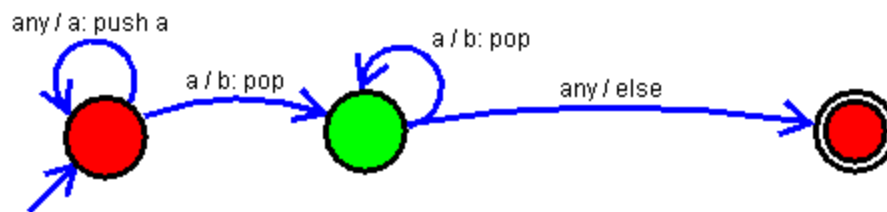
**AIM:**Design PDA using simulator to accept the input string  $a^n b^n$

$W = \{ aabb, aaabbb \}$



Automaton Simulator

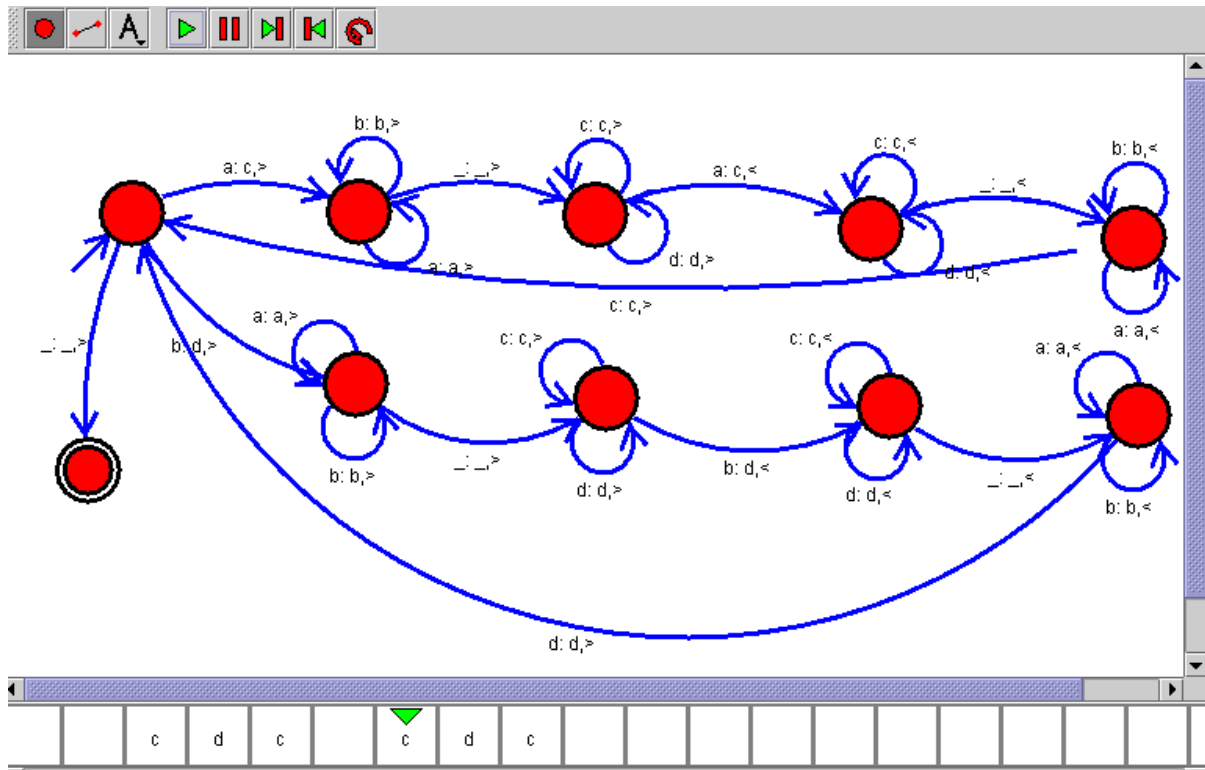
File Help





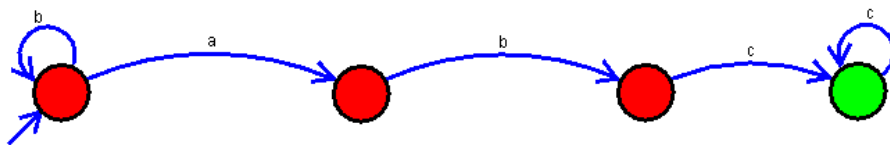
## EXP :29

**AIM:**Design TM using simulator to perform string comparison where  $w=\{aba\ aba\}$



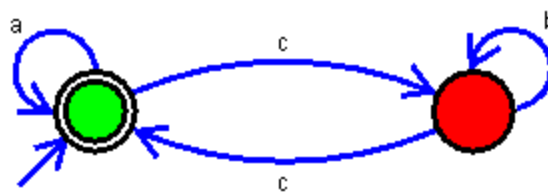
**EXP :30**

**AIM:** Design DFA using simulator to accept the string having 'abc' as substring over the set {a,b,c}

$$W = \{ \text{aaaabcccc}, \text{abcccccc} \}$$


