



**PES**

**UNIVERSITY**

CELEBRATING 50 YEARS

# Automata Formal Languages & Logic

## Introduction to the Course

---

**Prakash C O**

Department of Computer Science & Engineering

# Automata Formal Languages & Logic

## Introduction

---

### Dictionary Definition of an Automaton

***noun (plural automata)***

1. A moving mechanical device made in imitation of a human being.
2. A machine that performs a function according to a predetermined set of coded instructions.



# Automata Formal Languages & Logic

## Introduction

---

- The term “**Automata**” comes from the Greek word “**αὐτόματα**” which implies “**self-acting**”. An automaton (Automata in plural) is an abstract self-propelled machine which follows a predetermined sequence of operations automatically.
- **What is Automata Theory?**

**Study of abstract computing devices, or “machines”**

**Automaton = an abstract computing device**

Note: A “device” need not even be a physical hardware!

- **A fundamental question in computer science:**  
**Find out what different models of machines can do and cannot do**

# Automata Formal Languages & Logic

## Introduction

---

### What is Finite State Machine (FSM)?

- An automaton with a finite number of states is named a **Finite Automaton (FA)** or **Finite State Machine (FSM)**.
- **Finite-state machine (FSM)** or **finite-state automaton (FSA)** is **an (abstract) mathematical model of computation**. **FSM can be in exactly one of a finite number of states at any given time.**



# Automata Formal Languages & Logic

## Introduction

---

### What is Finite State Machine (FSM)? Cont..

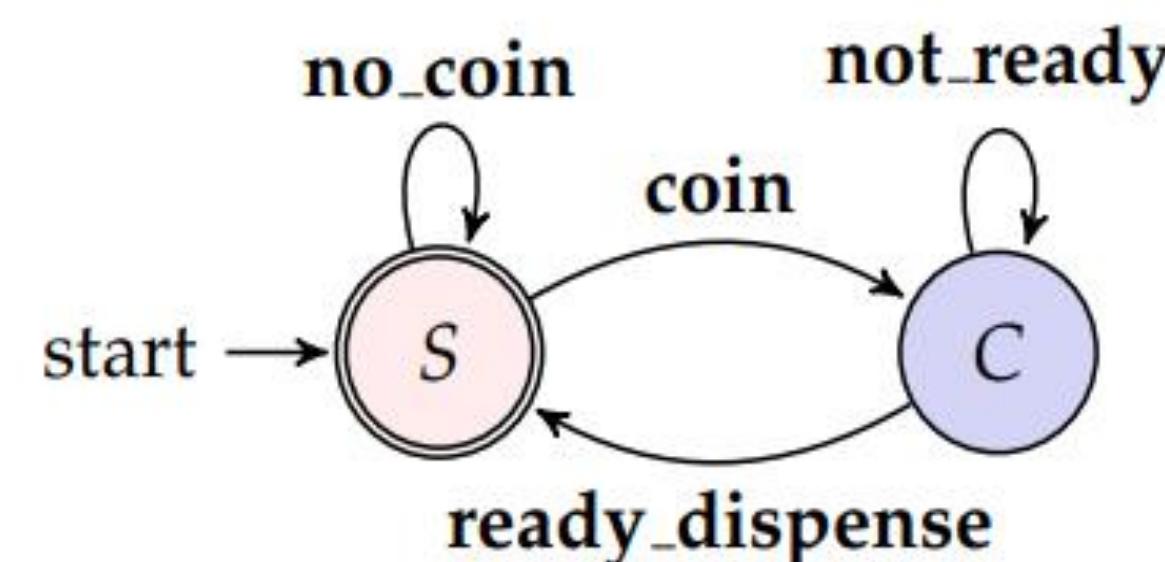
- The FSM can change from one state to another in response to sequence of inputs; the change from one state to another is called a **transition**.
- An FSM is defined by a list of its states, its initial state, its final state(s) and the inputs that trigger each transition.



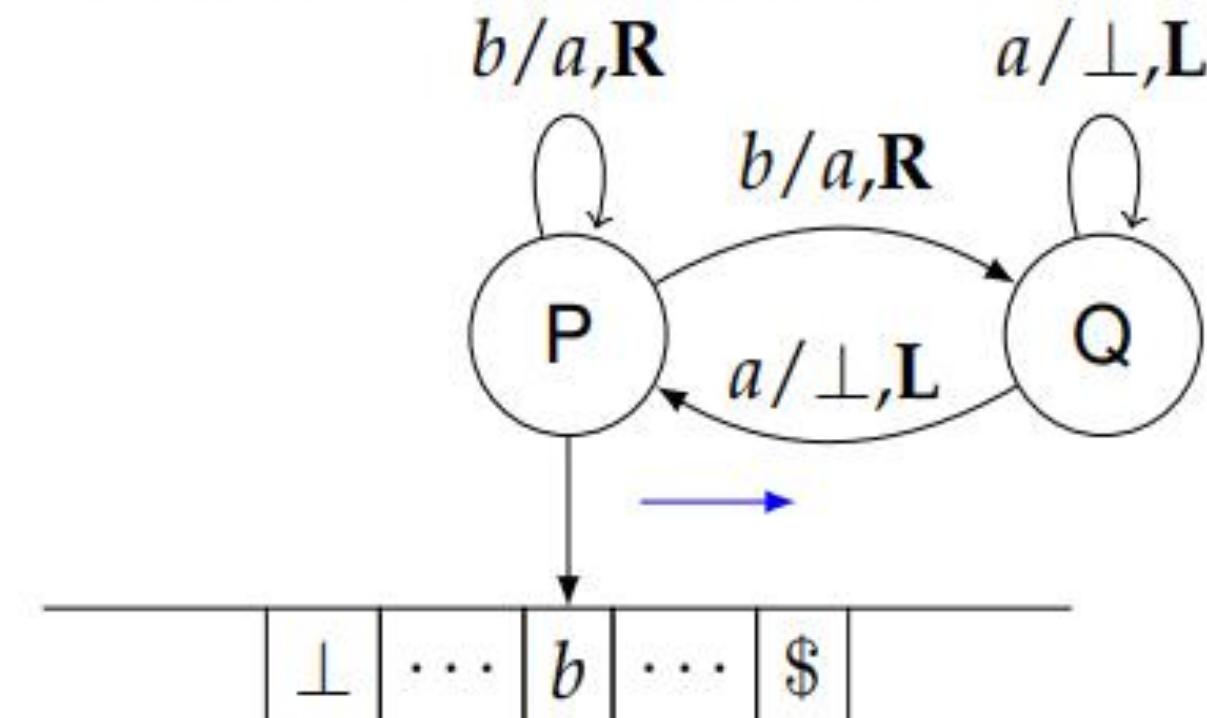
# Automata Formal Languages & Logic

## Introduction

**Finite instruction machine with finite memory (Finite State Automata)**



**Finite instruction machine with unbounded memory (Turing machine)**



The Turing Machine is a theoretical model of a simple computing device that can perform calculations and manipulate symbols on an infinite tape.

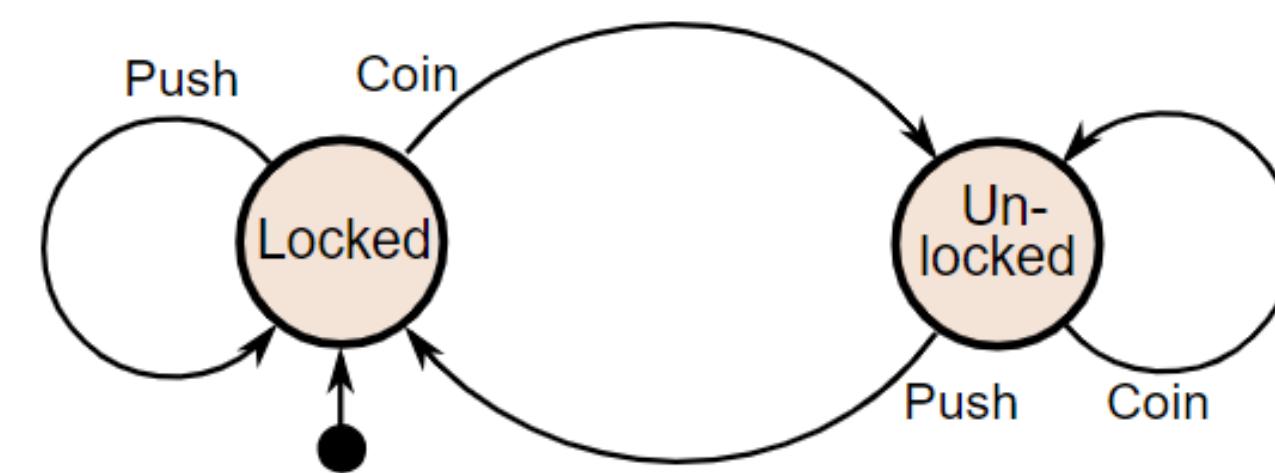
A Turing machine is an improved version of the Finite state Machine.

# Automata Formal Languages & Logic

## Introduction

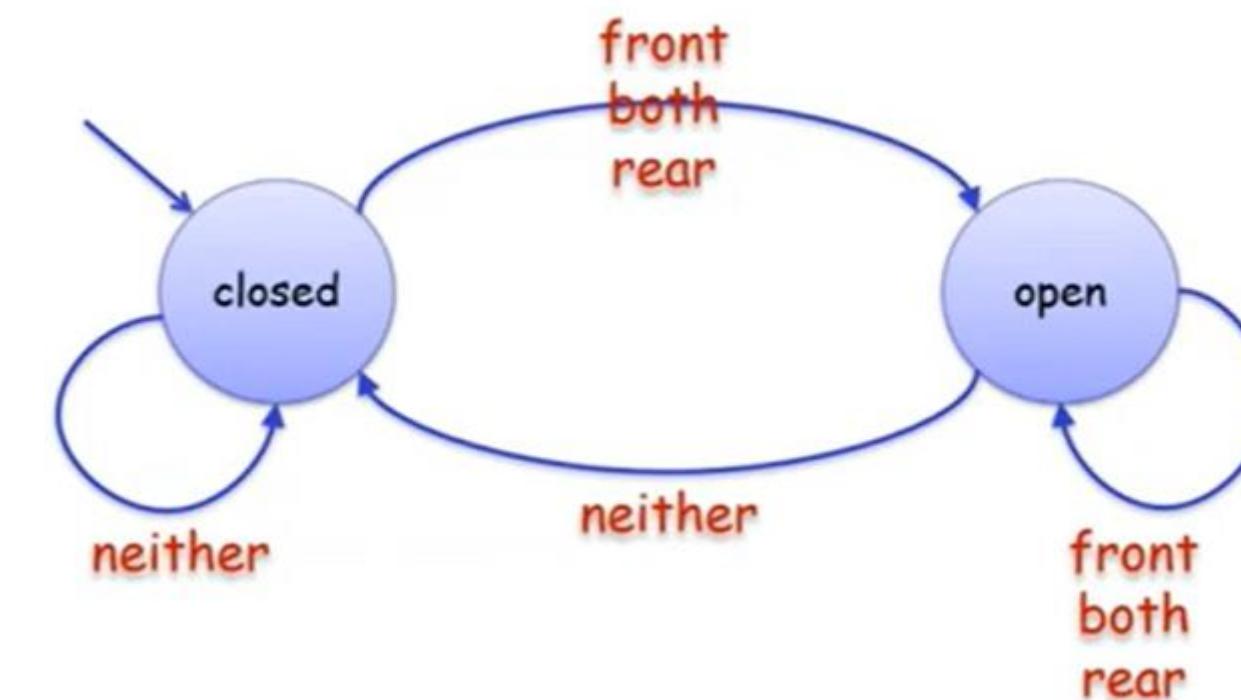
### Finite State Machine (Finite State Automata)

- Example 3: coin-operated turnstile



State diagram for a turnstile

- Example 4: Controller used for an automatic sliding door (simple example of a FSM)



# Automata Formal Languages & Logic

## Introduction

---

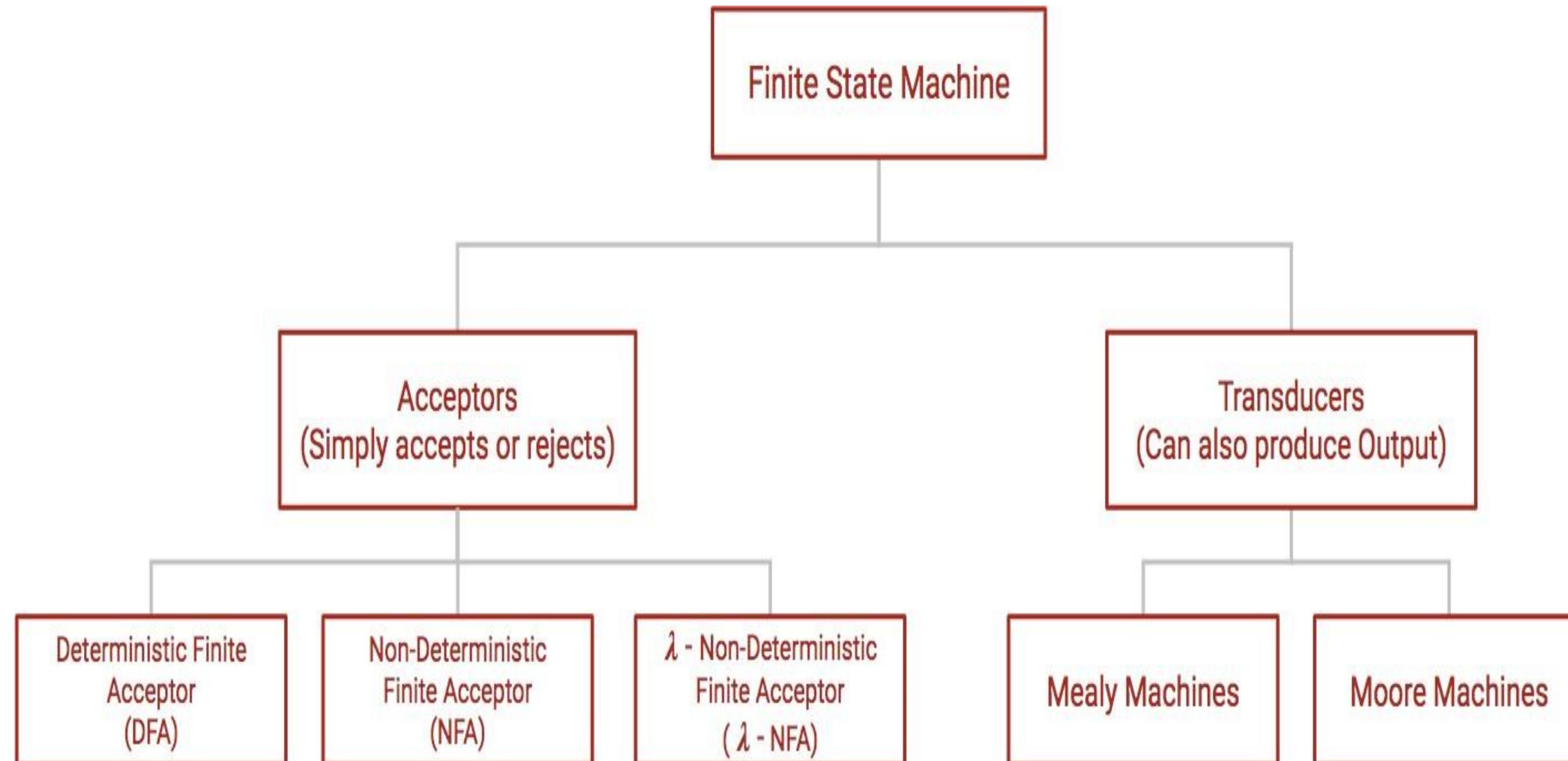
### Finite State Automata (FSA) / Finite State Machine (FSM)

- Through automata, computer scientists are able to understand how machines compute functions and solve problems and more importantly, what it means for a function to be defined as *computable* or for a question to be described as *decidable*.
  
- There are four major families of automaton:
  1. Finite-state machines (Acceptors/Recognizers and Transducers)
    1. Pushdown automata
    2. Linear-bounded automata
    3. Turing machine

# Automata Formal Languages & Logic

## Introduction

### Finite State Automata (FSA) / Finite State Machine (FSM)

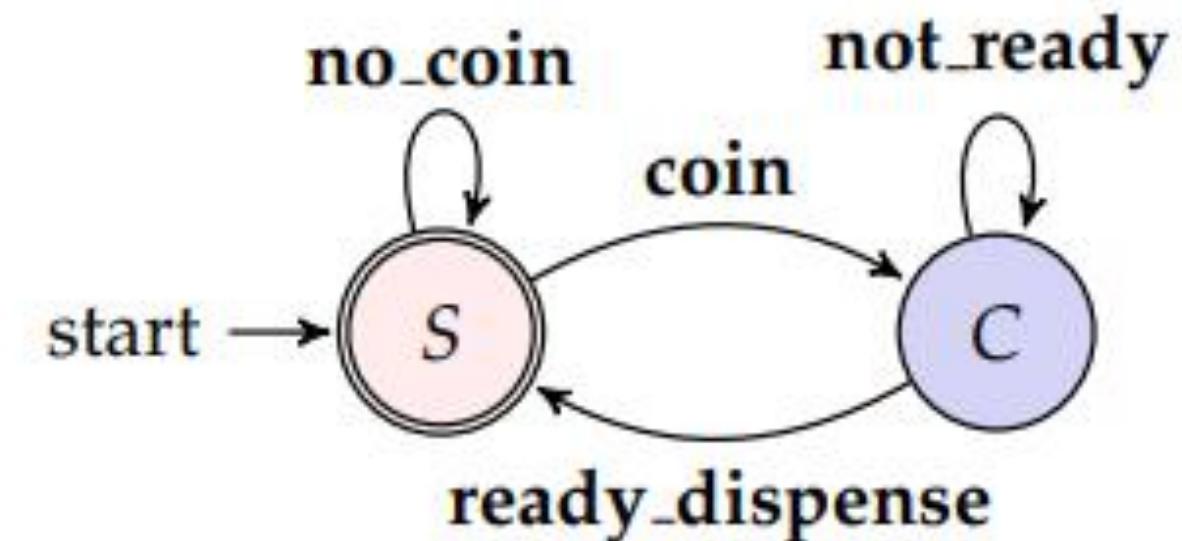
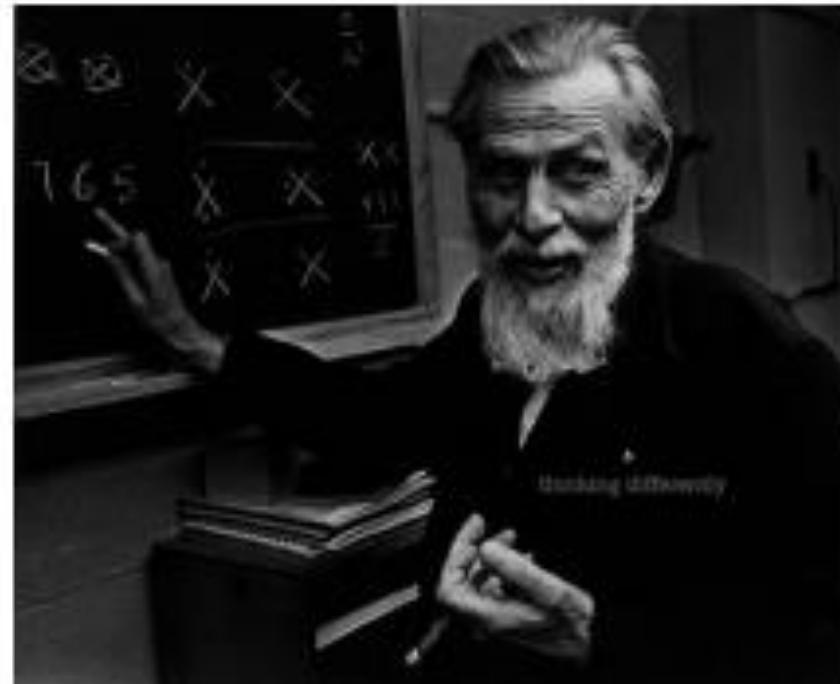


# Automata Formal Languages & Logic

## Introduction

---

## Finite State Automata/ Finite State Machine



**Introduced first by two neuro-psychologist Warren S. McCullough and Walter Pitts in 1943 as a model for human brain!**

Finite automata can naturally model microprocessors and even software programs working on variables with bounded domain - capture so-called regular sets of sequences that occur in many different fields (logic, algebra, regEx).

Applications in digital circuit/protocol verification, compilers, pattern recognition/matching, etc.

### Formal Languages:

- In logic, mathematics, computer science, and linguistics, **a formal language consists of words (or strings) whose letters are taken from an finite non-empty alphabet and are well-formed according to a specific set of rules called a formal grammar.**
  
- Words that belong to a particular formal language are sometimes called **well-formed words** or **well-formed formulas**. A formal language is often defined by means of a formal grammar such as a regular grammar or context-free grammar, which consists of its formation rules.

# Automata Formal Languages & Logic

## Introduction - Formal Languages

---

### Formal Languages:

- Automata theory is closely related to formal language theory. In this context, automata are used as finite representations of formal languages that may be infinite.
- Automata(or FSMs) are often classified by the class of formal languages they can recognize, as in the Chomsky hierarchy, which describes a nesting relationship between major classes of automata.
- A formal language is designed for use in situations in which natural language is unsuitable, as for example in mathematics, logic, or computer programming.  
The symbols and formulas of such languages stand in precisely specified syntactic and semantic relations to one another.

# Automata Formal Languages & Logic

## Introduction - Formal Languages

---

- Formal languages are languages which only consider about the well-formedness nothing else. That means the sufficient and necessary condition is to adhere to its rules.
- **Here are some examples of formal languages:**
  - $L = \Sigma^*$ , the set of *all* words over  $\Sigma$ ;
  - $L = \{a\}^* = \{a^n\}$ , where  $n$  ranges over the natural numbers and " $a^n$ " means "a" repeated  $n$  times (this is the set of words consisting only of the symbol "a");
  - The set of syntactically correct programs in a given programming language (the syntax of which is usually defined by a [context-free grammar](#));
  - The set of inputs upon which a certain [Turing machine](#) halts; or
  - The set of maximal strings of [alphanumeric ASCII](#) characters on this line, i.e., the set {the, set, of, maximal, strings, alphanumeric, ASCII, characters, on, this, line, i, e}.

# Automata Formal Languages & Logic

## Introduction - Formal Languages

---

➤ **Formal languages are normally defined in one of three ways**, all of which can be described by automata theory:

**1) Regular expressions** (only for regular languages)

- Regular Expressions are an algebraic way to describe regular languages.

**2) Standard Automata** - that accepts (or recognizes) the language.

**3) A formal grammar system** - that generates the language

- **Regular (Type-3) grammars** - generate the regular languages - recognized by FSM.
- **Context-free (Type-2) grammars** - generate the context-free languages - recognized by PDA.
- **Context-sensitive (Type-1) grammars** - generate context-sensitive languages - recognized by LBA.
- **Recursively enumerable (Type-0 or Unrestricted grammar) grammars** - generate recursively enumerable or Turing-recognizable languages - recognized by TM.

# Automata Formal Languages & Logic

## Introduction - Formal Languages

---

- Formal languages are normally defined in one of three ways, all of which can be described by automata theory:

**Formal Language  $L = \{\epsilon, a, aa, aaa, aaaa, \dots\}$**

- 1) Regular expressions (only for regular languages)

**Regex** →  $a^*$

- 2) Standard Automata - that accepts (or recognizes) the language



- 3) A formal grammar system

**Grammar** →  $S \rightarrow aS$   
 $S \rightarrow \epsilon$

# Automata Formal Languages & Logic

## Applications

---

### 1. Robotics and Control Systems:

FSMs are used to model the behaviour of Robots and Control systems.

### 2. Compiler Design:

Finite automata are extensively used in Compiler Design for Lexical analysis, which involves tokenizing and scanning the source code of a programming language.

Lexical analyzers employ finite automata to recognize and classify different lexical units such as keywords, identifiers, operators, literals and punctuation symbols.

- Lexical analysis (Finite state automata),
- Syntactical analysis (Pushdown automata),
- Code selection (Tree automata)

# Automata Formal Languages & Logic

## Applications

---

**3. Pattern Recognition:** Finite automata are employed in pattern-matching and recognition tasks. Applications include text processing, string matching, and searching algorithms.

**4. Implementing artificially intelligent agents in games,** but many games include characters with simple, state-based behaviors that are easily and effectively modeled using state machines.

**5. Natural Language Processing:** Finite automata are used in natural language processing for tasks such as text tokenization, morphological analysis, and part-of-speech tagging. They help in parsing and understanding the structure of natural language sentences.

# Automata Formal Languages & Logic

## Applications

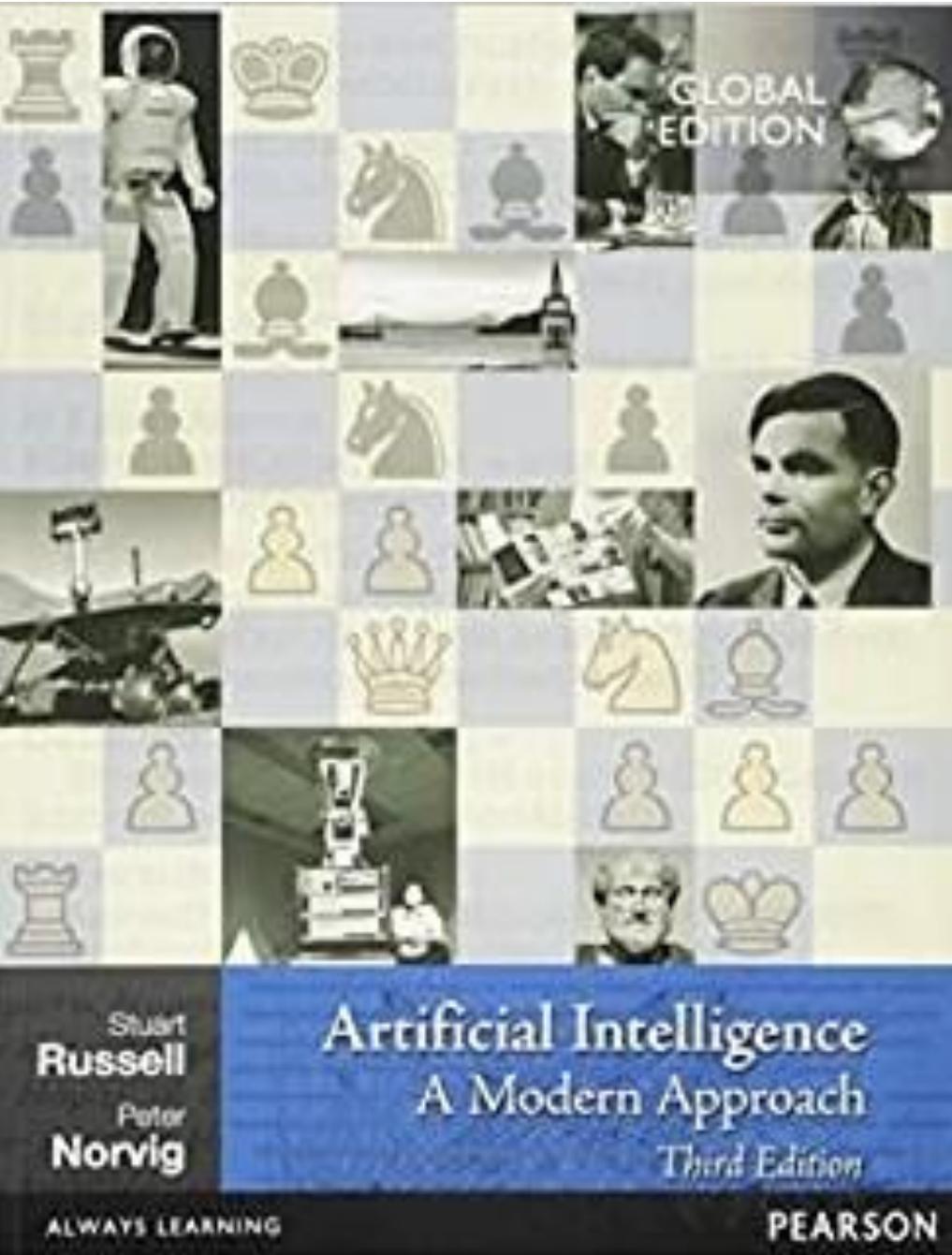
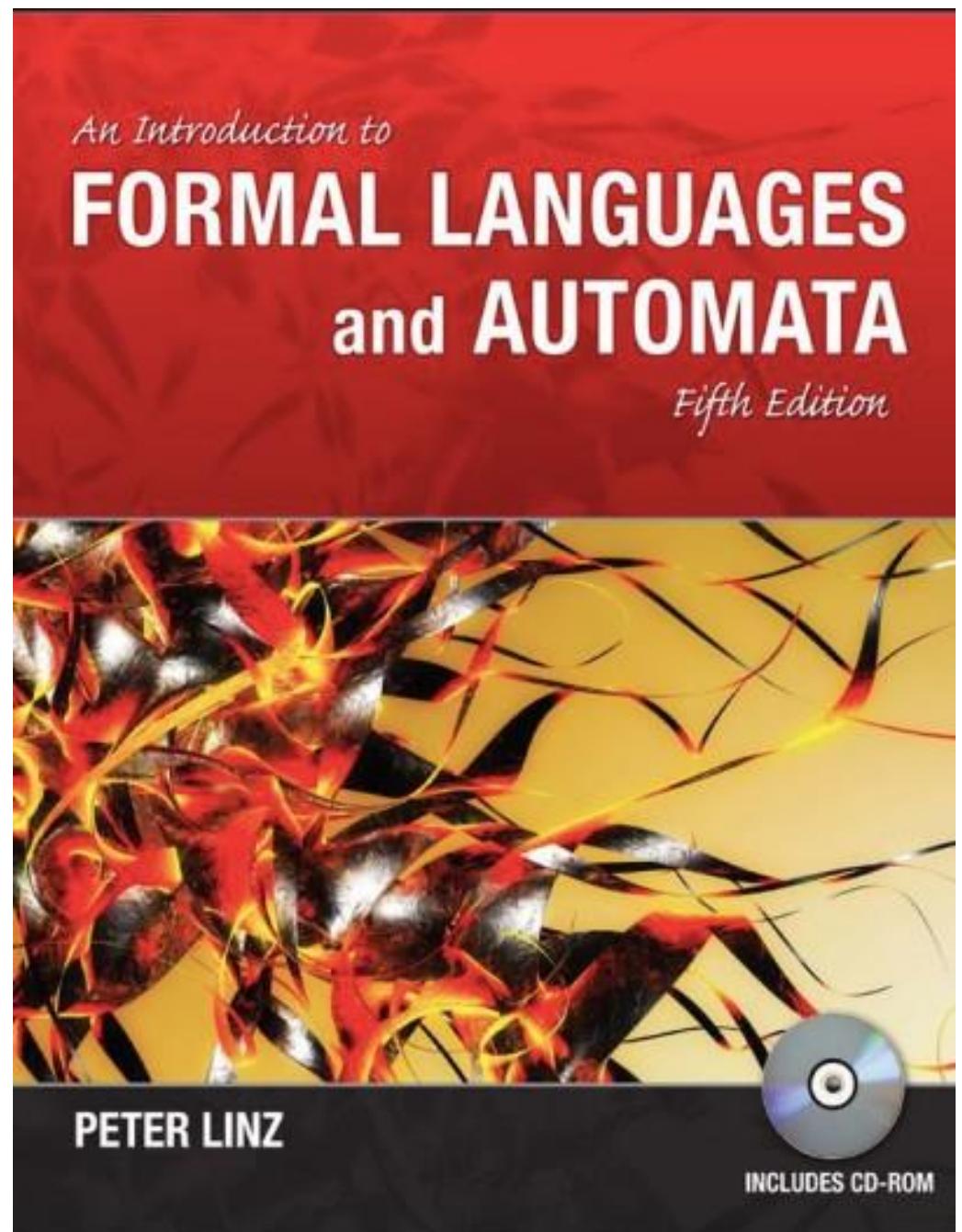
---

- . **6. Digital Circuit Design:** Finite automata can model the behavior of digital circuits and aid in their design and testing. Sequential circuits, such as counters and state machines, can be represented and analyzed using finite automata.
- . **7. Network Protocol Design and Analysis:** Network Protocol Design and Analysis: FSMs are employed to model and analyze communication protocols (e.g., TCP/IP, HTTP, ...), They help to ensure correct message sequencing and behaviour.
- . **8. Software Engineering:** State machines are extensively used in software development to model the behaviour of complex systems, such as user interfaces, protocol implementations, and control systems.
- 9. ....

# Automata Formal Languages & Logic

## Introduction - Course Textbook

---



# Automata Formal Languages & Logic

## Introduction - Course Evaluation Policy

---

### Evaluation Policy

| Evaluation component   | Test Mode  | Conducted for Marks      | Scaled to Marks |
|------------------------|--|--------------------------|-----------------|
| ISA-I                  | CBT  | 40                       | 20              |
| ISA-II                 | CBT  | 40                       | 20              |
| Experiential Learning  | Syntax validation - Coding Assignment in Class using PLY tool:<br><br>Building a foundation for further semester courses.    | 6 Marks                  | 10              |
|                        | Unit-wise Problem-solving Assignment in Class:<br><br>5 questions/unit, student should solve in a sheet of paper and submit. | 4 Marks<br>(1 Mark/Unit) |                 |
| Total (ISA)            |  | 50                       |                 |
| ESA                    | Pen and Paper  | 100                      | 50              |
| <b>Total (ISA+ESA)</b> |  |                          | <b>100</b>      |



# THANK YOU

---

Prakash C O

Department of Computer Science & Engineering

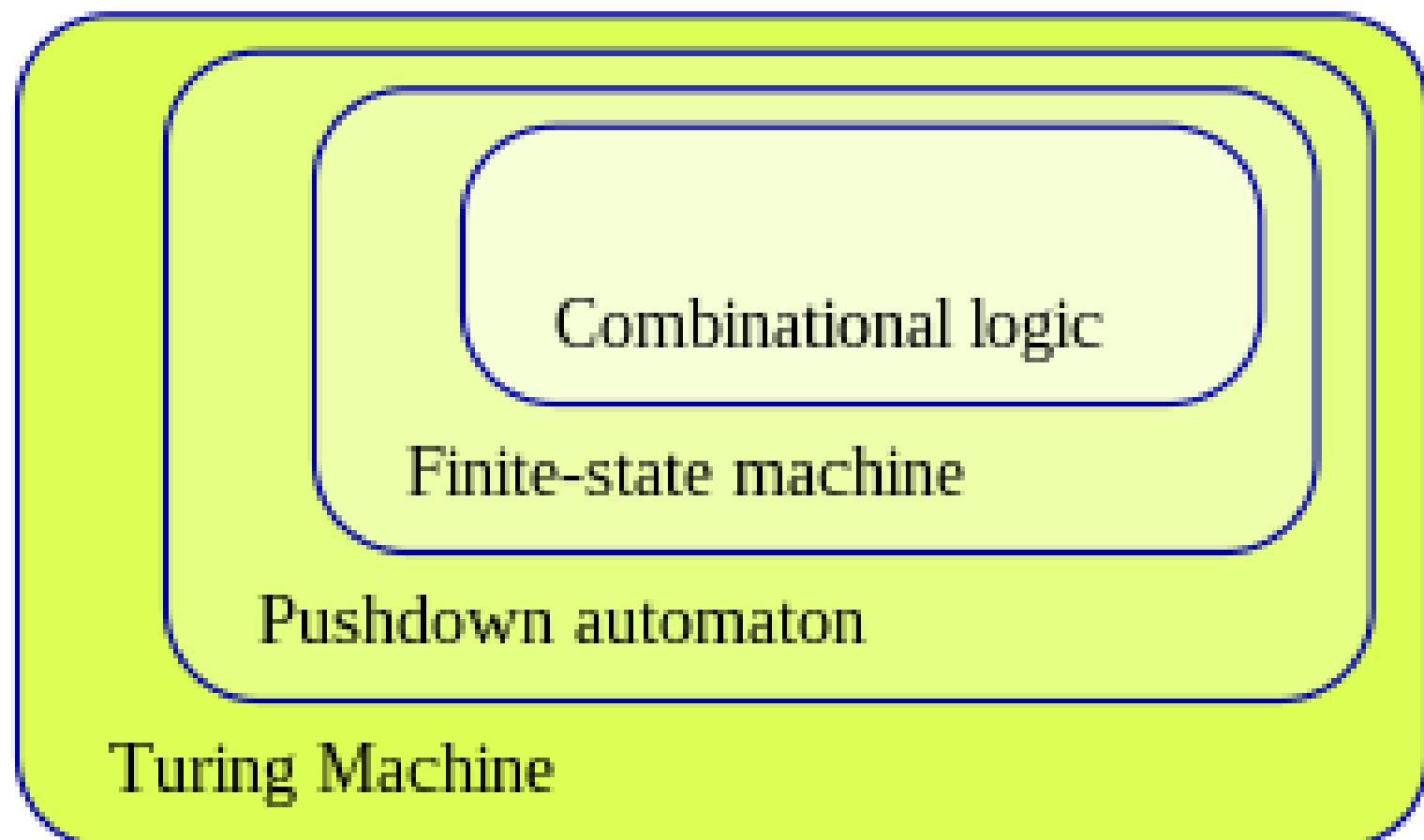
[coprakash@pes.edu](mailto:coprakash@pes.edu)

# Automata Formal Languages & Logic

## Introduction - Additional Info

---

Automata theory



In automata theory, **combinational logic** is a type of digital logic that is implemented by Boolean circuits, where the output is a pure function of the present input only. This is in contrast to sequential logic, in which the output depends not only on the present input but also on the history of the input. In other words, sequential logic has *memory* while combinational logic does not.

# Automata Formal Languages & Logic

## Introduction - Additional Info

---

### $\omega$ -automaton (or stream automaton)

- In automata theory, an  $\omega$ -automaton (or stream automaton) is a variation of a finite automaton that runs on infinite, rather than finite, strings as input.
- An  $\omega$ -automaton either accepts or rejects infinite inputs. Since  $\omega$ -automata do not stop, they have a variety of acceptance conditions rather than simply a set of accepting states.
- Classes of  $\omega$ -automata include the Büchi automata, Rabin automata, Streett automata, parity automata and Muller automata, each deterministic or non-deterministic. These classes of  $\omega$ -automata differ only in terms of acceptance condition.

# Automata Formal Languages & Logic

## Introduction - Additional Info

---

### Tree Automata:

- Most machine models like Turing machines and finite automata, work on inputs that are strings over some finite alphabet. However, often problem descriptions have inputs that are more structured than strings. For example, the input could be a graph, or a tree, or a partial order, etc.

# Automata Formal Languages & Logic

## Introduction - Additional Info

---

### Automata and Logic:

- The connection between automata and logic goes back to work of Biichi [Bii60] and Elgot [El61], who showed that finite automata and monadic second-order logic (interpreted over finite words) have the same expressive power, and that the transformations from logic formulas to automata and vice versa are effective.
  
- Mathematical logic and automata theory are two scientific disciplines with a fundamentally close relationship.

# Automata Formal Languages & Logic

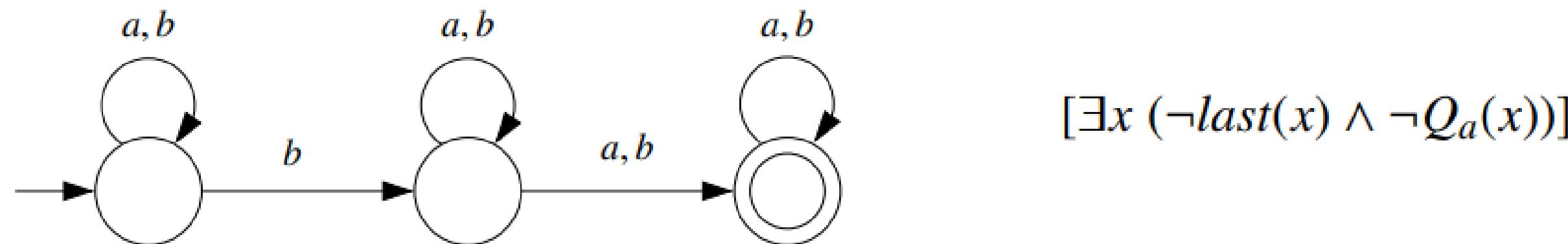
## Introduction - Additional Info

---

### Automata and Logic:

#### ➤ Example:

Intersecting the automata for  $\neg \text{last}(x)$  and  $\neg Q_a(x)$ , and subsequently projecting onto the first component, we get an automaton for  $\exists x (\neg \text{last}(x) \wedge \neg Q_a(x))$



Determinizing, complementing, and removing a useless state yields the following NFA for  $\neg \exists x (\neg \text{last}(x) \wedge \neg Q_a(x))$ :



# Automata Formal Languages & Logic

## Introduction - Additional Info

---

### Applications of FSMs/Automata Theory:

- <https://code.tutsplus.com/finite-state-machines-theory-and-implementation--gamedev-11867t>
- <https://robotika.sk/go/FSM/finite-state-automata.pdf>
- <https://people.computing.clemson.edu/~goddard/texts/theoryOfComputation/5.pdf>
- <https://www.cs.ucdavis.edu/~rogaway/classes/120/spring13/eric-applications.pdf>
- <https://yashindiasoni.medium.com/real-world-applications-of-automata-88c7ba254e80>



**PES**  
UNIVERSITY  
CELEBRATING 50 YEARS

# Automata Formal Languages & Logic

## Unit 1 - Mathematical Preliminaries

---

**Preet Kanwal**  
Department of Computer Science & Engineering

# Automata Formal Languages & Logic

## Mathematical Preliminaries

---

1. Sets

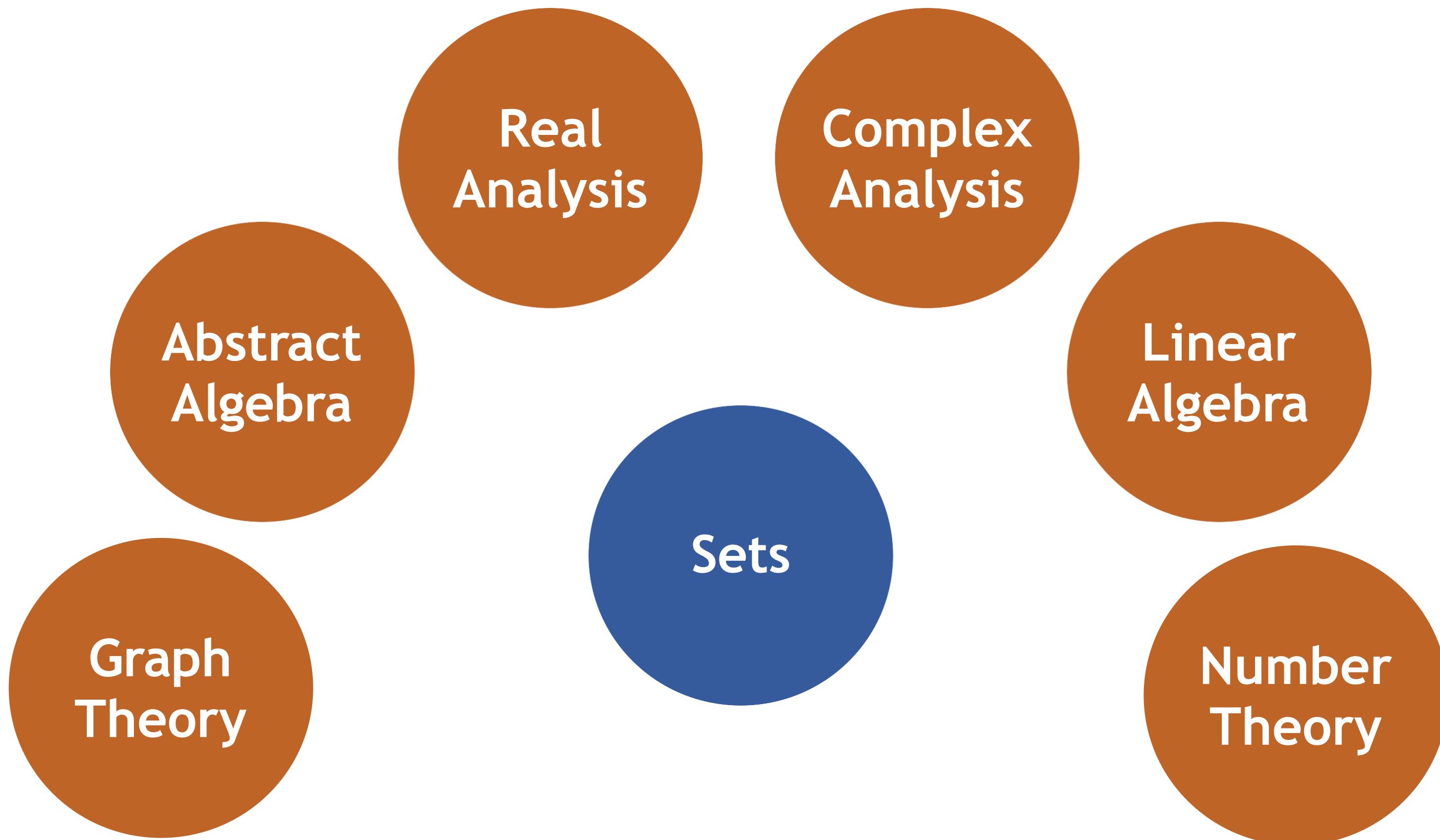
2. Functions and

3. Relations

# Automata Formal Languages & Logic

## Why are Sets important?

---



# Automata Formal Languages & Logic

## Sets

A **set** is an unordered collection of distinct objects, which may be anything (including other sets).

