

# Automata Based Programming Paradigm

## Summary

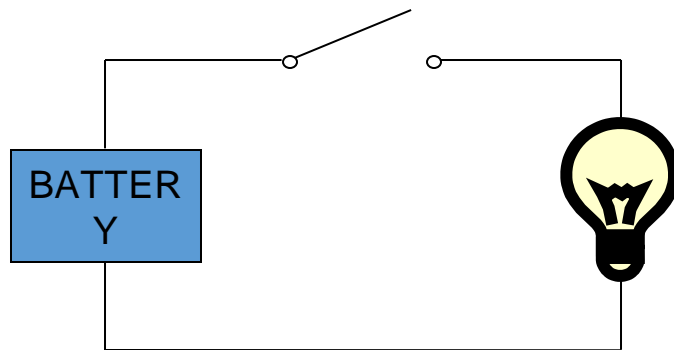
# Introduction

Automata-based programming is a programming paradigm in which the program or its part is thought of as a model of a finite state machine or any other formal automation.

## What is Automata Theory?

- Automata theory is the study of abstract computational devices
- Abstract devices are (simplified) models of real computations
- Computations happen everywhere: On your laptop, on your cell phone, in nature, ...

## Example:



input: switch

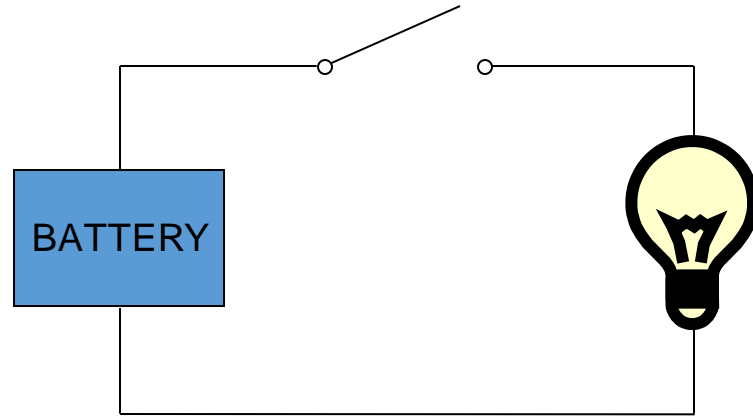
output: light bulb

actions: flip switch

states: on, off

# Simple Computer

Example:

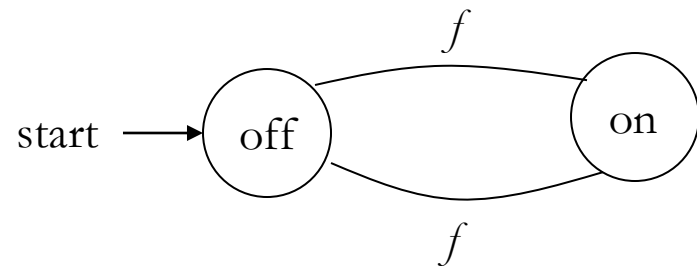


input: switch

output: light bulb

actions: flip switch

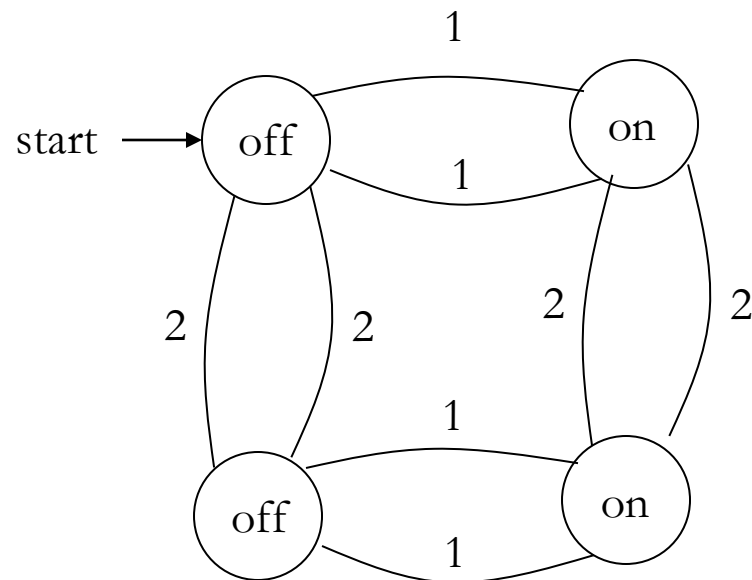
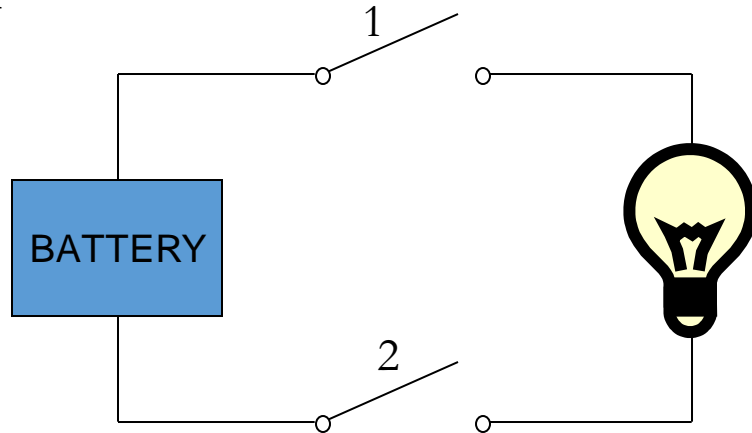
states: on, off



bulb is on if and only if there was an odd number of flips

# Another “computer”

Example:



inputs: switches 1 and 2

actions: 1 for “flip switch 1”

actions: 2 for “flip switch 2”

states: on, off

bulb is on if and only if both switches were flipped an odd number of times

# Types of Automata

finite automata	Devices with a finite amount of memory. Used to model “small” computers.
push-down automata	Devices with infinite memory that can be accessed in a restricted way. Used to model parsers, etc.
Turing Machines	Devices with infinite memory. Used to model any computer.

# Alphabets

A common way to talk about words, number, pairs of words, etc. is by representing them as strings

To define strings, we start with an alphabet

An **alphabet** is a finite set of symbols.

## Examples:

$\Sigma_1 = \{a, b, c, d, \dots, z\}$ : the set of letters in English

$\Sigma_2 = \{0, 1, \dots, 9\}$ : the set of (base 10) digits

$\Sigma_3 = \{a, b, \dots, z, \#\}$ : the set of letters plus the special symbol #

$\Sigma_4 = \{ (, ) \}$ : the set of open and closed brackets

# Strings

A **string** over alphabet  $\Sigma$  is a finite sequence of symbols in  $\Sigma$ .

The empty string will be denoted by  $\varepsilon$

## Examples:

abfbz is a string over  $S_1 = \{a, b, c, d, \dots, z\}$

9021 is a string over  $S_2 = \{0, 1, \dots, 9\}$

ab#bc is a string over  $S_3 = \{a, b, \dots, z, \#\}$

))()( is a string over  $S_4 = \{ (, ) \}$

# Languages

A **language** is a set of strings over an alphabet.

Languages can be used to describe problems with “yes/no” answers, for example:

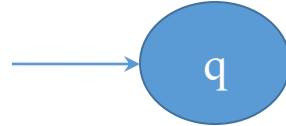
- $L_1 =$  The set of all strings over  $S_1$  that contain the substring “SRM”
- $L_2 =$  The set of all strings over  $S_2$  that are divisible by 7 = {7, 14, 21, ...}
- $L_3 =$  The set of all strings of the form  $s\#s$  where  $s$  is any string over {a, b, ..., z}
- $L_4 =$  The set of all strings over  $S_4$  where every ( can be matched with a subsequent )



# State transitions

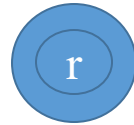
## Initial State

If any state  $q$  in  $Q$  is the initial state then it is represented by the circle with an arrow



## Final State

An arrow pointing to a filled circle nested inside another circle represents the object's final state.