## EXP :31

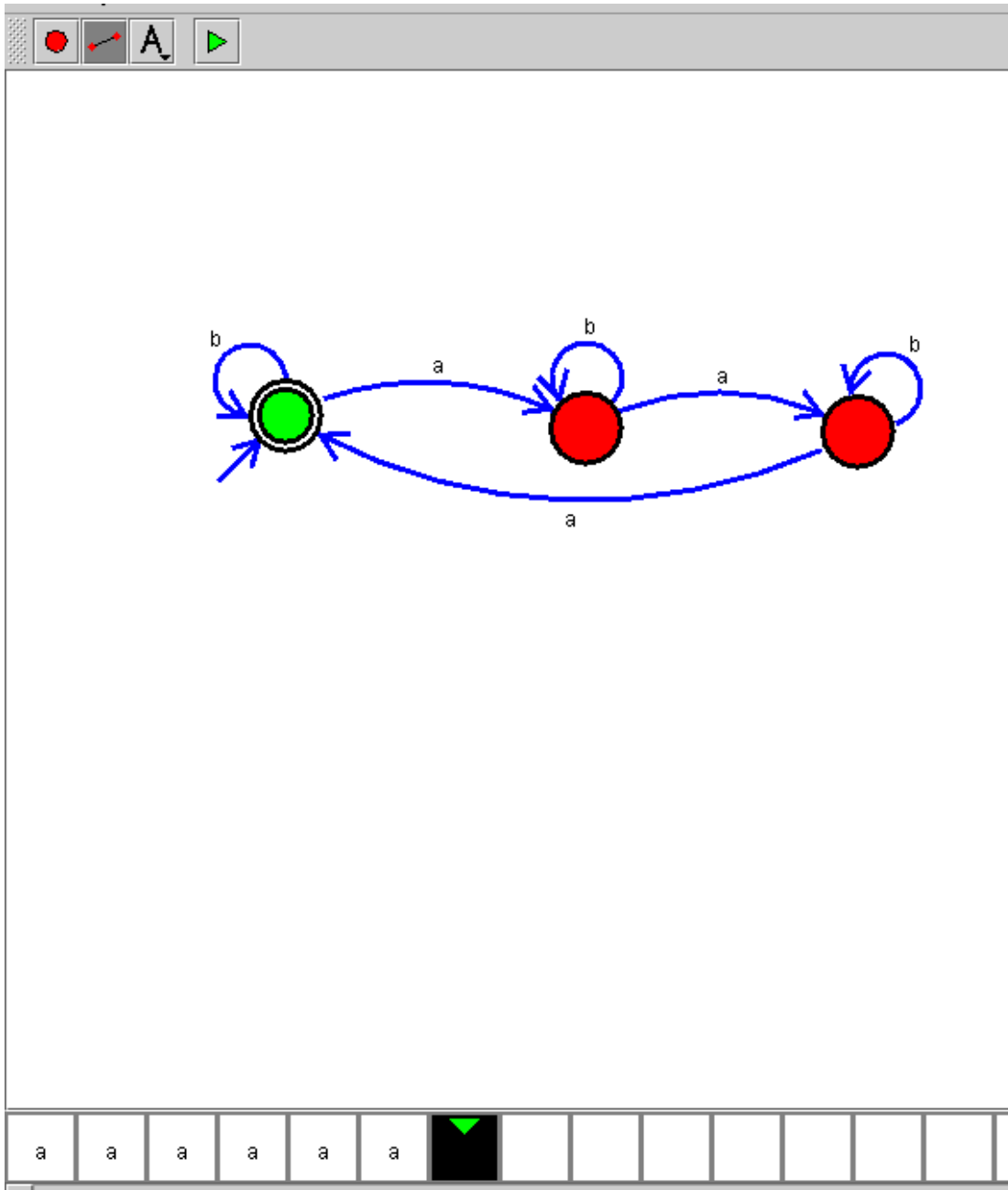
**AIM:**Design DFA using simulator to accept even number of c's over the set {a,b,c}



### EXP :32

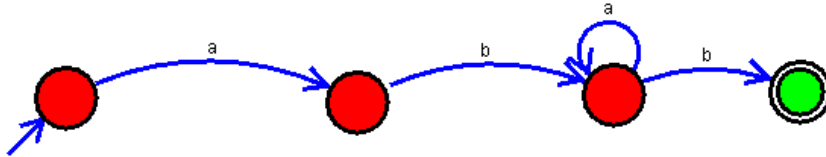
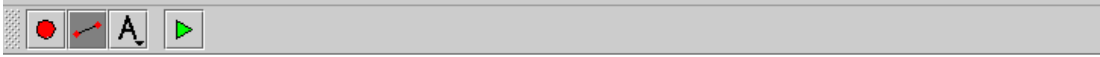
**AIM:** Design DFA using simulator to accept strings in which a's always appear tripled over input {a,b}

$W = \{aaa, ababa\}$



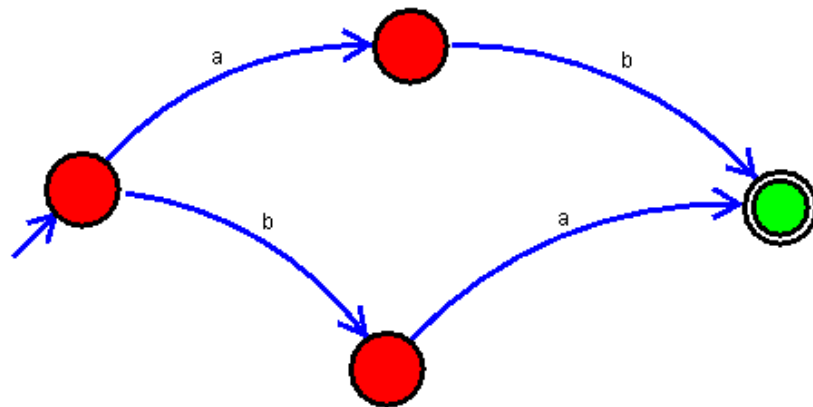
## EXP :33

**AIM:** Design NFA using simulator to accept the string the start with a and end with b over set {a,b} and check W= abaab