Example :



**Set Notation :**

Curly braces with commas separating out the elements

# Automata Formal Languages & Logic

## Sets

A **set** is an **unordered collection of distinct objects**, which may be anything (including other sets).



These are same sets



# Automata Formal Languages & Logic

## Set Representation

---

A set can be represented in any one of the following three ways or forms.

- (i) Descriptive form (or statement form)
- (ii) Set-builder form or Rule form
- (iii) Roster form or Tabular form

(i) **Descriptive form - verbal/english description of its elements.**

**Examples:**

- “The set of all even numbers.”
- “The set of all real numbers less than 150.”
- “The set of all negative integers.”

# Automata Formal Languages & Logic

## Set Representation

### (ii) Set-builder form or Rule form:

Let A = “The set of Even Natural Numbers”

In Set-builder form we write  $A = \{ n \mid n \in \mathbb{N} \text{ and } n \text{ is even} \}$

$\{ n \mid n \in \mathbb{N} \text{ and } n \text{ is even} \}$

The set of all  $n$   
where  
 $n$  is a natural  
number  
and  $n$  is even

# Automata Formal Languages & Logic

## Set Representation

---

(iii) **Roster form or Tabular form:** Listing the elements of a set inside a pair of braces {} is called the roster form.

For example:

Let  $A = \{x \mid x \text{ is an integer and } -1 \leq x < 5\}$

In roster form we write  $A = \{-1, 0, 1, 2, 3, 4\}$

# Automata Formal Languages & Logic

## Set Membership

---



$$\in \{ \text{1 rupee coin with thumbs up}, \text{2 rupee coin with flower}, \text{5 rupee coin with Lion Capital}, \text{10 rupee coin with Lion Capital} \}$$

Is



in the set?

# Automata Formal Languages & Logic

## Set Membership

---



notin

{ 1 ₹1 , 2 ₹2 , 5 ₹5 , 10 ₹10 }

Is



in the set?

The *cardinality* of a set S is the number of elements it contains, denoted by  $|S|$ .

### Examples:

- 1) if Set A = {*a, b, c, d, e*} then  $|A| = 5$
- 2) if Set B = {{*a, b*}}, {{*c, d, e, f, g*}}, {{*h*}} then  $|B| = 3$
- 3) if Set C = {  $n \in \mathbb{N} \mid n < 137$ } then  $|C| = 137$

# Automata Formal Languages & Logic

## Order of a Set / Cardinality

---

Question :

1) What is  $|N|$ ?

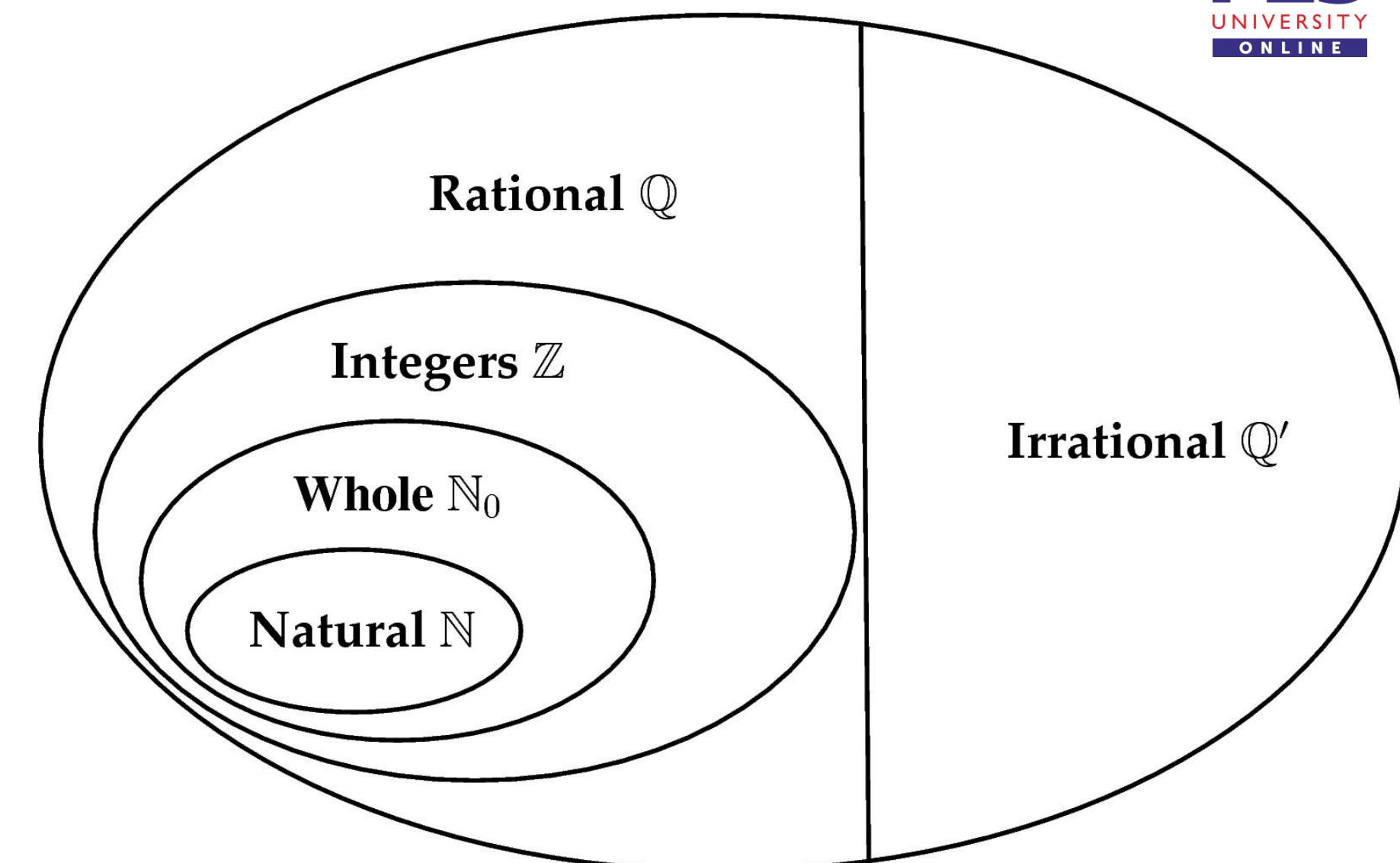
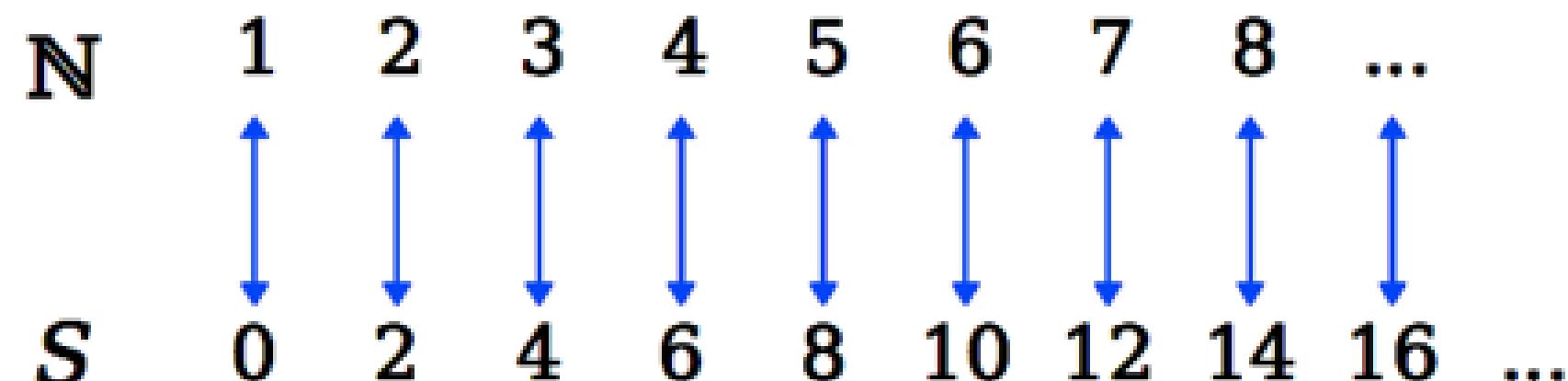
2) Consider the set  $S = \{ n \mid n \in N \text{ and } n \text{ is even} \}$

What is  $|S|$ ?

# Automata Formal Languages & Logic

## Order of a Set / Cardinality

Which infinity is larger?



$$S = \{ n \mid n \in \mathbb{N} \text{ and } n \text{ is even } \}$$

$$|S| = |\mathbb{N}| = \aleph_0$$

Note:  $\aleph_0$ (Aleph-null): the first transfinite cardinal number. It is also the cardinality of the natural numbers.

In mathematics, **transfinite numbers** or **infinite numbers** are numbers that are "infinite" in the sense that they are larger than all finite numbers.

# Automata Formal Languages & Logic

## Types of Sets

---

1. Empty Set or Null Set
2. Singleton Set
3. Finite Set
4. Infinite Set
5. Equivalent Set
6. Equal Sets
7. Disjoint Sets
8. Subsets
9. Proper Subset
10. Super Set
11. Universal Set
12. Power Set

# Automata Formal Languages & Logic

## Types of Sets

1. Empty Set or Null Set
2. Singleton Set
3. Finite Set
4. Infinite Set
5. Equivalent Set
6. Equal Sets
7. Disjoint Sets
8. Subsets
9. Proper Subset
10. Super Set
11. Universal Set
12. Power Set

A null set or an empty set is a valid set with no member.

{ }

Ø

We use this symbol to denote empty set

# Automata Formal Languages & Logic

## Types of Sets - Empty or Null Set

---

Question : Are these equal ?

$$\emptyset \text{ !} = \{ \emptyset \} ?$$

This set contains nothing at all.

This set has one element, which happens to be the empty set.

They are not equal

# Automata Formal Languages & Logic

## Types of Sets - Empty or Null Set

---

Question : Are these equal ?

2 ! = {2} ?

This is a number.

This is a set.  
It contains a number.

They are not equal

# Automata Formal Languages & Logic

## Types of Sets

1. Empty Set or Null Set
2. Singleton Set
3. Finite Set
4. Infinite Set
5. Equivalent Set
6. Equal Sets
7. Disjoint Sets
8. Subsets
9. Proper Subset
10. Super Set
11. Universal Set
12. Power Set

If a set contains only one element it is called to be a singleton set.

Example :

Set A = {1}

Set B = {a}

Set C = {4}

Set D = {car}

# Automata Formal Languages & Logic

## Types of Sets

1. Empty Set or Null Set
2. Singleton Set
3. Finite Set
4. Infinite Set
5. Equivalent Set
6. Equal Sets
7. Disjoint Sets
8. Subsets
9. Proper Subset
10. Super Set
11. Universal Set
12. Power Set

A set in which number of elements are finite (can be counted) are called Finite Sets.

### Example

Set A = {4, 8, 12, 16}

Set B = {90, 180}

# Automata Formal Languages & Logic

## Types of Sets

1. Empty Set or Null Set
2. Singleton Set
3. Finite Set
4. Infinite Set
5. Equivalent Set
6. Equal Sets
7. Disjoint Sets
8. Subsets
9. Proper Subset
10. Super Set
11. Universal Set
12. Power Set

A set in which number of elements are infinite are called Infinite Sets.

### Example

$N = \text{Set of all natural no's}$   
 $= \{1, 2, 3, \dots\}$

$Z = \text{Set of all integers}$   
 $= \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$

# Automata Formal Languages & Logic

## Types of Sets

1. Empty Set or Null Set
2. Singleton Set
3. Finite Set
4. Infinite Set
5. Equivalent Set
6. Equal Sets
7. Disjoint Sets
8. Subsets
9. Proper Subset
10. Super Set
11. Universal Set
12. Power Set

If the number of elements are same for two different sets, then they are called an equivalent sets.

**Example:**

If  $A = \{1, 2, 3, 4\}$  and  
 $B = \{\text{Red, Blue, Green, Black}\}$

Here, Set A and B are equivalent,  
since  $|A| = |B| = 4$

# Automata Formal Languages & Logic

## Types of Sets

1. Empty Set or Null Set
2. Singleton Set
3. Finite Set
4. Infinite Set
5. Equivalent Set
6. Equal Sets
7. Disjoint Sets
8. Subsets
9. Proper Subset
10. Super Set
11. Universal Set
12. Power Set

The two sets A and B are said to be equal if they have exactly the same elements, order of elements do not matter.

Example:

if  $A = \{1, 2, 3, 4\}$  and  $B = \{4, 3, 2, 1\}$

Then, Set A and B are equal & can be denoted as :  $A = B$

# Automata Formal Languages & Logic

## Types of Sets

1. Empty Set or Null Set
2. Singleton Set
3. Finite Set
4. Infinite Set
5. Equivalent Set
6. Equal Sets
7. Disjoint Sets
8. Subsets
9. Proper Subset
10. Super Set
11. Universal Set
12. Power Set

The two sets A and B are said to be disjoint if the set does not contain any common element.

**Example:**

Set A = {1,2,3,4} and Set B = {5,6,7,8} are disjoint sets, because there is no common element between them.

# Automata Formal Languages & Logic

## Types of Sets

1. Empty Set or Null Set
2. Singleton Set
3. Finite Set
4. Infinite Set
5. Equivalent Set
6. Equal Sets
7. Disjoint Sets
8. Subsets
9. Proper Subset
10. Super Set
11. Universal Set
12. Power Set

A set  $S$  is a subset of a set  $T$  (denoted  $S \subseteq T$ ) if all elements of  $S$  are also elements of  $T$ .

Examples:

1.  $\{ 1, 2, 3 \} \subseteq \{ 1, 2, 3, 4 \}$
2.  $\mathbb{N} \subseteq \mathbb{Z}$  (every natural number is an integer)
3.  $\mathbb{Z} \subseteq \mathbb{R}$  (every integer is a real number)

# Automata Formal Languages & Logic

## Types of Sets - Subsets

---

**Question :**

Are there any sets  $S$ , where  $\emptyset \subseteq S$ ?

# Automata Formal Languages & Logic

## Types of Sets

1. Empty Set or Null Set
2. Singleton Set
3. Finite Set
4. Infinite Set
5. Equivalent Set
6. Equal Sets
7. Disjoint Sets
8. Subsets
9. Proper Subset
10. Super Set
11. Universal Set
12. Power Set

if B is a proper subset of A (denoted as  $B \subset A$ ), then all elements of B are in A but A contains atleast one element that is not in B.

Example :

$$A = \{1, 3, 5\}$$

$$B = \{1, 5\}$$

$$C = \{1, 3, 5\}$$

then,

$$B \subset A$$

$$\text{But, } C \subseteq A$$

# Automata Formal Languages & Logic

## Types of Sets

1. Empty Set or Null Set
2. Singleton Set
3. Finite Set
4. Infinite Set
5. Equivalent Set
6. Equal Sets
7. Disjoint Sets
8. Subsets
9. Proper Subset
10. Super Set
11. Universal Set
12. Power Set

**if  $B \subseteq A$  ( $B$  is a subset of  $A$ ) then,  
 $A \supseteq B$  ( $A$  is a superset of  $B$ )**

**Here,**  
**if  $B \subset A$  ( $B$  is a proper subset of  $A$ ) then,  
 $A \supset B$  ( $A$  is a proper superset of  $B$ )**

**Consider Example :**

$$\begin{aligned}A &= \{1, 3, 5\} \\B &= \{1, 5\} \\C &= \{1, 3, 5\}\end{aligned}$$

**Here,  $A \supseteq C$  or  $C \supseteq A$  and  $A \supset B$**

# Automata Formal Languages & Logic

## Types of Sets - Subsets

---

**Question :**

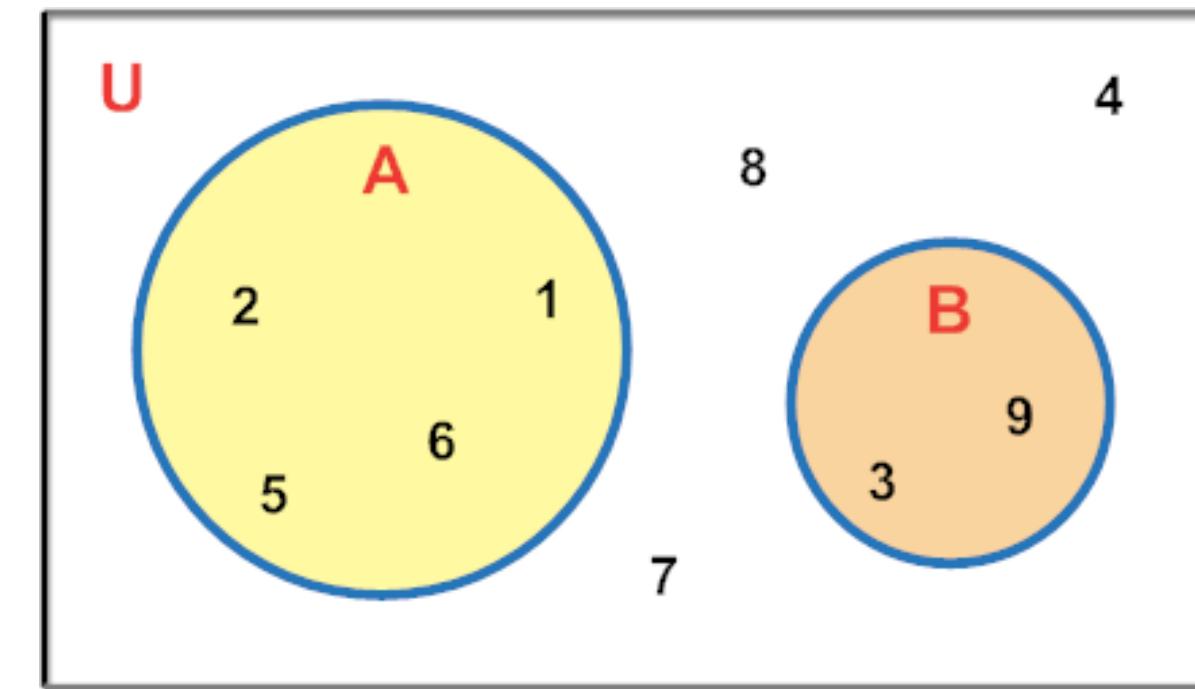
**Is every set a subset and superset of itself??**

# Automata Formal Languages & Logic

## Types of Sets

1. Empty Set or Null Set
2. Singleton Set
3. Finite Set
4. Infinite Set
5. Equivalent Set
6. Equal Sets
7. Disjoint Sets
8. Subsets
9. Proper Subset
10. Super Set
11. Universal Set
12. Power Set

A universal set, denoted by capital U is all the elements, or members, of any group under consideration, including its own elements.



# Automata Formal Languages & Logic

## Types of Sets

1. Empty Set or Null Set
2. Singleton Set
3. Finite Set
4. Infinite Set
5. Equivalent Set
6. Equal Sets
7. Disjoint Sets
8. Subsets
9. Proper Subset
10. Super Set
11. Universal Set
12. Power Set

Power set of any set  $S$  is the set of all subsets of  $S$ , including the empty set and  $S$  itself

$$S = \{a, b, c, d\}$$

$$P(S) = \{$$

$$\emptyset,$$

$$\{a\}, \{b\}, \{c\}, \{d\},$$

$$\{a, b\}, \{a, c\}, \{a, d\}, \{b, c\},$$

$$\{b, d\}, \{c, d\},$$

$$\{a, b, c\}, \{a, b, d\},$$

$$\{a, c, d\}, \{b, c, d\},$$

$$\{a, b, c, d\}$$

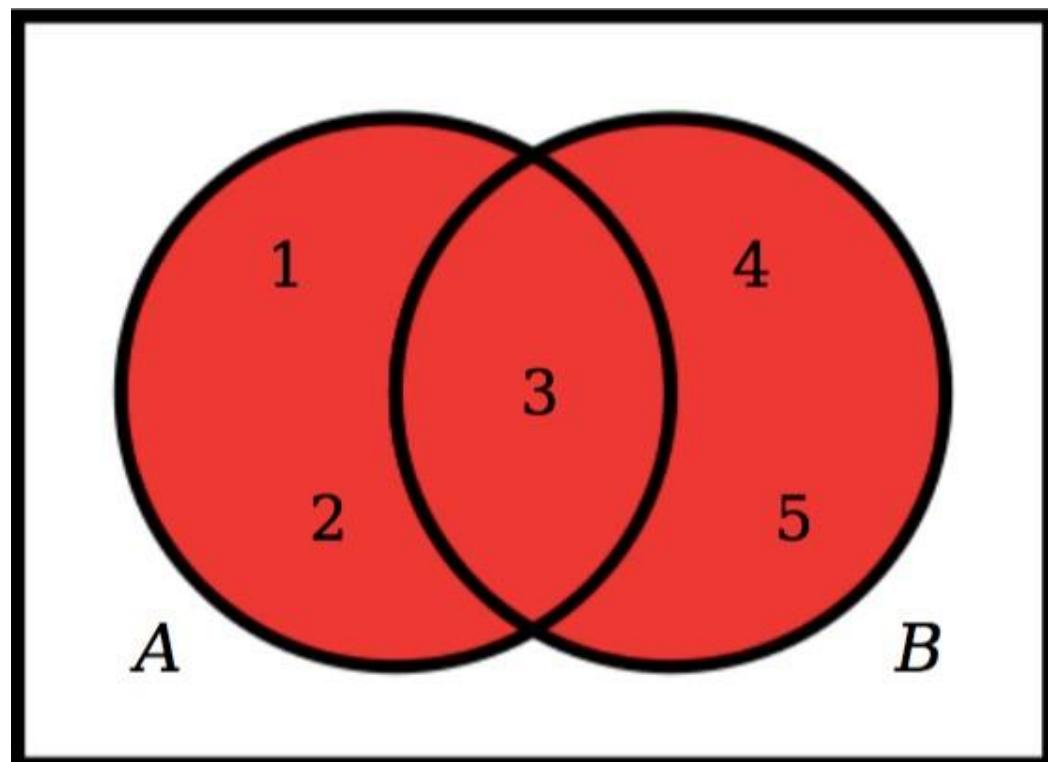
$$\}$$

$$|S| < |P(S)|$$

# Automata Formal Languages & Logic

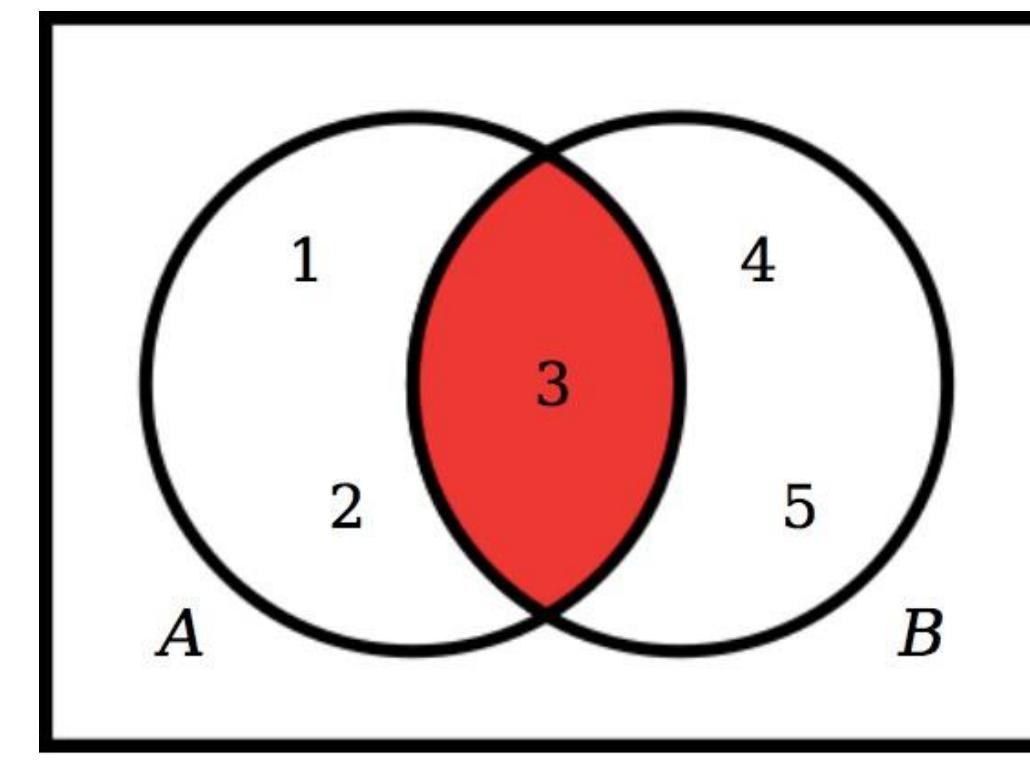
## Combining Sets - Operations on Set

---



$$\begin{aligned}A &= \{ 1, 2, 3 \} \\B &= \{ 3, 4, 5 \}\end{aligned}$$

Union  
 $A \cup B$   
 $\{ 1, 2, 3, 4, 5 \}$



$$\begin{aligned}A &= \{ 1, 2, 3 \} \\B &= \{ 3, 4, 5 \}\end{aligned}$$

Intersection  
 $A \cap B$   
 $\{ 3 \}$

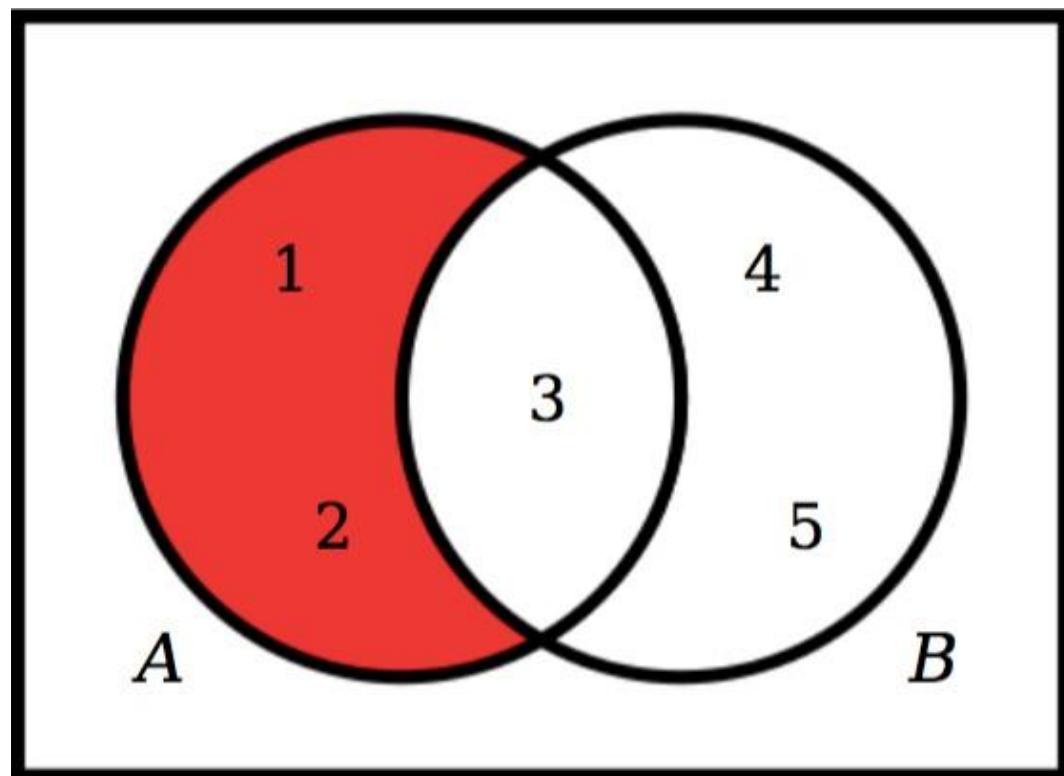
$$A \cup B = \{ x : x \in A \text{ or } x \in B \}$$

$$A \cap B = \{ x : x \in A \text{ and } x \in B \}$$

# Automata Formal Languages & Logic

## Combining Sets

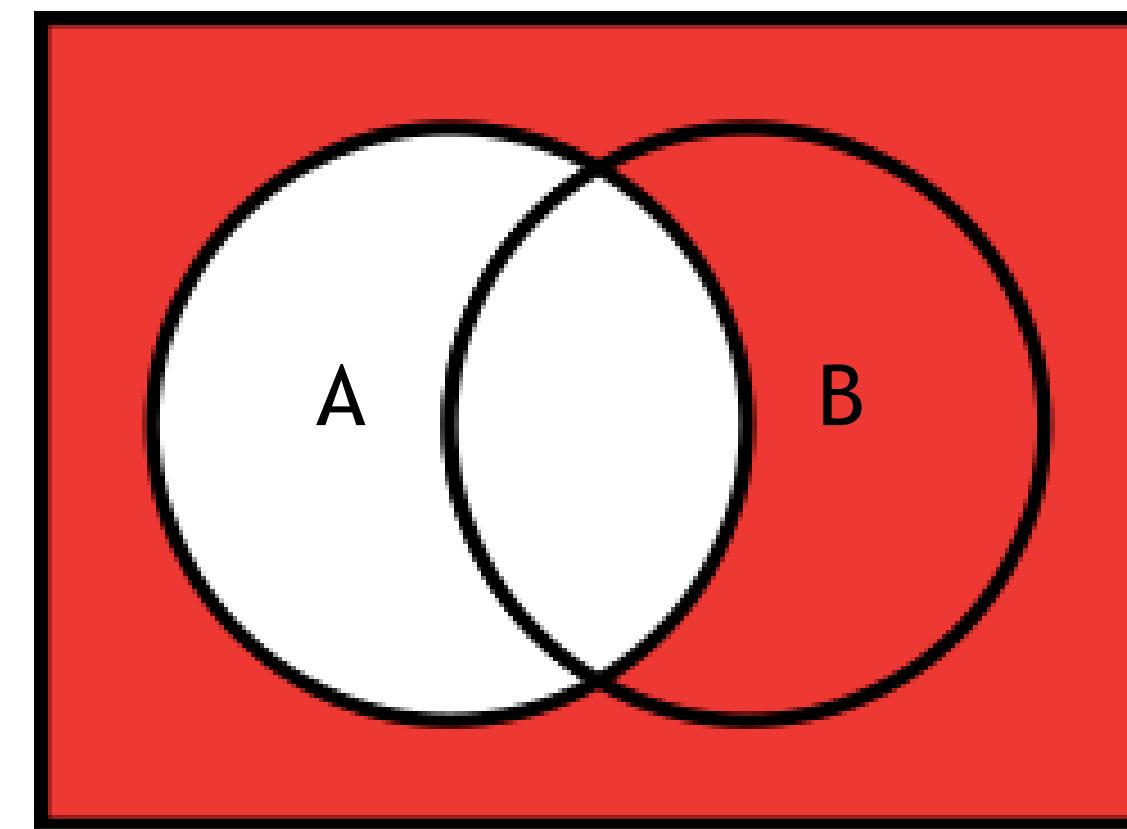
---



$$A = \{ 1, 2, 3 \}$$

$$B = \{ 3, 4, 5 \}$$

Difference  
 $A - B$   
 $\{ 1, 2 \}$



Compliment  
 $A' = \bar{A} = A^C$

$$A - B = \{ x : x \in A \text{ and } x \notin B \}$$

$$A' = \{ x : x \in U \text{ and } x \notin A \}$$

# Automata Formal Languages & Logic

## Set Identities

| <i>Identity</i>  | <i>Name</i>         |
|--|---------------------|
| $A \cap U = A$<br>$A \cup \emptyset = A$   | Identity laws       |
| $A \cup U = U$<br>$A \cap \emptyset = \emptyset$   | Domination laws     |
| $A \cup A = A$<br>$A \cap A = A$   | Idempotent laws     |
| $\overline{(\overline{A})} = A$  | Complementation law |
| $A \cup B = B \cup A$<br>$A \cap B = B \cap A$   | Commutative laws    |
| $A \cup (B \cup C) = (A \cup B) \cup C$<br>$A \cap (B \cap C) = (A \cap B) \cap C$                               | Associative laws    |
| $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$<br>$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$             | Distributive laws   |
| $\overline{A \cap B} = \overline{A} \cup \overline{B}$<br>$\overline{A \cup B} = \overline{A} \cap \overline{B}$ | De Morgan's laws    |
| $A \cup (A \cap B) = A$<br>$A \cap (A \cup B) = A$   | Absorption laws     |
| $A \cup \overline{A} = U$<br>$A \cap \overline{A} = \emptyset$   | Complement laws     |

$$\emptyset' = U$$

# Automata Formal Languages & Logic

## Cartesian Product of Sets

---

The Cartesian product of two sets A and B, denoted  $A \times B$ , is the set of all possible ordered pairs where the elements of A are first and the elements of B are second.

$$A \times B = \{ (a, b) : a \in A \text{ and } b \in B \}$$

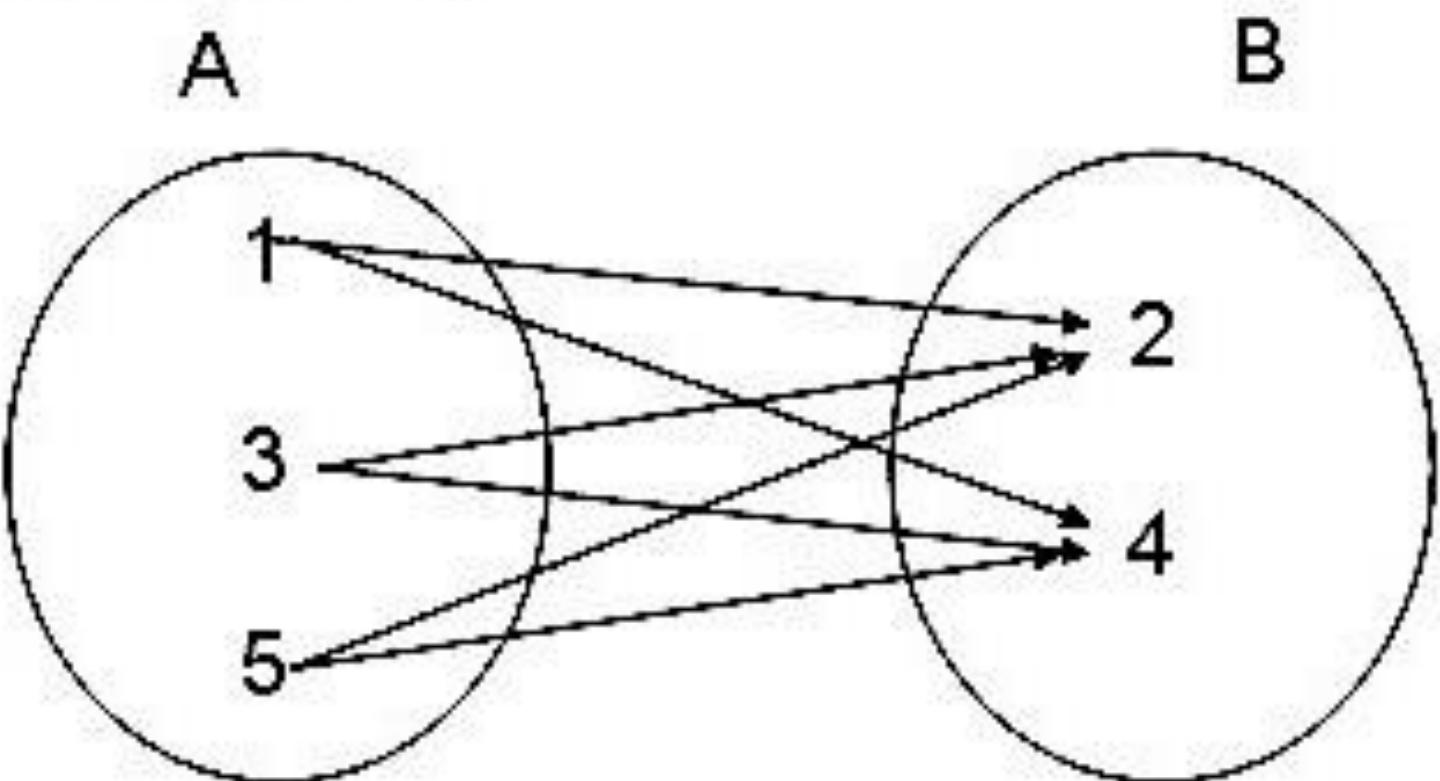
# Automata Formal Languages & Logic

## Cartesian Product of Sets

$A = \{1, 3, 5\}$  and  $B = \{2, 4\}$  then,  
 $A \times B = \{(a, b) : a \in A \text{ and } b \in B\}$

$$A \times B = \{(1, 2), (1, 4), (3, 2), (3, 4), (5, 2), (5, 4)\}$$

Arrow Diagram of  $A \times B$  :



$$\begin{array}{l} 1 \rightarrow 2 \\ 1 \rightarrow 4 \\ 3 \rightarrow 2 \\ 3 \rightarrow 4 \\ 5 \rightarrow 2 \\ 5 \rightarrow 4 \end{array}$$

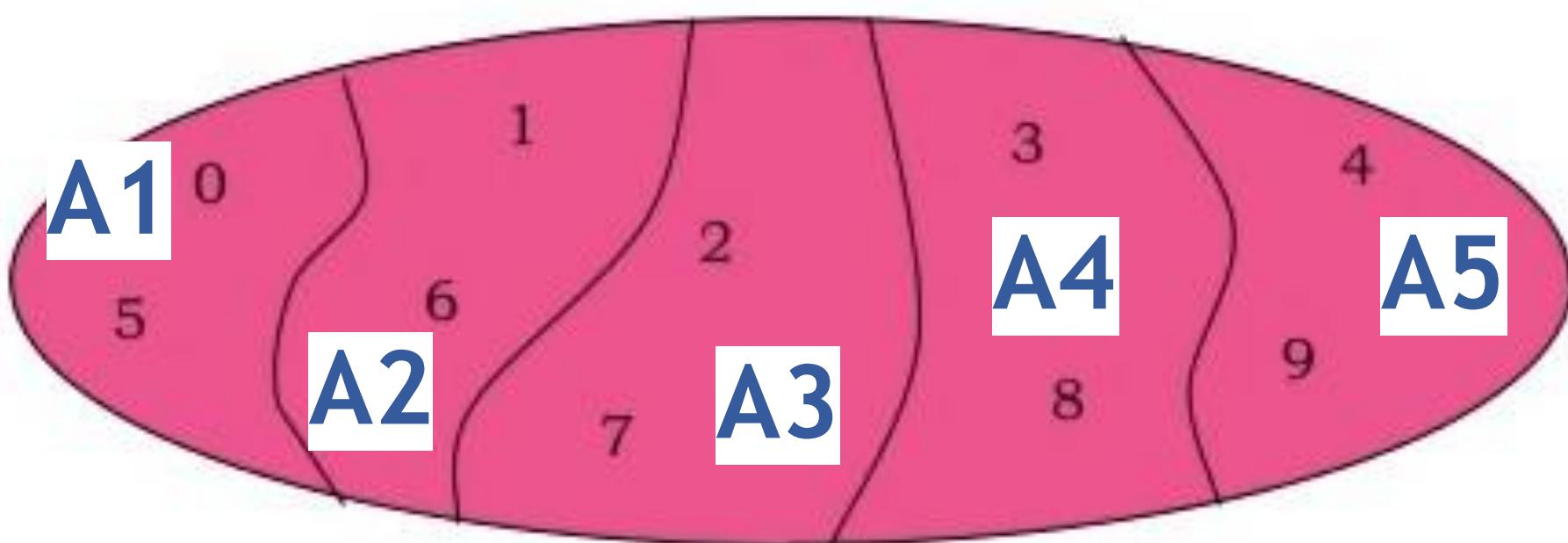
# Automata Formal Languages & Logic

## Partition of a Set

---

A partition of a set is a grouping of its elements into non-empty subsets, in such a way that every element is included in exactly one subset.

For example the regions A1, A2, A3, A4, A5 form partitions of Set A as shown below



A1, A2, A3, A4, A5 are disjoint subsets of A.

$$A = A1 \cup A2 \cup A3 \cup A4 \cup A5$$

# Automata Formal Languages & Logic

## Mathematical Preliminaries

---

**1. Sets**

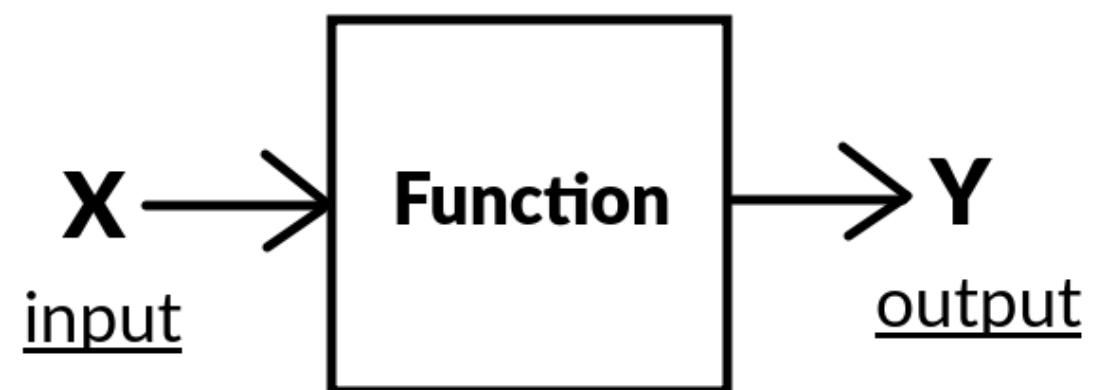
**2. Functions &**

**3. Relations**

# Automata Formal Languages & Logic

## Functions

---

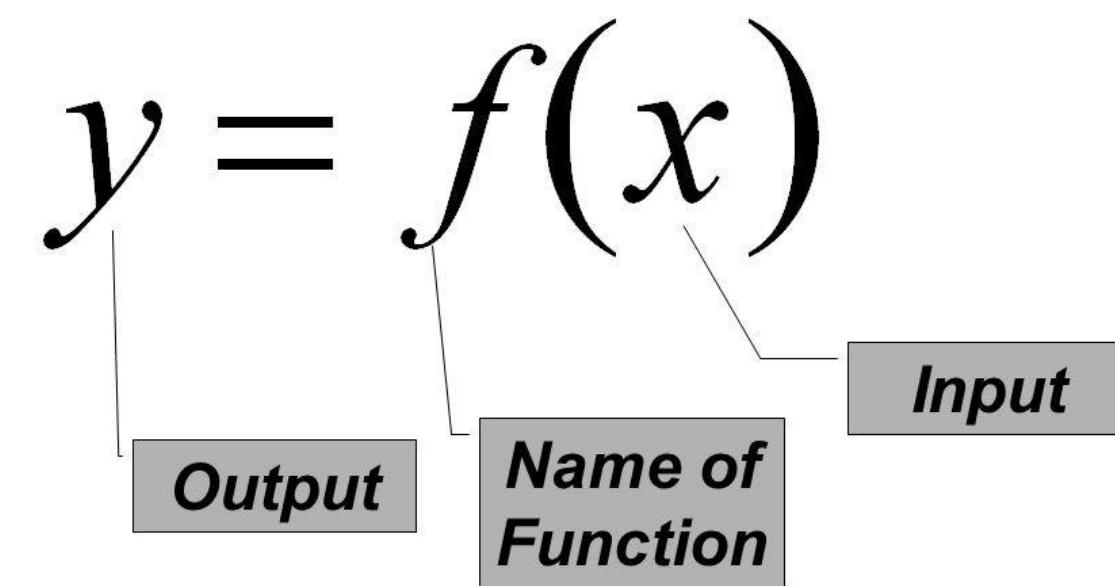


**Example :**

$$f(x) = \{x + 4, 0 < x < 5\}$$

$$f(2) = 6 = y$$

Function Notation



# Automata Formal Languages & Logic

## Functions

---

**Domain of a Function :**

Set of possible inputs to a function is called its domain.

**Range of a Function :**

Set of possible outputs from a function is called its range.