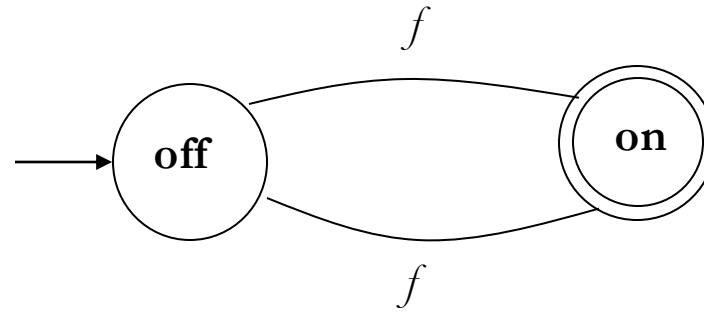


## Transition

A solid arrow represents the path between different states of an object. Label the transition with the event that triggered it and the action that results from it. A state can have a transition that points back to itself.



# Finite Automata



There are states off and on, the automaton starts in off and tries to reach the “good state” on

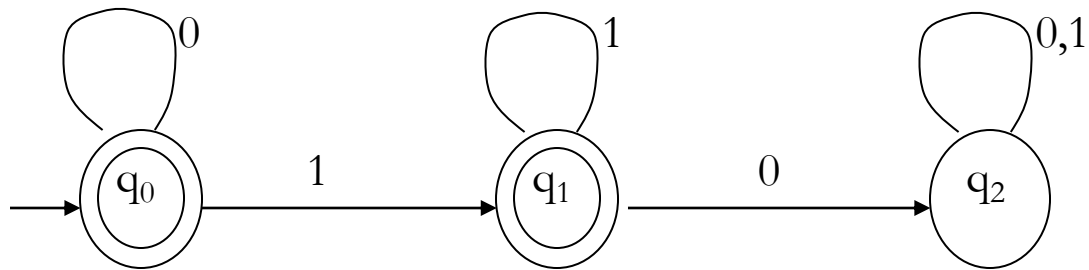
What sequences of fs lead to the good state?

Answer:  $\{f, fff, fffff, \dots\} = \{f^n : n \text{ is odd}\}$

This is an example of a deterministic finite automaton over alphabet  $\{f\}$

# Deterministic finite automata

- A deterministic finite automaton (DFA) is a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$  where
  - $Q$  is a finite set of states
  - $\Sigma$  is an alphabet
  - $\delta: Q \times \Sigma \rightarrow Q$  is a transition function
  - $q_0 \in Q$  is the initial state
  - $F \subseteq Q$  is a set of accepting states (or final states).
- In diagrams, the accepting states will be denoted by double loops



alphabet  $\Sigma = \{0, 1\}$

start state  $Q = \{q_0, q_1, q_2\}$

initial state  $q_0$

accepting states  $F = \{q_0, q_1\}$

transition function  $\delta$ :

		inputs	
		0	1
states	$q_0$	$q_0$	$q_1$
	$q_1$	$q_2$	$q_1$
	$q_2$	$q_2$	$q_2$

transition Function  $\delta$ :