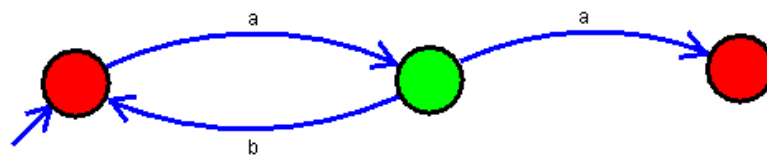
## EXP :34

**AIM:**Design NFA using simulator to accept the string that start and end with different symbols over the input {a,b}



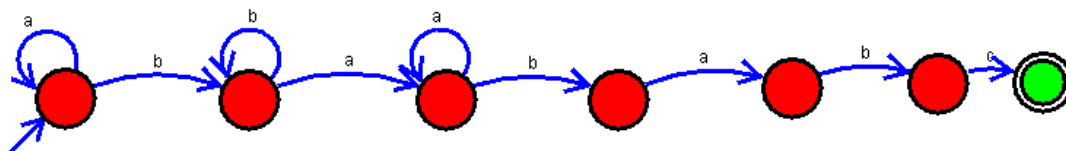
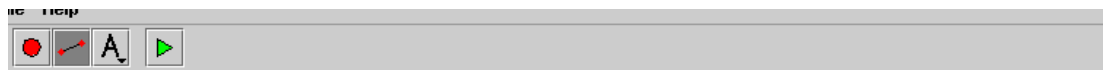
# EXP :35

**AIM:** Let L be regular language, L consist set of string over { a,b) number a's minus number b's less than or equal to 2. Design DFA to accept the the language L.



## EXP :36

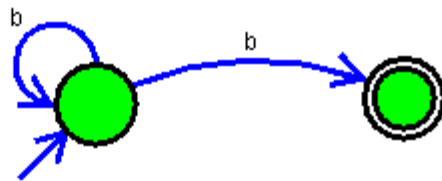
**AIM:** Design DFA using simulator to accept the string the end with abc over set {a,b,c} W= abbaababc





### EXP :37

**AIM:**Design NFA to accept any number of b's where input={a,b}



b	b	b	b							
---	---	---	---	--	--	--	--	--	--	--



**EXP NO : 38**

**CASE STUDY – REAL-TIME APPLICATION OF AUTOMATA THEORY**

**DATE :**

**THE AUTOMATIC TELLER MACHINE (ATM)**

**AIM:** To study and understand the formal design, specification and modelling of the ATM system using Finite State Machine

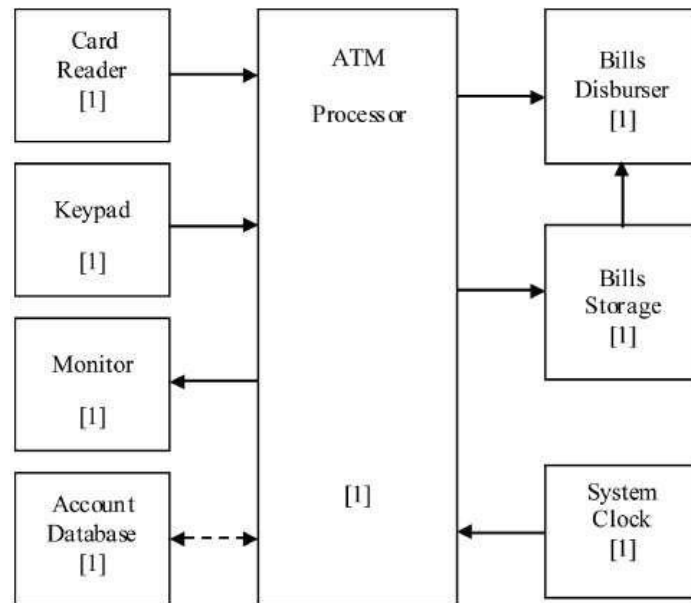
**FINITE STATE MACHINE (FSM) :**

A Finite State Machine is an abstract model of a system (physical, biological, mechanical, electronic, or software). Key components of a FSM are

- A finite number of states which represent the internal “memory” of the system by implicitly storing information about what has happened before
- Transitions which represent the “response” of the system to its environment. Transitions depend upon the current state of the machine as well as the current input and often result in a change of state.

**CONCEPTUAL MODEL OF THE ATM SYSTEM :**

An ATM system is a real-time front terminal of automatic teller services with the support of a central bank server and a centralized account database. The ATM provides money withdrawal and account balance management services. It encompasses an ATM processor, a system clock, a remote account database, and a set of peripheral devices such as the card reader, monitor, keypad, bills storage and bills disburser. The conceptual model of an ATM system is usually described by a Finite State Machine (FSM) which adopts a set of states and a set of transitions modelled by a transition diagram or a transition table to describe the basic behaviours of the ATM system,



### FORMAL DEFINITION OF THE FINITE STATE MACHINE :

Formal definition of the finite state machine can be written as

$$\text{ATM} = (Q, \Sigma, \delta, q_0, F)$$

where

- $Q \rightarrow$  set of states  $\{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$ 
  - S0 : System
  - S1 : Welcome
  - S2 : Check PIN
  - S3 : Input withdraw amount
  - S4 : Verify balance
  - S5 : Verify bills availability
  - S6 : Disburse bills
  - S7 : Eject card
- $\Sigma \rightarrow$  set of events that the ATM may accept and process
 
$$\Sigma = \{e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}$$
  - e0 : Start
  - e1 : Insert card
  - e2 : Correct PIN
  - e3 : Incorrect PIN
  - e4 : Request  $\leq$  max

- e5 : Request > max
- e6 : Cancel transaction
- e7 : Sufficient funds
- e8 : Insufficient funds
- e9 : Sufficient bills in ATM
- e10 : Insufficient bills in ATM