**Notation :**

$f : D \rightarrow R$

**Example :**

$$f(x) = \{x + 4, 0 < x < 5\}$$

$$\text{Domain} = \{1, 2, 3, 4\}$$

$$\text{Range} = \{5, 6, 7, 8\}$$

$$f : \{1, 2, 3, 4\} \rightarrow \{5, 6, 7, 8\}$$

# Automata Formal Languages & Logic

## Functions

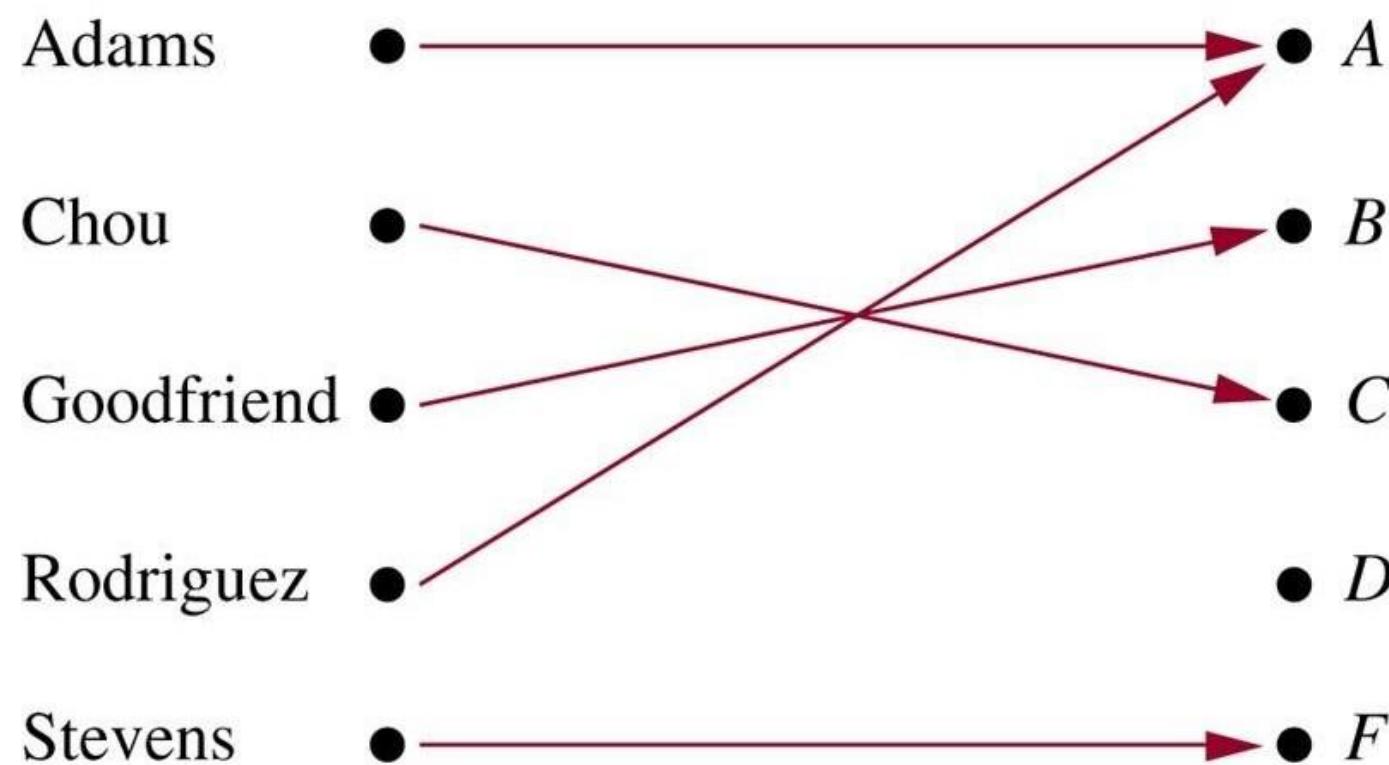
---

- A function  $f$  may not necessarily use all the elements of the Range

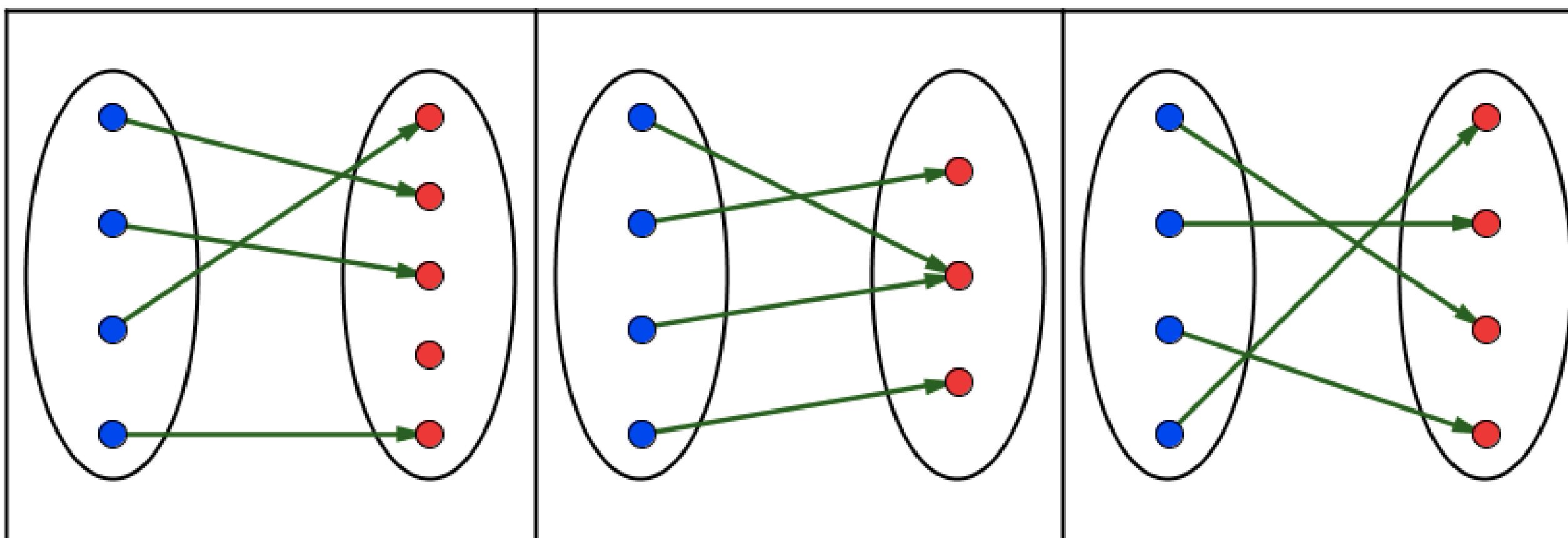
$f : \text{Student} \rightarrow \text{Grades}$

Domain of  $f$  : {Adams, Chou, Goodfriend, Rodriguez, Stevens}

Range of  $f$  : {A, B, C, F}



- 1) One-to-One (Injective) function
- 2) Onto (Surjective) function
- 3) One-to-One Correspondence (Bijective) function

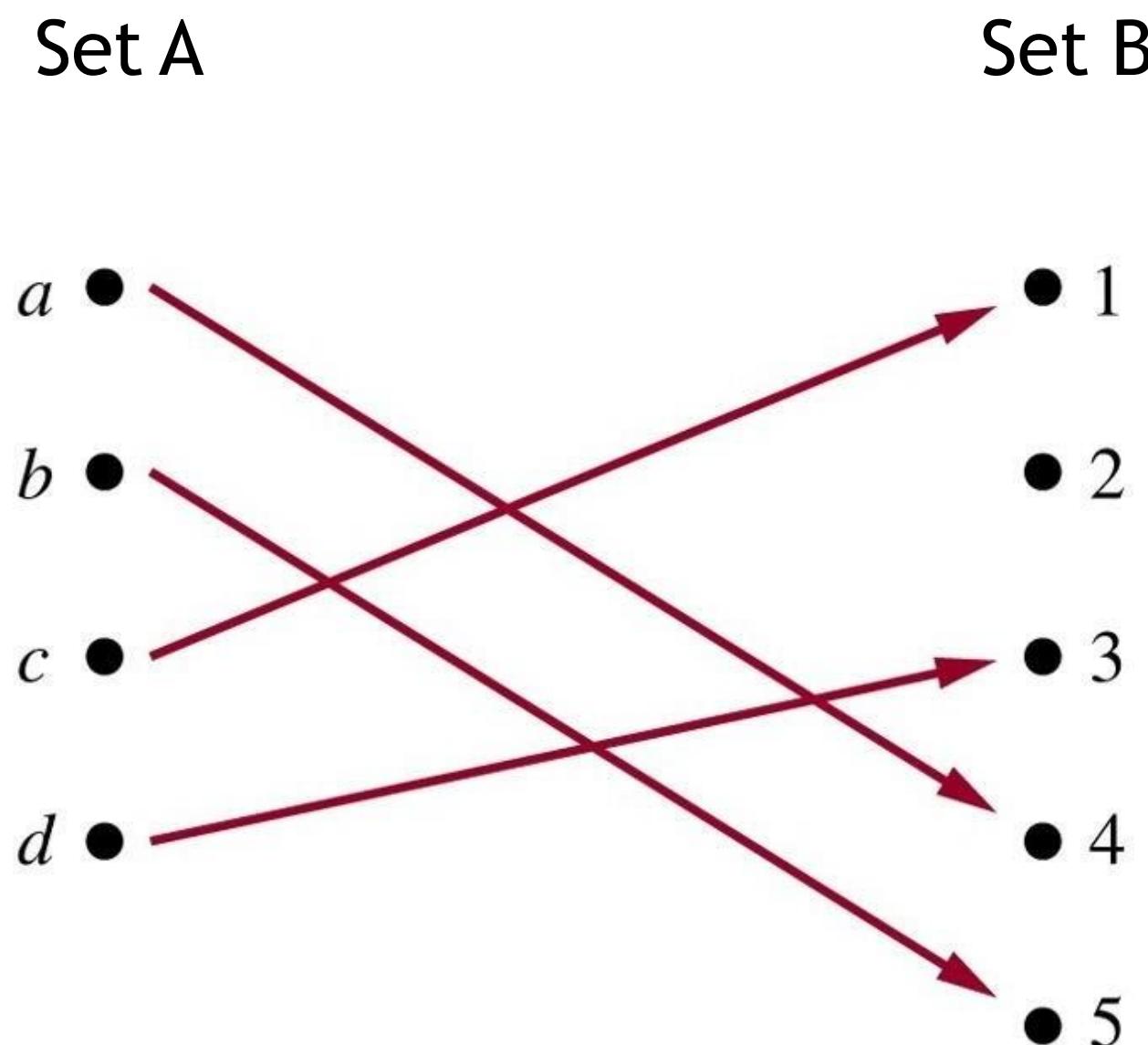


# Automata Formal Languages & Logic

## Types of Functions

---

1) One-to-One (Injective) function - A function  $f: A \rightarrow B$  is One to One if for each element of A there is a distinct element of B.



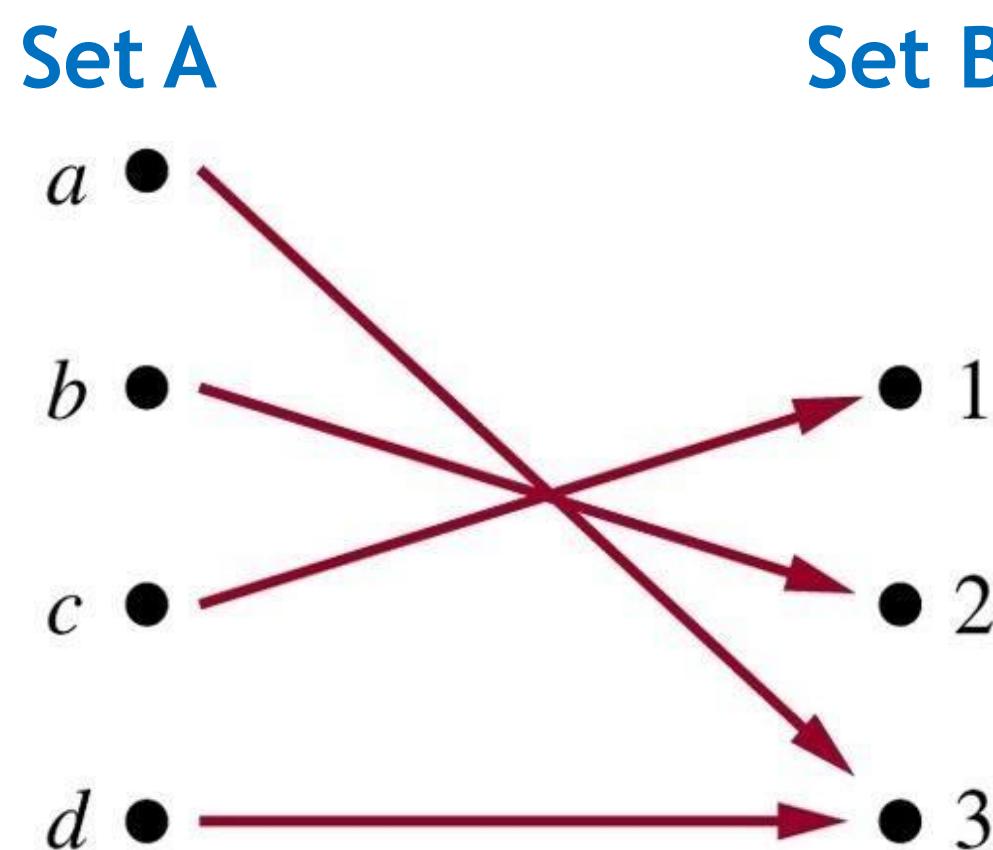
**Image(a) = 4**  
**Pre-image(4) = a**

# Automata Formal Languages & Logic

## Types of Functions

---

2) **Onto (Surjective) function** - all the elements in the Range (i.e., set B) must be used(mapped) and for every element of B, there is at least one or more than one element matching with A, then the function is said to be onto function or surjective function.

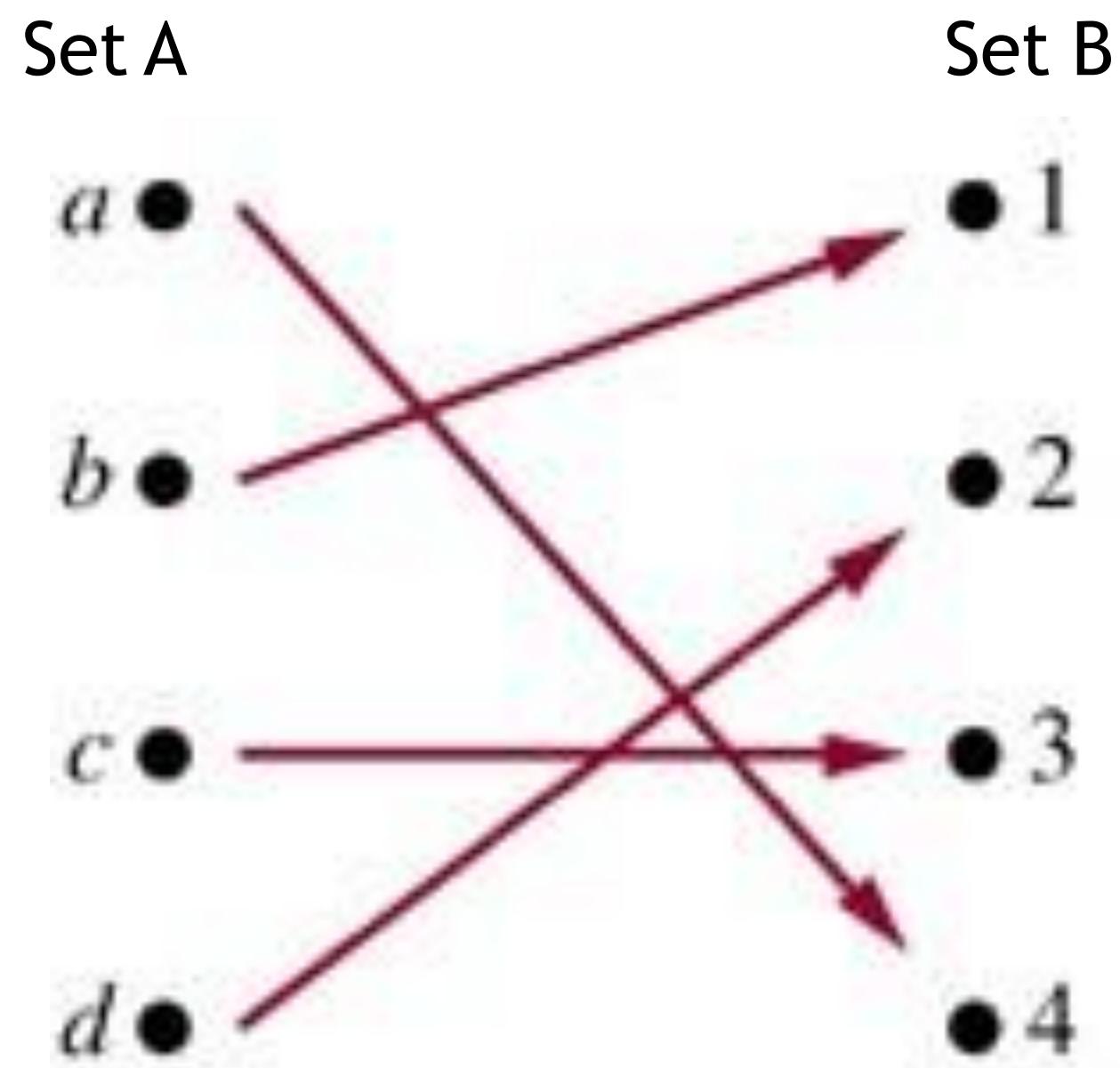


# Automata Formal Languages & Logic

## Types of Functions

---

3) One-to-One and Onto (Bijective) function - Both one-to-one and onto.



# Automata Formal Languages & Logic

## Mathematical Preliminaries

---

**1. Sets**

**2. Functions &**

**3. Relations**

# Automata Formal Languages & Logic

## Relations

---

A relation between two sets is a collection of ordered pairs  $(x,y)$ , containing one object from each set. Here  $x$  is from first set and  $y$  is from second set.

The set of the first components of each ordered pair is called the domain and the set of the second components of each ordered pair is called the range.

A function is a special relation in which each possible input maps to exactly one output value.

Example :

$$R = \{(1,2), (2,4), (3,6)\}$$

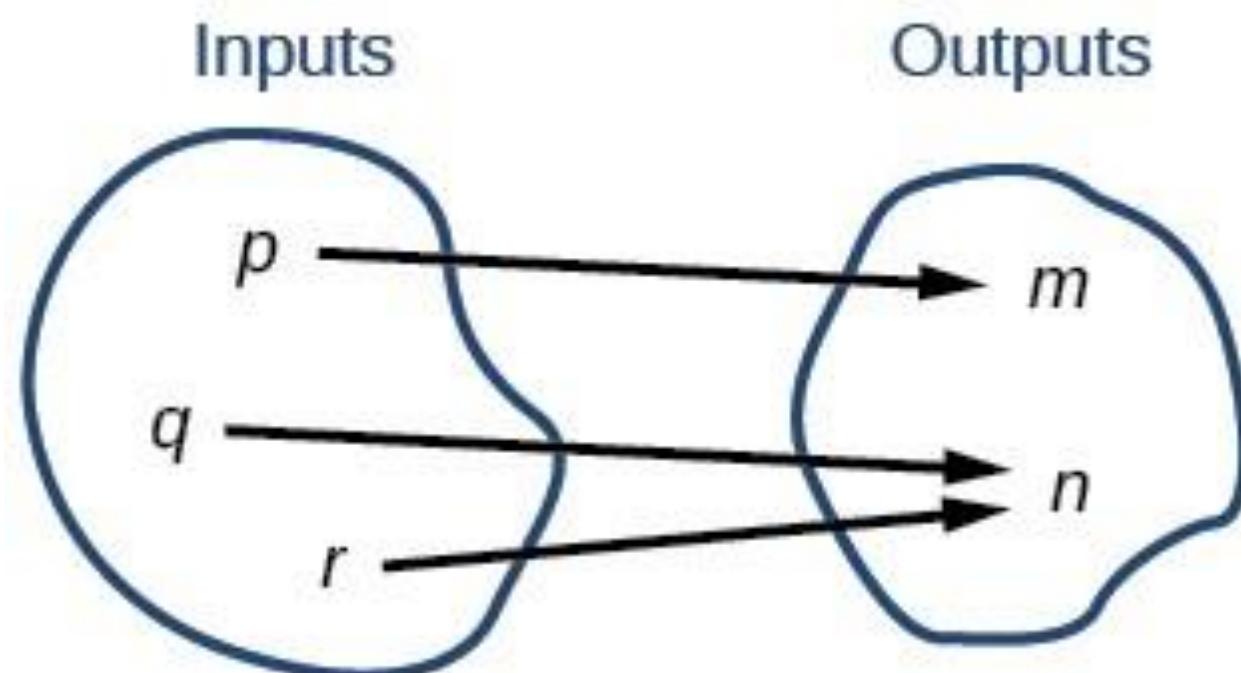
Domain is  $\{1,2,3\}$

Range is  $\{2,4,6\}$

# Automata Formal Languages & Logic

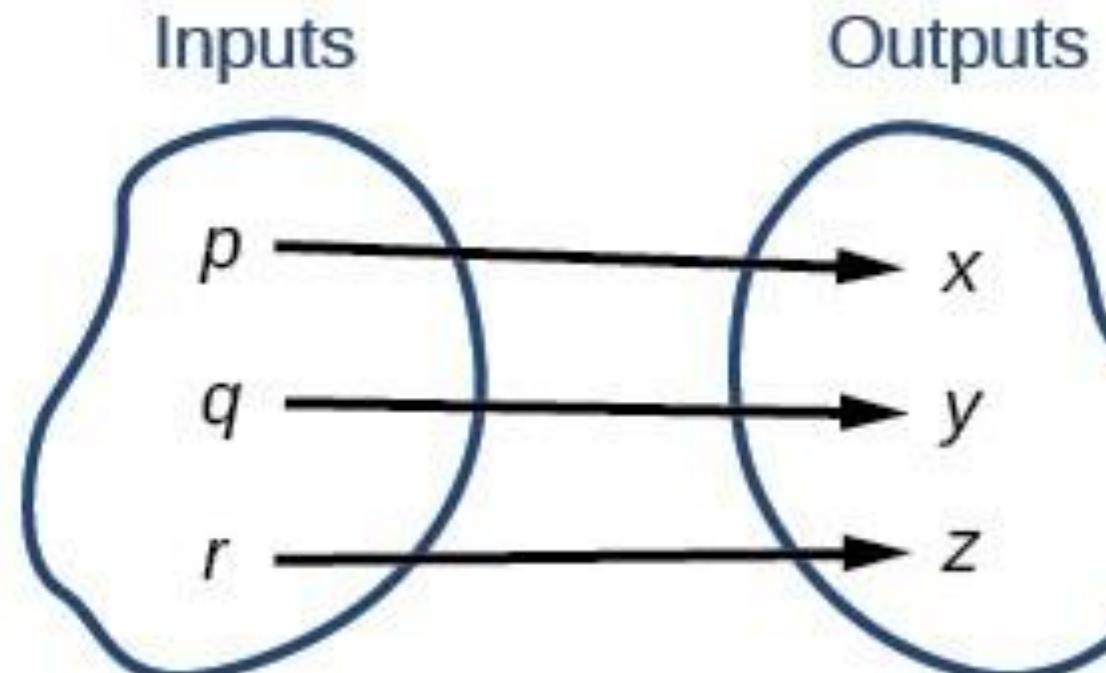
## Is this Relation a function?

**Relation is a Function**



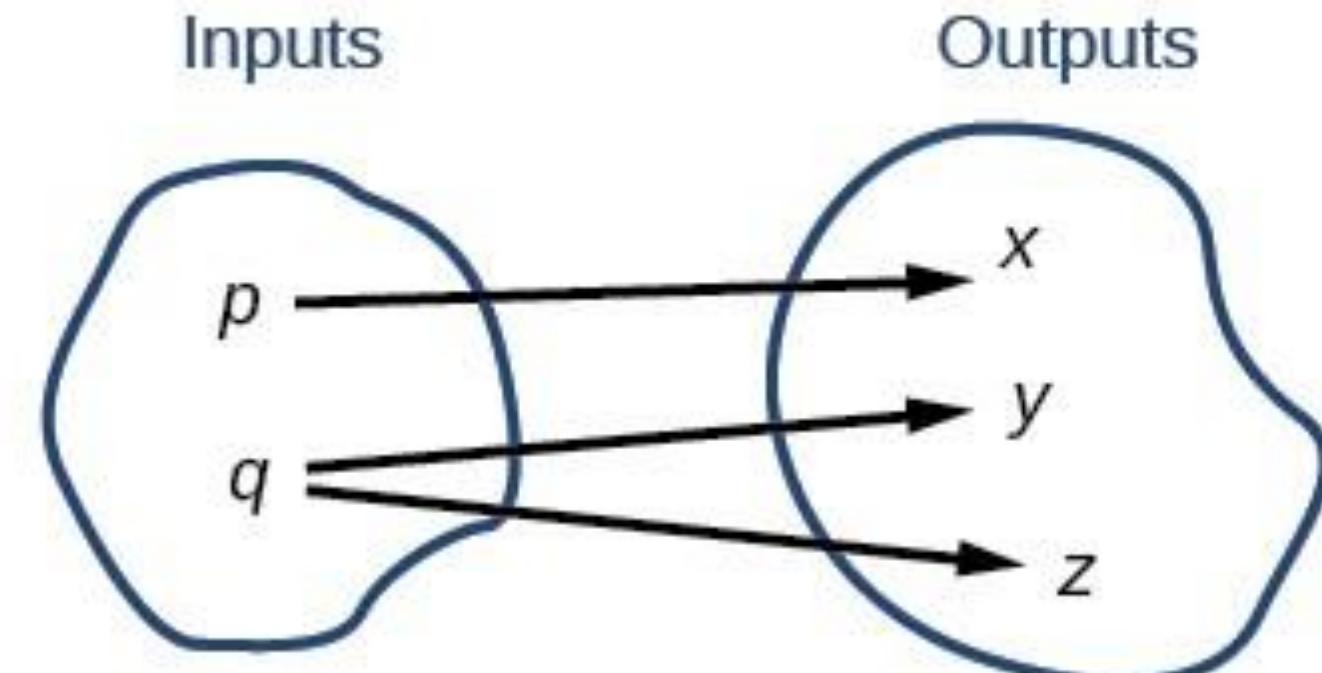
(a)

**Relation is a Function**



(b)

**Relation is NOT a Function**



(c)

# Automata Formal Languages & Logic

## Binary Relations

A binary relation is a relationship between two sets.

Formally, binary relation over sets X and Y is a subset of the Cartesian product  $X \times Y$ ;

Element a is related to b by R is denoted by  $aRb$ .

$aRb$  denotes  $(a, b) \in R$

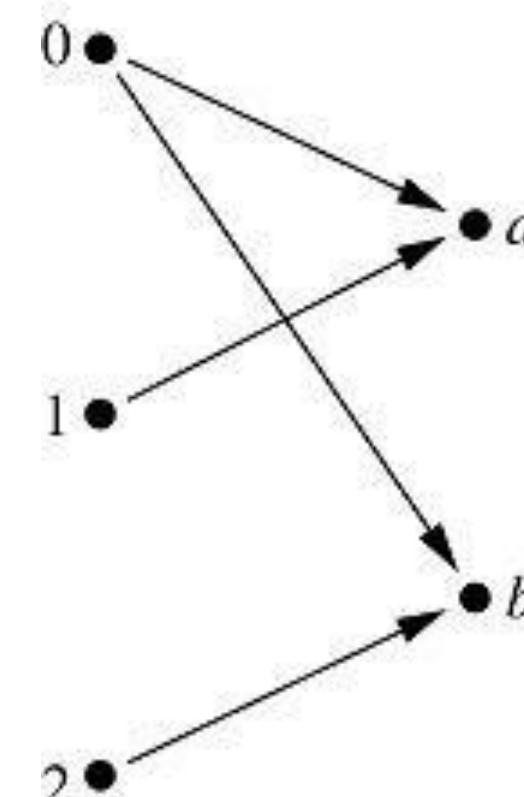
Eg:

$$A = \{0, 1, 2\}$$

$$B = \{a, b\}$$

$$A \times B = \{(0, a), (1, a), (0, b), (1, b), (2, a), (2, b)\}$$

$$R = \{(0, a), (1, a), (0, b), (2, b)\} \subseteq A \times B$$



| R | a | b |
|---|---|---|
| 0 | x | x |
| 1 | x |   |
| 2 |   | x |

# Automata Formal Languages & Logic

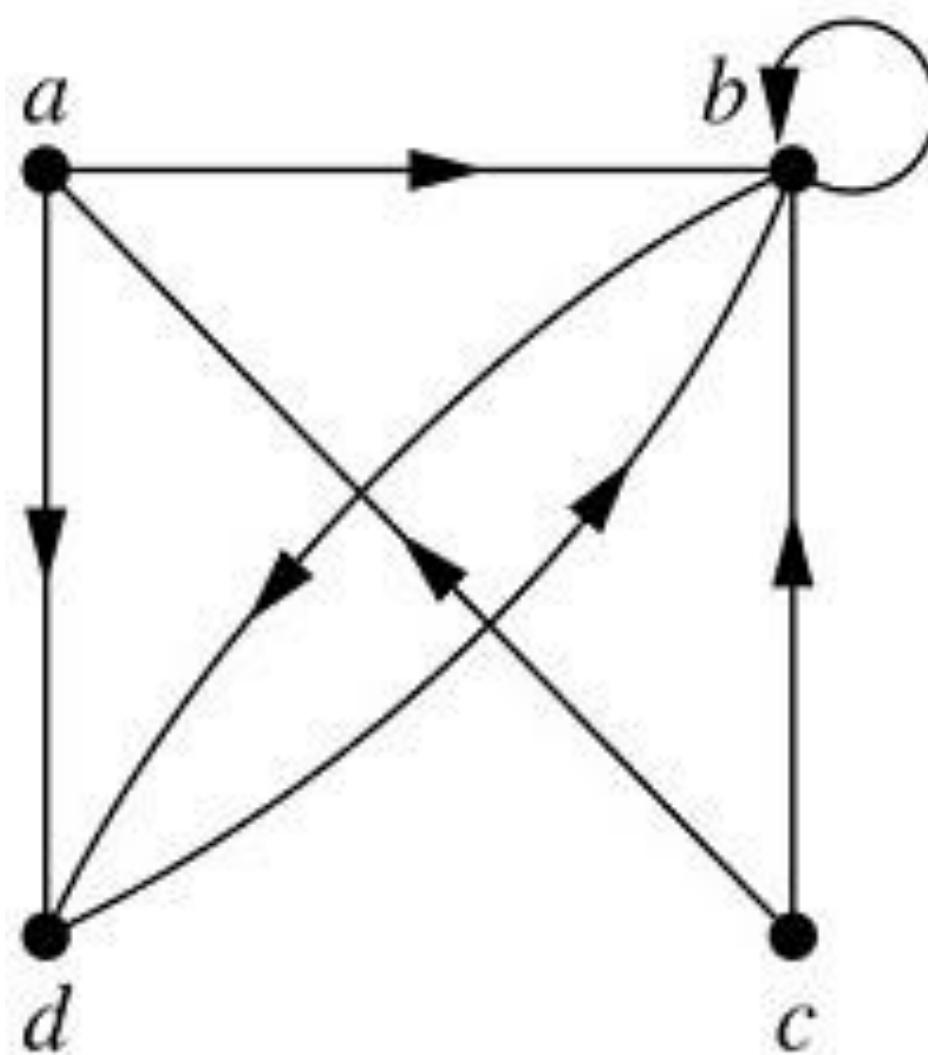
## Representing a Relation

### Representing a Relation :

1) Matrix form

2) Directed Graph (Digraph)

|   | a | b | c | d |
|---|---|---|---|---|
| a | 0 | 1 | 0 | 1 |
| b | 0 | 1 | 0 | 1 |
| c | 1 | 1 | 0 | 0 |
| d | 0 | 1 | 0 | 0 |



Example :

Set A = {a, b, c, d}

Relation R on Set A is a relation from A to A ( $\subseteq A \times A$ )

$R = \{(a, d), (a, b), (b, b), (b, d), (c, b), (c, a), (d, b)\}$

# Automata Formal Languages & Logic

## Properties of a Relation

---

A relation R on a set A ( $A = \{1, 2, 3, 4\}$ ) is

**Reflexive** : iff  $(a,a) \in R$  for every element  $a \in A$ .

$R = \{(1, 1), (2, 2), (3, 3), (4, 4), (1, 2)\}$  is reflexive

**Symmetric** : iff  $(b,a) \in R$  whenever  $(a,b) \in R$ , for all  $a, b \in A$ .

$R = \{(1, 2), (2, 1), (1, 4), (4, 1), (3, 3)\}$  is symmetric

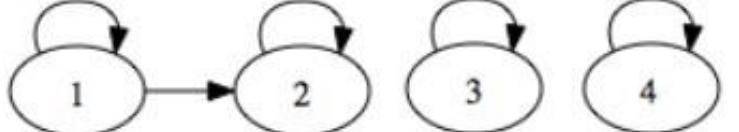
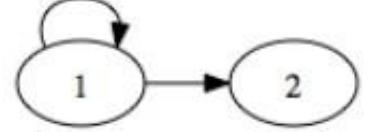
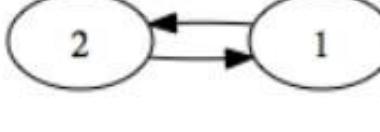
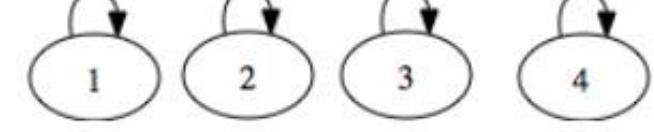
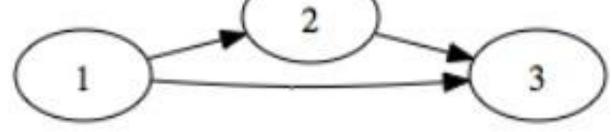
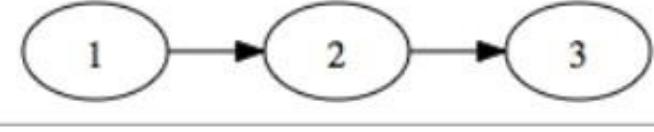
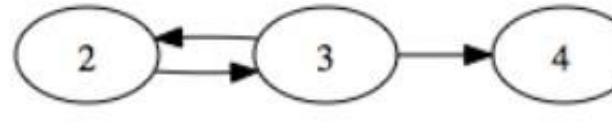
**Transitive** : iff  $(a,c) \in R$  whenever  $(a,b) \in R$  and  $(b,c) \in R$ , for all  $a, b, c \in R$ .

$R = \{(1, 2), (2, 1), (1, 1)\}$

# Automata Formal Languages & Logic

## Properties of a Relation

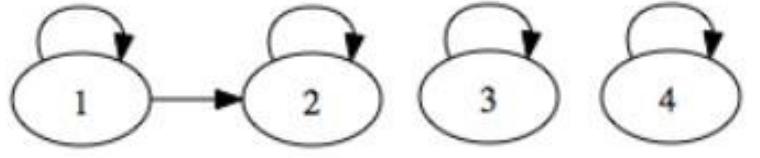
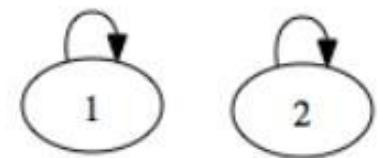
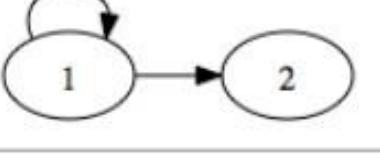
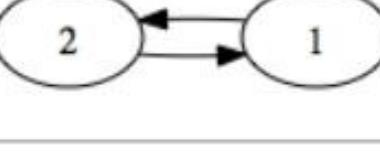
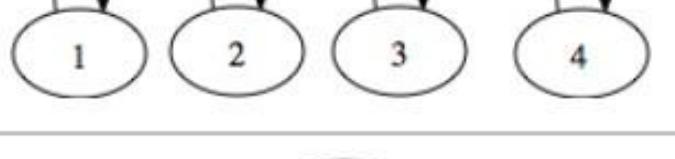
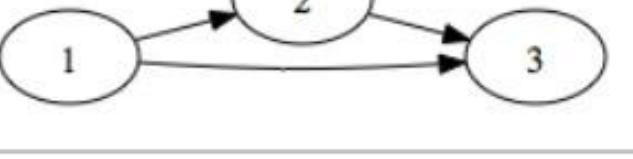
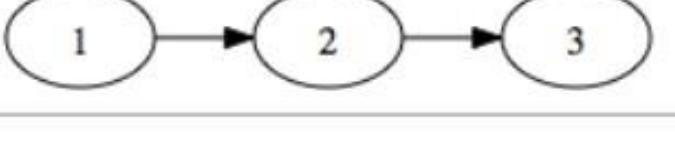
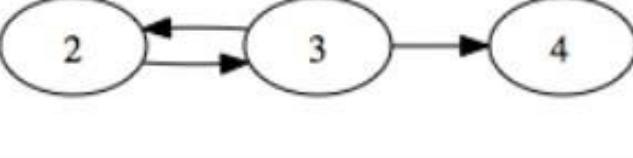
### Practice

| Relation   | Reflexive | Symmetric | Transitive |
|--|-----------|-----------|------------|
|  <pre> graph LR     1((1)) --&gt; 2((2))     2((2)) --&gt; 3((3))     1((1)) --- 1loop(( ))     </pre>                              |           |           |            |
|  <pre> graph LR     1((1)) --- 1loop(( ))     2((2)) --- 2loop(( ))     </pre>  |           |           |            |
|  <pre> graph LR     1((1)) --&gt; 2((2))     </pre>   |           |           |            |
|  <pre> graph LR     2((2)) --&gt; 1((1))     </pre>   |           |           |            |
|  <pre> graph LR     1((1)) --- 1loop(( ))     2((2)) --- 2loop(( ))     3((3)) --- 3loop(( ))     4((4)) --- 4loop(( ))     </pre> |           |           |            |
|  <pre> graph LR     1((1)) --&gt; 2((2))     2((2)) --&gt; 3((3))     </pre>   |           |           |            |
|  <pre> graph LR     1((1)) --&gt; 2((2))     2((2)) --&gt; 3((3))     3((3)) --&gt; 1((1))     </pre>                              |           |           |            |
|  <pre> graph LR     2((2)) --&gt; 3((3))     3((3)) --&gt; 4((4))     4((4)) --&gt; 2((2))     </pre>                              |           |           |            |

# Automata Formal Languages & Logic

## Properties of a Relation

### Practice

| Relation   | Reflexive | Symmetric | Transitive |
|--|-----------|-----------|------------|
|   | Y         | N         | Y          |
|   | Y         | Y         | Y          |
|   | N         | N         | Y          |
|   | N         | Y         | N          |
|  | Y         | Y         | Y          |
|  | N         | N         | Y          |
|  | N         | N         | N          |
|  | N         | N         | N          |

# Automata Formal Languages & Logic

## Equivalence Relations

Denoted by  $x \equiv y$

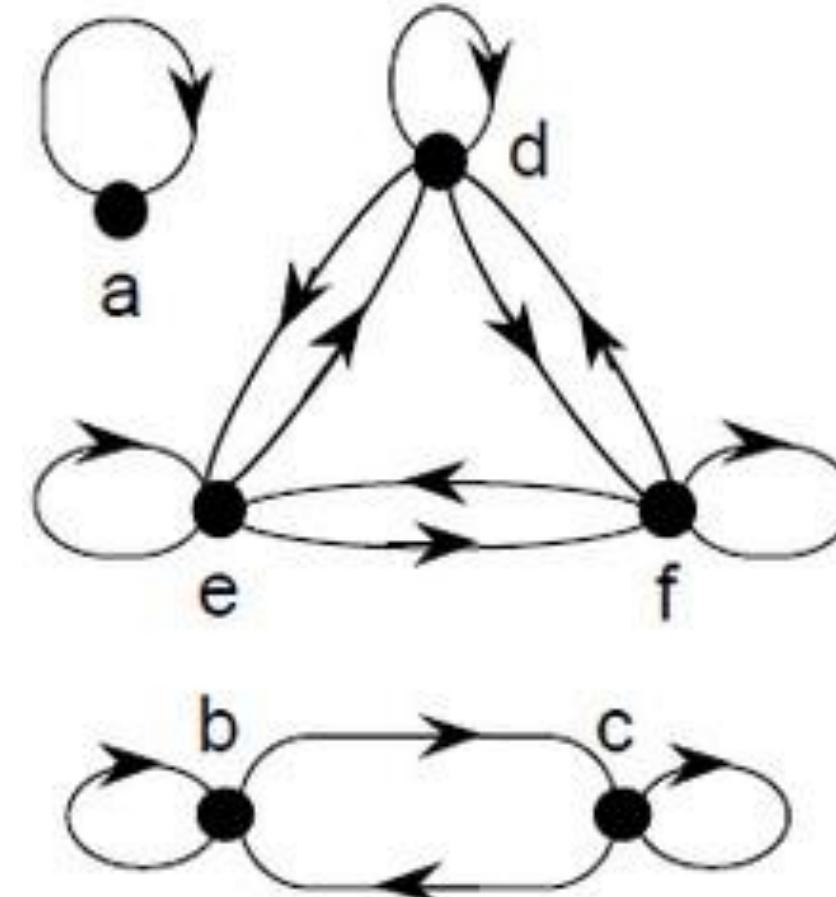
A relation  $R$  on a set  $A$  is called an equivalence relation if it is

- reflexive,
- symmetric and
- transitive.

Example: Let  $S = \{a, b, c, d, e, f\}$ .

Equivalence Relation  $R$  on set  $S$  be

$\{ (a,a), (b,b), (b,c), (c,b), (c,c),$   
 $(d,d), (d,e), (d,f), (e,d), (e,e), (e,f), (f,d),$   
 $(f,e), (f,f) \}$



|   | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| a | 1 |   |   |   |   |   |
| b |   | 1 | 1 |   |   |   |
| c |   |   | 1 | 1 |   |   |
| d |   |   |   | 1 | 1 | 1 |
| e |   |   |   | 1 | 1 | 1 |
| f |   |   |   | 1 | 1 | 1 |

# Automata Formal Languages & Logic

## Equivalence Classes

Let R be an equivalence relation on a set A.

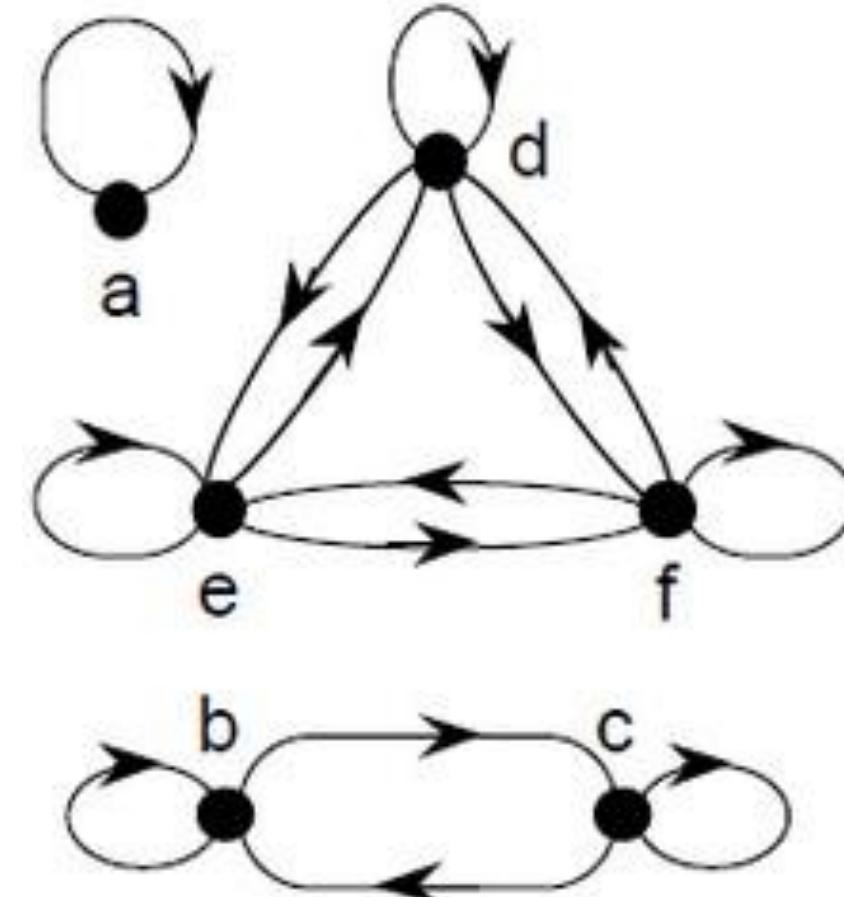
The set of all the elements that are related to an element 'a' of A is called equivalence class of 'a'.

Denoted by  $[a]_R$  or  $[a]$  when the relation is implicit.