$$\delta(q_0, 0) = q_0$$

$$\delta(q_0, 1) = q_1$$

$$\delta(q_1, 0) = q_2$$

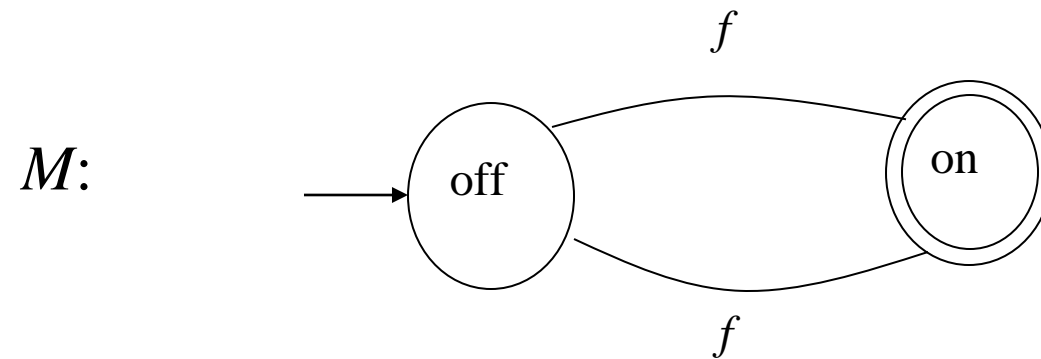
$$\delta(q_1, 1) = q_1$$

$$\delta(q_2, 0) = q_2$$

$$\delta(q_2, 1) = q_2$$

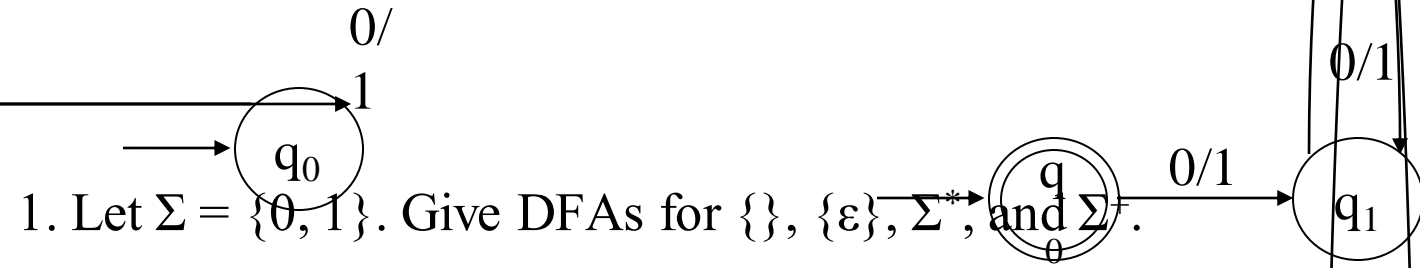
# Language of a DFA

The **language of a DFA**  $(Q, S, d, q_0, F)$  is the set of all strings over  $S$  that, starting from  $q_0$  and following the transitions as the string is read left to right, will reach some accepting state.

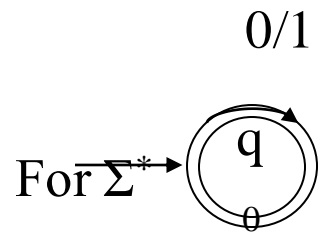


- Language of  $M$  is  $\{f, fff, fffff, \dots\} = \{f^n: n \text{ is odd}\}$

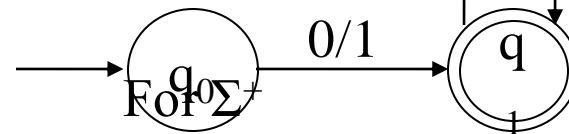
# Example of DFA



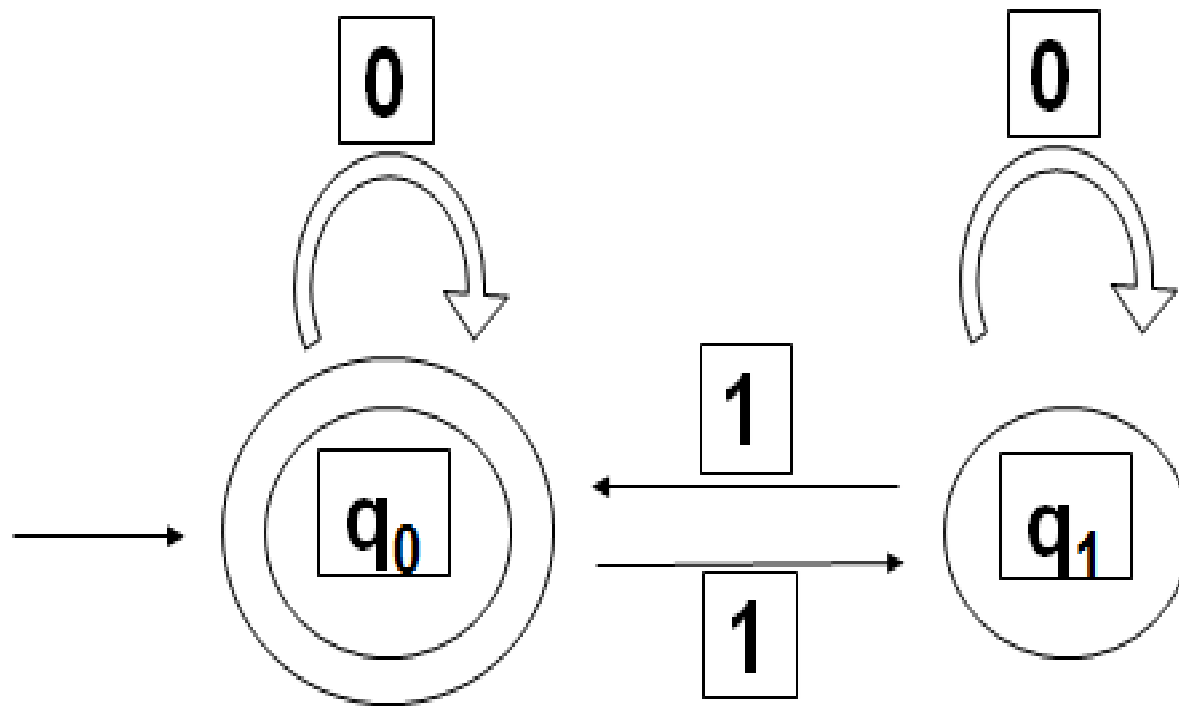
For  $\{\}$ :