- $q_0 \rightarrow$  start state

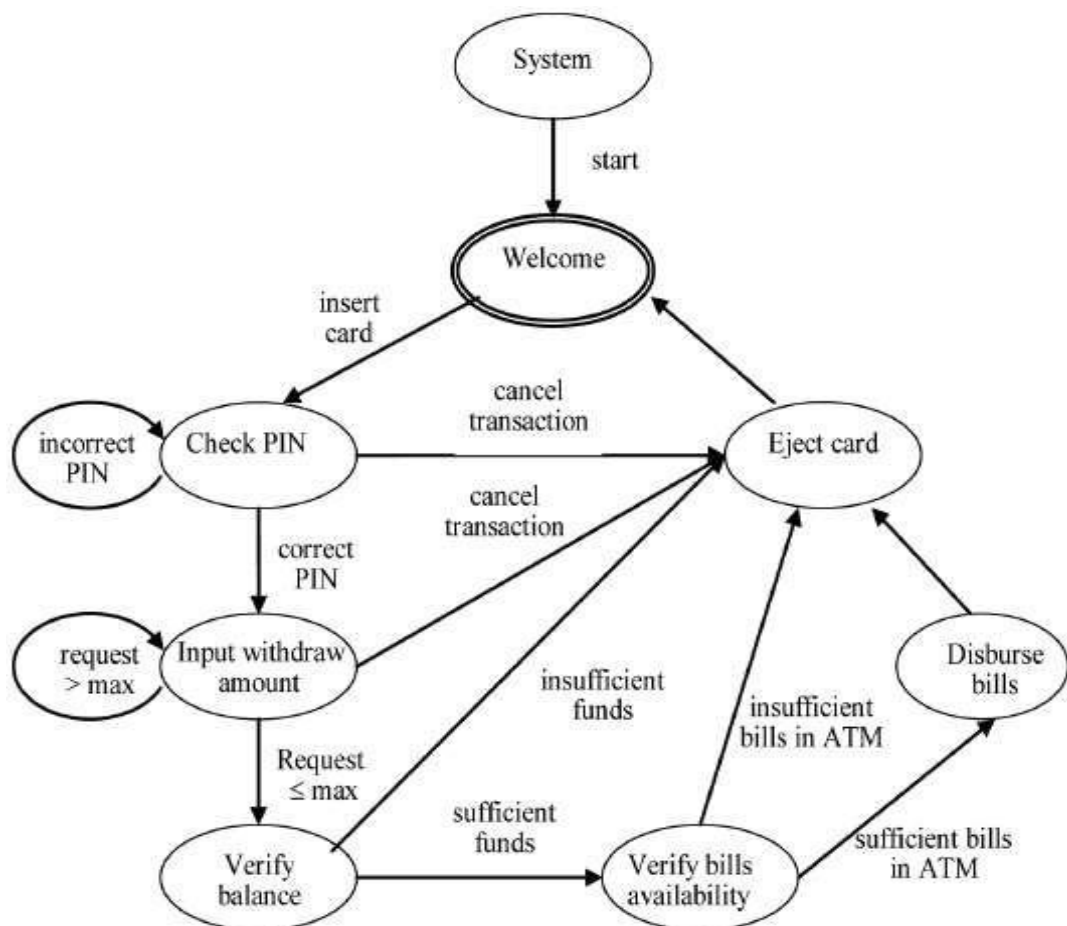
$q_0 = s_1$  (Welcome)

- $F \rightarrow$  set of accepting states

$F = \{s_1\}$

- $\delta \rightarrow$  transition function that determines the next state of the FSM on the basis of the current state and the incoming event.  $\delta$  is defined as shown in the transition diagram and the transition table given below.

### Transition Diagram :



**State Transition Table :**

<b>Present state</b>	<b>Input (Event)</b>	<b>Next State</b>
s0	e0	s1
s1	e1	s2
s2	e2	s3
s2	e3	s2
s2	e6	s7
s3	e4	s4
s3	e5	s3
s3	e6	s7
s4	e7	s5
s4	e8	s7
s5	e9	s6
s5	e10	s7
s6	-	s7
s7	-	s1

**RESULT:**

The conceptual model of ATM system, its configuration, basic behaviours and logical relationships among components of the ATM system are studied.

**EXP NO : 39**  
**DATE :**

## **PATTERN SEARCHING**

**AIM:** To study and understand the formal design, specification and modelling of Pattern Searching / Text Searching using Finite State Machine

### **FINITE STATE MACHINE (FSM) :**

A Finite State Machine is an abstract model of a system (physical, biological, mechanical, electronic, or software). Key components of a FSM are

- A finite number of states which represent the internal “memory” of the system by implicitly storing information about what has happened before
- Transitions which represent the “response” of the system to its environment. Transitions depend upon the current state of the machine as well as the current input and often result in a change of state.

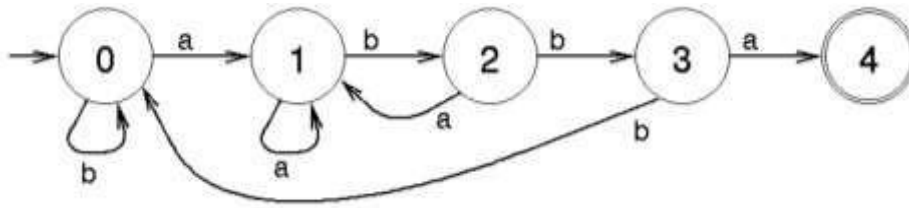
### **CONCEPTUAL MODEL OF THE PATTERN MATCHING SYSTEM :**

Pattern searching is an important problem in computer science. When we search for a string in notepad / word file or browser or database, pattern searching algorithms are used to show the results. Finite Automata (FA) can also be used for pattern searching. In FA-based pattern searching algorithm, we preprocess the pattern and build a 2D array that represents a Finite Automata. Construction of the FA is the main tricky part of this algorithm. Once this FA is built, the searching is simple. In search, we simply need to start from the first state of the automata and the first character of the text. At every step, we consider the next character of text, look for the next state in the built FA and move to a new state. If we reach the final state, then the pattern is found in the text. The time complexity of the search process is  $O(n)$ .