Example :

$$[b] = \{b, c\}$$

$$[d] = \{d, e, f\}$$



|   | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| a | 1 |   |   |   |   |   |
| b |   | 1 | 1 |   |   |   |
| c |   |   | 1 | 1 |   |   |
| d |   |   |   | 1 | 1 | 1 |
| e |   |   |   | 1 | 1 | 1 |
| f |   |   |   | 1 | 1 | 1 |



# THANK YOU

---

Preet Kanwal

Department of Computer Science & Engineering

[preetkanwal@pes.edu](mailto:preetkanwal@pes.edu)



**PES**  
UNIVERSITY

CELEBRATING 50 YEARS

# Automata Formal Languages & Logic

## Unit 1 - Basic Notations

---

**Prakash C O**  
Department of Science & Engineering

- Alphabet ( $\Sigma$ )
- String or word (w)
- Empty String [ $\epsilon$  (“epsilon”) or  $\lambda$  (“lambda”)]
- Length of a String ( |w| )
- Powers of an Alphabet ( $\Sigma^k$ )
- Kleene Closure or Kleene Star or Kleene operator(\*)
- Kleene Plus (+)
- Zero or one occurrence (?)
- Language ( L )

# Automata Formal Languages & Logic

## Unit 1 - Basic Notations

---

### Alphabet ( $\Sigma$ ):

- An alphabet is a finite, non-empty set of symbols.
- We use the symbol  $\Sigma$  (sigma) to denote an alphabet.
- Examples:
  - Binary:  $\Sigma = \{0,1\}$
  - All lower case letters:  $\Sigma = \{a,b,c,\dots,z\}$
  - Alphanumeric:  $\Sigma = \{a-z, A-Z, 0-9\}$
  - DNA molecule letters:  $\Sigma = \{a,c,g,t\}$

...

# Automata Formal Languages & Logic

## Unit 1 - Basic Notations

---

### String or word ( $w$ ):

- A string or word is a finite sequence of symbols chosen from  $\Sigma$
- Empty string is  $\epsilon$  (“epsilon”) or  $\lambda$  (“lambda”)
- Length of a string  $w$ , denoted by “ $|w|$ ”, is equal to the number of (non-  $\epsilon$ ) characters in the string

E.g., if  $w = 010100$  then  $|w| = 6$

if  $w = 01\epsilon 0\epsilon 1\epsilon 00\epsilon$  then  $|w| = ?$

if  $w = \epsilon$  then  $|w| = ?$

- $xy$  = concatenation of two strings  $x$  and  $y$

# Automata Formal Languages & Logic

## Unit 1 - Basic Notations

---



### Powers of an alphabet:

- Let  $\Sigma$  be an alphabet.
- $\Sigma^k$  = the set of all strings of length  $k$

**Example:** if  $\Sigma = \{a, b\}$  then,

$\Sigma^0 = \{\lambda\}$  or  $\{\varepsilon\}$       i.e., the set of all strings of length 0

$\Sigma^1 = \{a, b\}$       i.e., the set of all strings of length 1

$\Sigma^2 = \{aa, bb, ab, ba\}$       i.e., the set of all strings of length 2

.....

# Automata Formal Languages & Logic

## Unit 1 - Basic Notations

---

**Kleene star (or Kleene closure):** Kleene star is a unary operation, either on sets of strings or on sets of symbols or characters.

The application of the Kleene star to an alphabet  $\Sigma$  is written as  $\Sigma^*$

If  $\Sigma$  is an alphabet, then  $\Sigma^*$  is the set of all strings over symbols in  $\Sigma$ , including the empty string  $\epsilon$ .

The definition of Kleene star on  $\Sigma$  is

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$$

**Kleene plus:** The Kleene plus omits the  $\Sigma^0$  term in the above union. In other words, the Kleene plus on  $\Sigma$  is

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots \quad \text{i.e., the set of strings of length } > 0$$

# Automata Formal Languages & Logic

## Unit 1 - Basic Notations

---

**Language(L):** L is said to be a language over alphabet  $\Sigma$ , only if  $L \subseteq \Sigma^*$

⇒ this is because  $\Sigma^*$  is the set of all strings (of all possible length including 0) over the given alphabet  $\Sigma$

### Examples:

Let L be *the* language of all strings consisting of  $n$  0's followed by  $n$  1's (i.e.,  $0^n 1^n$ ):

$$L = \{\epsilon, 01, 0011, 000111, \dots\}$$

Let L be *the* language of all strings of with equal number of 0's and 1's:

$$L = \{\epsilon, 01, 10, 0011, 1100, 0101, 1010, 1001, \dots\}$$

**Definition:**  $\emptyset$  denotes the Empty language, Let  $L = \{\epsilon\}$ ; Is  $L = \emptyset$ ?

# Automata Formal Languages & Logic

## Unit 1 - Basic Notations

---



### Finite or Infinite Languages:

Finite Language has a finite set of words/strings.

$L = \text{set of string over } \Sigma = \{0, 1\} \text{ such that } |w| = 2$

$L = \{00, 01, 10, 11\}$

Infinite Language has a infinite set of words/strings.

$L = \text{set of string over } \Sigma = \{0, 1\} \text{ such that } |w| > 2$

$L = \{000, 101, 010, 011, 100, 110, 001, 111, 1100, \dots\}$

# Automata Formal Languages & Logic

## Unit 1 - Basic Notations

---



### The Membership Problem

Given a string  $w \in \Sigma^*$  and a language  $L$  over  $\Sigma$ , decide whether  $w \in L$  or not.

Example:

Let  $w = 100011$  and  $\Sigma = \{0,1\}$

Q) Is  $w \in$  the language of strings with equal number of 0s and 1s?



**PES**  
**UNIVERSITY**

CELEBRATING 50 YEARS

**THANK YOU**

---

**Prakash C O**

Department of Computer Science and Engineering

**coprakash@pes.edu**

**+91 98 8059 1946**



# Automata Formal Languages & Logic

## Unit 1 – DFA

---

**Preet Kanwal and Prakash C O**

Department of Computer Science & Engineering

# Automata Formal Languages & Logic

---

## Unit 1: Deterministic Finite Automata (DFA)

Preet Kanwal and Prakash C O

Department of Computer Science & Engineering

### Formal Definition of a DFA

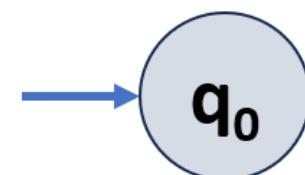
A DFA can be represented by a 5-tuple  $M=(Q, \Sigma, \delta, q_o, F)$  where

- **$Q$  is a finite set of states.**
- **$\Sigma$  is a finite non-empty set of input symbols called the alphabet.**
- **$\delta$  is the transition function where  $\delta: Q \times \Sigma \rightarrow Q$**
- **$q_o$  is the initial state from where any input is processed ( $q_o \in Q$ ).**
- **$F$  is a set of final state/states ( $F \subseteq Q$ ).**

Machine M can be defined in 3 different ways:

1. Transition Diagram or State Transition Diagram or State Diagram
2. Transition Table
3. Transition Function

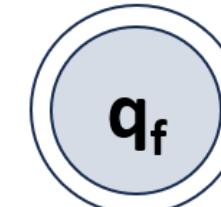
- Each finite automaton consists of a set of *states* connected by *transitions*.
- Symbols used in Constructing a Finite Automata as Transition Diagram:



**Start state**



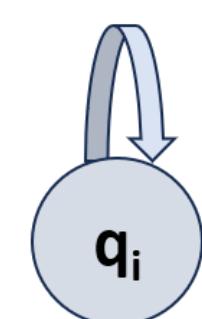
**State i**



**Final (or Accepting) state**



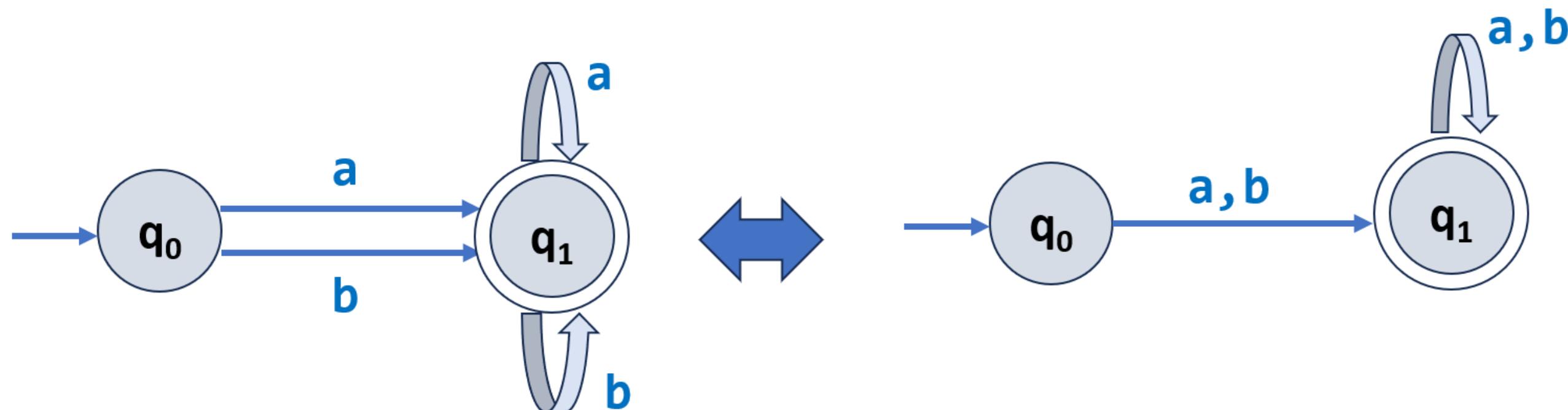
**Transition:**  $\delta(q_i, a) = q_j$



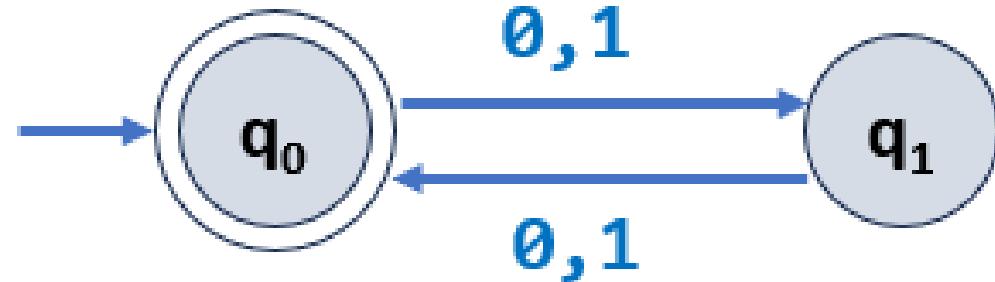
**Self loop**

**Transition:**  $\delta(q_i, a) = q_i$

- The below Transition Diagrams are equivalent (or same):



DFA:



The automaton in the above figure can be represented as  $(Q, \Sigma, \delta, q_o, F)$  where

- $Q = \{q_0, q_1\}$ ,
- $\Sigma = \{0, 1\}$ ,
- $q_o = q_0$ ,
- $F = \{q_0\}$ , and
- Transition function  $\delta$  is such that

$$\delta(q_0, 0) = q_1, \delta(q_0, 1) = q_1, \text{ and } \delta(q_1, 0) = q_0, \delta(q_1, 1) = q_0.$$

Let's draw the state diagram of the following automaton  $(Q, \Sigma, \delta, q_o, F)$

$$Q = \{q_1, q_2, q_3\}$$

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

$\delta$  is given in a tabular form below (i.e., **transition table**):

| $\delta$          | 0               | 1 |
|-------------------|-----------------|---|
| $\rightarrow q_1$ | $q_1 \quad q_2$ |   |
| $q_2^*$           | $q_3 \quad q_2$ |   |
| $q_3$             | $q_2 \quad q_2$ |   |

$q_1$  is the initial state, and  $F = \{ q_2 \}$ .

**What does it accept?**

- In automata theory, a finite-state machine is called a deterministic finite automaton (DFA),
  1. If each of its transitions is uniquely determined by its source state and input symbol, and
  2. For every state, a transition is required for all the input symbols in  $\Sigma$ .

### The Transition Function of a DFA is $\delta: Q \times \Sigma \rightarrow Q$

- In DFA, for each state there must be exactly one transition defined for each symbol in  $\Sigma$ . This is the “deterministic” part of DFA.  
**or**
- In DFA, the transition for every state and input symbol is uniquely defined, there is no ambiguity or choice in the transitions.  
**or**
- DFA is a finite-state machine that accepts or rejects a given string of symbols, by running through a state sequence uniquely determined by the string. *Deterministic* refers to the uniqueness of the computation run.

**The Word (or string) acceptance by FA / FSA / FSM :**

- A finite automaton does not accept as soon as it enters an accepting state.
- A finite automaton accepts if its run ends in an accepting state.

### The Language acceptance by FA / FSA / FSM :

- The language accepted by a DFA  $M = (Q, \Sigma, \delta, q_0, F)$  is the set of all strings on  $\Sigma$  accepted by  $M$ .
  
- So, for an automaton to accept a particular language (e.g.,  $\{a^n b^m \mid n, m \geq 1\}$ ), it must not only accept every string in the language, but also reject every string not in the language.

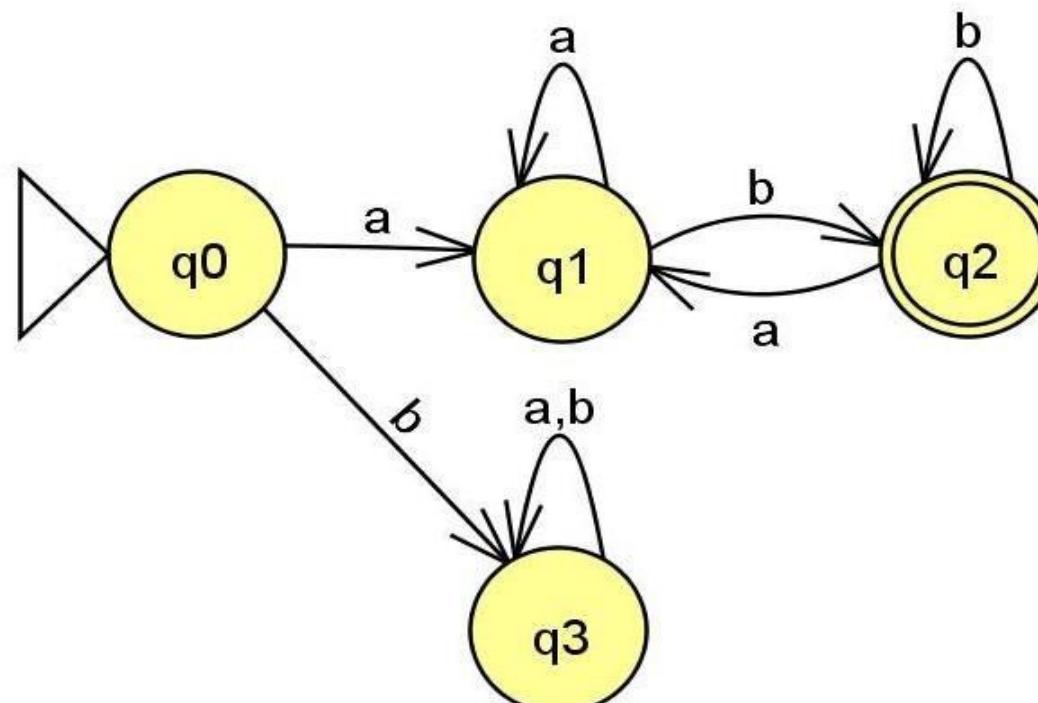
### Computation or run of a DFA

- For an input string, DFA will create an unique ‘computation sequence’.
- A DFA accepts a string if run (or computation) leads to some accept state.

The Language of a DFA  $M$  is  $L(M) = \{w \mid \text{run of } M \text{ on } w \text{ is accepting}\}$ .

### Computation or run of a DFA

- Example: Given  $w = \text{aababb}$  and Machine  $M$



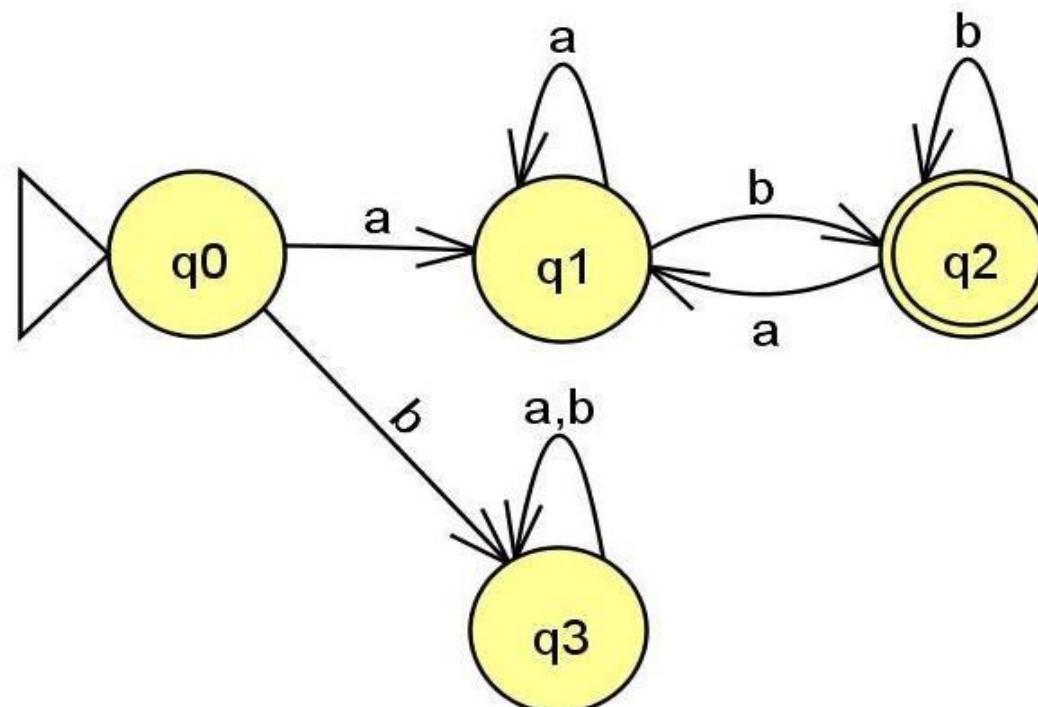
A run of  $M$  on  $w$  is

$$q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{b} q_2$$

Accepted run/Computation because  $q_2 \in F$

### Computation or run of a DFA

- Example: Given  $w = aabbba$  and Machine  $M$



A run of  $M$  on  $w$  is

$$q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{b} q_2 \xrightarrow{b} q_2 \xrightarrow{a} q_1$$

Rejected run/Computation because  $q_1 \notin F$

### Approach to Construct a DFA that recognises a language L:

- **Step-1:** Try to enumerate all the Strings in the language i.e.,
  - \* Specify the minimal string and then the next minimal strings, ...
  - \* Enumerate the strings in the order of increasing length
  - \* If possible discover a pattern in the enumerated strings
- **Step-2:** Draw a DFA skeleton of the automata based on the minimal string or the pattern discovered.
- **Step-3:** Complete the DFA

### Example 1:

Construct DFA for the Language of strings of length 2 , over  $\Sigma = \{a,b\}$  .

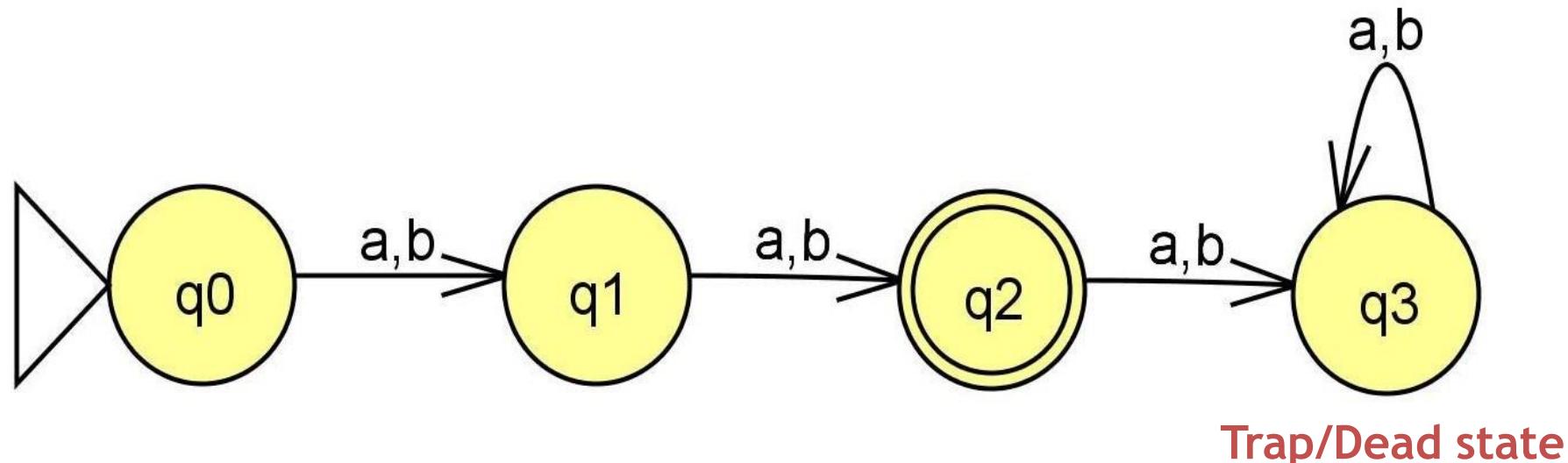
### Solution :

Note Transition diagram is a directed graph.

### Example 1:

Construct DFA for the Language of strings of length 2 , over  $\Sigma = \{a,b\}$  .

Solution :  $L = \{aa, bb, ab, ba\}$



Note1: strings of length 1 are rejected in  $q_1$  and strings of length>2 are rejected in  $q_3$

Note2: For every state, transition is required for all the input symbols in  $\Sigma$

# Automata Formal Languages and Logic

## Unit 1 - Deterministic Finite Acceptor/Automata

---



**Example 1a:**

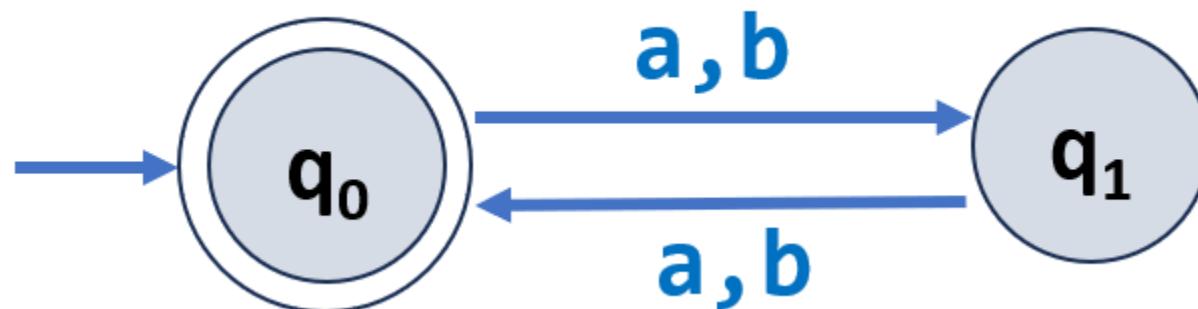
**Construct DFA for the Language of strings of even length, over  $\Sigma = \{a, b\}$ .**

**Solution :**

### Example 1a:

Construct DFA for the Language of strings of even length, over  $\Sigma = \{a, b\}$ .

Solution :  $L = \{\epsilon, aa, bb, ab, ba, aaaa, bbbb, abbb, \dots\}$



### Example 1b:

Construct DFA for the Language of strings having even number of a's, over  $\Sigma = \{a, b\}$ .

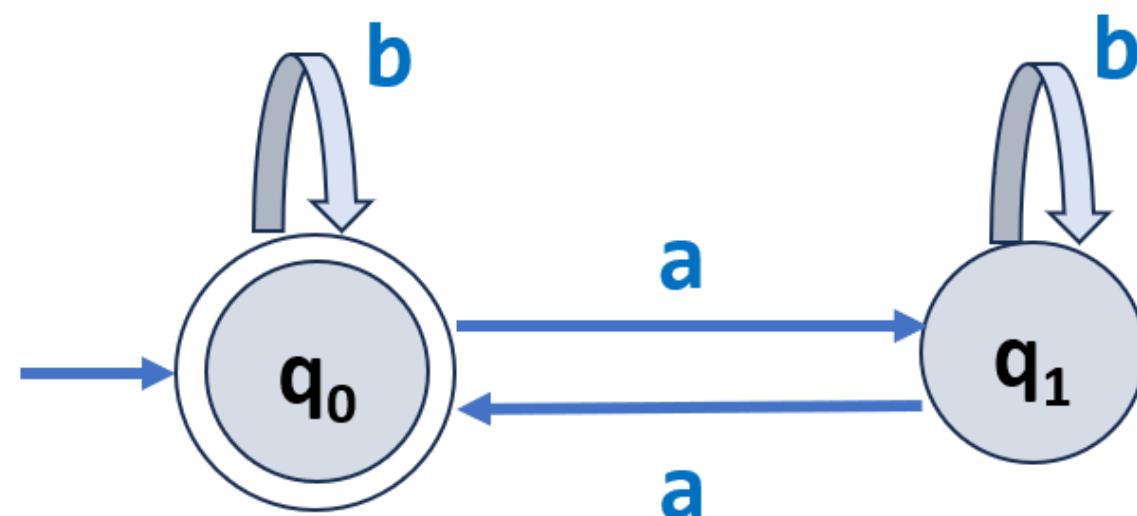
Solution :

### Example 1b:

Construct DFA for the Language of strings having even number of a's, over  $\Sigma = \{a, b\}$ .

Solution :

$L = \{\epsilon, aa, aaaa, aaaaaaa, aaaaaaaaa, \dots, b, bb, bbb, bbbb, bbbbb, \dots, aab, baa, aba, \dots \dots \}$



# Automata Formal Languages and Logic

## Unit 1 - Deterministic Finite Acceptor/Automata

---



### Exercise:

Construct DFA for the Language of strings having odd number of a's, over  $\Sigma = \{a, b\}$ .

### Solution :

**Example 2:**

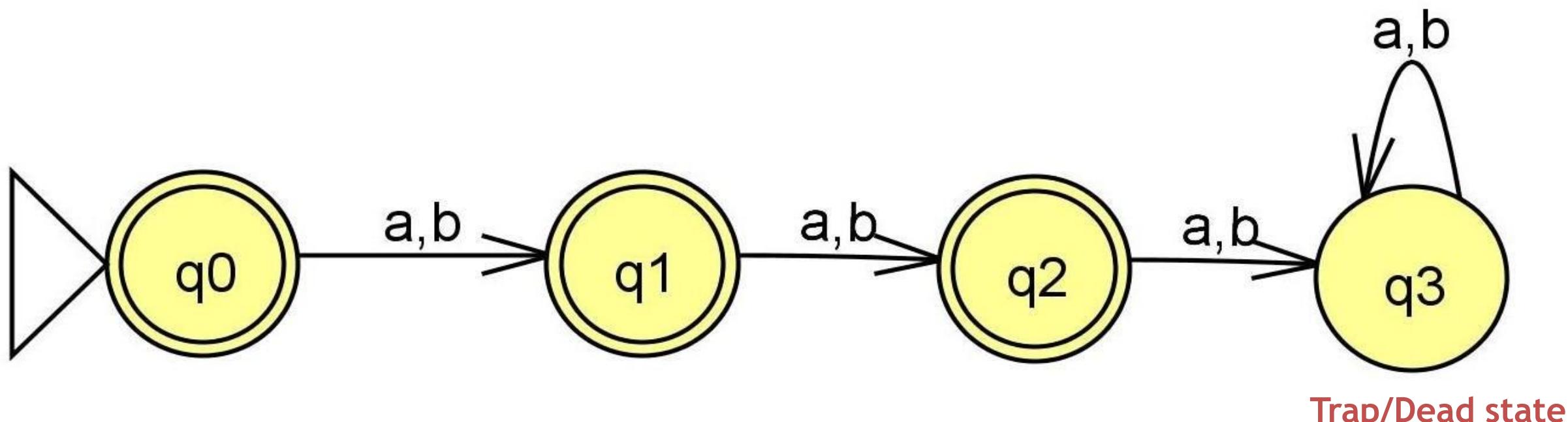
**Construct DFA for the language  $L = \{ w \mid w \in \{a, b\}^* \text{ and } |w| \leq 2 \}$**

**Solution :**

### Example 2:

Construct DFA for the language  $L = \{ w \mid w \in \{a, b\}^* \text{ and } |w| \leq 2 \}$

Solution :  $L = \{\epsilon, a, b, aa, bb, ab, ba\}$



**Trap state** is one from which we cannot escape, that is, it has either no transitions defined or transitions for all symbols goes to itself.

# Automata Formal Languages and Logic

## Unit 1 - Deterministic Finite Acceptor/Automata

---



**Example 3:**

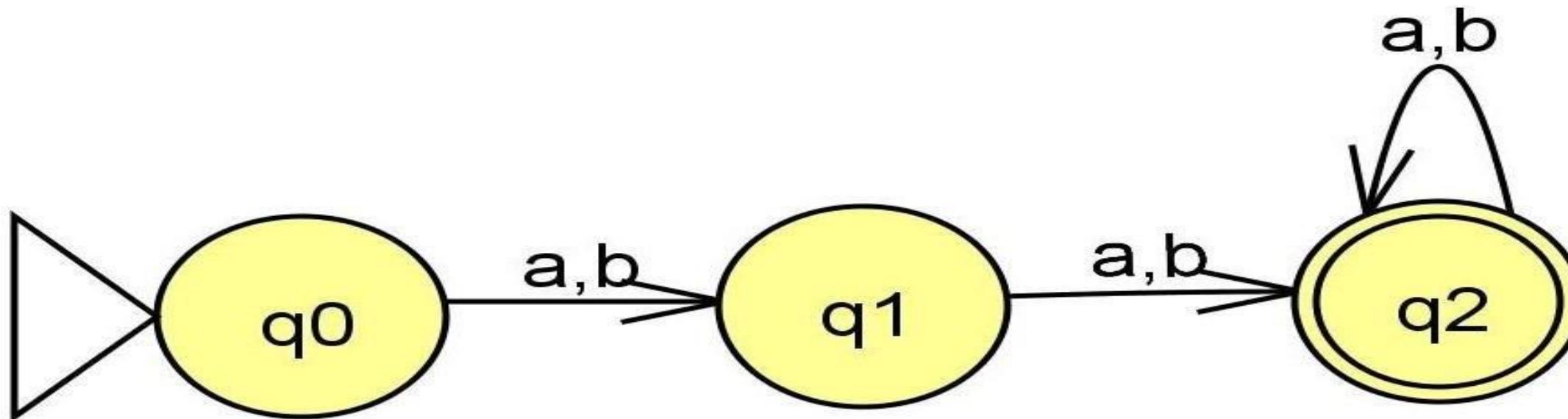
**Construct DFA for the language  $L = \{ w \mid w \in \{a, b\}^* \text{ and } |w| \geq 2 \}$**

**Solution :**

### Example 3:

Construct DFA for the language  $L = \{ w \mid w \in \{a, b\}^* \text{ and } |w| \geq 2 \}$

Solution :  $L = \{ aa, bb, ab, ba, aaa, aba, aab, \dots \}$



**Example 4:**

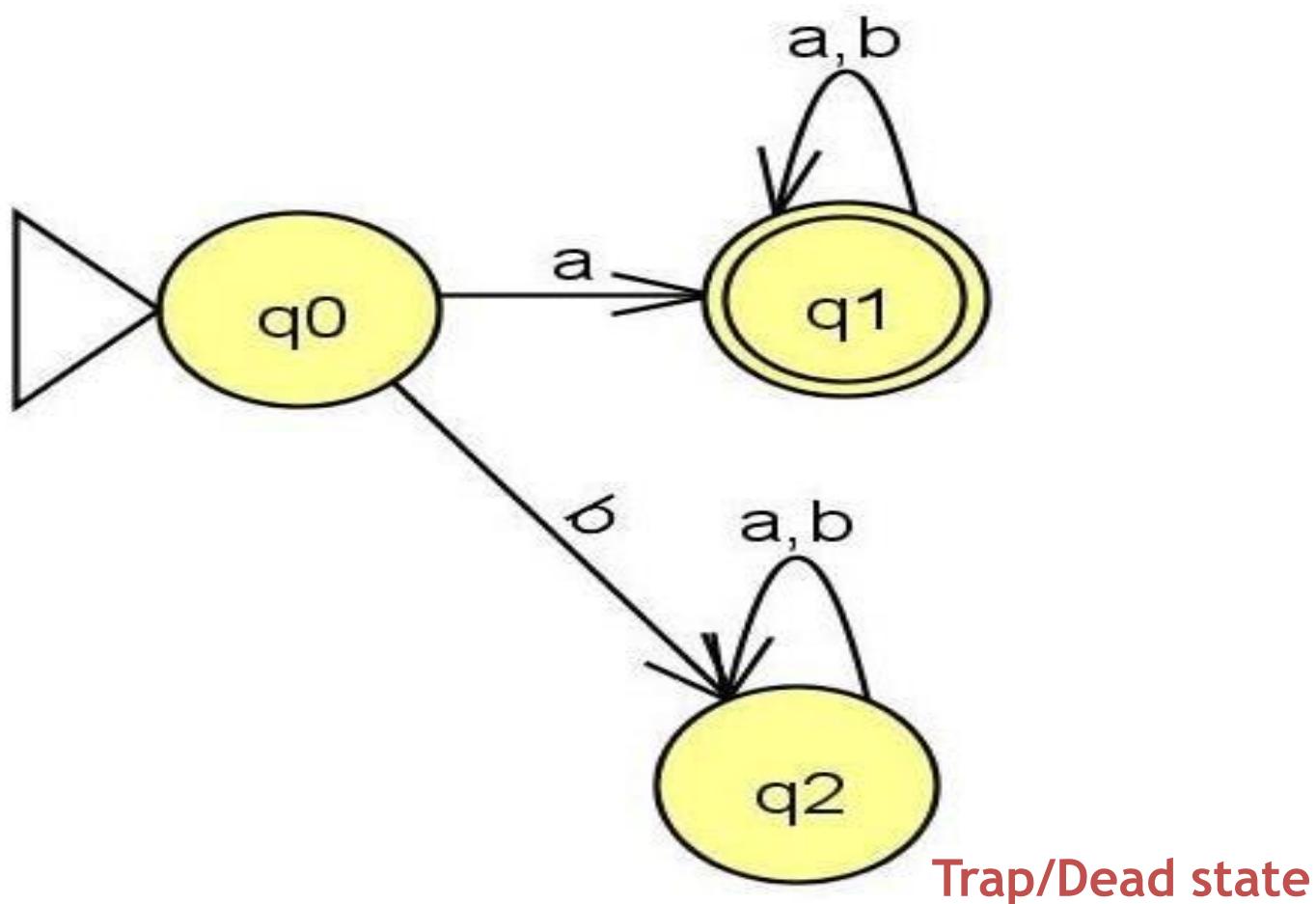
**Construct DFA for the language of strings starts with a, over  $\Sigma=\{a,b\}$ .**

**Solution :**

### Example 4:

Construct DFA for the language of strings starts with a, over  $\Sigma = \{a, b\}$ .

Solution :  $L = \{ a, aa, aaa, \dots, ab, abb, \dots, aba, aab, \dots \}$



### Example 5:

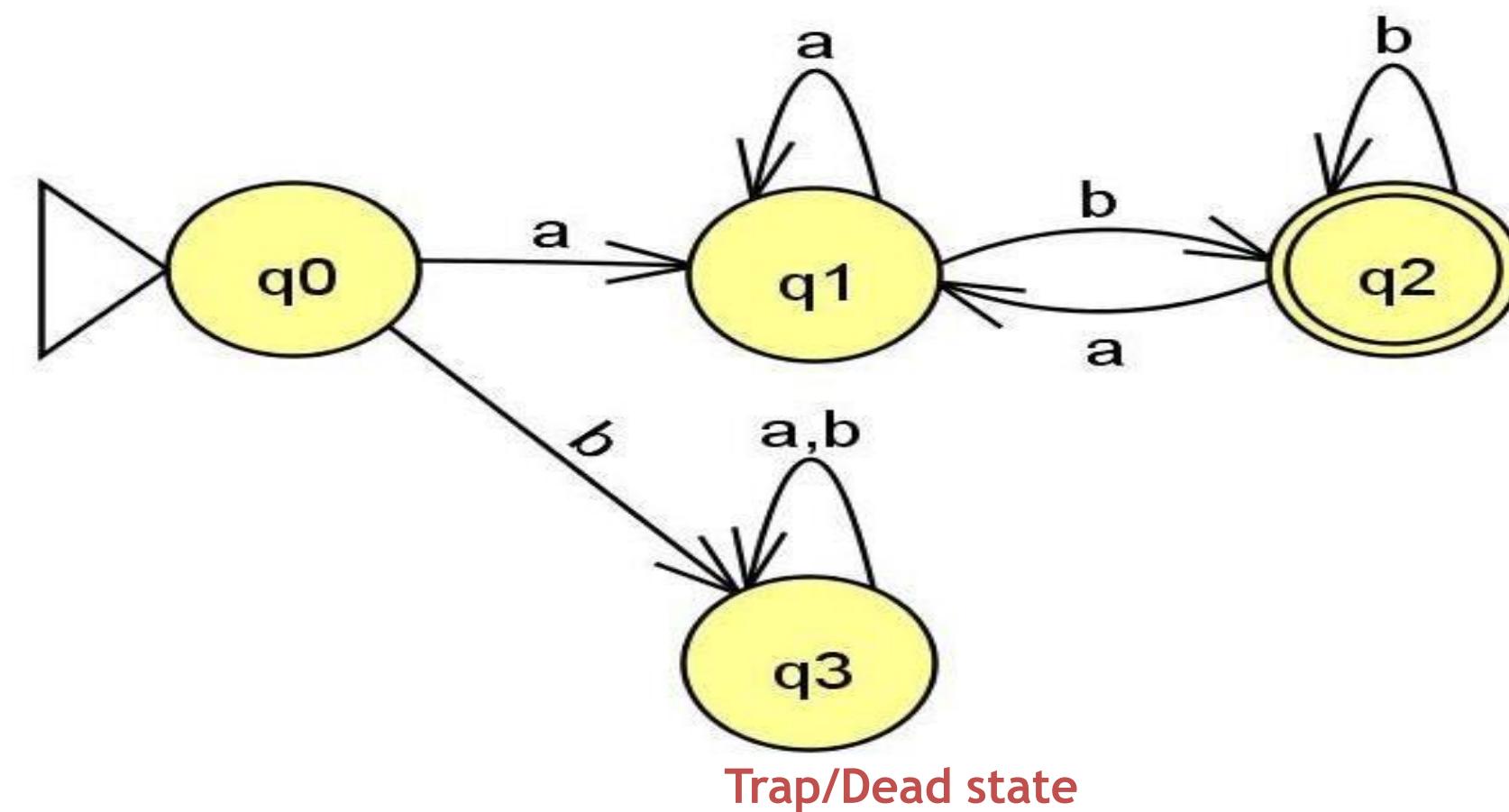
Construct DFA for the language of strings starts with a and ends with b, over  $\Sigma = \{a, b\}$ .

### Solution:

### Example 5:

Construct DFA for the language of strings starts with a and ends with b, over  $\Sigma = \{a, b\}$ .

Solution:  $L = \{ ab, aab, abb, abab, aaab, abbb, aabb, \dots \}$



### Example 6:

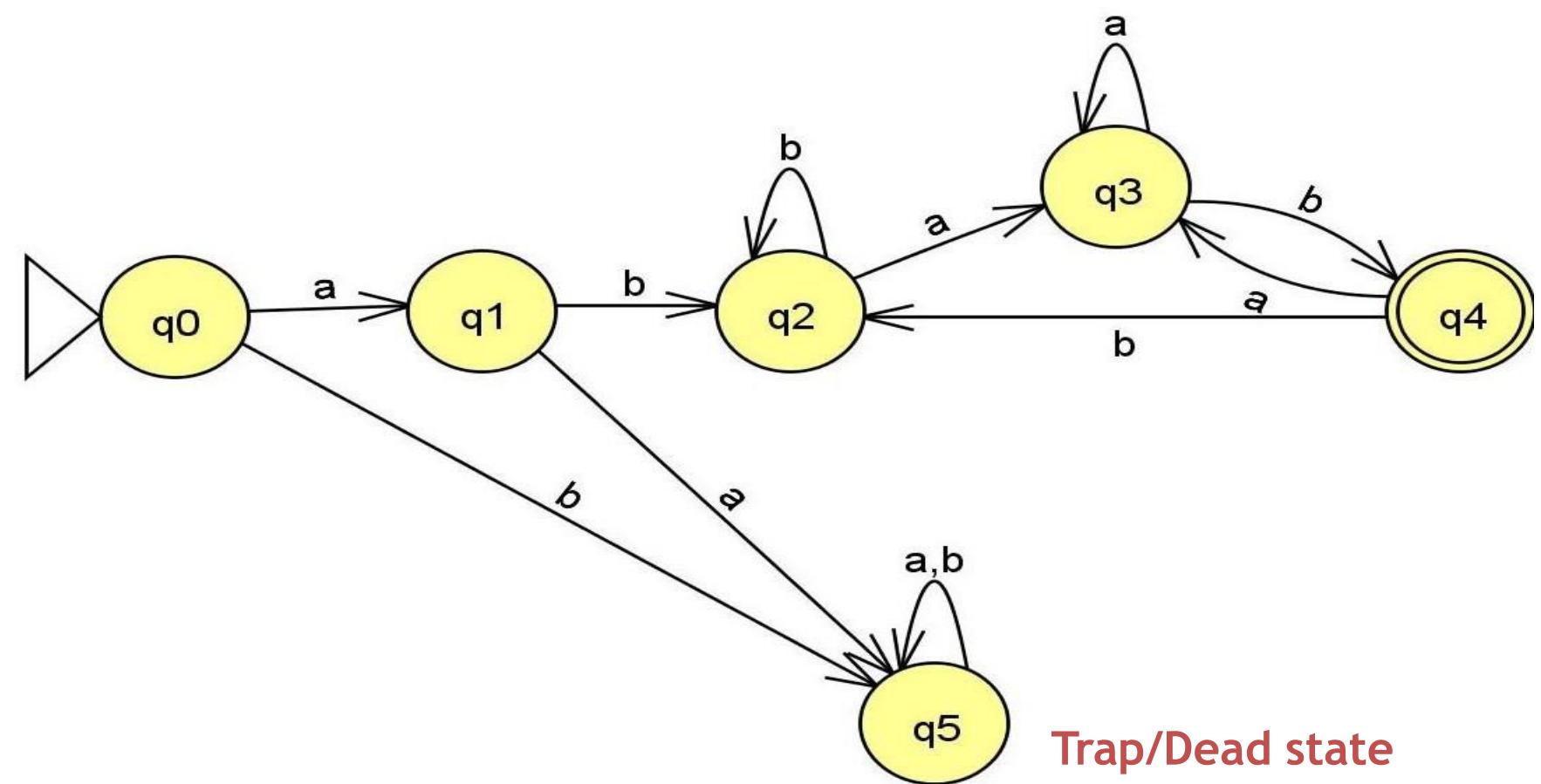
The language with strings over  $\Sigma = \{a,b\}$  where every string starts with ab and ends with ab, over  $\Sigma = \{a,b\}$ .

**Solution:**  $L = \{abab, abaab, abbab, \dots\}$  If minimal string in the language is abab then DFA is

### Example 6:

The language with strings over  $\Sigma = \{a,b\}$  where every string starts with ab and ends with ab, over  $\Sigma = \{a,b\}$ .