For  $\{\epsilon\}$ :



# Example of DFA

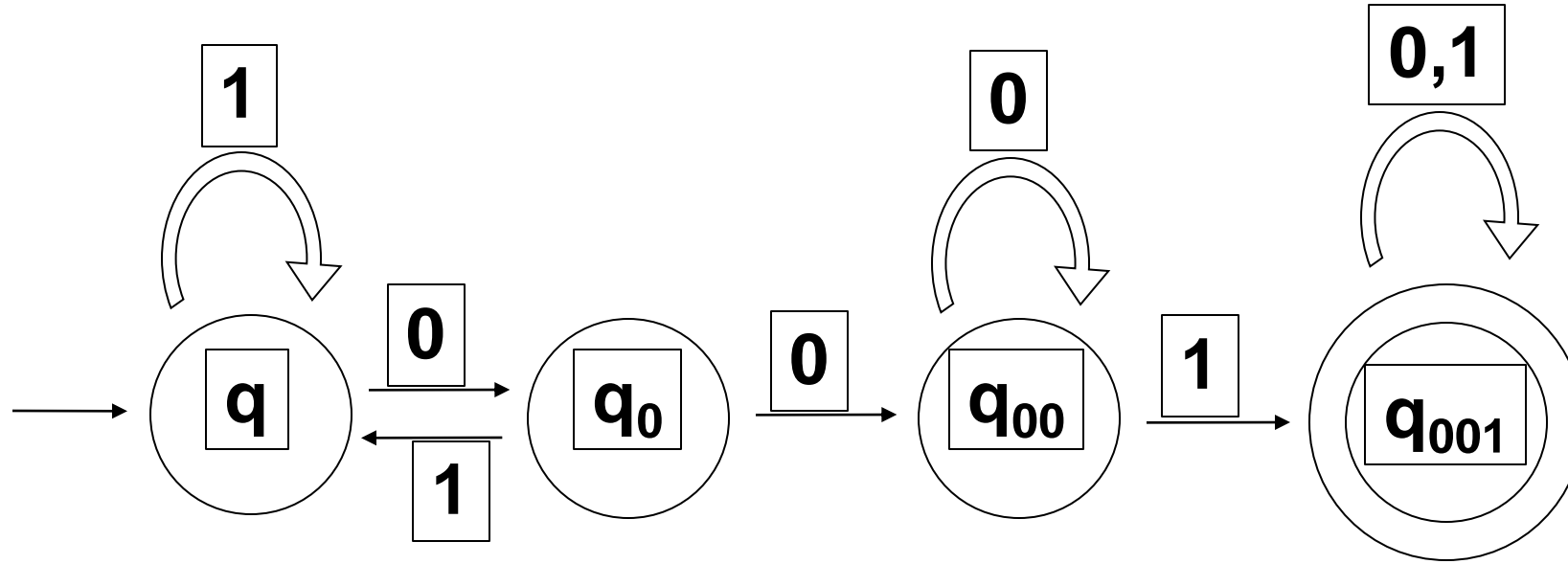


$L(M) =$

$\{ w \mid w \text{ has an even number of 1s} \}$

# Example of DFA

Build an automaton that accepts all and only those strings that contain 001





# Example of DFA using Python

```
from automata.fa.dfa import DFA
# DFA which matches all binary strings ending in an odd number of '1's
dfa = DFA(
    states={'q0', 'q1', 'q2'},
    input_symbols={'0', '1'},
    transitions={
        'q0': {'0': 'q0', '1': 'q1'},
        'q1': {'0': 'q0', '1': 'q2'},
        'q2': {'0': 'q2', '1': 'q1'}
    },
    initial_state='q0',
    final_states={'q1'}
)
dfa.read_input('01') # answer is 'q1'
dfa.read_input('011') # answer is error
print(dfa.read_input_stepwise('011'))
```

```
if dfa.accepts_input('011'):
    print('accepted')