### **FORMAL DEFINITION OF THE FINITE AUTOMATA**

#### **A Simple Example:**

Suppose a text file consists of only a's and b's and the search is for the string “abba”. The corresponding finite automata will be as follows:



Start searching for the string from the initial state 0 and when the final state 4 is reached, the search is successful.

### A Complex Example :

For example, let us design a Finite Automata for accepting the keywords “ezhil”, “hills” and “lssbus” in a text file. The Nondeterministic Finite Automata (NFA) can be built quickly for the given problem. Based on the number of keywords and its length, the size of the NFA may vary.

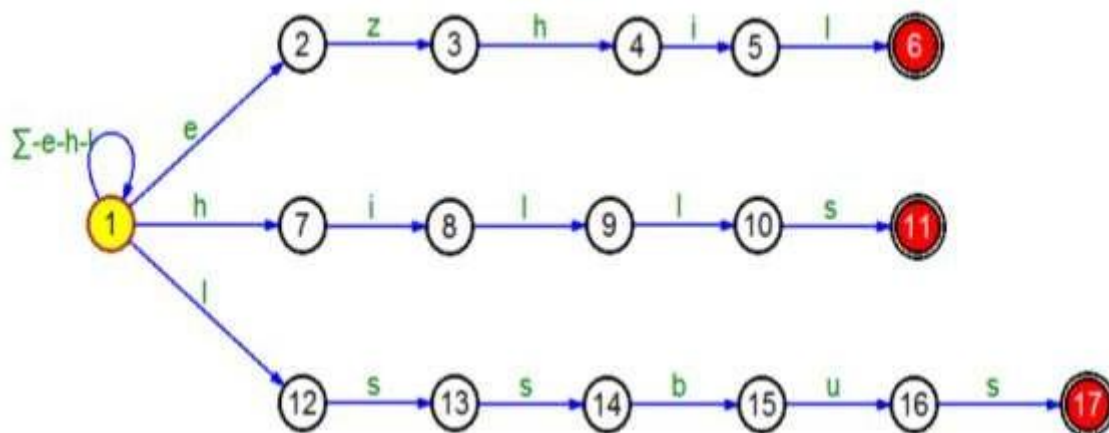
Formal definition of the NFA can be written as

$$N = (Q, \Sigma, \delta, q_0, F)$$

where

- $Q \rightarrow$  set of states  $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17\}$
- $\Sigma \rightarrow$  input alphabet  $\{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$
- $q_0 \rightarrow$  initial state. 1
- $F \rightarrow$  set of final states  $\{6, 11, 17\}$
- $\delta \rightarrow$  transition function which is shown below as transition diagram and table

### Transition Diagram :





### State Transition Table :

Input → States	e	z	h	i	l	s	b	u	Σ-e-z-h-i- l-s-b-u
→1	1,2	1	1,7	1	1,12	1	1	1	1
2		3							
3			4						
4				5					
5					6				
*6									
7				8					
8					9				
9					10				
10						11			
*11									
12						13			
13						14			
14							15		
15								16	
16						17			
*17									

### Sample Input :

If the given input is

that

*dr ezhilarasu umadevi went to nilgiri hills*

*by lssbus. Ezhil visited many places in hills.*

*On the way, he saw one flex which has the*

*word ezhillssbus*

Starting from the initial state, taking the input character by character, if one of the final states is reached, the input word will be accepted.

### SAMPLE C PROGRAM :

```
// C program for Finite Automata Pattern searching Algorithm
```

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

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

```
#define
```

```
NO_OF_CHARS 256
```

```
int getNextState(char *pat, int M, int state, int x)
```

```
{
```

```
    // If the character c is same as next character
```

```
    // in pattern, then simply increment state
```

```
    if (state < M && x ==
```

```
        pat[state]) return
```

**state+1;**

```

// ns stores the result which is next state
int ns, i;

// ns finally contains the longest prefix
// which is also suffix in "pat[0..state-1]c"

// Start from the largest possible value
// and stop when you find a prefix which is also suffix
for (ns = state; ns > 0; ns--)
{
    if (pat[ns-1] == x)
    {
        for (i = 0; i < ns-1; i++)
            if (pat[i] !=
                pat[state-ns+1+i])
                break;
        if (i == ns-1)
            return ns;
    }
}

return 0;
}

/* This function builds the TF table which represents Finite
Automata for a given pattern */

void computeTF(char *pat, int M, int TF[][NO_OF_CHARS])
{
    int state, x;
    for (state = 0; state <= M; ++state)
        for (x = 0; x < NO_OF_CHARS; ++x)
            TF[state][x] = getNextState(pat, M, state, x);
}

/* Prints all occurrences of pat in txt */

void search(char *pat, char *txt)
{
    int M =
        strlen(pat); int N
        = strlen(txt);

    int TF[M+1][NO_OF_CHARS];

```

```
computeTF(pat, M, TF);
```

```
// Process txt over FA.
```

```
int i, state=0;
```

```
for (i = 0; i < N; i++)
```

```
{
```

```
    state
```

```
    =
```

```
    TF[s                                i-M+1);
```

```
    tate]
```

```
    [txt[
```

```
    i]]; if
```

```
    (stat
```

```
    e ==
```

```
    M)
```

```
        printf ("\n  
        Pattern found at  
        index %d",
```

```
}
```

```
}
```

```
// Driver program to test above function
```

```
int main()
```

```
{
```

```
    char *txt = "AABAACAADAABAAABAA";
```

```
    char *pat =
```

```
    "AABA";
```

```
    search(pat, txt);
```

```
    return 0;
```

```
}
```

## **RESULT:**

The conceptual model of the pattern searching system, the method for constructing NFA for the required keywords and searching for the required keywords in a text file are studied.