**Solution:**  $L=\{abab, abaab, abbab, \dots\}$  If minimal string in the language is abab then DFA is



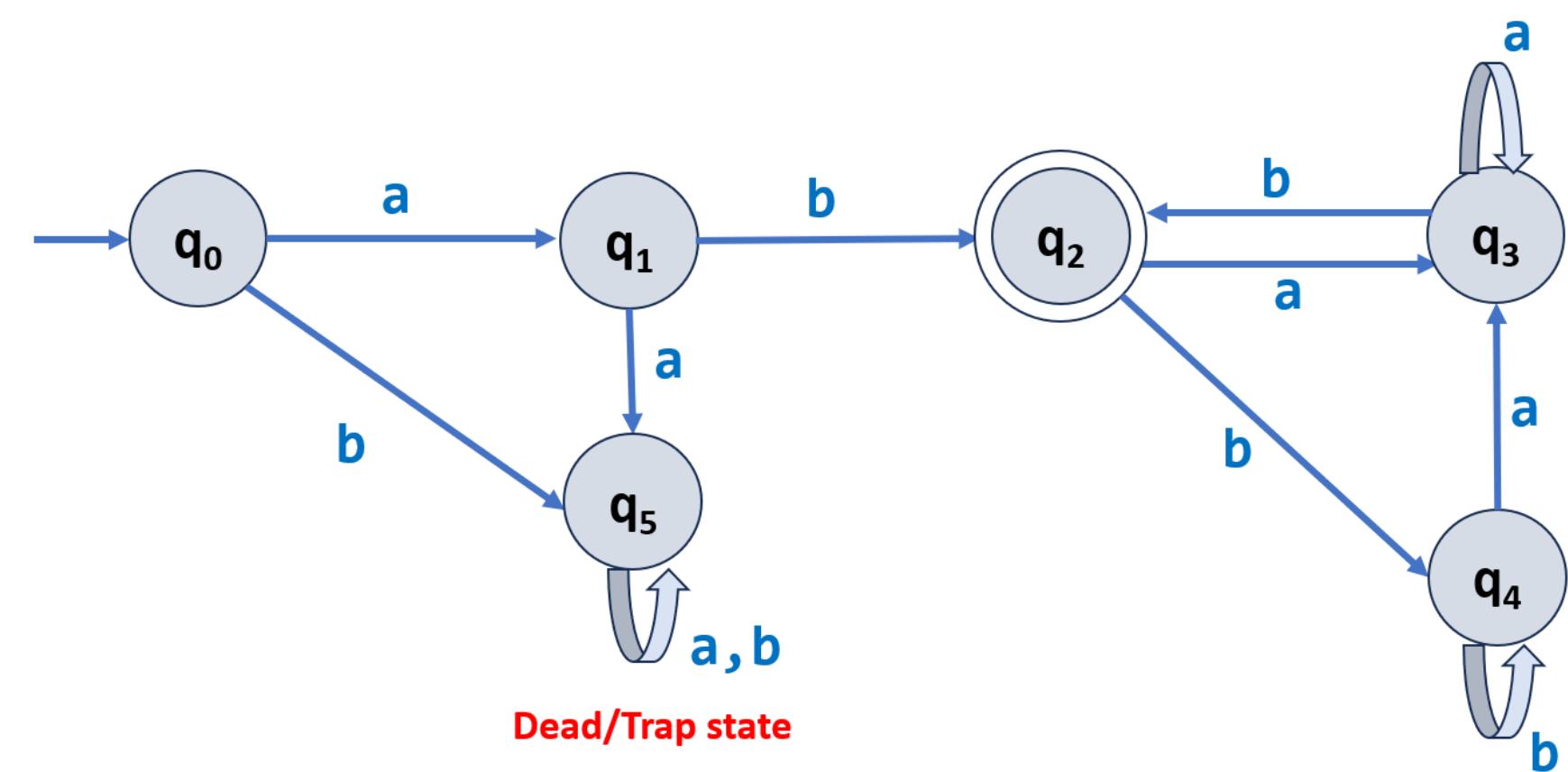
### Example 6a:

The language with strings over  $\Sigma = \{a,b\}$  where every string starts with ab and ends with ab, over  $\Sigma = \{a,b\}$ .

**Solution:**  $L=\{ab, abab, abaab, abbab, \dots\}$

If minimal string in the

language is ab then DFA is



### Example 7:

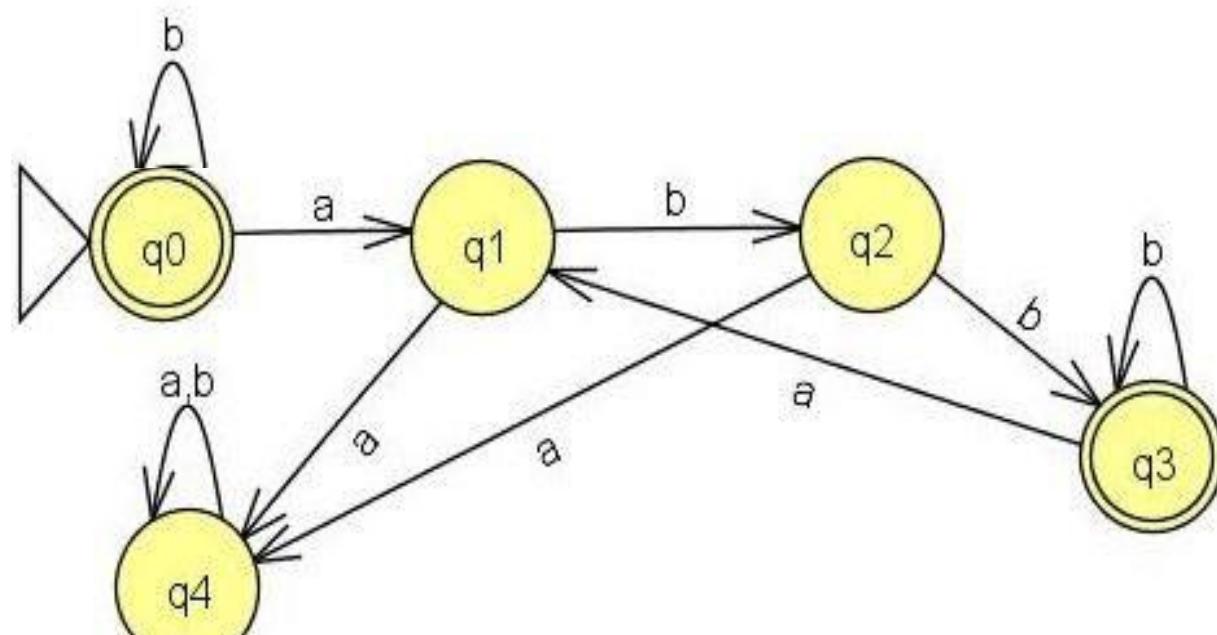
The language of strings over  $\Sigma = \{a,b\}$ , where every a in the string is followed by bb.

### Solution :

### Example 7:

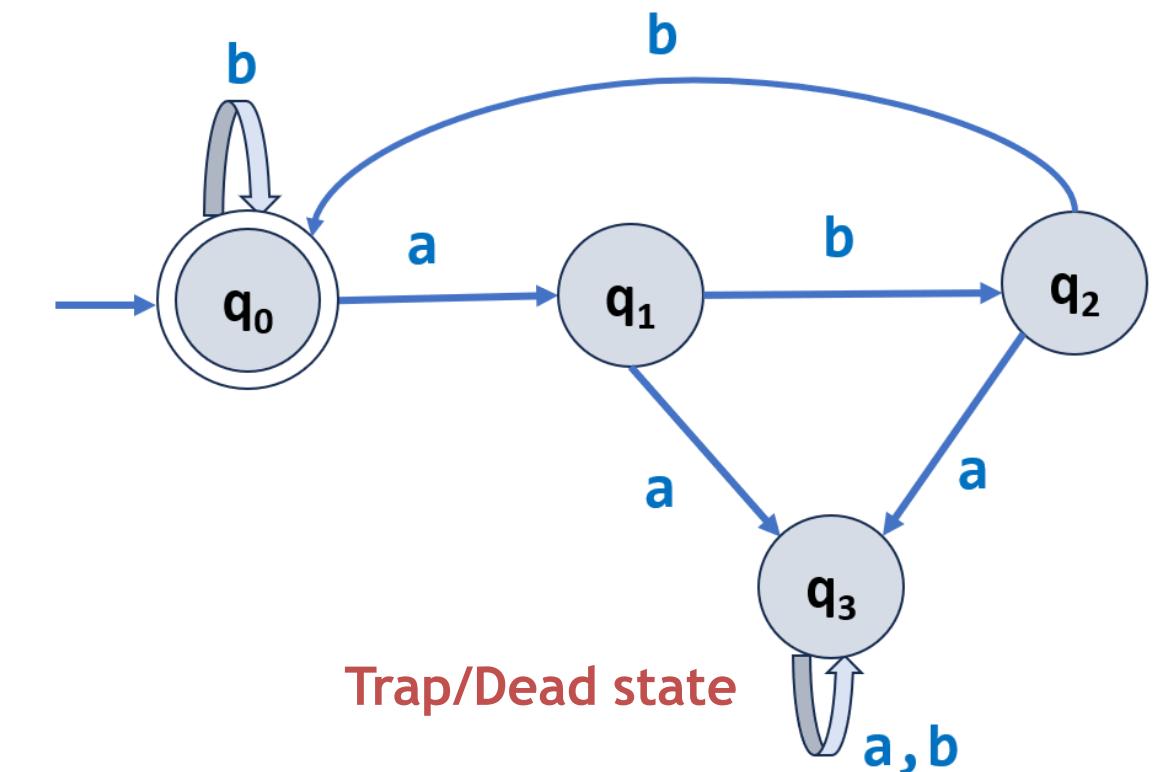
The language of strings over  $\Sigma = \{a, b\}$ , where every a in the string is followed by bb.

**Solution :**  $L = \{\epsilon, b, bb, bbb, \dots, abb, babb, bbabb, babb, \dots\}$



Trap/Dead state

or



Trap/Dead state

### Example 7a:

The language of strings over  $\Sigma = \{a,b\}$ , where every a in the string is never followed by bb.

Solution :

### Example 8:

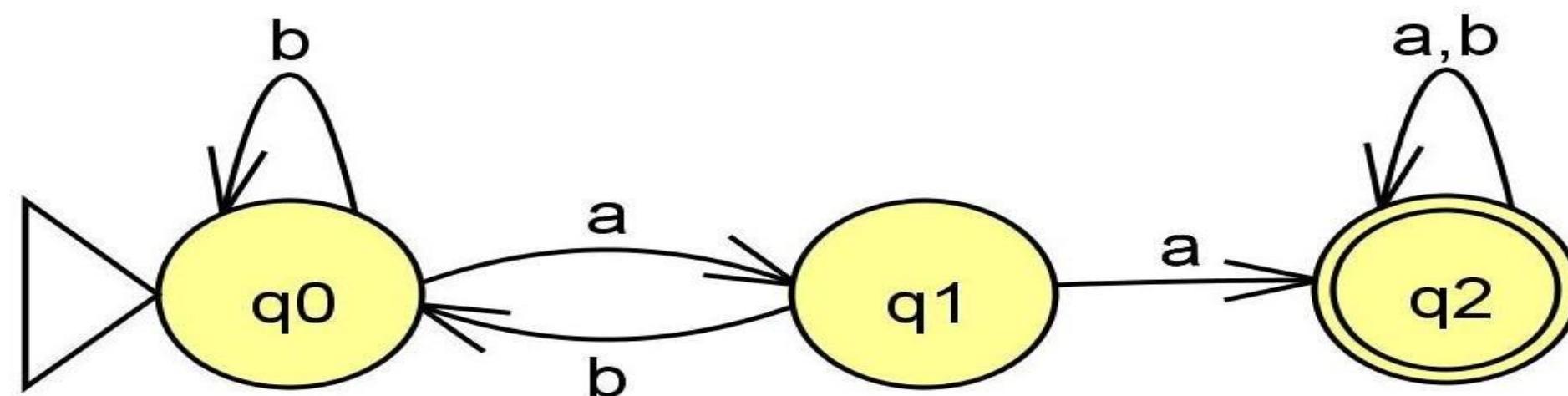
The language of strings over  $\Sigma = \{a,b\}$  where every string must contain “aa” as the substring.

### Solution :

### Example 8:

The language of strings over  $\Sigma = \{a, b\}$  where every string must contain “aa” as the substring.

Solution :  $L = \{aa, aaa, aaaa, \dots, baa, aab, aabb, bbaa, baab, \dots\}$



**Example 9:**

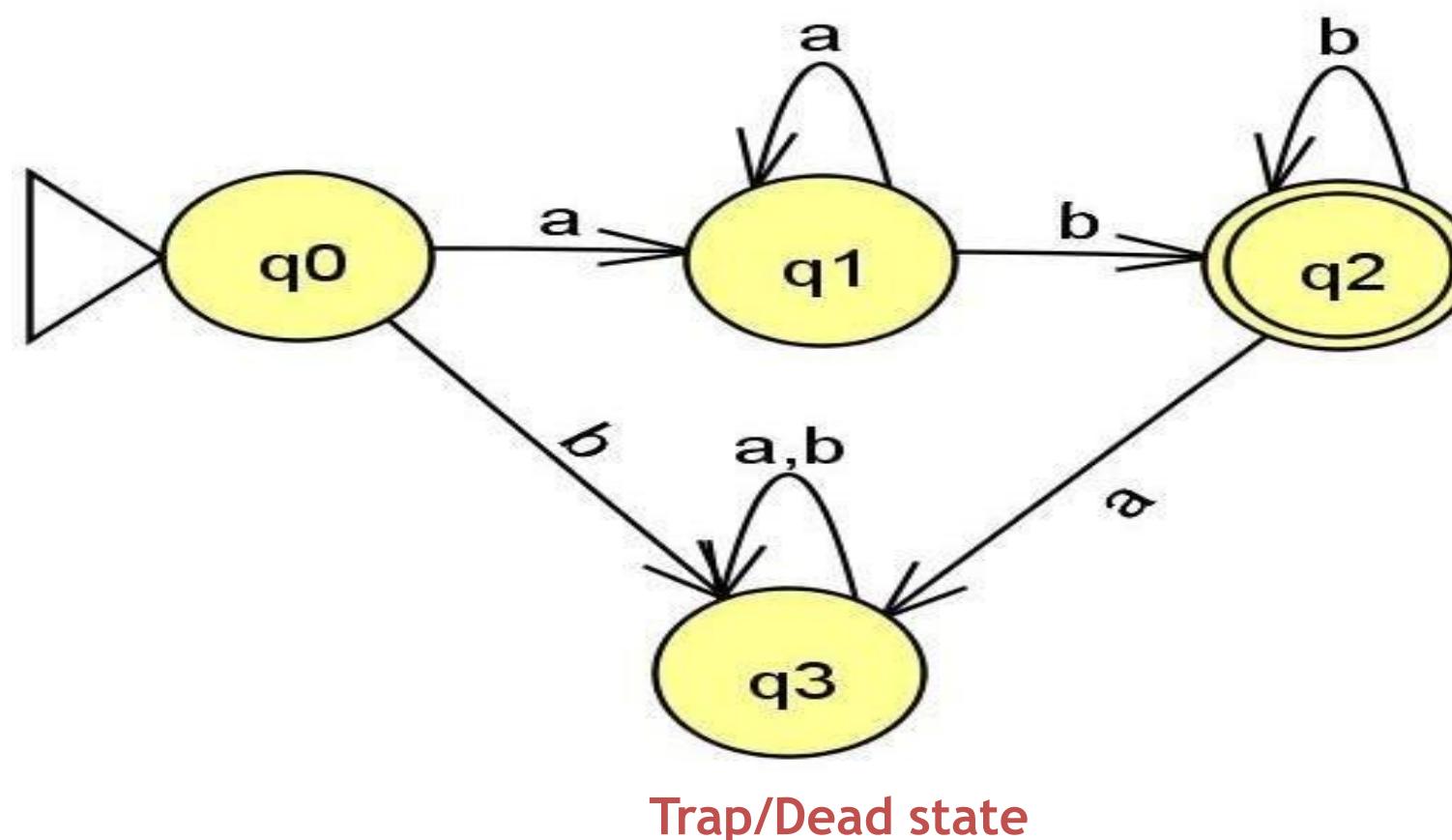
**Construct DFA for the language of strings over  $\Sigma = \{a,b\}$  of the form  $a^n b^m$  where  $n, m \geq 1$ .**

**Solution:**

### Example 9:

Construct DFA for the language of strings over  $\Sigma = \{a,b\}$  of the form  $a^n b^m$  where  $n, m \geq 1$ .

Solution :  $L = \{ ab, abb, aab, aabb, abbb, aaab, aaabbb, \dots \}$



### Example 10:

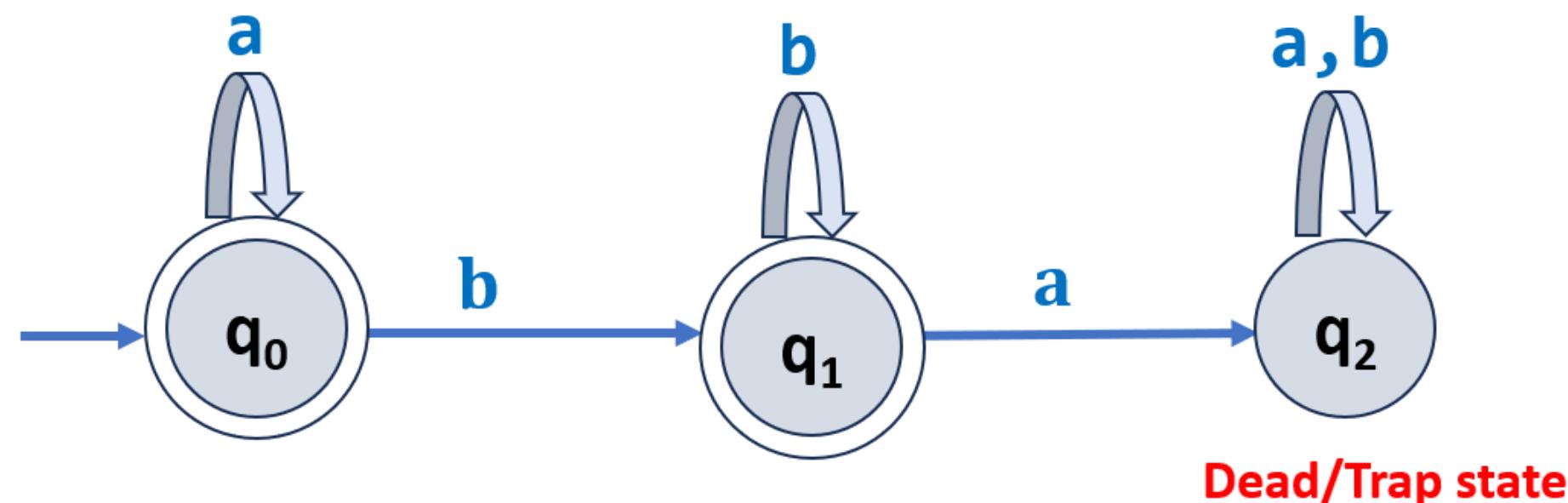
**Construct DFA for the language of strings over  $\Sigma = \{a,b\}$  of the form  $a^n b^m$  where  $n, m \geq 0$ .**

**Solution :**

### Example 10:

Construct DFA for the language of strings over  $\Sigma = \{a,b\}$  of the form  $a^n b^m$  where  $n, m \geq 0$ .

Solution :  $L = \{\epsilon, a, aa, aaa, \dots, b, bb, bbb, \dots, ab, abb, aab, aabb, abbb, aaab, aaabb, \dots\}$



**Example 11:**

Construct DFA for the language of strings over  $\Sigma = \{a, b\}$  where  $n_a(w) \bmod 2 = 0$ .

**Solution :**  $L = \{\epsilon, aa, aaaa, aaaaaaa, \dots, b, bb, bbb, \dots, aab, aba, baa, \dots\}$

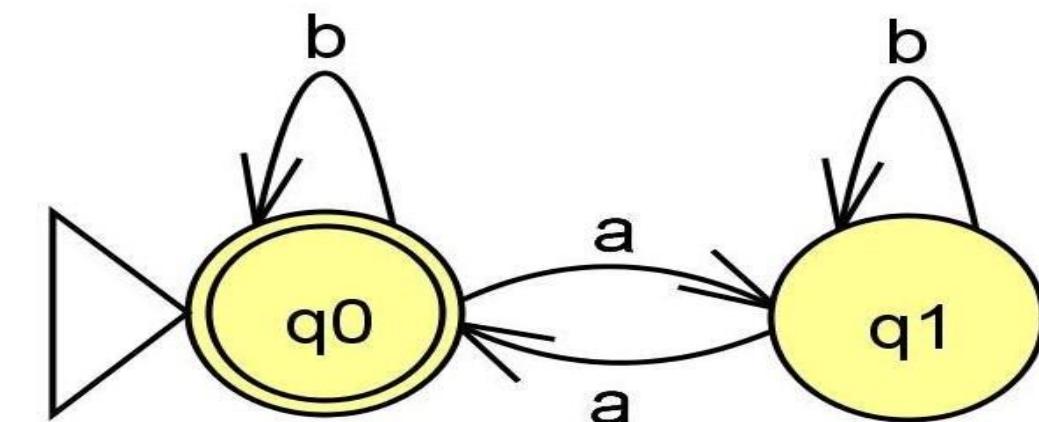
### Example 11:

Construct DFA for the language of strings over  $\Sigma = \{a, b\}$  where  $n_a(w) \bmod 2 = 0$ .

**Solution :**  $L = \{\epsilon, aa, aaaa, aaaaaa, \dots, b, bb, bbb, \dots, aab, aba, baa, \dots\}$

#### Explanation:

- In the DFA the states should represent the information about the string processed so far.
- In this example, states have to represent only the number of a's seen so far is even or odd, so only need two states  $q_0$  and  $q_1$ . One state to represent “even” and another state to represent “odd”.
- Before reading any symbol, the number of a's processed so far is 0, which is even. Hence, the initial state  $q_0$  should represent even. We want the DFA to accept the strings in  $L$  (strings with even number of a's). Hence, we should choose  $q_0$  to also represent our accepting state.



### Example 12:

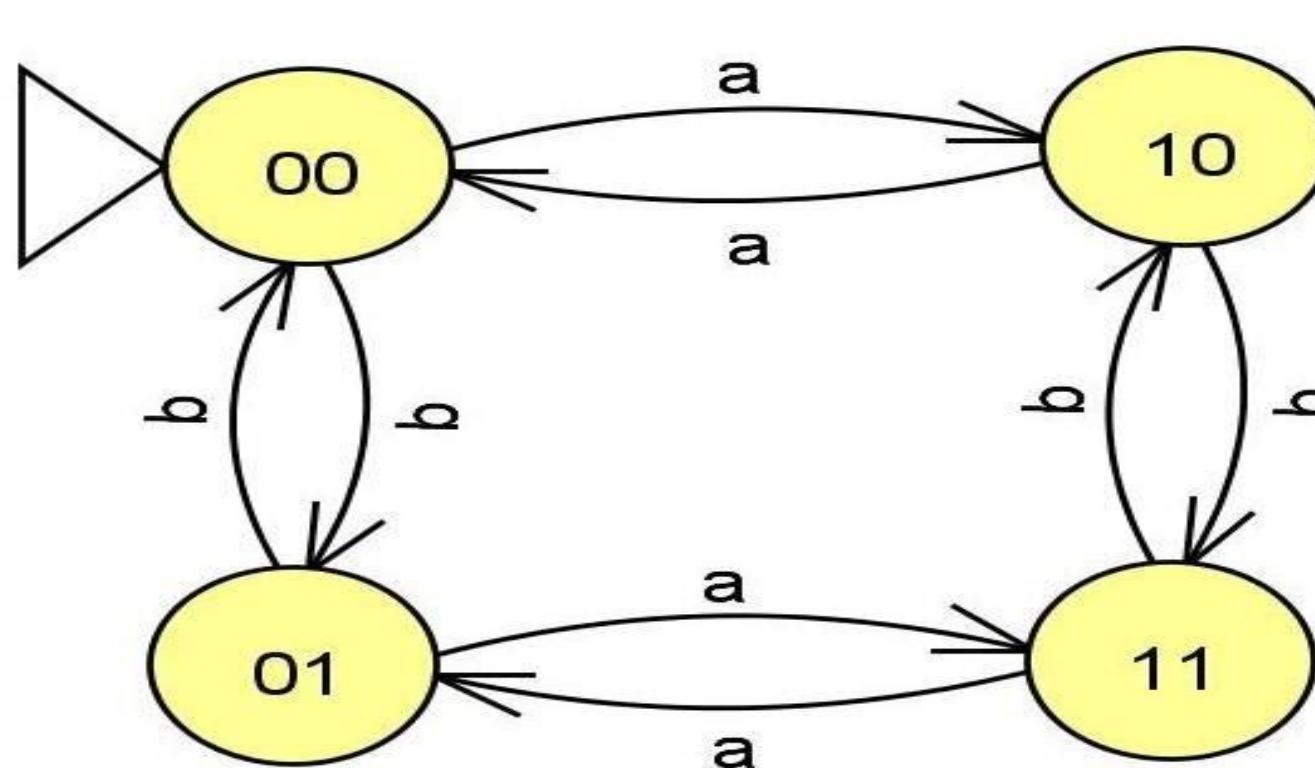
Construct DFA for the language of strings over  $\Sigma=\{a,b\}$  where,  $n_a(w) \bmod 2 = 0$  and  $n_b(w) \bmod 2 = 0$

### Solution:

### Example 12:

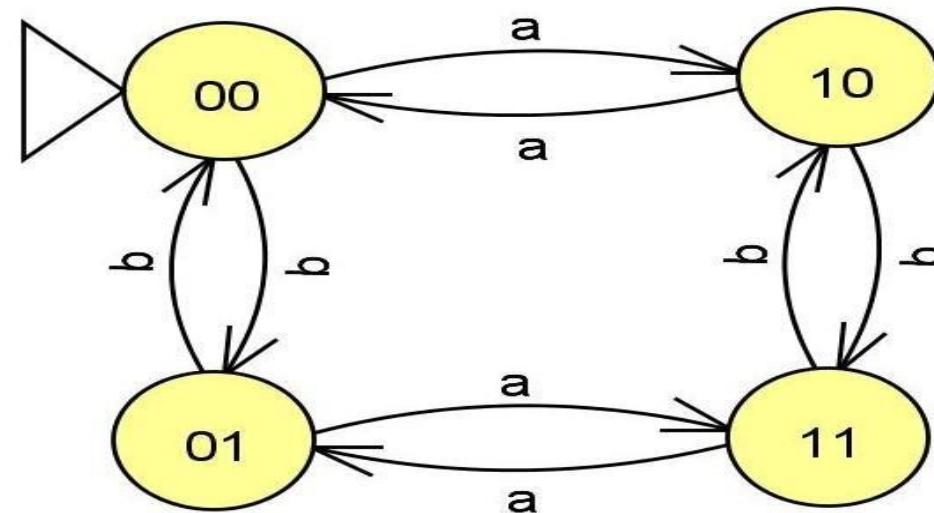
Construct DFA for the language of strings over  $\Sigma=\{a,b\}$  where,  $n_a(w) \bmod 2 = 0$  and  $n_b(w) \bmod 2 = 0$

### Solution:



This state represents, seen so far odd number of a's and even number of b's.

Note: State 00 is the final state.



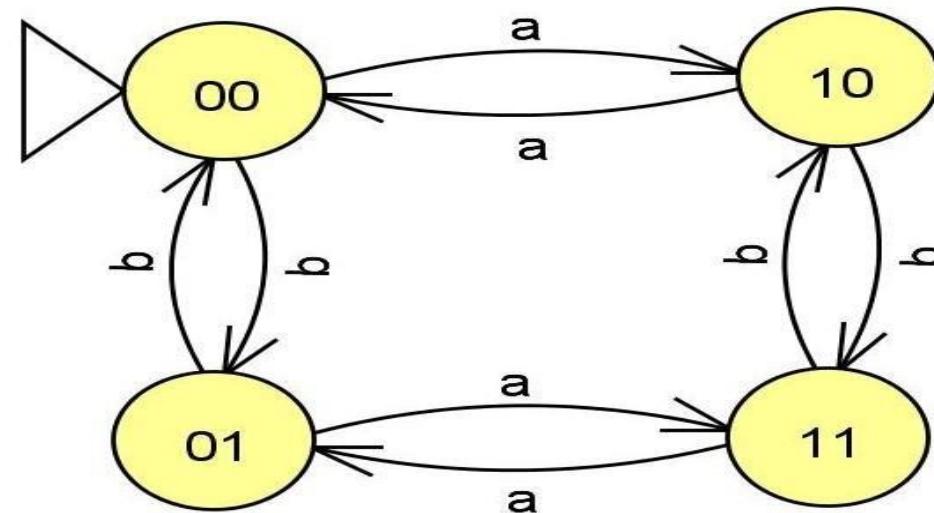
Note:

- State **00** is final state for strings having Even number of a's and Even number of b's. (i.e.,  $n_a(w) \bmod 2 = 0$  and  $n_b(w) \bmod 2 = 0$ )

$L = \{ \epsilon, aa, aaaa, aaaaaaa, \dots,$   
 $bb, bbbb, bbbbbbb, \dots,$   
 $aabb, bbaa, abab, baba, abba, baab, \dots \}$

- State **10** is final state for strings having odd number of a's and Even number of b's. (i.e.,  $n_a(w) \bmod 2 = 1$  and  $n_b(w) \bmod 2 = 0$ )

$L = \{ a, aaa, aaaaa, aaaaaaaaa, \dots,$   
 $abb, bab, bba, abbbb, babbb, \dots,$   
 $aaabb, bbaaa, \dots \}$



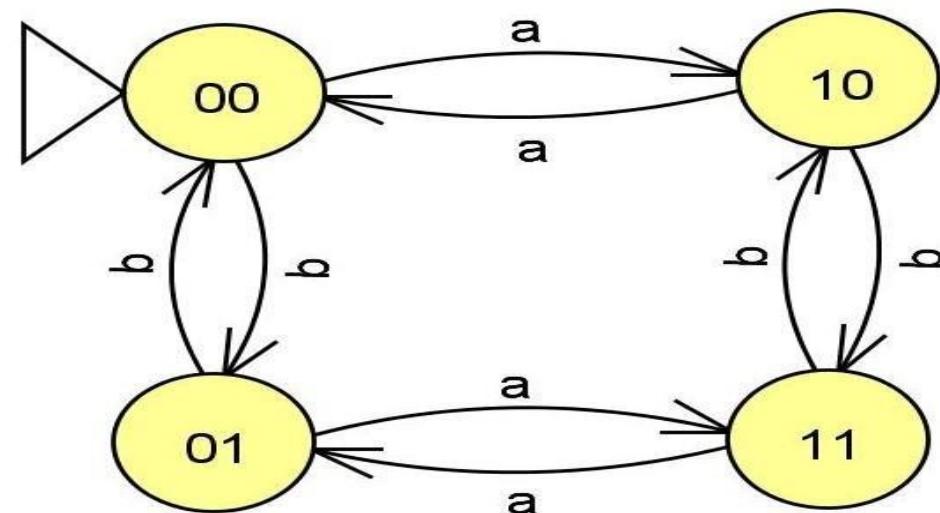
Note:

- **State 01 is final state** for strings having **Even number of a's and odd number of b's.** (i.e.,  $n_a(w) \bmod 2 = 0$  and  $n_b(w) \bmod 2 = 1$ )

$L = \{ b, bbb, bbbbb, bbbbbbb, \dots,$   
 $\quad baa, aba, aab, baaaa, abaaa, \dots, bbbbaa, aabbb, \dots \}$

- **State 11 is final state** for strings having **Odd number of a's and odd number of b's.** (i.e.,  $n_a(w) \bmod 2 = 1$  and  $n_b(w) \bmod 2 = 1$ )

$L = \{ ab, aaabbb, aaaaabbbbb, aaaaaaabbbbbbb, \dots,$   
 $\quad ba, bbbaaa, bbbbbaaaaa, bbbbbbbaaaaaaa, \dots,$   
 $\quad ababab, ababababab, \dots, bababa, bababababa, \dots, aabbba, abbbaa, \dots \}$



Note:



1<sup>st</sup> bit is related to symbol b's remainders [i.e.,  $n_b(w) \bmod k$ ]

2<sup>nd</sup> bit is related to symbol a's remainders [i.e.,  $n_a(w) \bmod k$ ]

In case of mod 2 - Remainder is 0 or 1

- Remainder 0 means Even number of a's or b's
- Remainder 1 means Odd number of a's or b's

In case of mod 3 - Remainder is 0 or 1 or 2

### Example 13:

Construct DFA for the language of strings over  $\Sigma=\{a,b\}$  where,  $n_a(w) \bmod 3 = 0$  and  $n_b(w) \bmod 2 = 0$ .

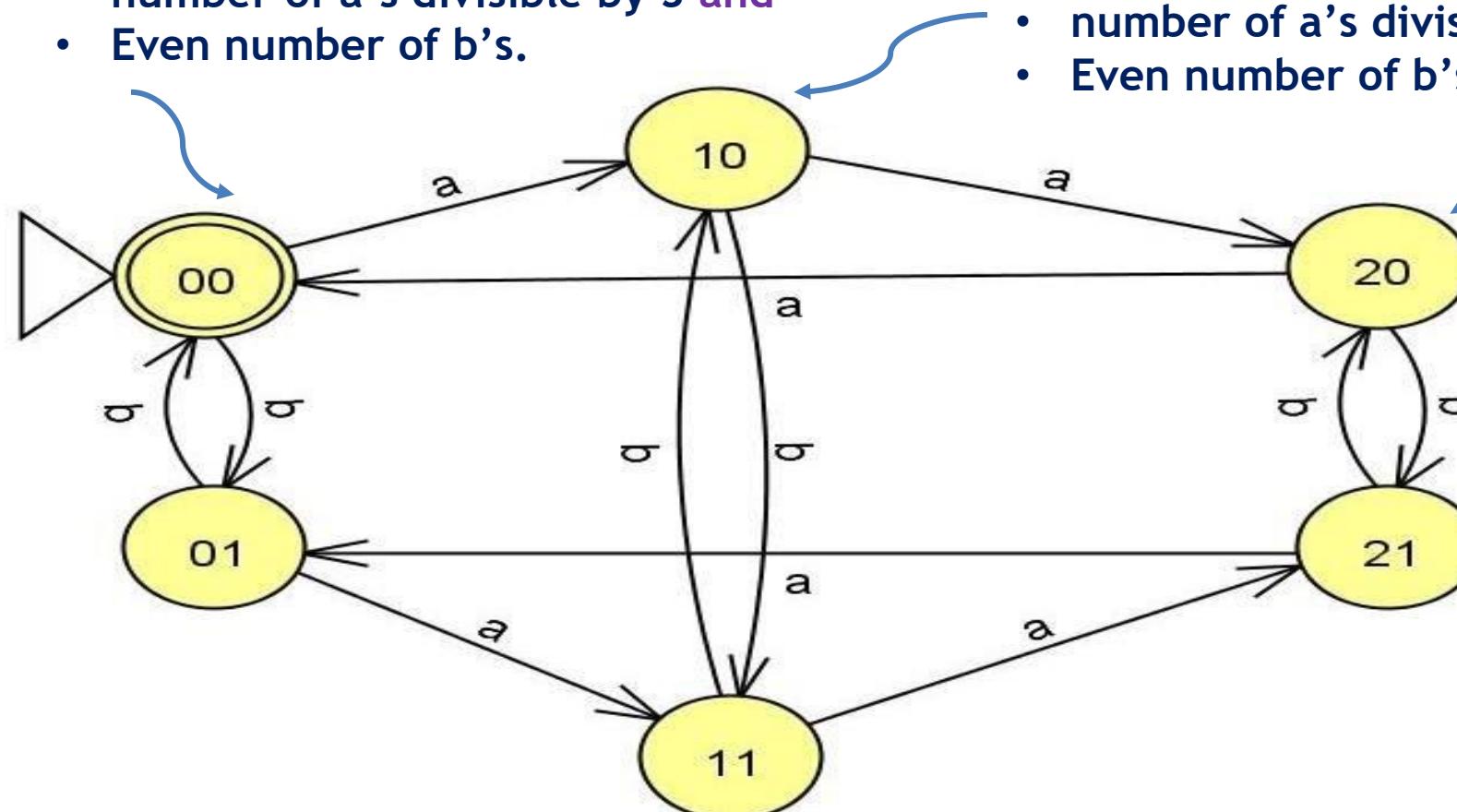
Solution :

### Example 13:

Construct DFA for the language of strings over  $\Sigma = \{a, b\}$  where,  $n_a(w) \bmod 3 = 0$  and  $n_b(w) \bmod 2 = 0$ .

### Solution :

This state represents, seen so far  
 • number of a's divisible by 3 and  
 • Even number of b's.



This state represents, seen so far  
 • number of a's divisible by 3 + 1 and  
 • Even number of b's.

This state represents, seen so far  
 • number of a's divisible by 3 + 2 and  
 • Even number of b's.

This state represents, seen so far  
 • number of a's divisible by 3 + 2 and  
 • Odd number of b's.

### Example 14:

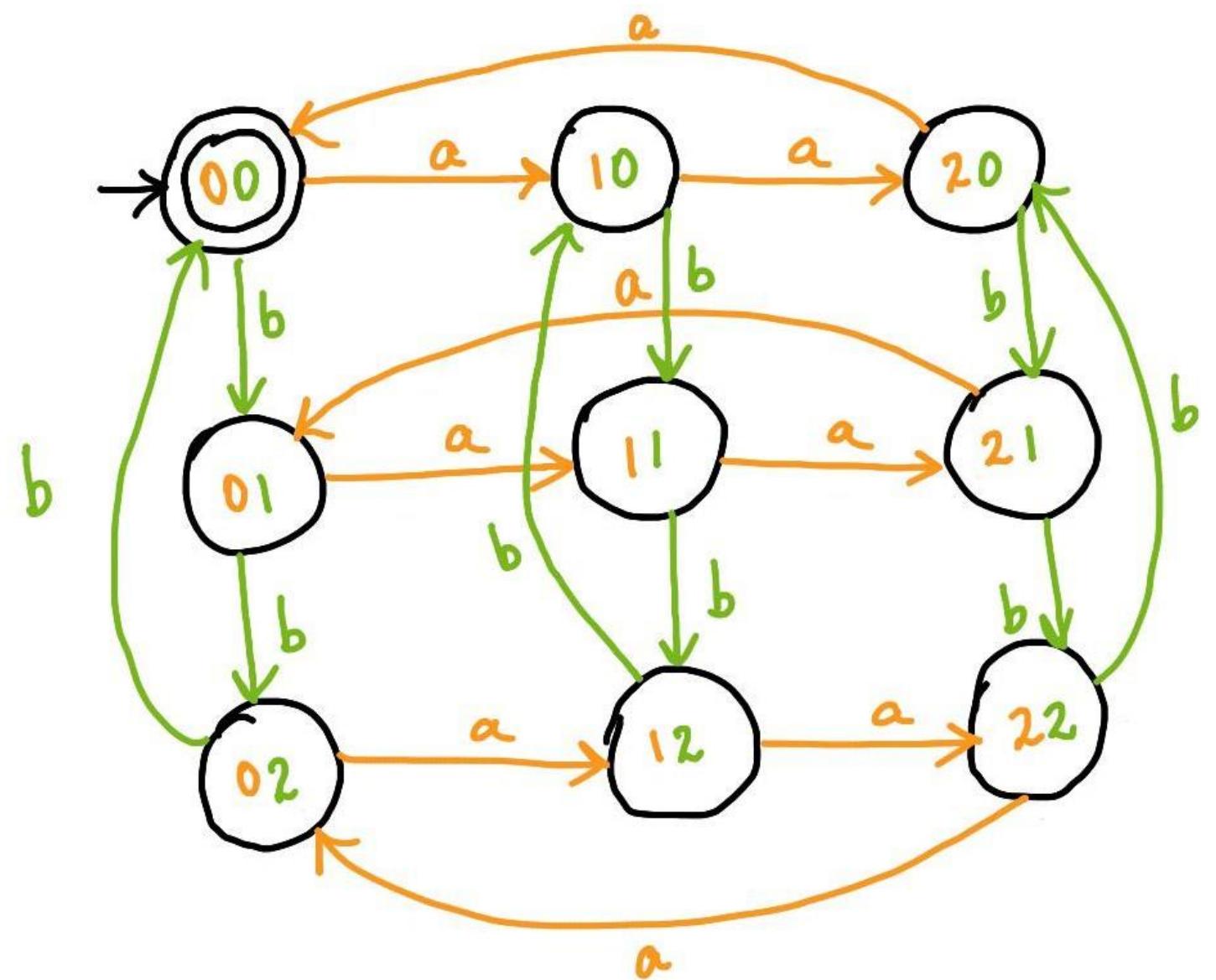
Construct DFA for the language of strings over  $\Sigma=\{a,b\}$  where,  $n_a(w) \bmod 3 = 0$  and  $n_b(w) \bmod 3 = 0$

Solution :

### Example 14:

Construct DFA for the language of strings over  $\Sigma = \{a, b\}$  where,  $n_a(w) \bmod 3 = 0$  and  $n_b(w) \bmod 3 = 0$

Solution :



### Example 15:

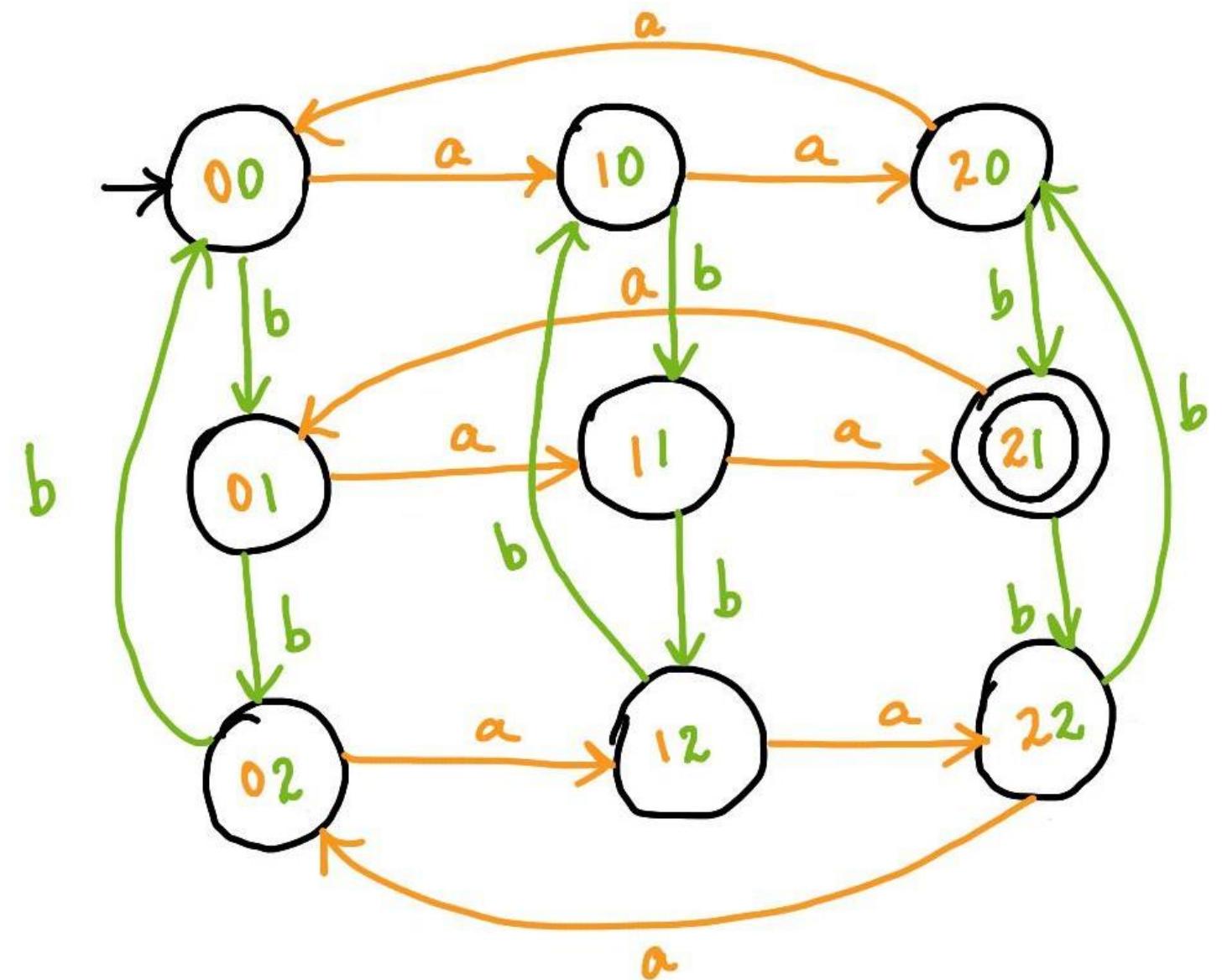
Construct DFA for the language of strings over  $\Sigma = \{a, b\}$  where,  $n_a(w) \bmod 3 = 2$  and  $n_b(w) \bmod 3 = 1$

### Solution :

### Example 15:

Construct DFA for the language of strings over  $\Sigma = \{a, b\}$  where,  $n_a(w) \bmod 3 = 2$  and  $n_b(w) \bmod 3 = 1$

Solution :



### Example 16:

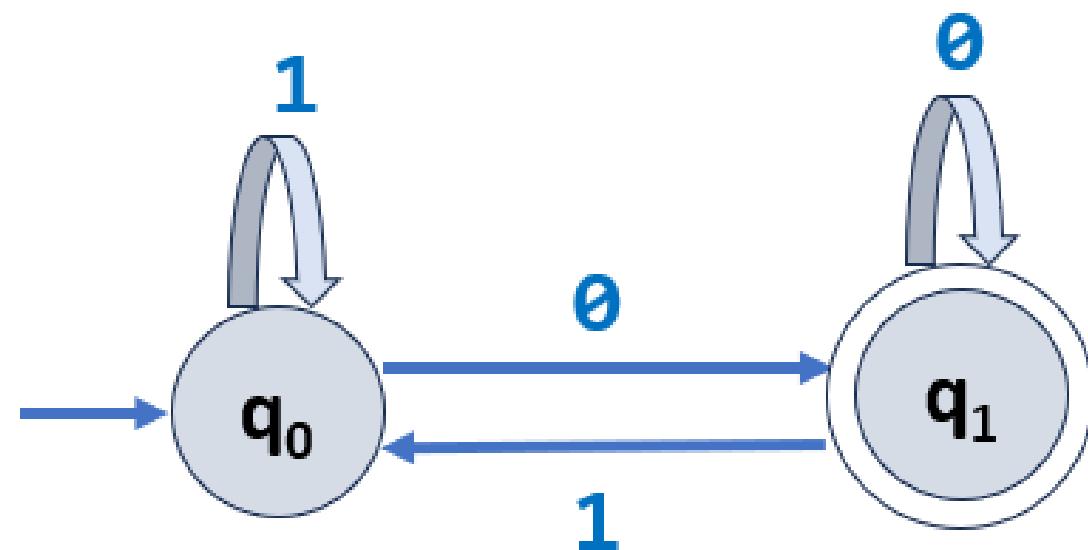
Construct DFA for a binary number divisible by 2 and  $\Sigma = \{0,1\}$ . i.e.,  $w \bmod 2 = 0$