else:
    print('rejected')
```

# NDFA

## Non-Deterministic Finite Automata

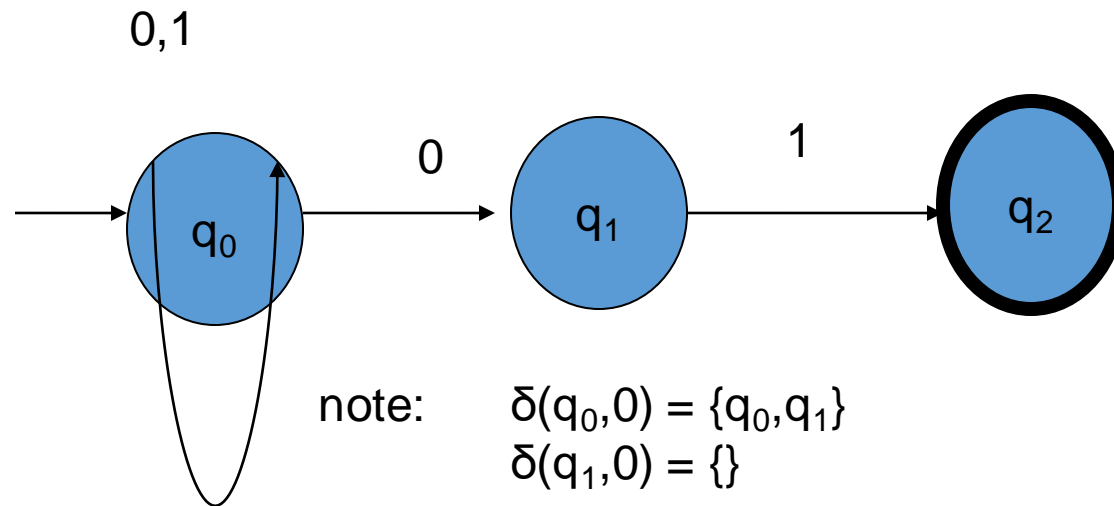
A nondeterministic finite automaton (NFA) over an alphabet  $A$  is similar to a DFA except that **epsilon-edges are allowed**, there is no requirement to emit edges from a state, and multiple edges with the same letter can be emitted from a state.

A nondeterministic finite automaton  $M$  is a five-tuple  $M = (Q, \Sigma, \delta, q_0, F)$ , where:

- $Q$  is a finite set of states of  $M$
- $\Sigma$  is the finite input alphabet of  $M$
- $\delta: Q \times \Sigma \rightarrow 2^Q$  [power set of  $Q$ ], is the state transition function mapping a state-symbol pair to a subset of  $Q$
- $q_0$  is the start state of  $M$
- $F \subseteq Q$  is the set of accepting states or final states of  $M$