**EXP NO :40**

**DATE :**

## **VENDING MACHINE**

**AIM:** To study and understand the formal design, specification and modelling of Vending Machine using Finite State Machine

### **FINITE STATE MACHINE (FSM) :**

A Finite State Machine is an abstract model of a system (physical, biological, mechanical, electronic, or software). Key components of a FSM are

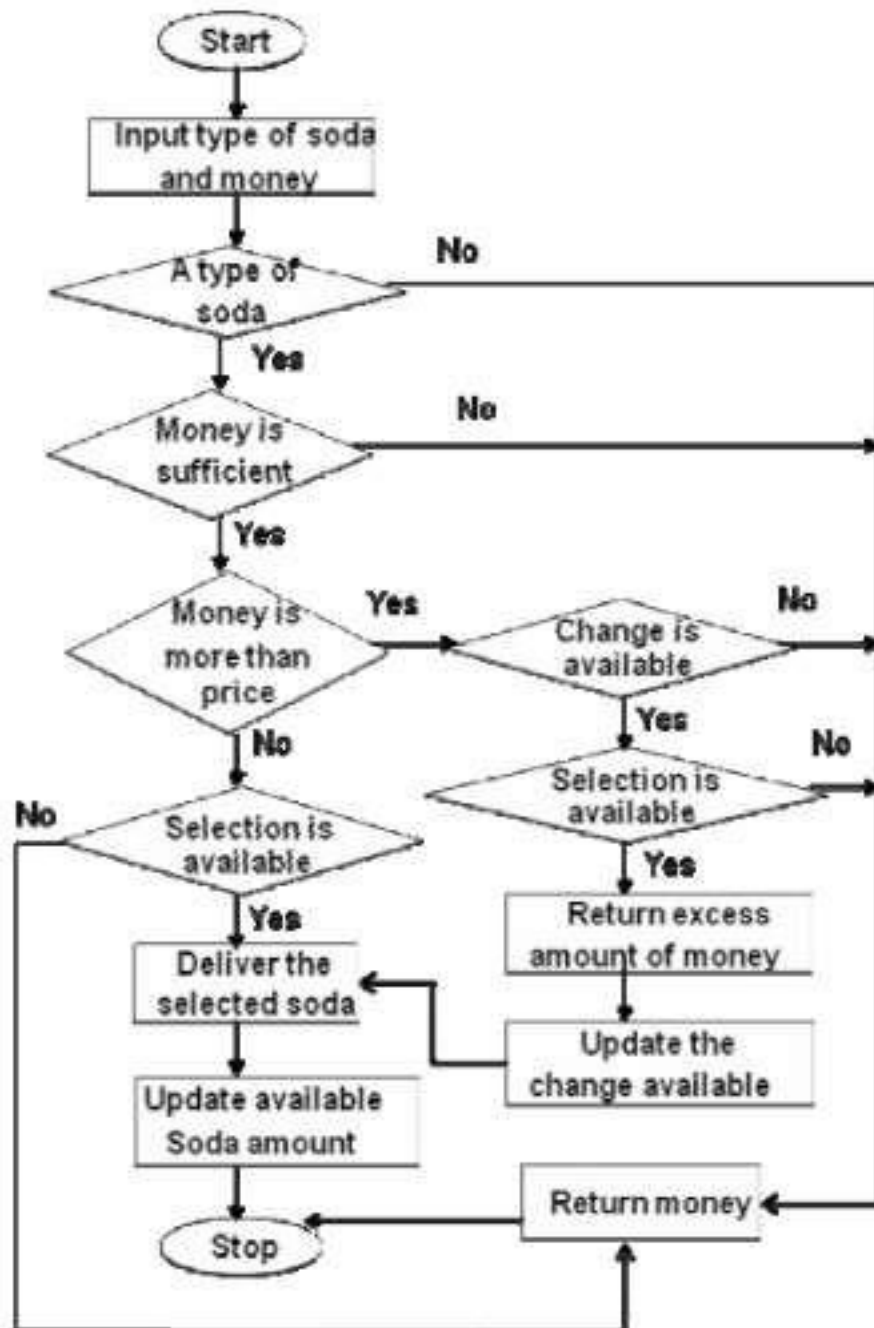
- A finite number of states which represent the internal “memory” of the system by implicitly storing information about what has happened before
- Transitions which represent the “response” of the system to its environment. Transitions depend upon the current state of the machine as well as the current input and often result in a change of state.

### **CONCEPTUAL MODEL OF THE VENDING MACHINE :**

Vending machines (VM) are electronic devices used to provide different products such as snacks, coffee, tickets, etc. Vending machines provide several different types of items when money is inserted into it. The Vending machines are more practical, easy to use and accessible for user than the standard purchasing method. The efficient implementation of these machines can be in different ways by using microcontroller and FPGA board. They are designed to be able to accept money and serve product according to the amount of money inserted. The basic operation of VM is given below.