Solution :

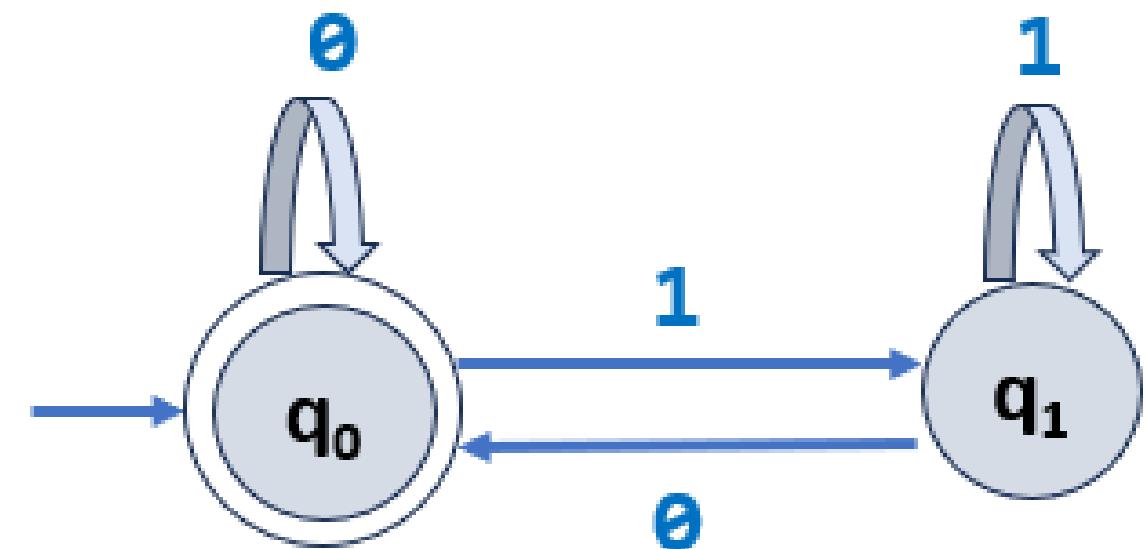
### Example 16:

Construct DFA for a binary number divisible by 2 and  $\Sigma = \{0,1\}$ . (i.e.,  $w \bmod 2 = 0$ )

**Solution :**  $L = \{ 0, 10, 100, 110, 1000, 1010, \dots \}$



or



**Example 16a:**

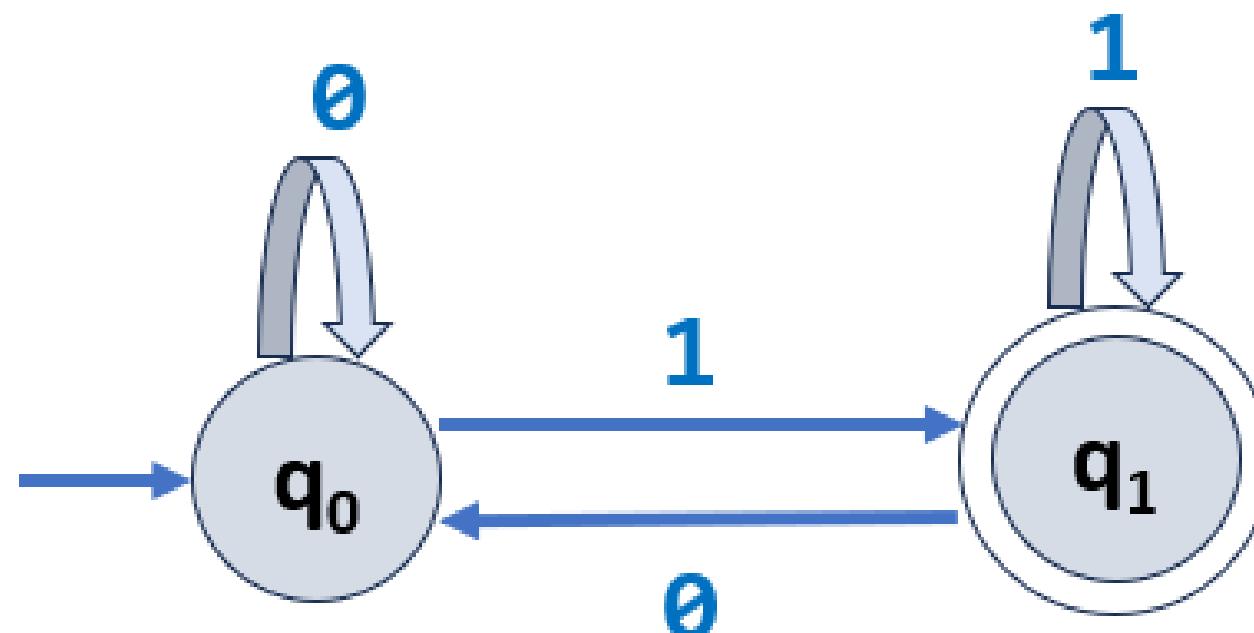
**Construct DFA for a binary number where  $w \bmod 2 = 1$**

**Solution :**

### Example 16a:

Construct DFA for a binary number where  $w \bmod 2 = 1$

Solution :



### Example 17:

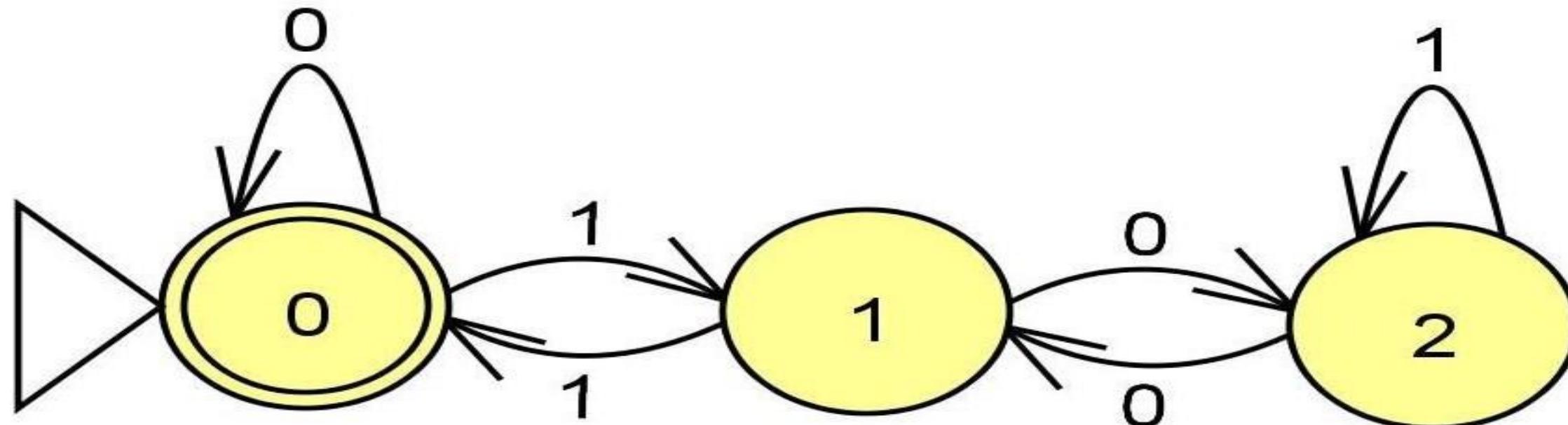
Construct a DFA for binary number divisible by 3. (i.e.,  $w \bmod 3 = 0$ ).

Solution :

### Example 17:

Construct a DFA for binary number divisible by 3. (i.e.,  $w \bmod 3 = 0$ ).

Solution :



### Example 18:

Construct a DFA for binary number divisible by 4. (i.e.,  $w \bmod 4 = 0$ ).

Solution :

### Exercise:

1. Construct DFAs to recognize C-language

- Identifiers,
- Integers
- Floating point numbers.

2. Construct DFAs to recognize C-language keywords

- for
- while
- switch

3. Construct a DFA to recognize all the relational operators in C-language.

### Exercise:

4. Construct DFAs for the validation of

- PAN number
- Aadhar number
- Email-id
- Phone number

5. ...

### Limitations of DFA (or FSMs):

1. No stored program concept - the machine is the computation. No auxiliary memory.
2. A DFA is not powerful enough to recognize many context-free languages because a DFA can't count. But counting is not enough - consider a language of palindromes, containing strings of the form  $ww_R$ . Such a language requires more than an ability to count; it requires a stack.

### Limitations of DFA (or FSMs): cont...

Non-regular languages are those for which no DFA exists. These languages have patterns or structures that cannot be recognized by a finite automaton.

#### Which languages CANNOT be described by any RE or DFA?

- Bit strings with equal number of 0s and 1s.
- Nested parenthesis matching.
- Decimal strings that represent prime numbers.
- Genomic strings that are Watson-Crick complemented palindromes.
- Many more. . . .

### Note:

- Every NFA has an equivalent DFA (accepting the same language).
- The languages accepted by Finite State Machines (DFA, NFA,  $\epsilon$ -NFA ...) are referred to as Regular languages (can be described by regular expressions).
- Regular Expression is a concise way to describe a set of strings.
- For any DFA, there exists a regular expression to describe the same set of strings; for any regular expression, there exists a DFA that recognizes the same set.
- DFA that accepts nothing has no final states.
- There is a unique minimal DFA for every regular language.



**PES**  
**UNIVERSITY**

CELEBRATING 50 YEARS

**THANK YOU**

---

**Preet Kanwal and Prakash C O**  
Department of Computer Science & Engineering

# Automata Formal Languages and Logic

## Unit 1 - Deterministic Finite Acceptor/Automata



| Criteria         | DFA  | NFA   |
|------------------|--|---|
| State transition | Each input symbol leads to exactly one state | One input symbol can lead to one, more or no states |



# Automata Formal Languages & Logic

## Unit 1 – NFA and $\epsilon$ -NFA

---

**Prakash C O**

Department of Computer Science & Engineering

# Automata Formal Languages & Logic

---

## Unit 1: NFA and $\epsilon$ -NFA

Prakash C O

Department of Computer Science & Engineering

### Formal Definition of an NFA

NFA can be represented by a 5-tuple  $M = (Q, \Sigma, \delta, q_o, F)$  where

- **Q** is a finite set of states.
- **$\Sigma$**  is a finite non-empty set of input symbols called the alphabet.
- **$\delta$**  is the transition function where  $\delta: Q \times \Sigma \rightarrow 2^Q$
- **$q_o$**  is the initial state from where any input is processed ( $q_o \in Q$ ).
- **F** is a set of final state/states ( $F \subseteq Q$ ).

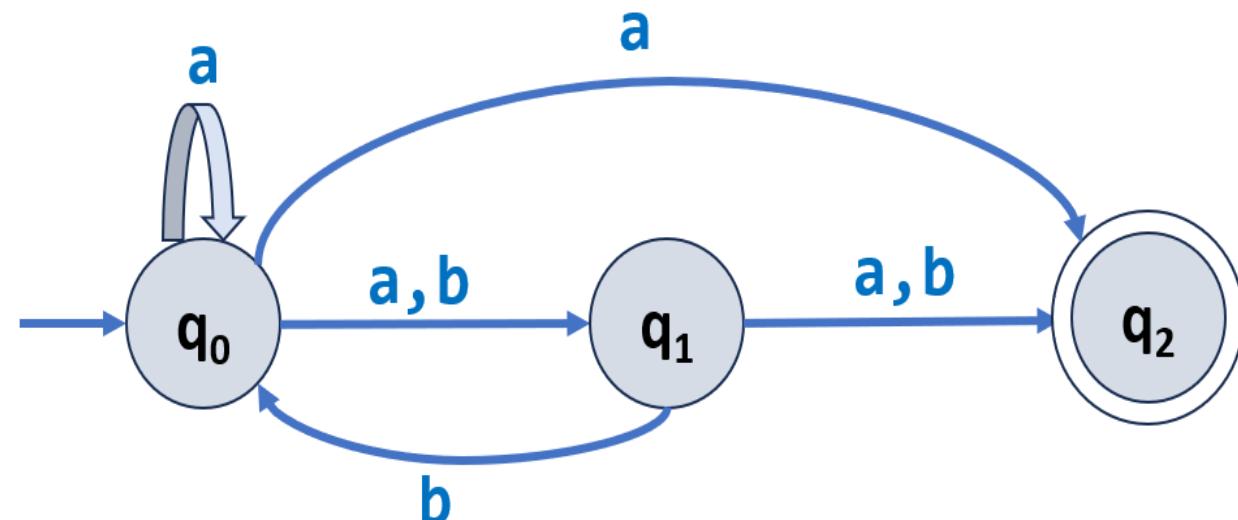
### Why NFA?

- NFAs were introduced in 1959 by Michael O. Rabin and Dana Scott, who also showed their equivalence to DFAs.
- NFAs are used in the implementation of Regular Expressions (i.e., Thompson's construction is used to convert RE to NFA).
- Constructing an NFA to recognize a given language is sometimes much easier than constructing a DFA for that language.
- DFAs have been generalized to nondeterministic finite automata (NFA). NFA provides a nice way of ‘abstraction of information’.

The Transition Function of an NFA is  $\delta: Q \times \Sigma \rightarrow 2^Q$

- Here the power set (i.e.,  $P(Q)=2^Q$ ) has been taken because in case of an NFA, from a state, transition can occur to any combination of  $Q$  states.

- Example:



$$2^Q = 2^3 = \{ \emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\} \}$$

- In an NFA, a (state, input\_symbol) combination may lead to several states simultaneously.
 
$$\delta(q_0, a) = \{q_0, q_1, q_2\}$$

$$\delta(q_2, a) = \emptyset$$

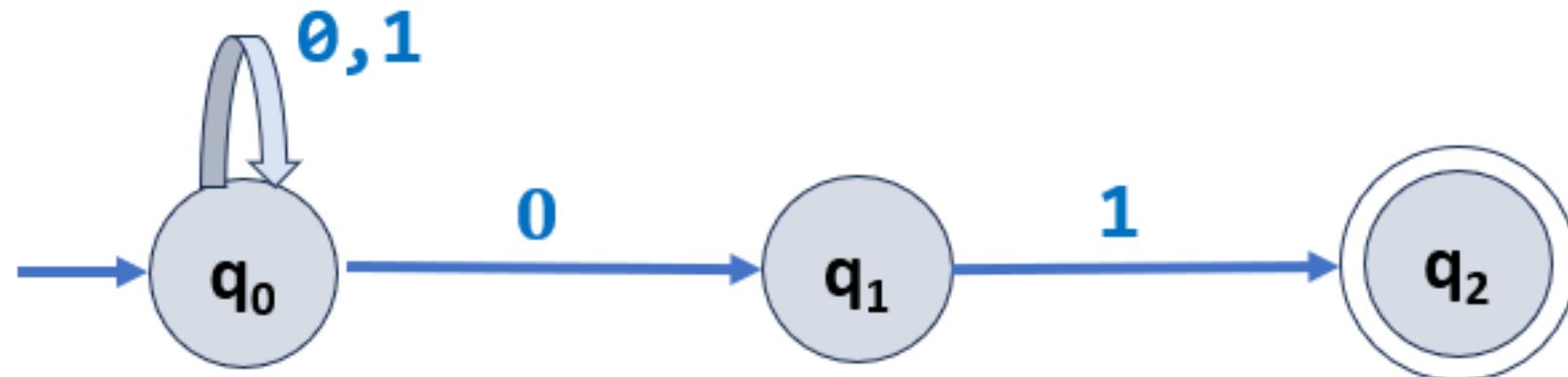
### Why NFA is called Non-deterministic?

- In automata theory, a finite-state machine is called a deterministic finite automaton (DFA), if
  - each of its transitions is uniquely determined by its source state and input symbol, and
  - reading an input symbol is required for each state transition.
- A nondeterministic finite automaton (NFA), does not need to obey the above restrictions of DFA. In particular, every DFA is also an NFA. NFA may have several arrows of the same label starting from a state.
- A Finite Automata(FA) is said to be non-deterministic if there is more than one possible transition from a state on the same input symbol.

### Computation in NFA

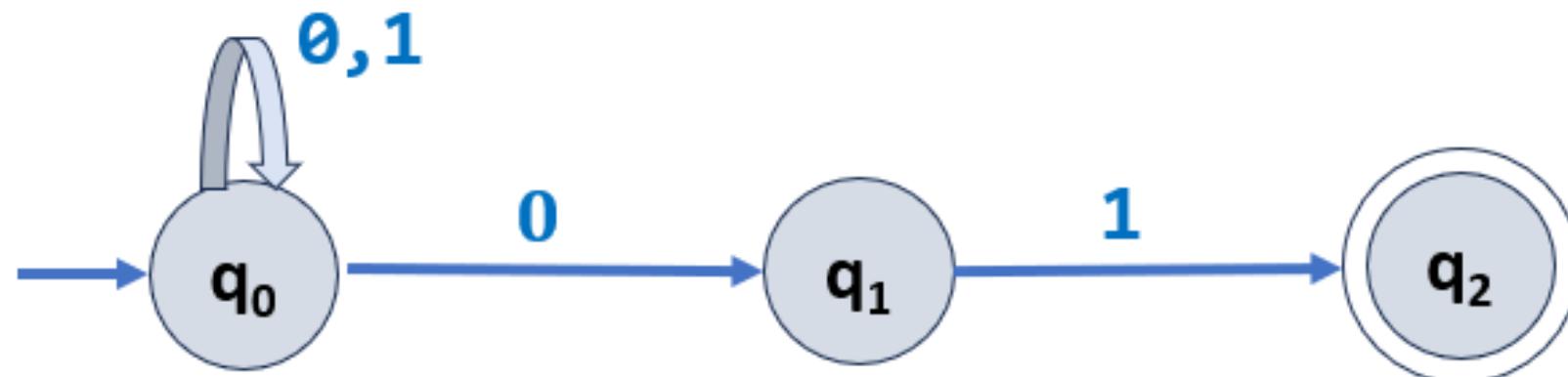
- For an input string, NFA will create a ‘computation tree’ rather than a ‘computation sequence’ in case of DFA.  
**An NFA accepts a string if any one of the paths in the tree leads to some accept state.**
- Nondeterminism in NFA causes multiple runs. There can be several or no runs for a given input word.  
**If there is at least one run that leads to an accepting state, the machine will accept the input string.**

➤ **Example 1:** Consider  $L = \{ x \in \{0, 1\}^* \mid \text{where } x \text{ ends with } 01\}$

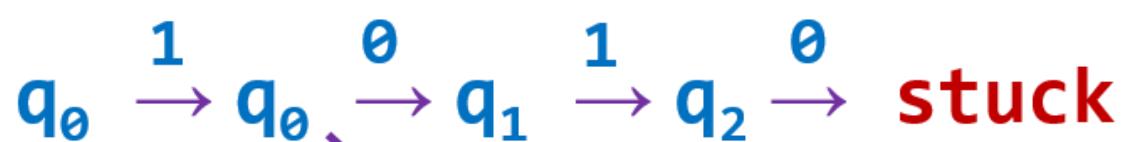


Let 10101 be an input word. The following are potential runs.

➤ **Example 1:** Consider  $L = \{ x \in \{0, 1\}^* \mid \text{where } x \text{ ends with } 01\}$



Let 10101 be an input word. The following are potential runs.



Unfinished runs are unacceptable

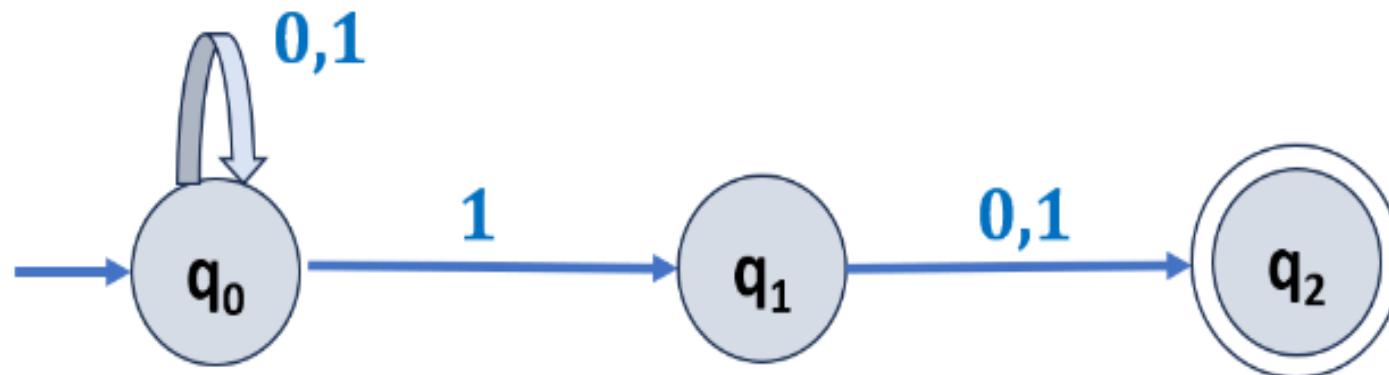


Accepted run/Computation



Rejected run/Computation

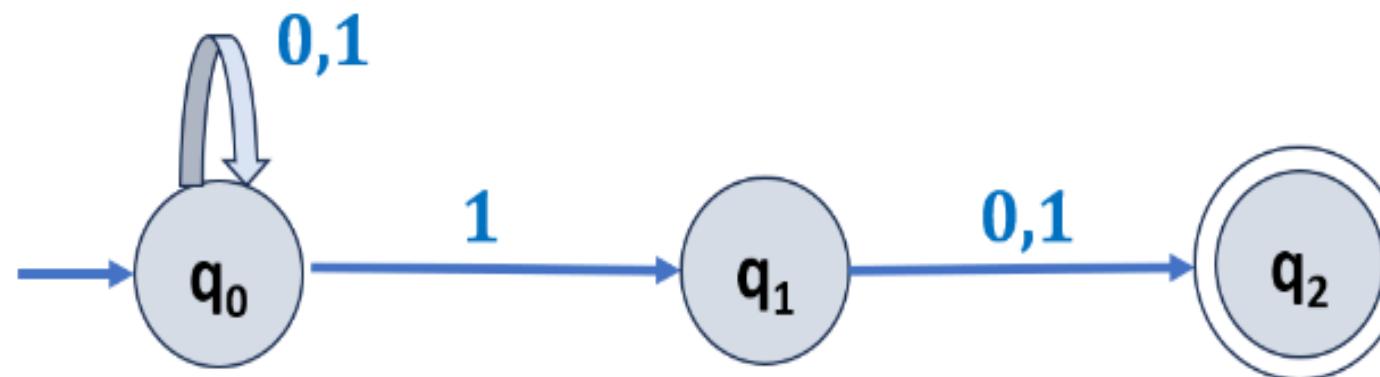
➤ **Example 2:** Consider  $L = \{x \in \{0, 1\}^* \mid \text{2nd symbol from the right is } 1\}$



Computation tree for the input string 11010

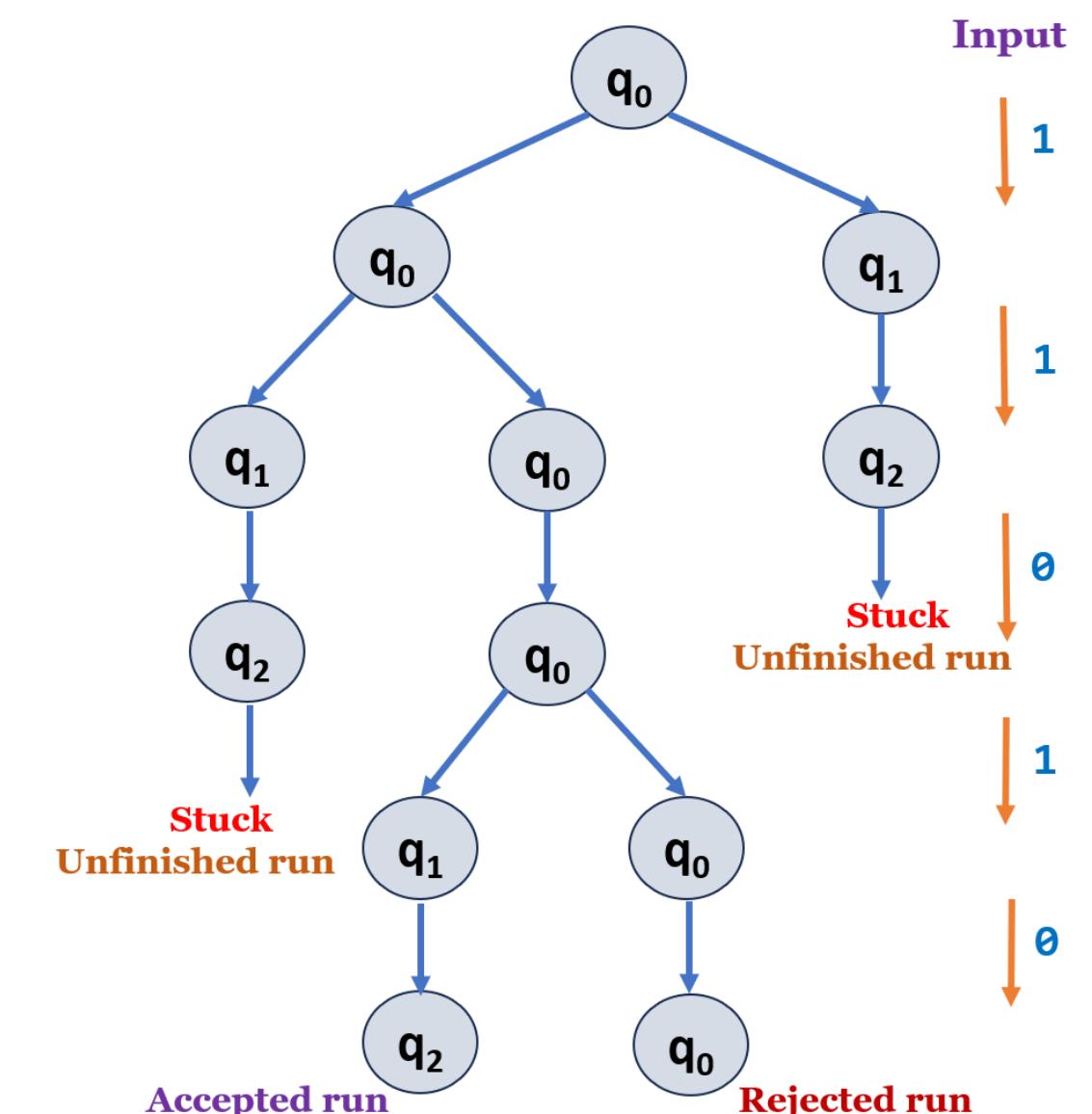
- The NFA consumes a string of input symbols, one by one. In each step, **whenever two or more transitions are applicable**, it "clones" itself into appropriately many copies, each one following a different transition.
- 
- If no transition is applicable, the current copy is in a dead end (or stuck), it "dies". If, after consuming the complete input, any of the copies is in an accept state, the input is accepted, else, it is rejected.

➤ Example 2: Consider  $L = \{x \in \{0, 1\}^* \mid \text{2nd symbol from the right is } 1\}$



Computation tree for the input string 11010

- The NFA consumes a string of input symbols, one by one. In each step, **whenever two or more transitions are applicable**, it "clones" itself into appropriately many copies, each one following a different transition.
- 
- If no transition is applicable, the current copy is in a dead end (or stuck), it "dies". If, after consuming the complete input, any of the copies is in an accept state, the input is accepted, else, it is rejected.



### NFA Construction:

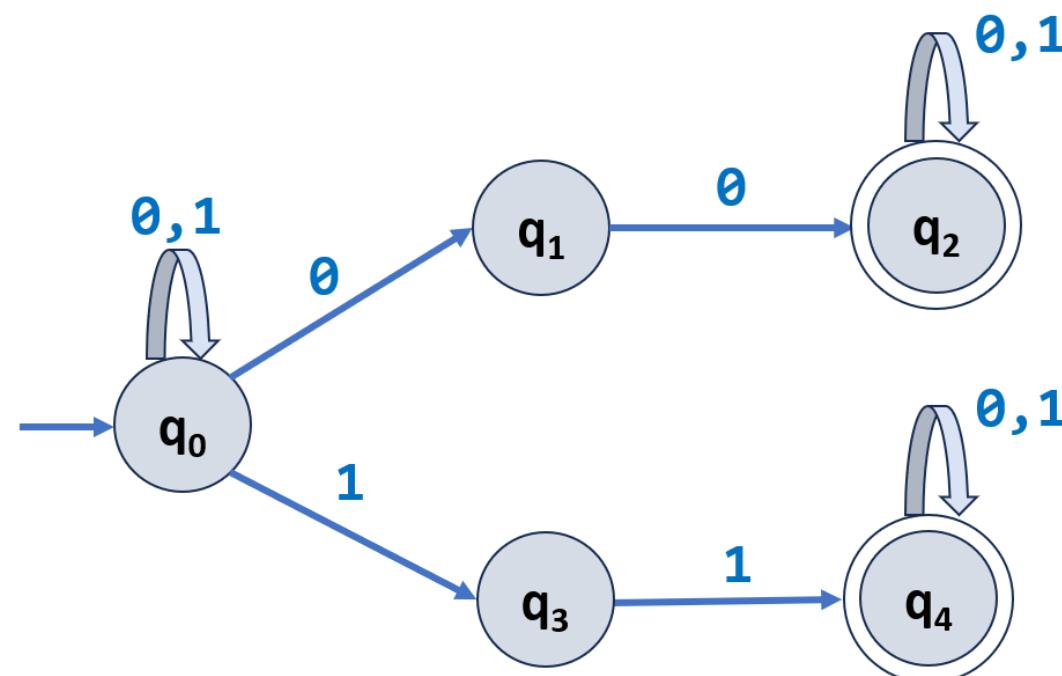
**Example 1:** Construct NFA for the language of binary strings having at least a pair of '00' or a pair of '11'

**Solution:**

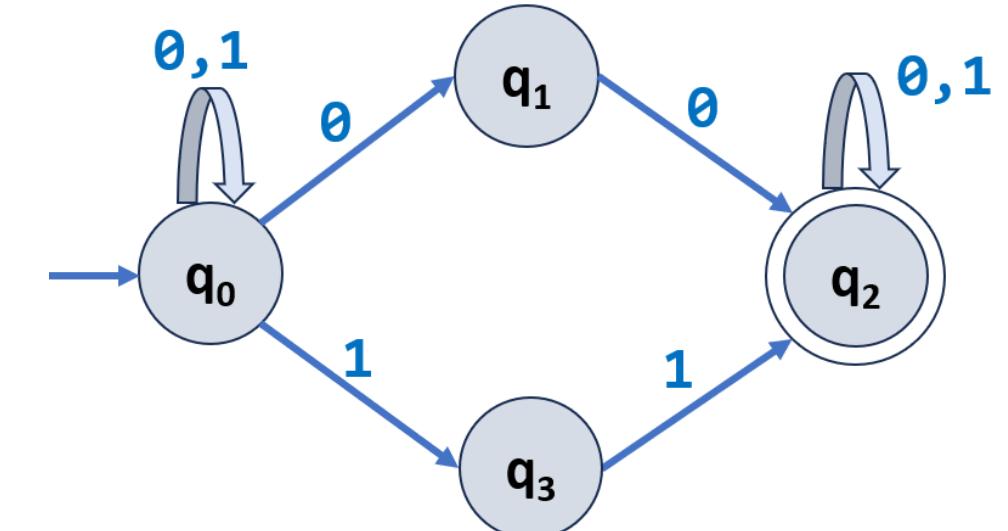
### NFA Construction:

**Example 1:** Construct NFA for the language of binary strings having at least a pair of '00' or a pair of '11'

### Solution:



or

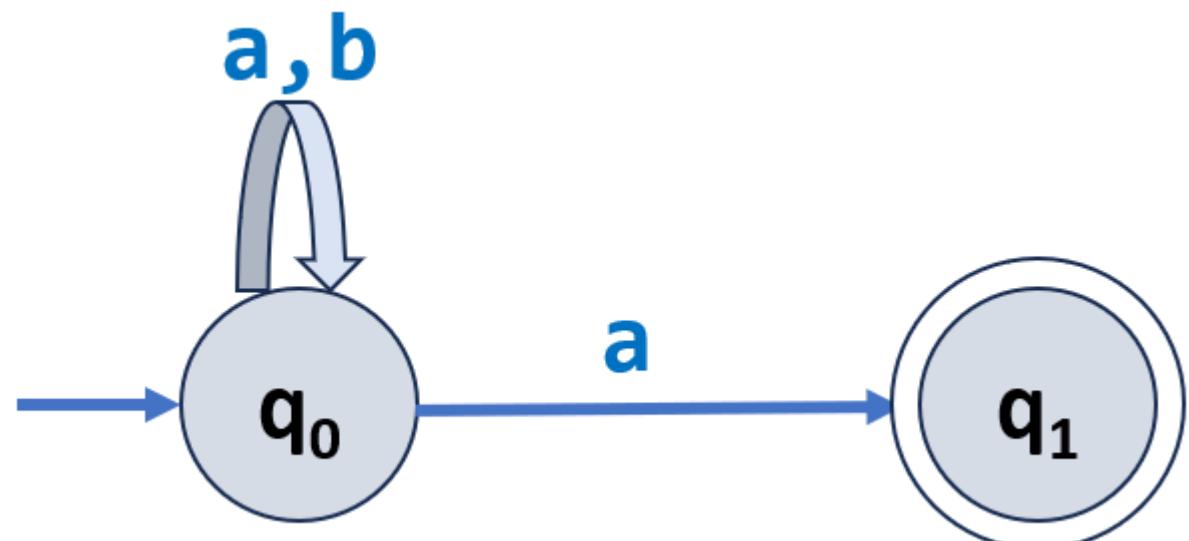


**Example 2: Construct NFA for the language of strings ends with a, over  $\Sigma = \{a, b\}$**

**Solution:**

**Example 2:** Construct NFA for the language of strings ends with a, over  $\Sigma = \{a, b\}$

**Solution:**  $L = \{ a, aa, ba, bba, baa, aaa, aba, aaaa, aaba, abba, baba, bbba, \dots \}$

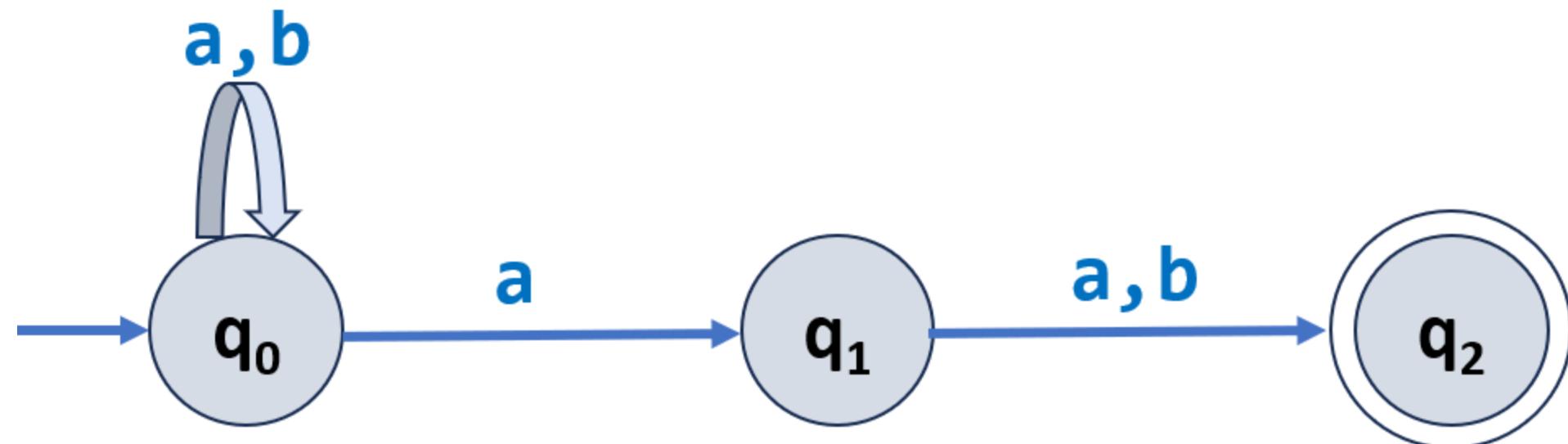


**Example 3:** Construct NFA for the language of strings, where the second symbol from the RHS of a string is 'a' and  $\Sigma = \{a, b\}$

**Solution:**

**Example 3:** Construct NFA for the language of strings, where the second symbol from the RHS of a string is 'a' and  $\Sigma = \{a, b\}$

**Solution:**  $L=\{ aa, ab, aab, baa, bab, aaa, aaaa, aaab, abab, baab, bbab, \dots \}$



**Example 4:** Design an NFA in which all the string contain a substring 1110

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

**Solution:**

**Example 5:** Design an NFA with  $\Sigma = \{0, 1\}$  accepts all string in which the third symbol from the right end is always 0.

**Solution:**

### DFA and NFA Equivalence

- NFAs and DFAs are equivalent in that if a language is recognized by an NFA, it is also recognized by a DFA and vice versa.
- The establishment of NFA and DFA equivalence is important and useful.
  - It is useful because constructing an NFA to recognize a given language is sometimes much easier than constructing a DFA for that language.
  - It is important because NFAs can be used to reduce the complexity of the mathematical work required to establish many important properties in automata theory.
  - For example, it is much easier to prove closure properties of regular languages using NFAs than DFAs.

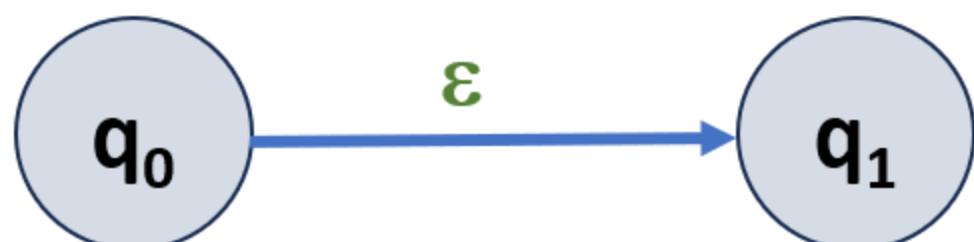
### Formal Definition of an $\epsilon$ -NFA (or $\lambda$ -NFA)

$\epsilon$ -NFA can be represented by a 5-tuple  $M = (Q, \Sigma, \delta, q_0, F)$  where

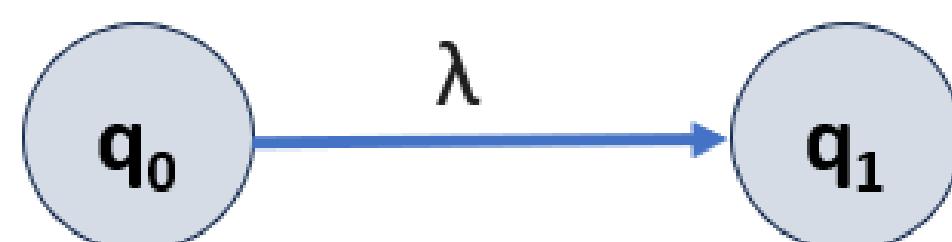
- **$Q$  is a finite set of states.**
- **$\Sigma$  is a finite non-empty set of input symbols called the alphabet.**
- **$\delta$  is the transition function where  $\delta: Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$  or  $\delta: Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$**
- **$q_0$  is the initial state from where any input is processed ( $q_0 \in Q$ ).**
- **$F$  is a set of final state/states ( $F \subseteq Q$ ).**

### $\epsilon$ -transitions (or $\lambda$ -transitions) (or Null moves):

- Let us add another feature to our automaton.
- Let us allow it to jump states without reading inputs. We will call such moves  $\epsilon$ -transitions (or  $\lambda$ -transitions) .



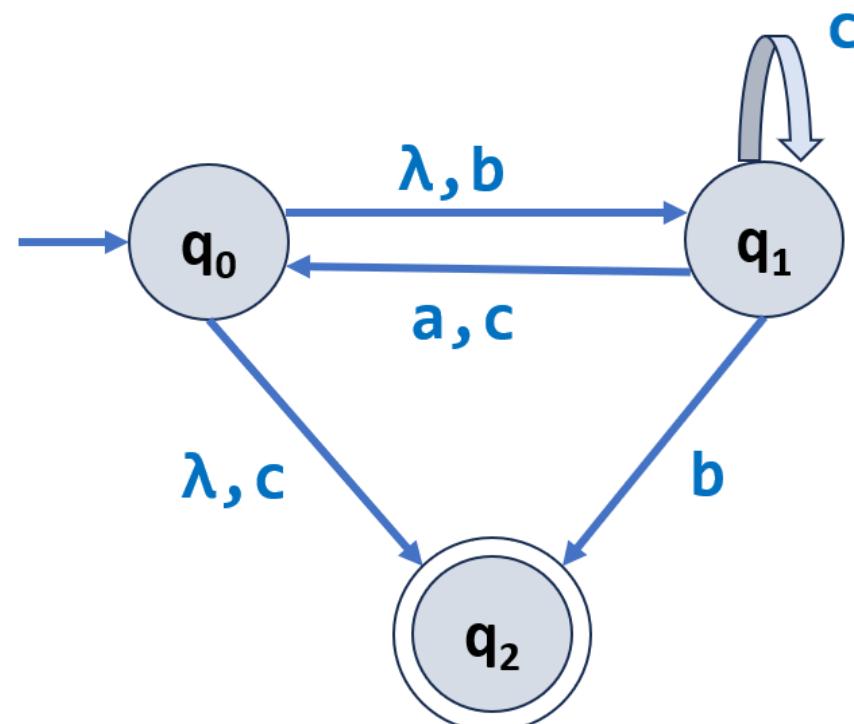
or



### $\epsilon$ -closure (or $\lambda$ -closure) of a state or set of states:

- For a state  $q$ , let  $\epsilon$ -closure( $q$ ) (or  $\lambda$ -closure( $q$ )) denote the set of states that are reachable from  $q$  by  $\epsilon$ -transitions (or  $\lambda$ -transitions) including the state  $q$  itself.