# Example NDFA

- NFA that recognizes the language of strings that end in 01

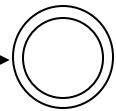


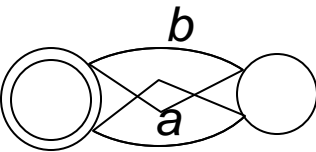
transition Table

	inputs	
	0	1
states $q_0$	$q_1, q_0$	$q_0$
$q_1$	$\emptyset$	$q_2$
$q_2$	$\emptyset$	$\emptyset$

# Examples

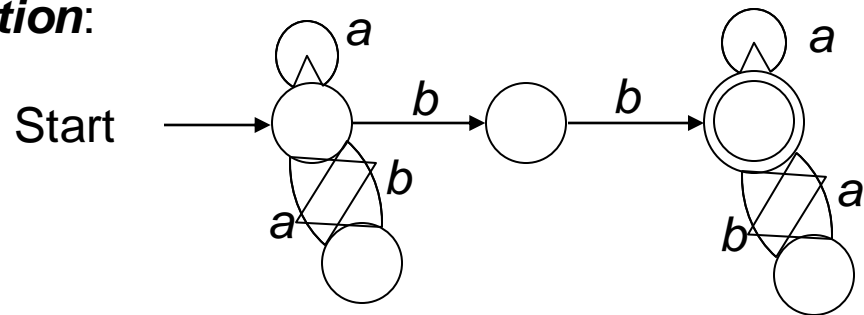
**Solutions:** (a): Start  $\longrightarrow$  

(b) Start  $\longrightarrow$    
:

(c): Start  $\longrightarrow$  

Construct an NFA to recognize the language  $(a + ba)^*bb(a + ab)^*$ .

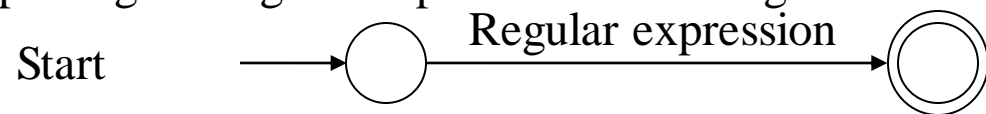
**A solution:**



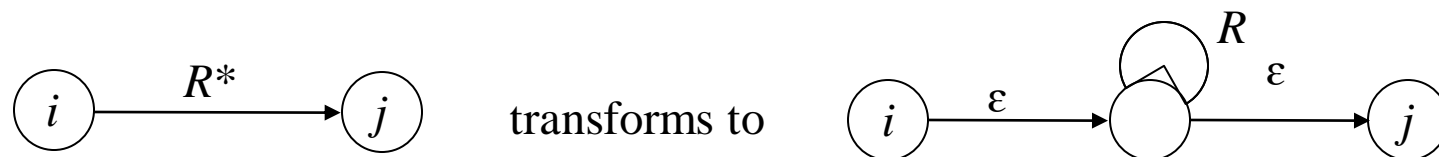
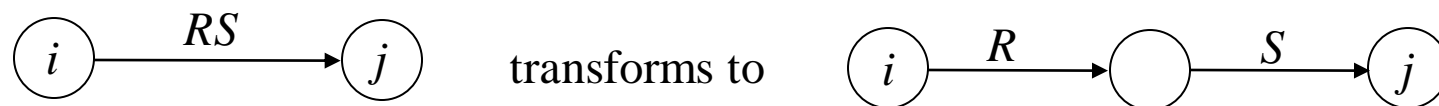
# Examples

## Algorithm: Transform a Regular Expression into a Finite Automaton

Start by placing the regular expression on the edge between a start and final state:

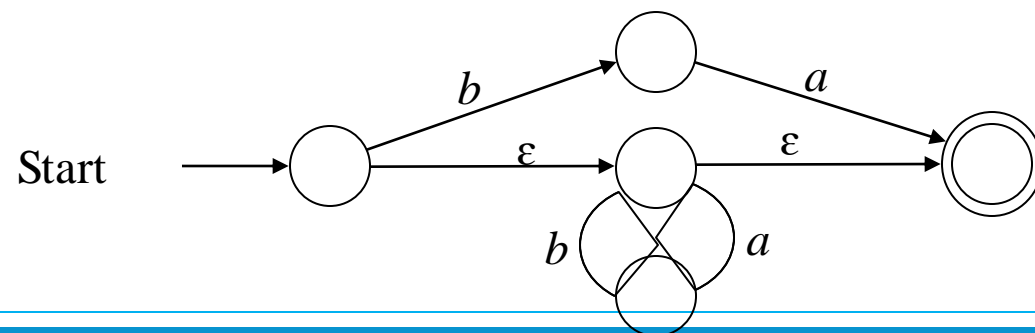


Apply the following rules to obtain a finite automaton after erasing any  $\emptyset$ -edges.



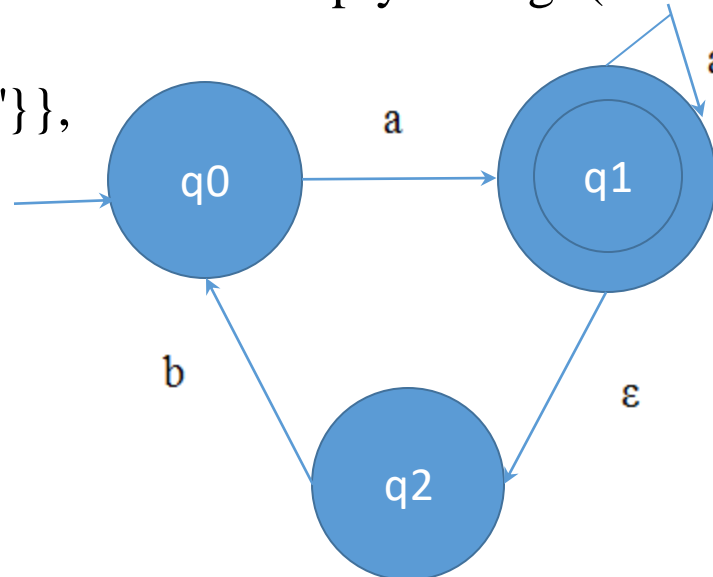
*Quiz.* Use the algorithm to construct a finite automaton for  $(ab)^* + ba$ .

**Answer:**



# Example of NFA using Python

```
from automata.fa.nfa import NFA
# NFA which matches strings beginning with 'a', ending with 'a', and
# containing
# no consecutive 'b's
nfa = NFA(
    states={'q0', 'q1', 'q2'},
    input_symbols={'a', 'b'},
    transitions={
        'q0': {'a': {'q1'}},
        # Use " as the key name for empty string (lambda/epsilon)
        'q1': {'a': {'q1'}, '"': {'q2'}},
        'q2': {'b': {'q0'}}
    },
    initial_state='q0',
    final_states={'q1'}
)
```



```
nfa.read_input('aba')
ANSWER :{'q1', 'q2'}
```

```
nfa.read_input('abba')
ANSWER: ERROR
```

```
nfa.read_input_stepwise('aba')
```

```
if nfa.accepts_input('aba'):
    print('accepted')
else:
    print('rejected')
nfa.validate()
```

# Difference between DFA and NFA

S. No.	DFA	NFA
1.	For Every symbol of the alphabet, there is only one state transition in DFA.	We do not need to specify how does the NFA react according to some symbol.
2.	DFA cannot use Empty String transition.	NFA can use Empty String transition.
3.	DFA can be understood as one machine.	NFA can be understood as multiple little machines computing at the same time.
4.	Backtracking is allowed in DFA.	Backtracking is not always allowed in NFA.

S. No.	Title	NFA	DFA
1.	Power	Same	Same
2.	Supremacy	Not all NFA are DFA.	All DFA are NFA
3.	Transition Function	Maps $Q \rightarrow (\Sigma \cup \{\lambda\} \rightarrow 2^Q)$ , the number of next states is zero or one or more.	$Q \times \Sigma \rightarrow Q$ , the number of next states is exactly one
4.	Time complexity	The time needed for executing an input string is more as compare to DFA.	The time needed for executing an input string is less as compare to NFA.
5.	Space	Less space requires.	More space requires.



*Thank You*