- The user inserts money and the money counter sends to the control unit, the amount of money inserted in the VM by the user.
- The operation buttons are active to choose the products that people like. According to the VM's internal program, VM dispenses the products when people insert the correct amount.
- If the program is designed to return the change, VM will return the change.
- When selected product is not available, VM will reject the service.

# FLOW CHART FOR THE OPERATION OF SODA VENDING MACHINE :



## FORMAL DEFINITION OF THE FINITE AUTOMATA:

A DFA that describes the behaviour of a vending machine which accepts dollars and quarters and charges \$1.25 per soda. Once the machine receives at least \$1.25, corresponding to the final states, it will allow the user to select a soda. Self loops represent ignored input. The machine will not dispense a soda until at least \$1.25 has been deposited and it will not accept more money once it has already received greater than or equal to \$1.25.

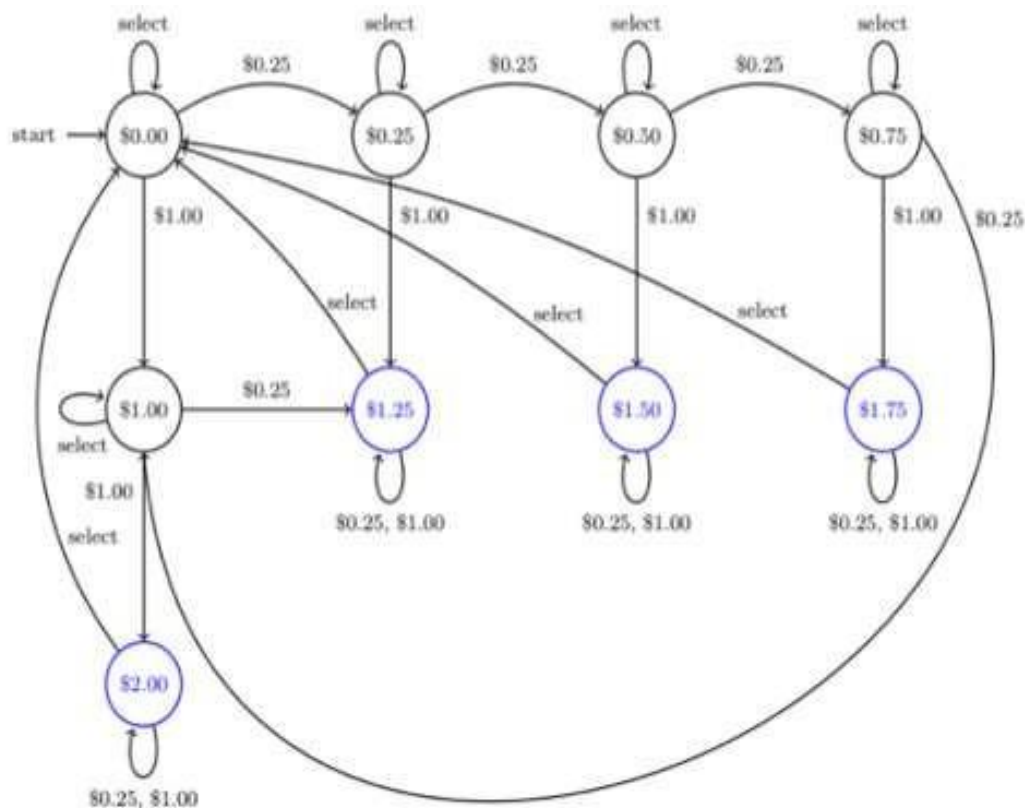
Formal definition of the DFA can be written as

$$D = (Q, \Sigma, \delta, q_0, F)$$

where

- $Q \rightarrow$  set of states  $\{\$0.00, \$0.25, \$0.50, \$0.75, \$1.00, \$1.25, \$1.50, \$1.75, \$2.00\}$
- $\Sigma \rightarrow$  input alphabet  $\{\$0.25, \$1.00, \text{select}\}$
- $q_0 \rightarrow$  initial state.  $\$0.00$
- $F \rightarrow$  set of final states
- $\delta \rightarrow$  transition function which is shown below as transition diagram

### Transition Diagram :



### RESULT:

The conceptual model of the vending machine is studied using FSM automata theory. Constructing a FSM which uses fewer states enables the machine to provide fast response serving.



**EXP NO : 41**

**DATE :**

## **NATURAL LANGUAGE PROCESSING**

**AIM:** To study and understand the formal design, specification and modelling of Natural Language Processing using Automata Theory

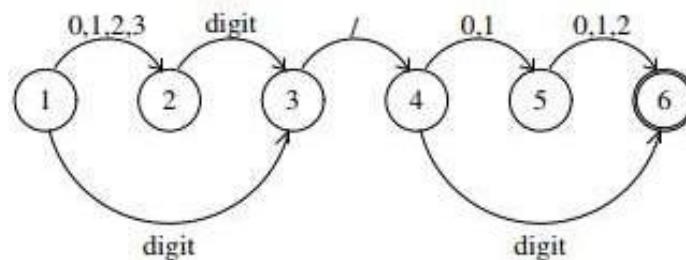
### **NATURAL LANGUAGE PROCESSING (NLP) :**

Natural language processing is a field of linguistics and computer science which focuses on processing natural language. Natural languages are human spoken languages like English, Telugu and Tamil, in opposition to artificial languages such as computer languages C or Java. The main goal of NLP is to make human languages automatically processable. It implies finding techniques to convert an utterance which can be either spoken or written into formal data. Formal data are a representation of that utterance that can be processed using a computer and with no or minimal supervision. Some part of natural language processing relies on automata theory.