#### Example 1:



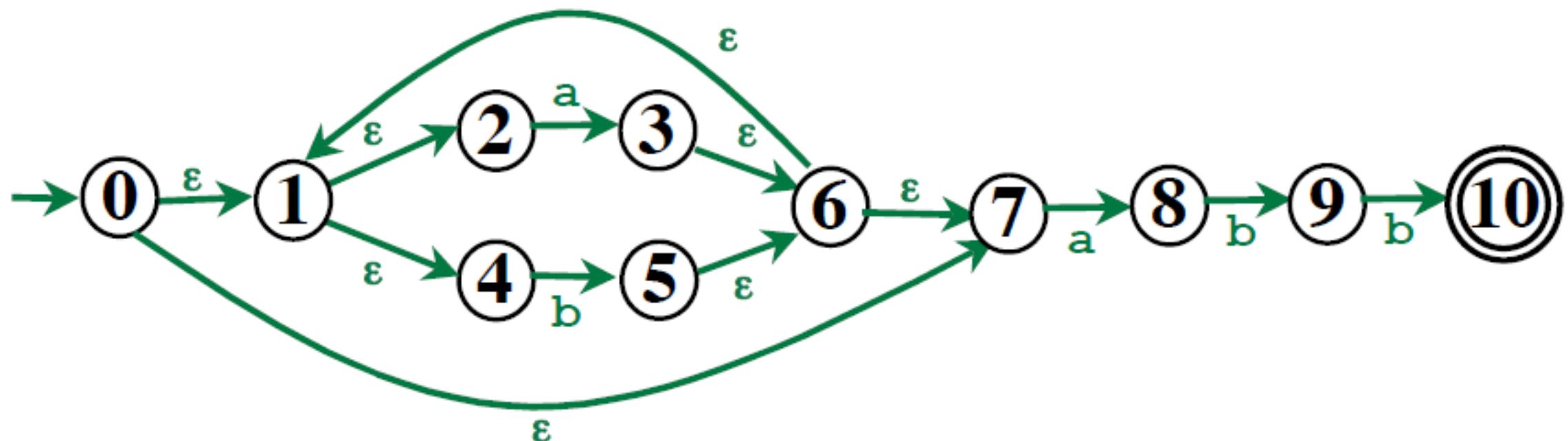
$$\lambda\text{-closure}(q_0) = \{q_0, q_1, q_2\}$$

$$\lambda\text{-closure}(q_1) = \{q_1\}$$

$$\lambda\text{-closure}(q_2) = \{q_2\}$$

**$\epsilon$ -closure (or  $\lambda$ -closure) of a state or set of states:**

➤ Example 2:

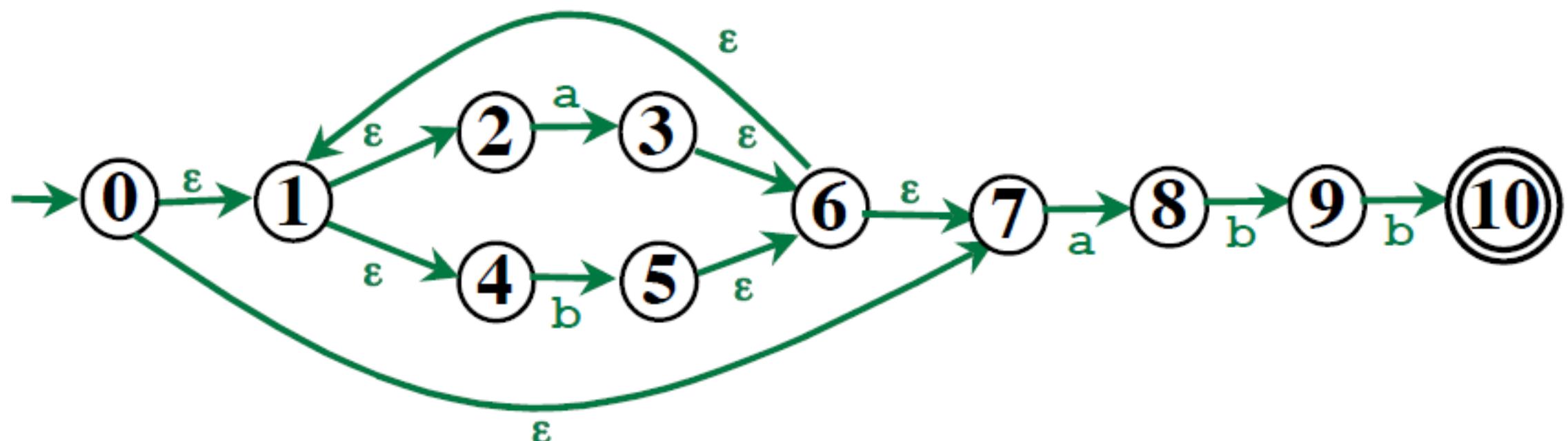


**$\epsilon$ -closure(0) =**

**$\epsilon$ -closure(6) =**

**$\epsilon$ -closure (or  $\lambda$ -closure) of a state or set of states:**

➤ Example 2:

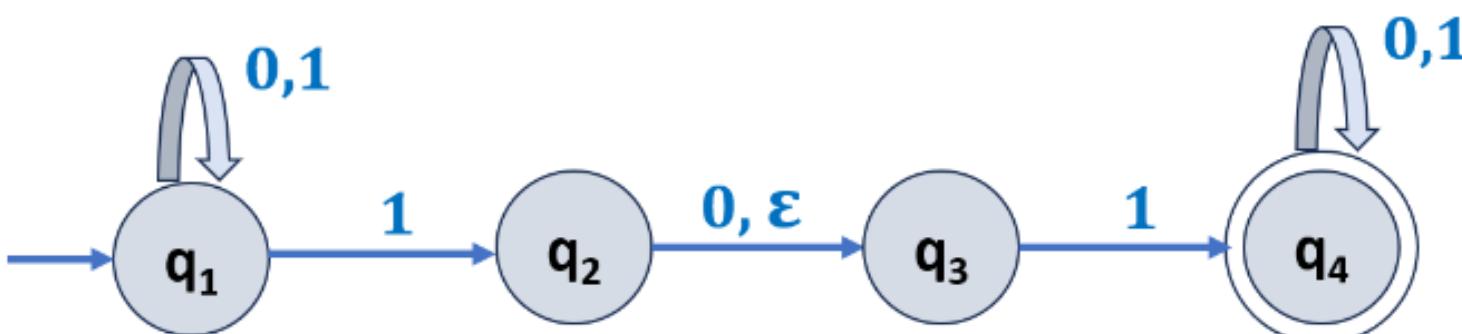


$$\epsilon\text{-closure}(0) = \{ 0 \ 1 \ 2 \ 4 \ 7 \}$$

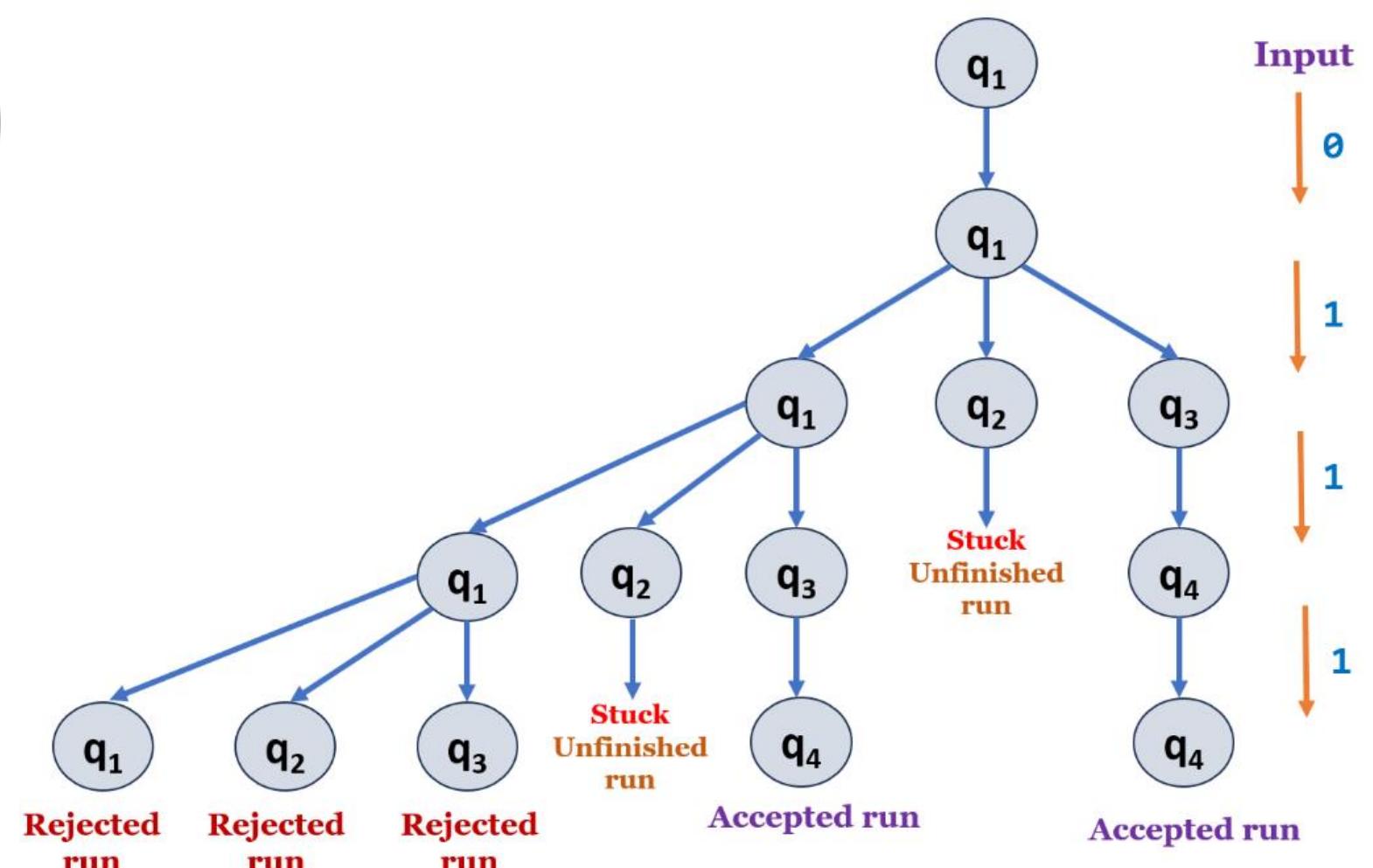
$$\epsilon\text{-closure}(6) = \{ 6 \ 7 \ 1 \ 2 \ 4 \}$$

### Computation or Run in an $\epsilon$ -NFA

**Example:** Consider  $L = \{ x \in \{0, 1\}^* \mid \text{where } x \text{ contains the substring } 101 \text{ or } 11 \}$

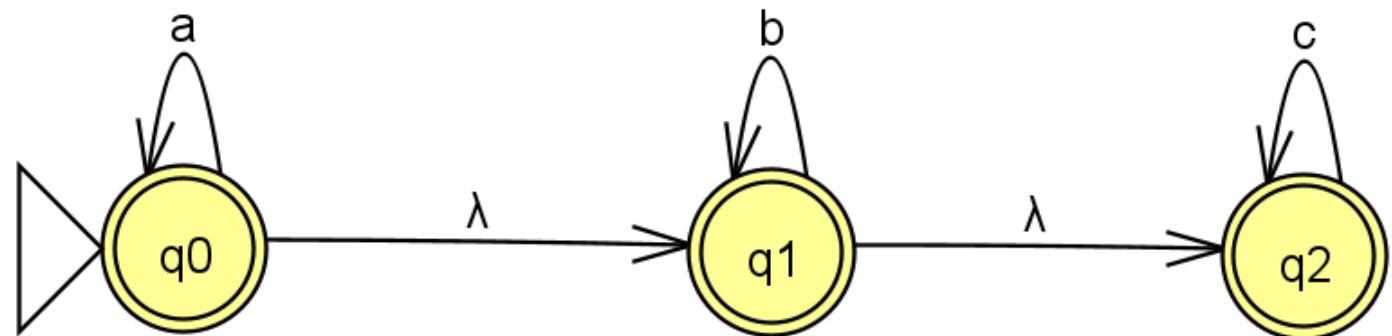


Computation tree for the input string 0111



### Computation or Run in an $\epsilon$ -NFA

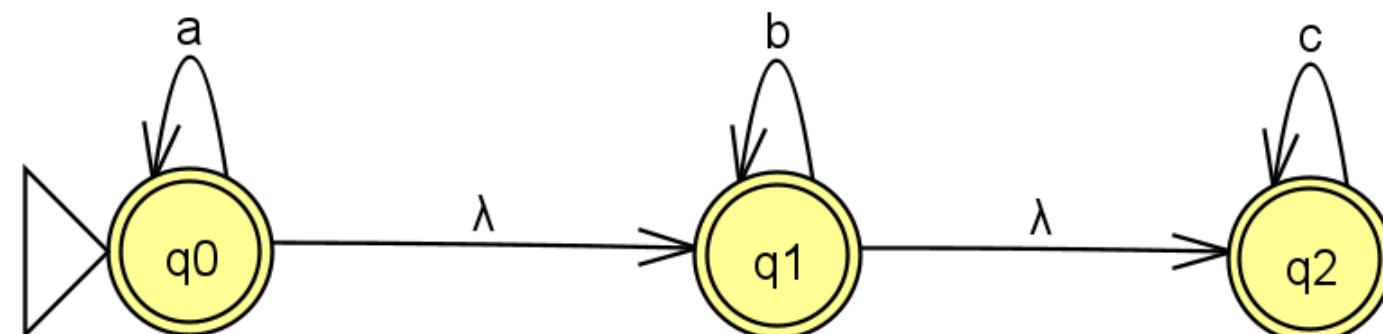
**Exercise:** Consider the language of strings over  $\Sigma = \{a, b, c\}$  of the form  $a^n b^m c^k$  where  $n, m, k \geq 0$ .



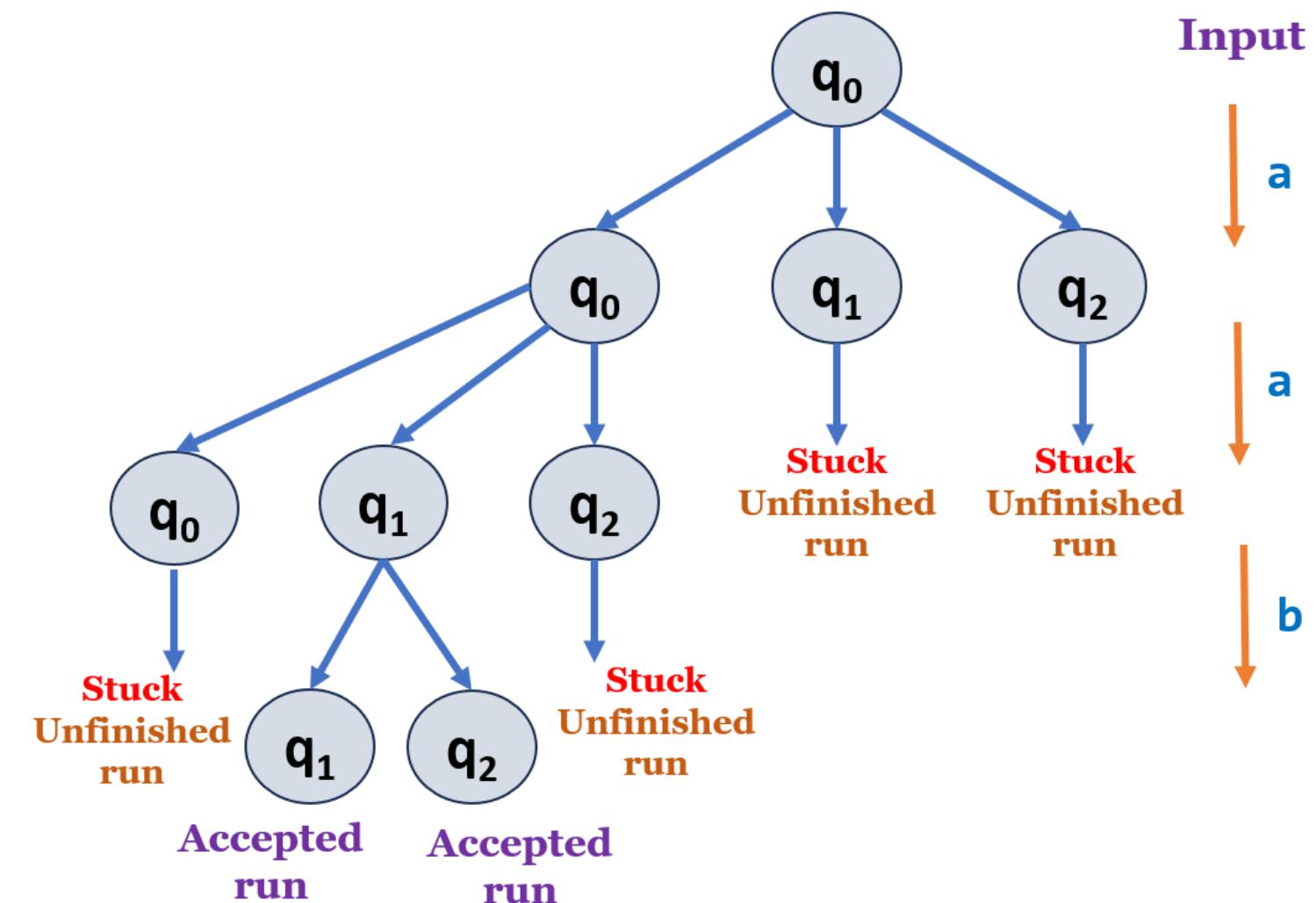
Construct Computation tree for the input string aab.

### Computation or Run in an $\epsilon$ -NFA

**Exercise:** Consider the language of strings over  $\Sigma = \{a, b, c\}$  of the form  $a^n b^m c^k$  where  $n, m, k \geq 0$ .



Computation tree for the input string aab.

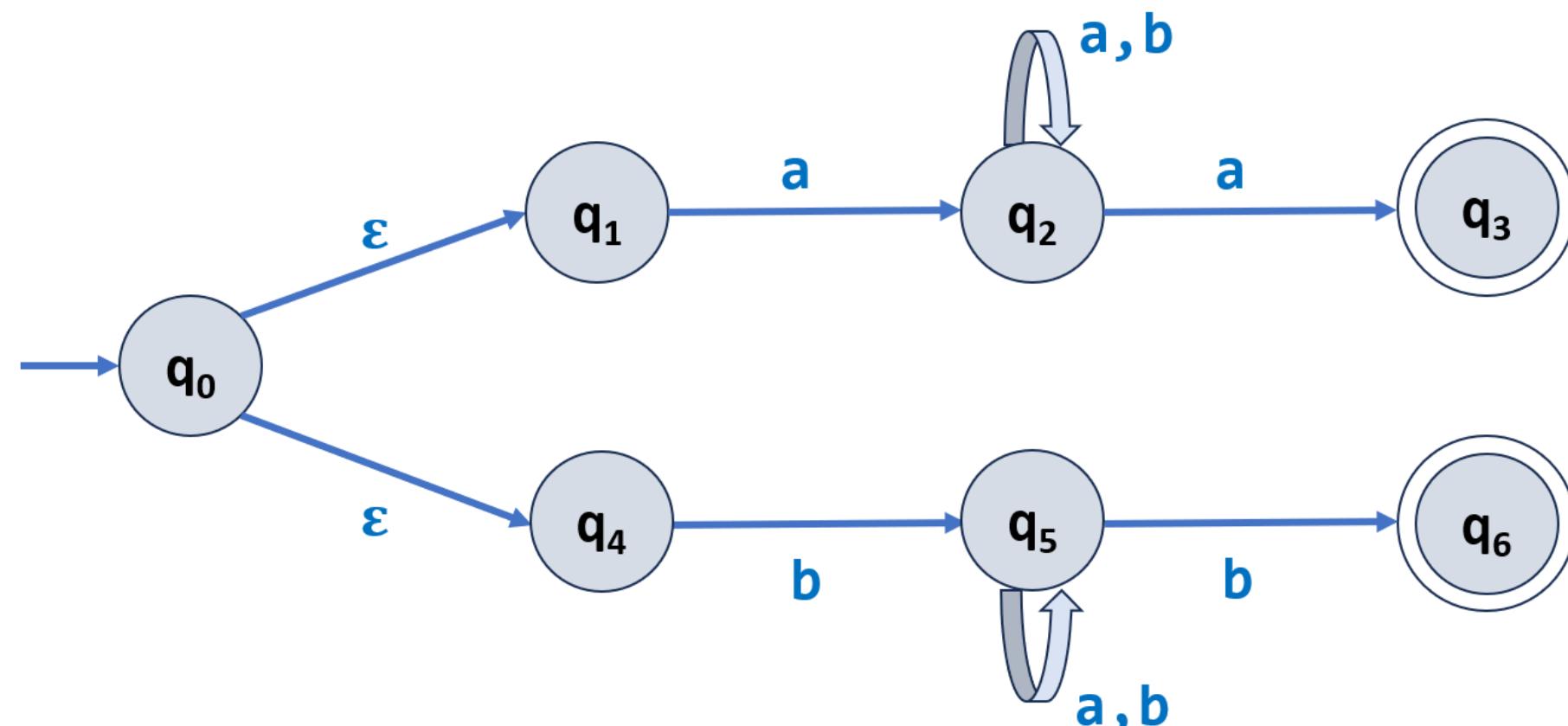


**Example 1:** Construct  $\epsilon$ -NFA for a language of strings that begins and ends with the same symbol, and  $\Sigma = \{a, b\}$

**Solution:**

**Example 1:** Construct  $\epsilon$ -NFA for a language of strings that begins and ends with the same symbol, and  $\Sigma = \{a, b\}$

**Solution:**  $L=\{ aa, bb, aba, bab, aaa, bbb, \dots \}$



**Example 2: Construct  $\epsilon$ -NFA that recognizes integers without leading zeros.**

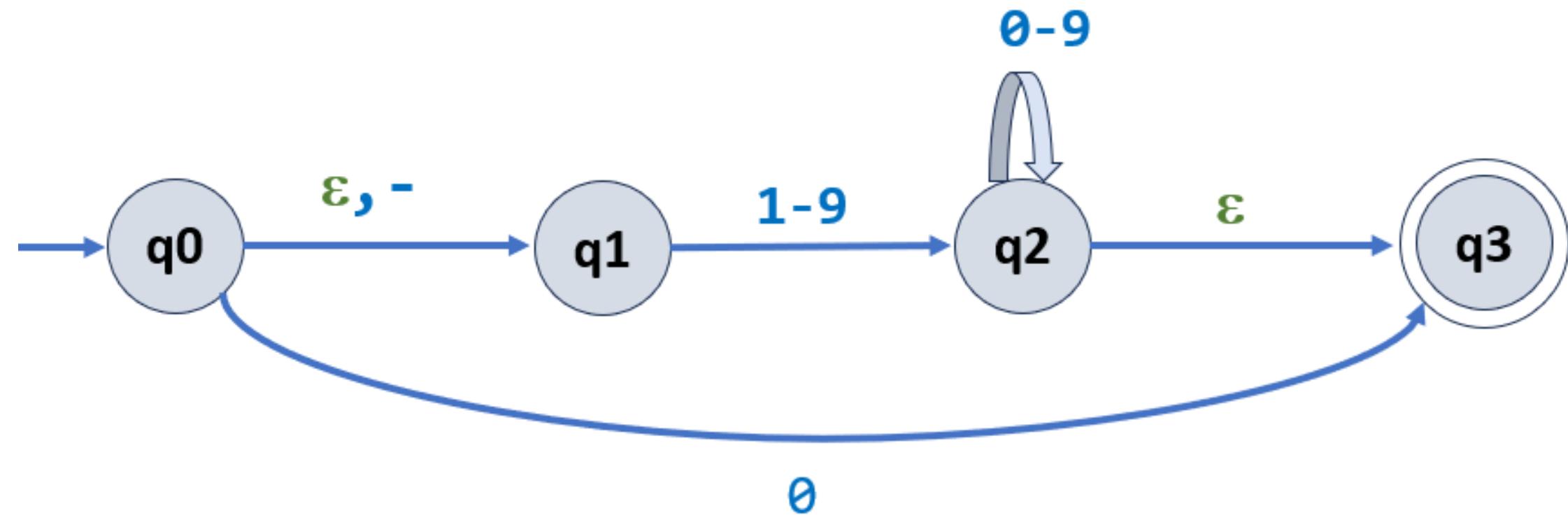
$$\Sigma = \{ -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

**Solution:** Example integers: ... , -2, -1, 0, 1, 2, ... , 101, 102, ...

**Example 2: Construct  $\epsilon$ -NFA that recognizes integers without leading zeros.**

$$\Sigma = \{ -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

**Solution:** Example integers: ... , -2, -1, 0, 1, 2, ... , 101, 102, ...

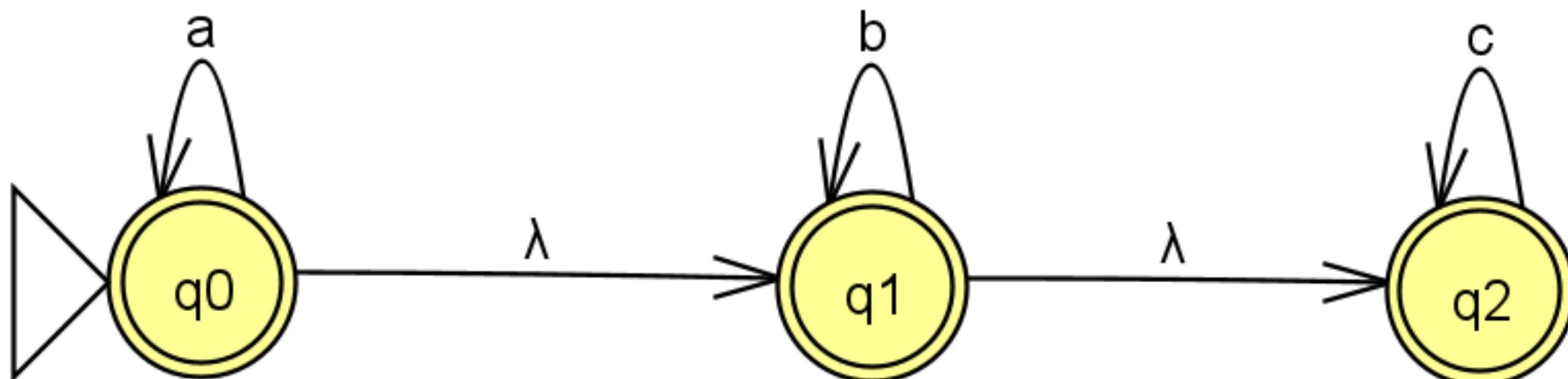


**Example 3: Construct  $\lambda$ -NFA for the language  $L=\{ a^n b^m c^p \mid n, m, p \geq 0 \}$**

**Solution:**  $L=\{\varepsilon, a, aa, aaa, aaaa, \dots, b, bb, bbb, bbbb, \dots, c, cc, ccc, cccc, \dots, ab, bc, ac, \dots, abc, \dots\}$

**Example 3: Construct  $\lambda$ -NFA for the language  $L=\{ a^n b^m c^p \mid n, m, p \geq 0 \}$**

**Solution:**  $L=\{\epsilon, a, aa, aaa, aaaa, \dots, b, bb, bbb, bbbb, \dots, c, cc, ccc, cccc, \dots, ab, bc, ac, \dots, abc, \dots\}$



- DFA may be cumbersome to specify for complex systems and may even be unknown, but NFA provides a nice way of ‘abstraction of information’.
- Any language that can be accepted by a DFA can also be accepted by an NFA, i.e., Every NFA has an equivalent DFA (accepting the same language).
- DFA's, NFA's, and  $\epsilon$ -NFA's all accept exactly the same set of languages: the regular languages.
- NFA are often more convenient to design than DFA and may have fewer states compared to DFA.
- In real-time, only Deterministic Finite Automata (DFAs) can be implemented.



# THANK YOU

---

**Prakash C O**

Department of Computer Science & Engineering

[coprakash@pes.edu](mailto:coprakash@pes.edu)

- Using subset construction method, every NFA can be translated to a DFA that recognizes the same language. DFAs, and NFAs as well, recognize exactly the set of regular languages.
  
- NFAs have been generalized in multiple ways, e.g.,  $\epsilon$ -NFA, finite-state transducers, pushdown automata, alternating automata,  $\omega$ -automata, and probabilistic automata.

### Note:

- "Deterministic" means "if you put the system in the same situation twice, it is guaranteed to make the same choice both times".
- "Non-deterministic" means "not deterministic", or in other words, "if you put the system in the same situation twice, it might or might not make the same choice both times".
- A non-deterministic finite automaton (NFA) can have multiple transitions out of a state. This means there are multiple options for what it could do in that situation. It is not forced to always choose the same one; on one input, it might choose the first transition, and on another input it might choose the same transition.
- Here you can think of "situation" as "what state the NFA is in, together with what symbol is being read next from the input". Even when both of those are the same, a NFA still might have multiple matching transitions that can be taken out of that state, and it can choose arbitrarily which one to take. In contrast, a DFA only has one matching transition that can be taken in that situation, so it has no choice -- it will always follow the same transition whenever it is in that.



# Automata Formal Languages & Logic

---

**Preet Kanwal**

Department of Computer Science & Engineering

# Automata Formal Languages & Logic

---

## Unit 1

**Preet Kanwal**

Department of Computer Science & Engineering

### NFA $\rightarrow$ DFA (Subset Construction)

- The initial state is the start state, plus all states reachable from the start state via  $\lambda$ -transitions (Called  $\lambda$ -closure).
- Transition from a state  $S$  on character  $a$  is found by following all possible transitions on  $a$  for each state in  $S$ , then taking the set of states reachable from there by  $\lambda$ -transitions.
- Accepting states are any set of states where some state in the set is an accepting state.

### NFA → DFA (Subset Construction)

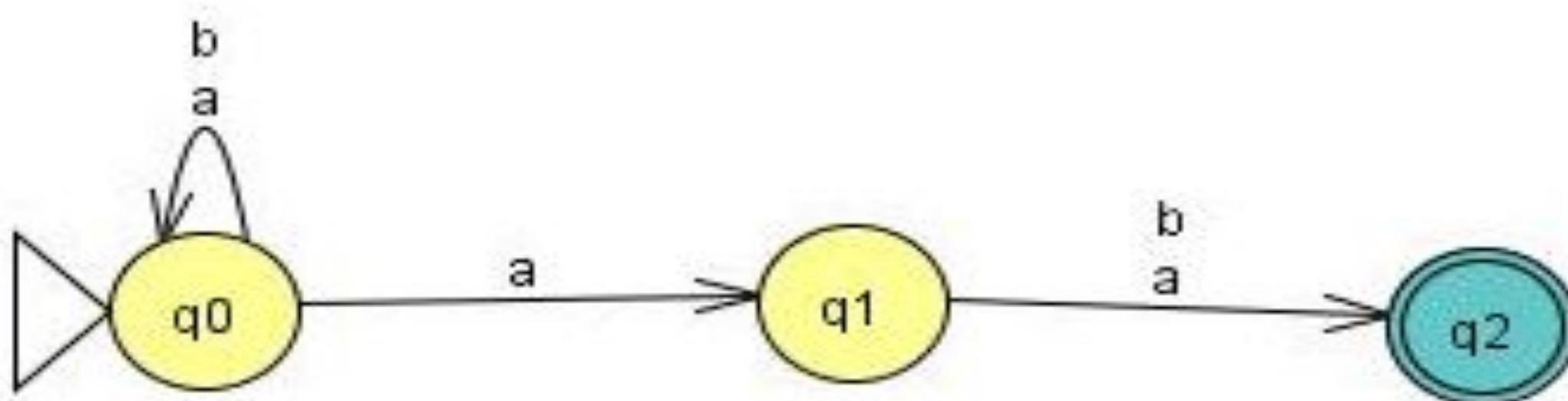
- In converting an NFA to a DFA, the DFA's states correspond to set of NFA states.
- In the worst-case, the construction can result in a DFA that is exponentially larger than the original NFA .
- Minimization of a DFA ensures that the resulting DFA (after minimization) has the least possible states. The advantages of having a minimal DFA are: Faster Execution: The more the number of states the more time the DFA will take to process a string, hence minimization ensures faster execution.

### NFA $\rightarrow$ DFA (Subset Construction)

Given an NFA with states  $Q$ , inputs  $\Sigma$ , transition function  $\delta_N$ , start state  $q_0$ , and final states  $F$ , construct equivalent DFA with:

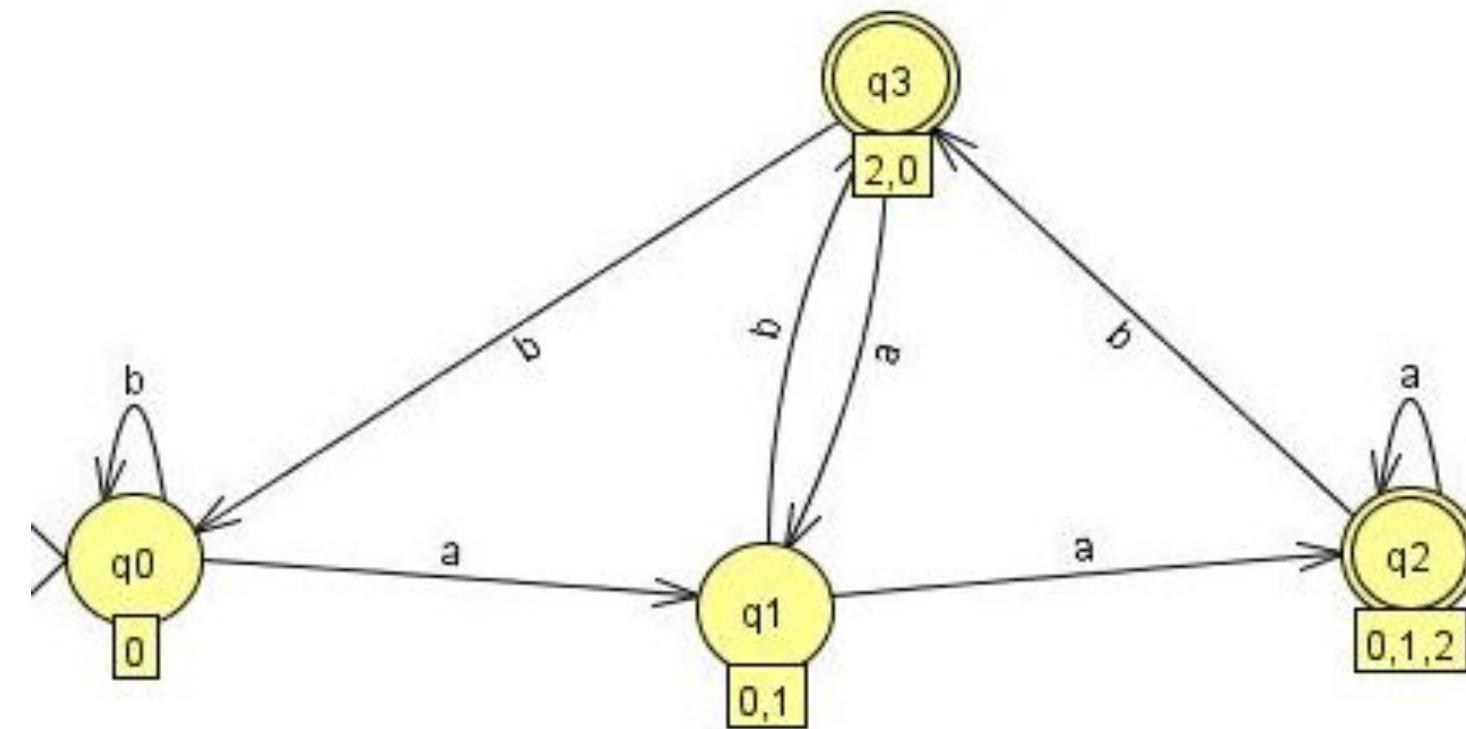
- \* States  $2^Q$  (Set of subsets of  $Q$ ).
- \* Inputs  $\Sigma$ .
- \* Start state  $\{q_0\}$ .
- \* Final states = all those with a member of  $F$ .
- The DFA states have names that are sets of NFA states.
- But as a DFA state, an expression like  $\{p,q\}$  must be read as a single symbol, not as a set.

**Example 1:** Convert the following NFA,  $\Sigma = \{a, b\}$ ,  $L = \{\text{Strings where the second symbol from RHS is } a\}$  to DFA.



### Solution:

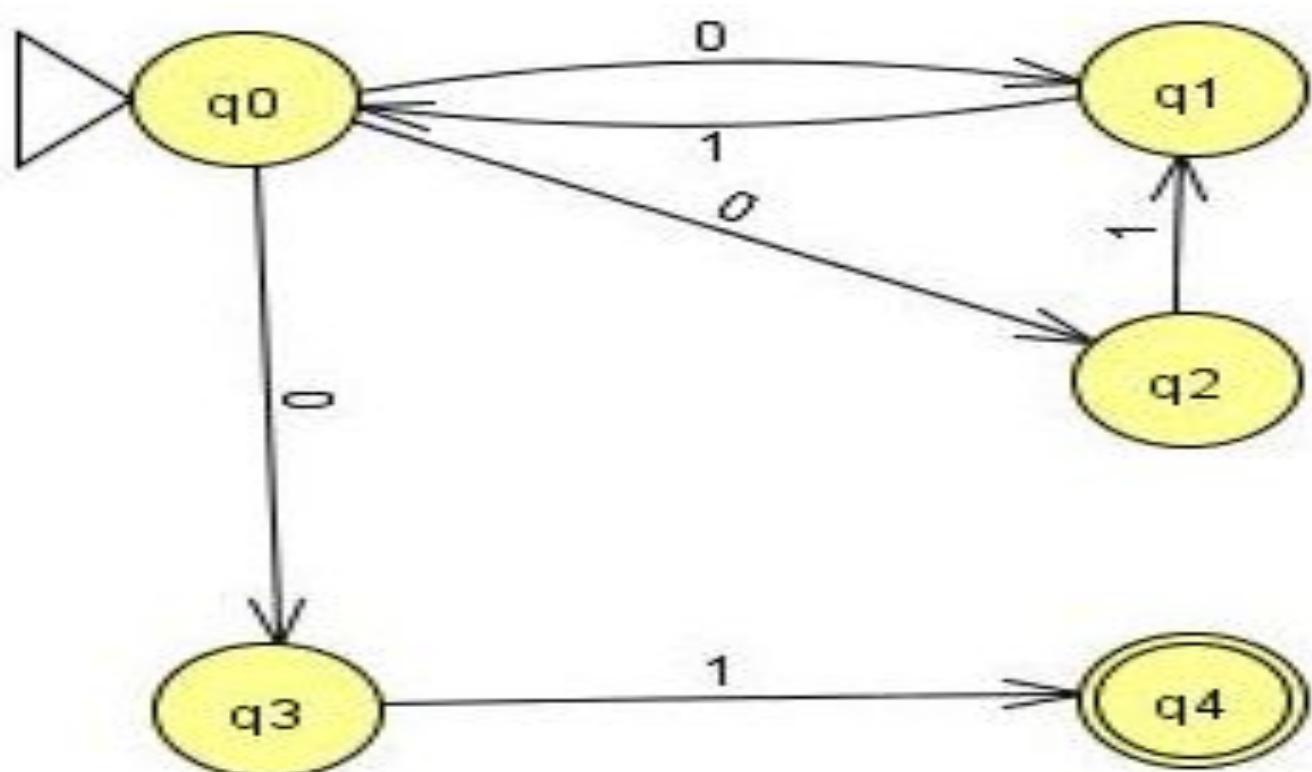
|            | a          | b       |
|------------|------------|---------|
| q0         | {q0,q1}    | q0      |
| {q0q1}     | {q0,q1,q2} | {q0,q2} |
| {q0,q1,q2} | {q0,q1,q2} | {q0,q2} |
| {q0,q2}    | {q0q1}     | q0      |



States= [{q0}, {q0q1},{q0q1q2},{q1q2}]

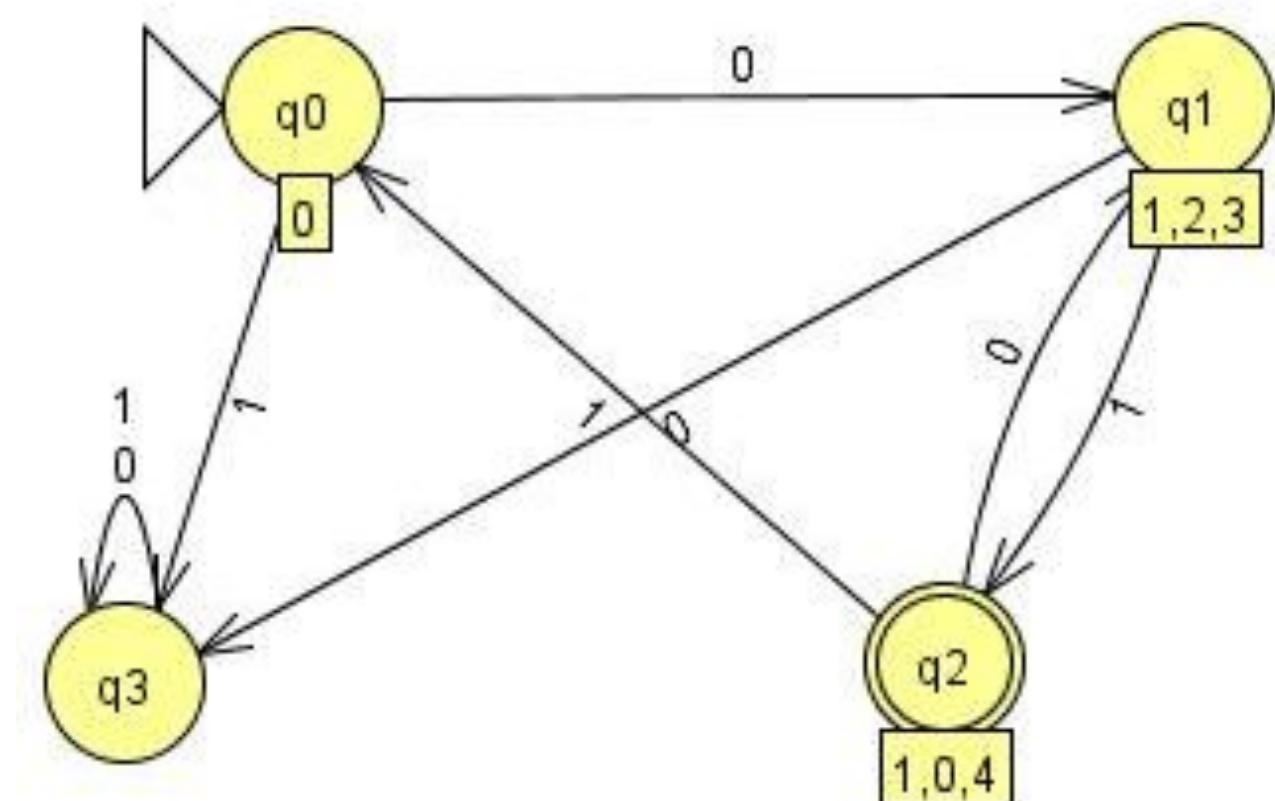
Can rename the states as q0=q0, {q0q1}=q1, {q0,q1,q2}=q2, {q0,q2}=q3

Example 2: Convert the following NFA to DFA.



### Solution:

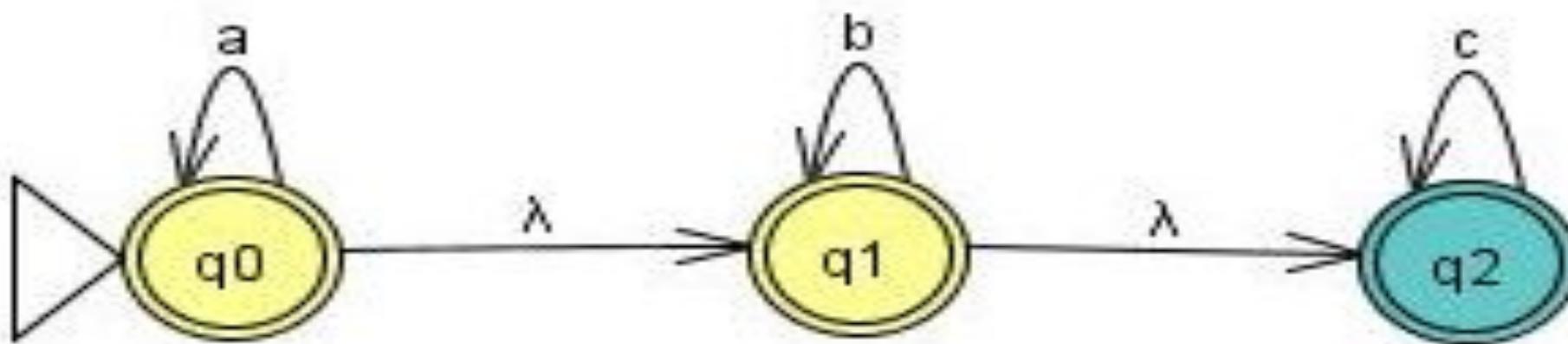
|             | 0          | 1          |
|-------------|------------|------------|
| ---> q0     | {q1,q2,q3} | $\Phi$     |
| {q1,q2,q3}  | $\Phi$     | {q0,q1,q4} |
| *{q0,q1,q4} | {q1,q2,q3} | q0         |
| $\Phi$      | $\Phi$     | $\Phi$     |



States:  $\{\{q0\}, \{q0q1q2\}, \{q0q1q4\}, \Phi\}$

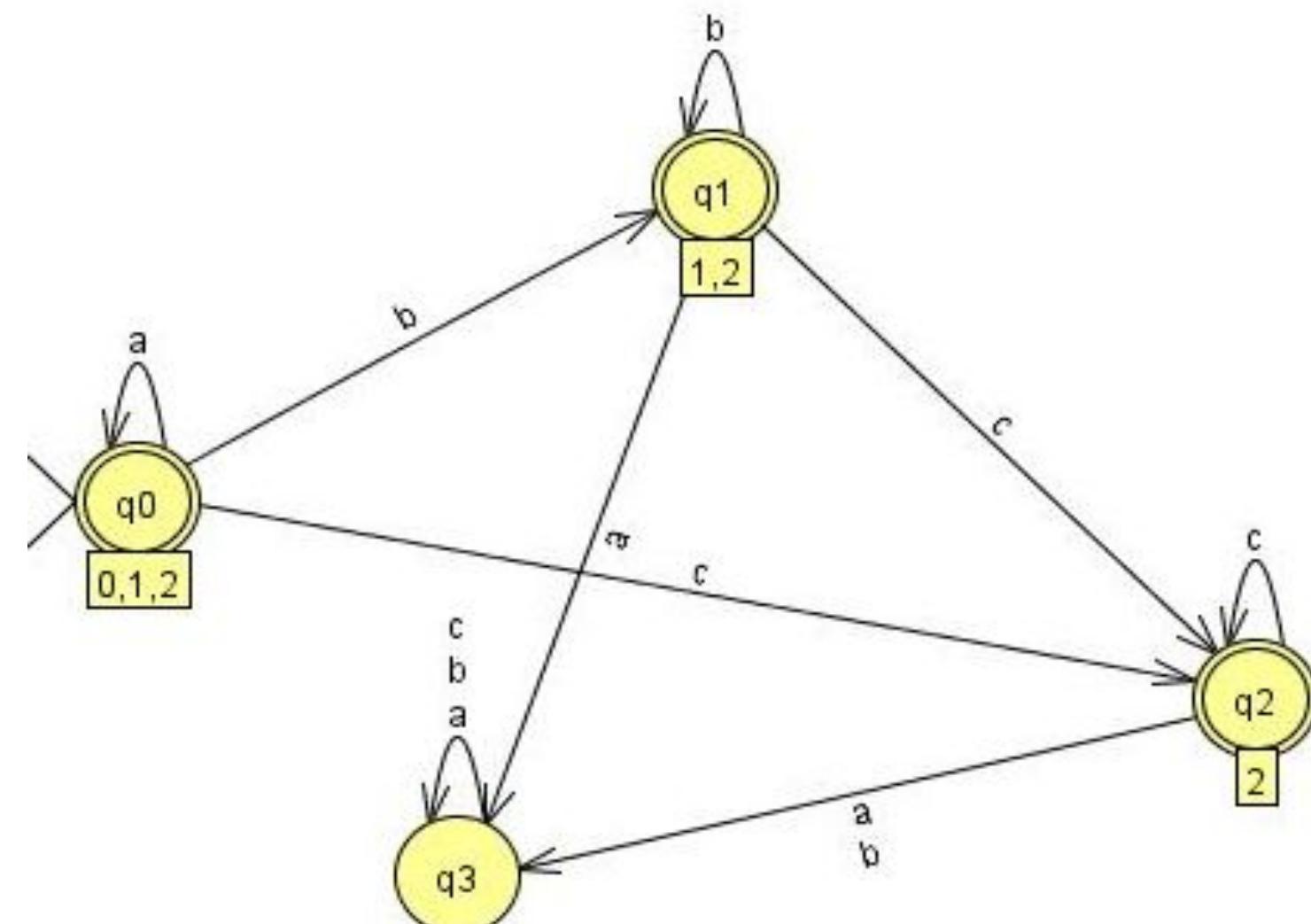
Rename the states as  $q0 = q0, q0q1q2 = q1, q0q1q4 = q2, \Phi = q3$

Example 3: Convert the following  $\lambda$ -NFA  $L=\{a^n b^m c^k, n,m,k \geq 0\}$  to DFA.