### **FINITE AUTOMATA FOR SIMPLE STRUCTURES :**

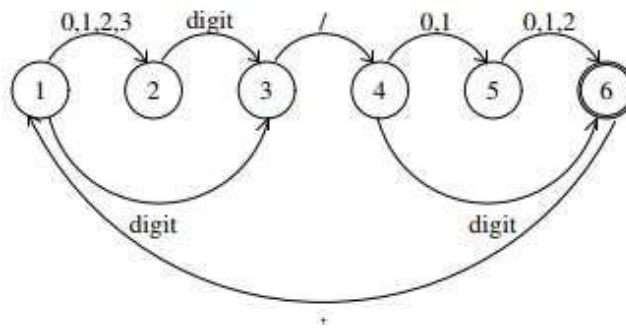
#### **Automata for Recognizing Dates :**

Suppose we want to recognize dates (just day and month pairs) written in the format day/month. The day and the month may be expressed as one or two digits (e.g. 11/2, 1/12 etc.). This format corresponds to the following simple FSA, where each character corresponds to one transition:



This is a NFA. For example, an input starting with the digit 3 will move the FSA to both state 2 and state 3.

Suppose we want to recognize a comma-separated list of such dates, the FSA can be designed as shown below. It has a cycle and can accept a sequence of indefinite length.

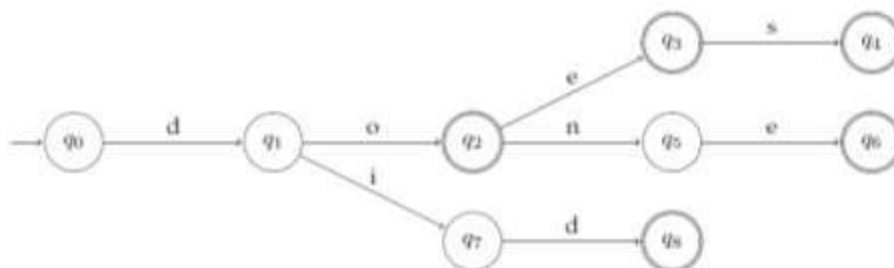


### Lexicon :

In linguistics, a lexicon is the vocabulary of a person, group or language. It contains all the minimal grammatical elements of a language. Sample lexicon for the word do and some of its derivatives is shown below:

do  
did  
does  
done

Efficient representation of lexicons are possible using finite state automata as shown below.



Constructing a Deterministic Finite Automata can improve the access and minimizing the automata can reduce the number of states considerably.

### Can Finite Automata model the syntax of natural languages?

The syntax of natural languages cannot be described by Finite Automata. Strings having infinite recursion cannot be generated by a FSM. However, FSMs are very useful for partial grammars which don't require full recursion. For representing complex structures, context free grammars are useful.

## CONTEXT FREE GRAMMARS :

In formal language theory, context-free grammar (CFG) is a certain type of formal grammar. A set of production rules that describe all possible strings in a given formal language. Languages generated by context-free grammars are known as context-free languages.

CFGs arise in linguistics where they are used to describe the structure of sentences and words in a natural language, and they were in fact invented by the linguist Noam Chomsky for this purpose.

In computer science, as the use of recursively-defined concepts increased, they are used more and more. In an early application, grammars are used to describe the structure of programming languages. In a newer application, they are used in an essential part of the Extensible Markup Language (XML) called the *Document Type Definition*.

A CFG has four components:

1. A set of non-terminals
2. A set of terminals
3. A set of rules (productions)
4. A start symbol

A Simple CFG for a fragment of English

S → NP VP

VP → VP PP

VP → V

VP → V NP

VP → V VP

NP → NP PP

PP → P NP

;;; lexicon

V → can

V → fish

NP → fish

NP → rivers

NP → pools

NP → December

NP → Scotland

NP → it

NP → they

P → in

The rules with terminal symbols on the RHS correspond to the lexicon. Here are some strings which the grammar generates, along with their bracketings:

**they fish**

(S (NP they) (VP (V fish)))

**they can fish**

(S (NP they) (VP (V can) (VP (V fish))))

**they fish in rivers**

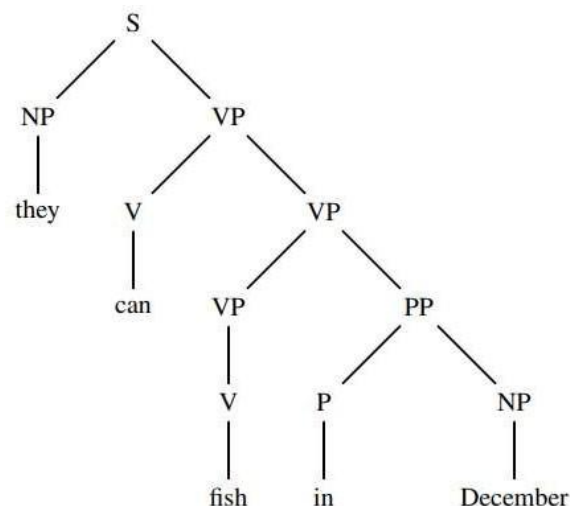
(S (NP they) (VP (VP (V fish)) (PP (P in) (NP rivers))))

**they fish in rivers in December**

(S (NP they) (VP (VP (V fish)) (PP (P in) (NP (NP rivers) (PP (P in) (NP December))))))

**Parse Tree**

A CFG only defines a language. It does not say how to determine whether a given string belongs to the language it defines. To do this, a parser can be used whose task is to map a string of words to its parse tree. A parse tree or derivation tree is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar.



(S (NP they) (VP (V can) (VP (VP (V fish)) (PP (P in) (NP December))))))

**RESULT:**

The application of finite automata and Context Free Grammars in the field of natural language processing is studied.