### Solution:

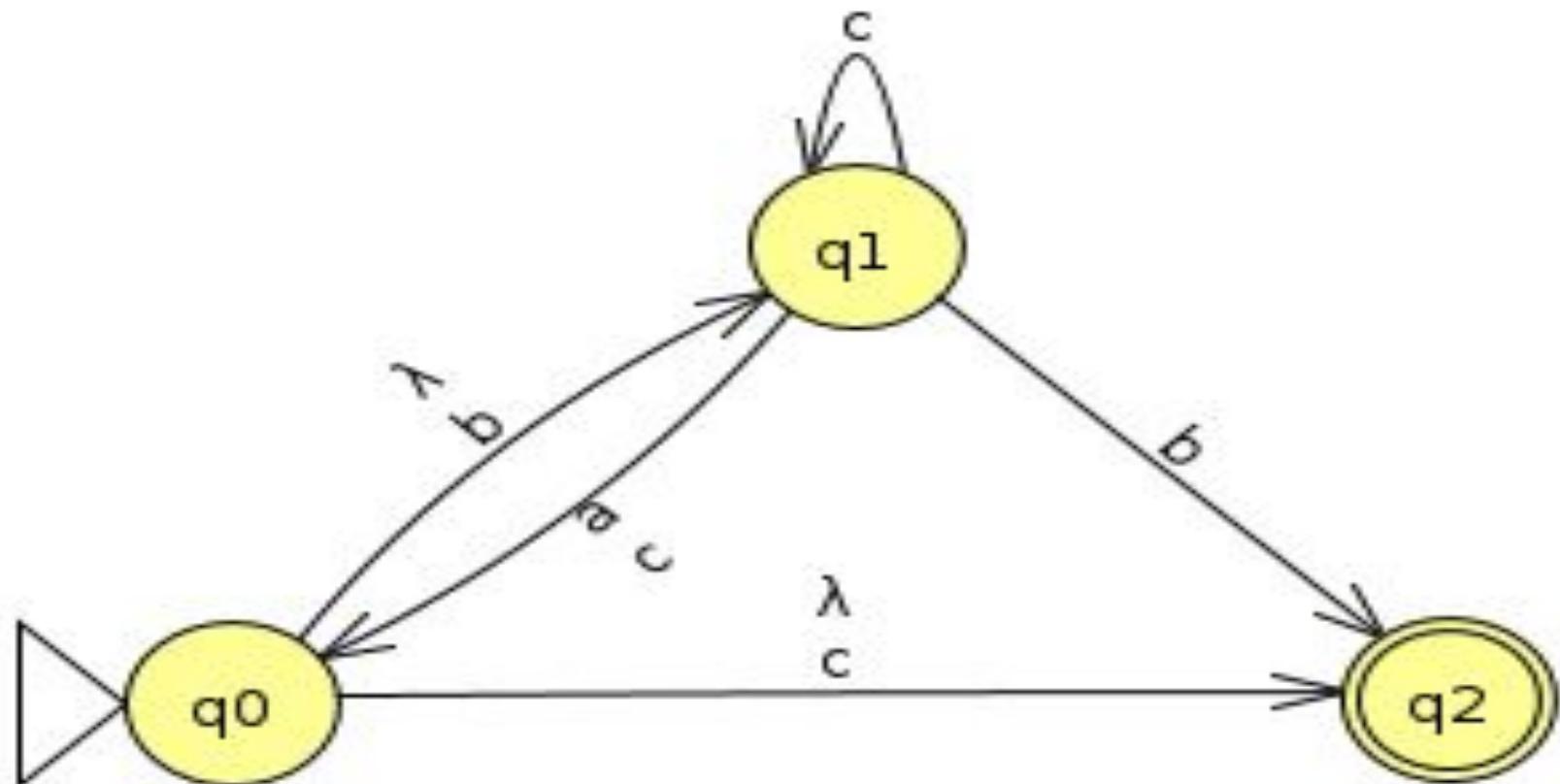
|                                 | a                 | b             | c      |
|---------------------------------|-------------------|---------------|--------|
| $\rightarrow^* \{q_0 q_1 q_2\}$ | $\{q_0 q_1 q_2\}$ | $\{q_1 q_2\}$ | $q_2$  |
| $*\{q_1 q_2\}$                  | $\Phi$            | $\{q_1 q_2\}$ | $q_2$  |
| $*q_2$                          | $\Phi$            | $\Phi$        | $q_2$  |
| $\Phi$                          | $\Phi$            | $\Phi$        | $\Phi$ |



Rename the states as:

$\{q_0 q_1 q_2\}$  as  $q_0$ ,  $\{q_1 q_2\}$  as  $q_1$ ,  $q_2$  as

Example 4: Convert the following  $\lambda$ -NFA to DFA.

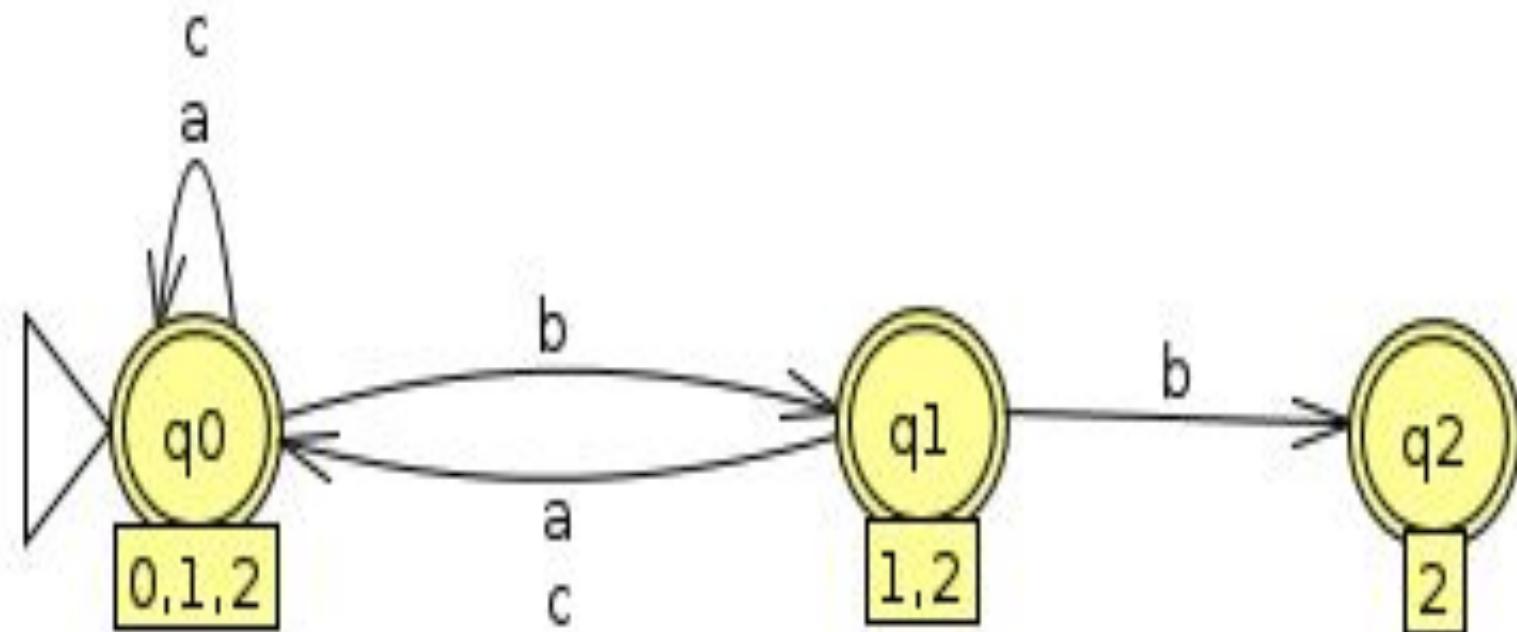


### Solution:

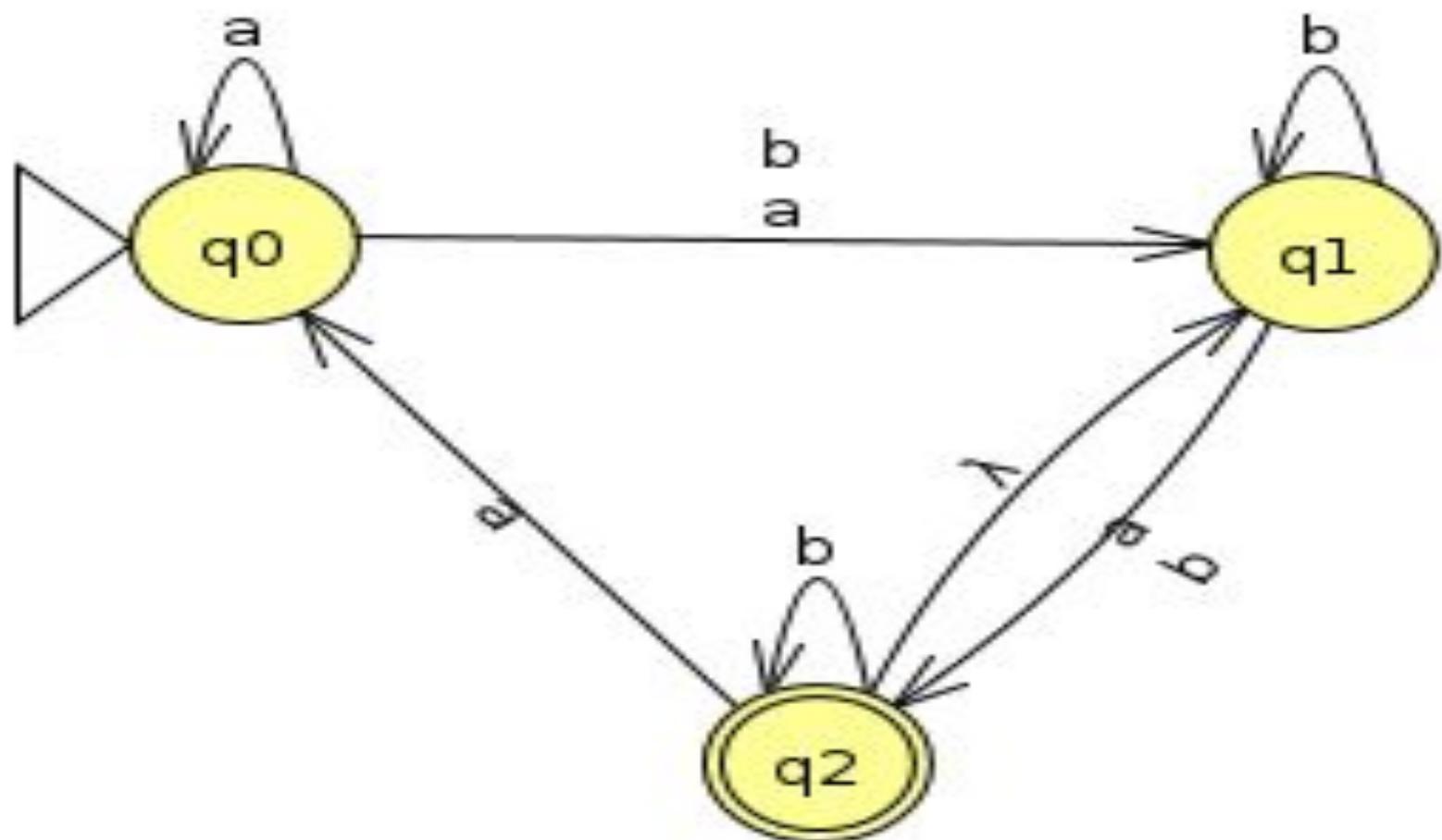
Transition Table

|         | a                                   | b        | c      |
|---------|-------------------------------------|----------|--------|
| ->      | *{q0q1q2}                           | {q0q1q2} | {q1q2} |
| *{q1q2} | {q0q1q2}                            | q2       | $\Phi$ |
| *q2     | $\Phi$                              | $\Phi$   | $\Phi$ |
| $\Phi$  | <input checked="" type="checkbox"/> | $\Phi$   | $\Phi$ |

DFA



Example 5: Convert the following  $\lambda$ -NFA to DFA.

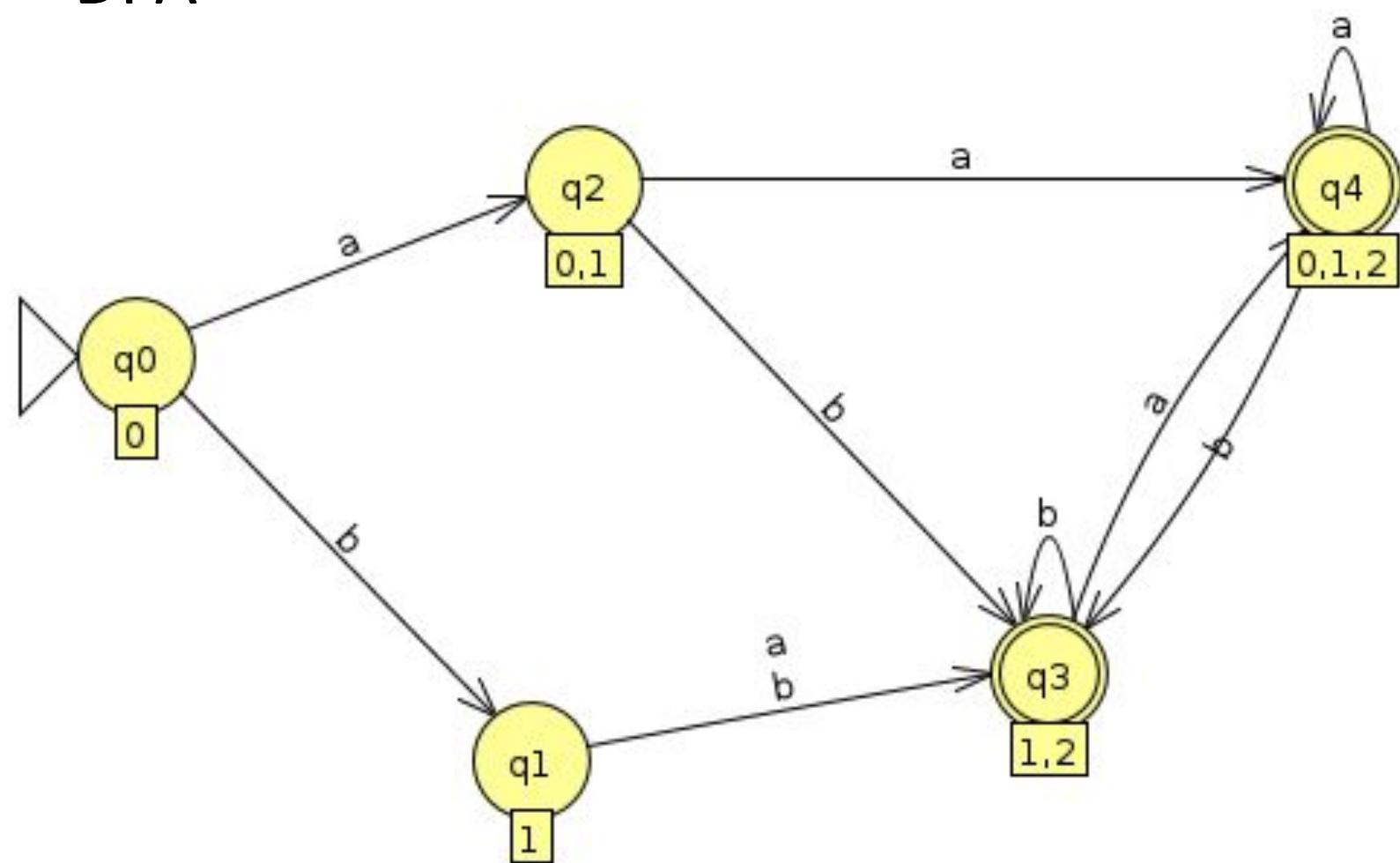


### Solution:

Transition Table

|           | a        | b      |
|-----------|----------|--------|
| ->q0      | {q0q1}   | q1     |
| q1        | {q1q2}   | {q1q2} |
| {q0q1}    | {q0q1q2} | {q0q2} |
| *{q1q2}   | {q0q1q2} | {q1q2} |
| *{q0q1q2} | {q0q1q2} | {q1q2} |

DFA





**THANK YOU**

---

**Preet Kanwal**

Department of Computer Science & Engineering

**[preetkanwal@pes.edu](mailto:preetkanwal@pes.edu)**

**+91 80 6666 3333 Extn 724**



# Automata Formal Languages & Logic

---

**Preet Kanwal**

Department of Computer Science & Engineering

# Automata Formal Languages & Logic

---

## Unit 1

**Preet Kanwal**

Department of Computer Science & Engineering

### Minimization of DFA (Table Filling Algorithm)

- In converting an NFA to a DFA, the DFA's states correspond to set of NFA states.
- In the worst-case, the construction can result in a DFA that is exponentially larger than the original NFA .
- Minimization of a DFA ensures that the resulting DFA (after minimization) has the least possible states.
- The advantages of having a minimal DFA are: Faster Execution: The more the number of states the more time the DFA will take to process a string, hence minimization ensures faster execution.

### Minimization of DFA (Table Filling Algorithm)

**Step I : Eliminate Unreachable States**

**Step II : Mark Distinguishable Pair of States  
(Final, Non-Final States)**

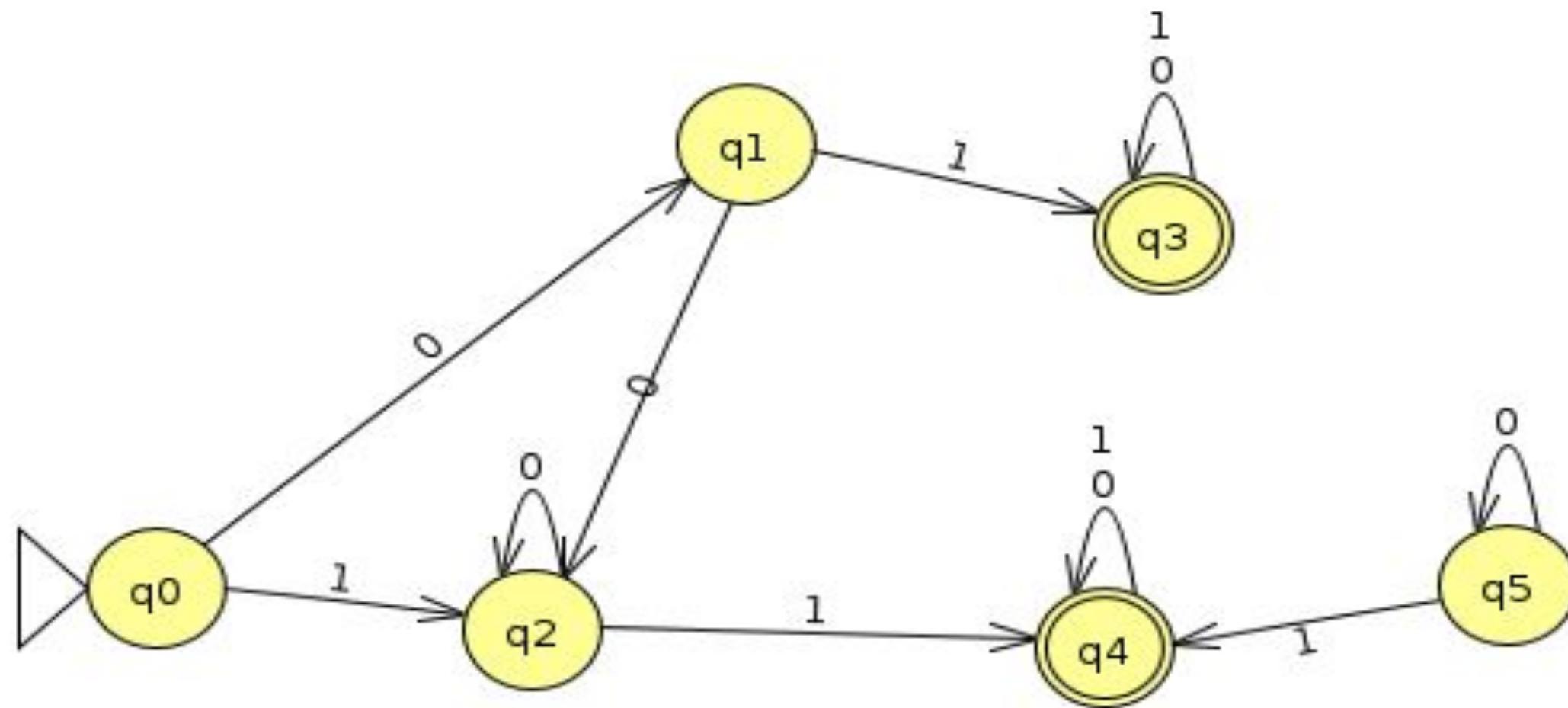
**Step III : If there are any Unmarked pairs (M, N) make  
the following Check :**

if  $\delta(M, a) = X$  and  $\delta(N, a) = Y \& \& (X, Y)$  is marked then,  
Mark (M, N) also.

Repeat Step III until no new states can be marked.

**Step IV : Combine all the unmarked pairs and make them a single state in the  
minimized DFA.**

Example 1: Minimize the following DFA.

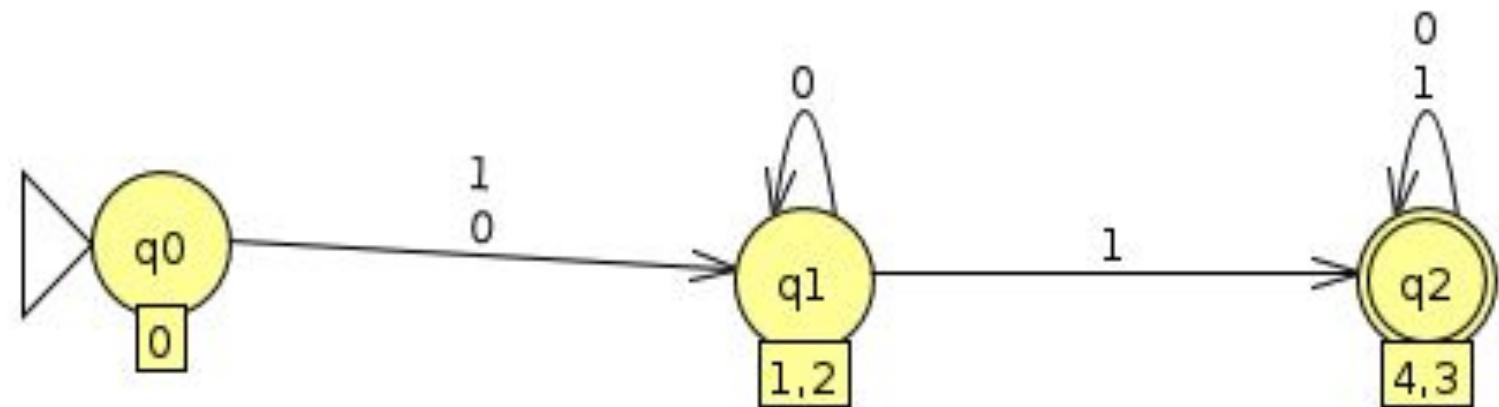


### Solution:

Final Table

|             |           |           |             |
|-------------|-----------|-----------|-------------|
| <b>q1</b>   | X         |           |             |
| <b>q2</b>   | X         |           |             |
| * <b>q3</b> | X         | X         | X           |
| * <b>q4</b> | X         | X         | X           |
|             | <b>q0</b> | <b>q1</b> | <b>q2</b>   |
|             |           |           | * <b>q3</b> |

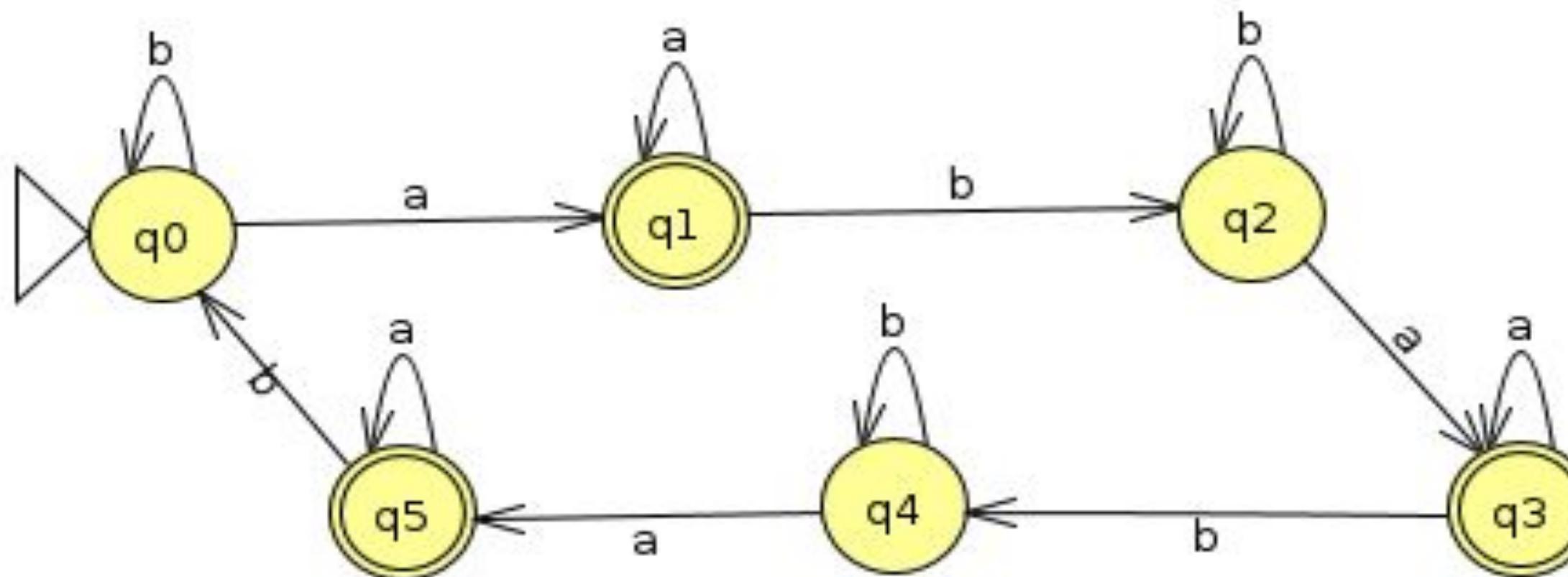
Minimized DFA



States merged :

- {q1, q2}
- {q3, q4}

Example 2 : Minimize the following DFA.

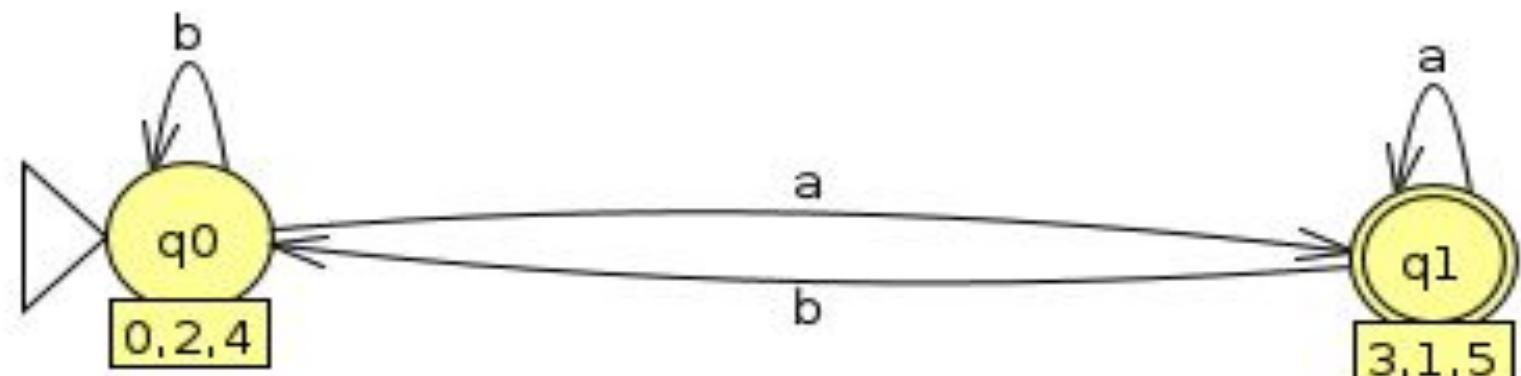


### Solution:

Final Table

|     |    |     |    |     |    |
|-----|----|-----|----|-----|----|
| *q1 | X  |     |    |     |    |
| q2  |    | X   |    |     |    |
| *q3 | X  |     | X  |     |    |
| q4  |    | X   |    | X   |    |
| *q5 | X  |     | X  |     | X  |
|     | q0 | *q1 | q2 | *q3 | q4 |

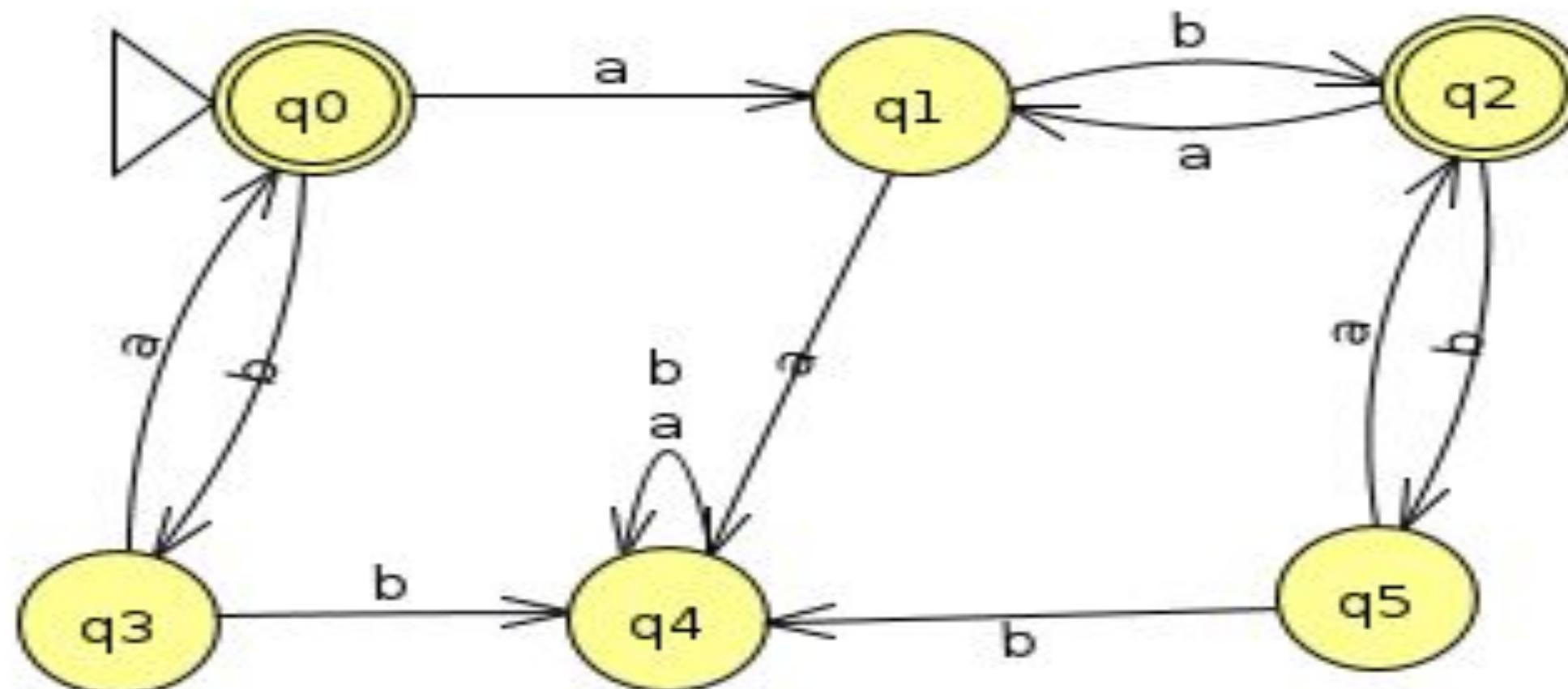
Minimized DFA



States merged :

- {q0,q4}, {q0, q2} and {q2, q4}
- {q1,q5}, {q1,q3} and {q3,q5}

Example 3: Minimize the following DFA.

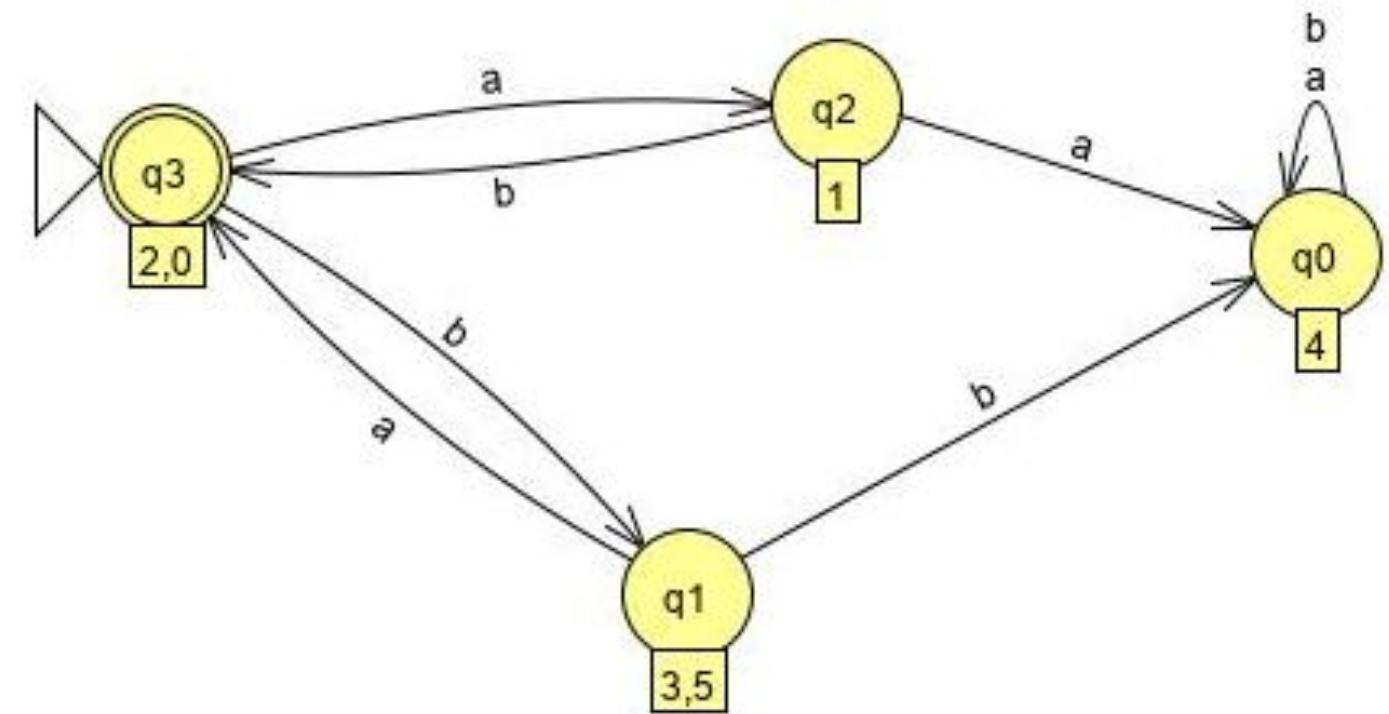


**Solution:**

Minimized DFA

Final Table

|    |    |    |    |    |    |
|----|----|----|----|----|----|
| q1 | X  |    |    |    |    |
| q2 |    | X  |    |    |    |
| q3 | X  | X  | X  |    |    |
| q4 | X  | X  | X  |    |    |
| q5 | X  | X  | X  |    | X  |
|    | q0 | q1 | q2 | q3 | q4 |



States merged :

- {q0, q2}
- {q3, q5}



**THANK YOU**

---

**Preet Kanwal**

Department of Computer Science & Engineering

**[preetkanwal@pes.edu](mailto:preetkanwal@pes.edu)**

**+91 80 6666 3333 Extn 724**



# Automata Formal Languages & Logic

---

**Preet Kanwal**

Department of Computer Science & Engineering

# Automata Formal Languages & Logic

---

## Applications of Finite State Machines

**Preet Kanwal**

Department of Computer Science & Engineering

# Automata Formal Languages and Logic

## Applications of Finite State Machines

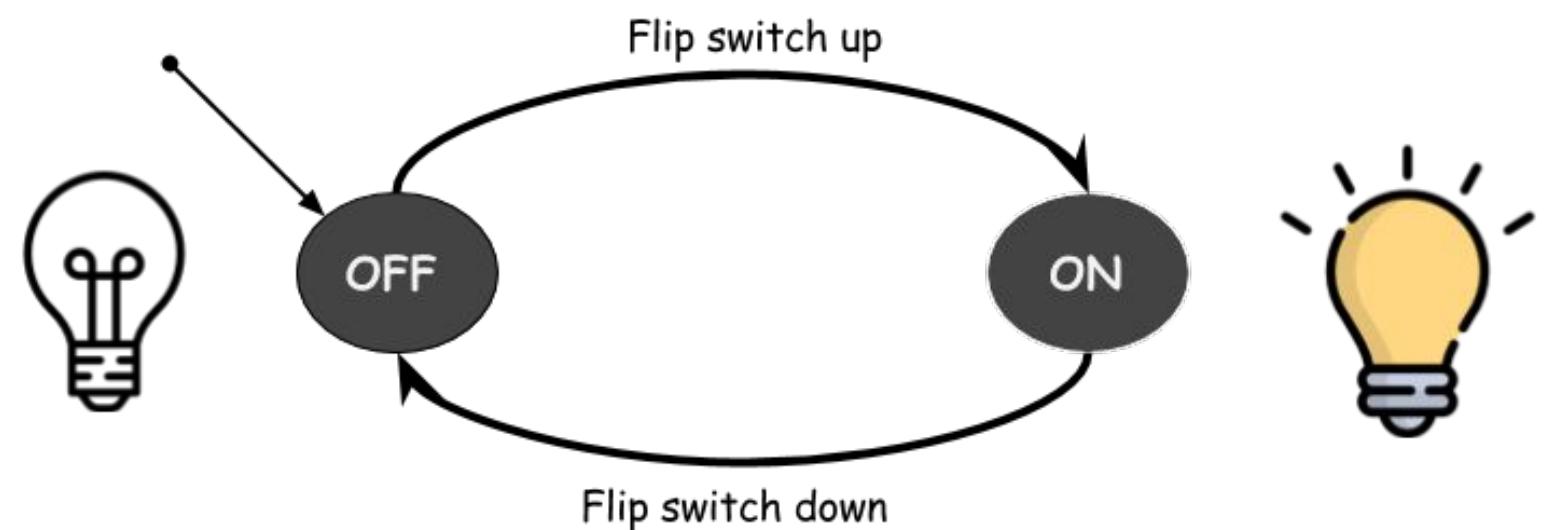
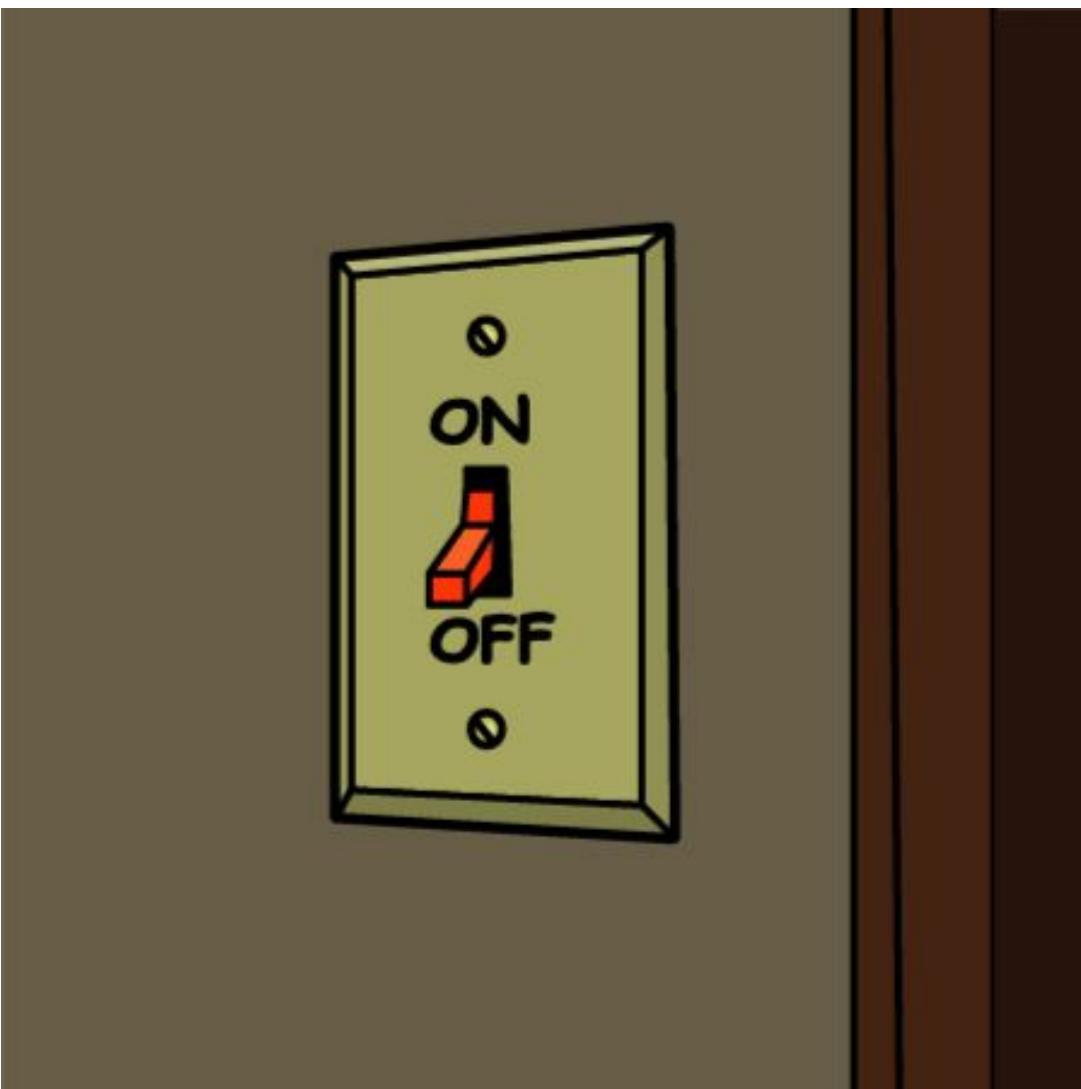
---



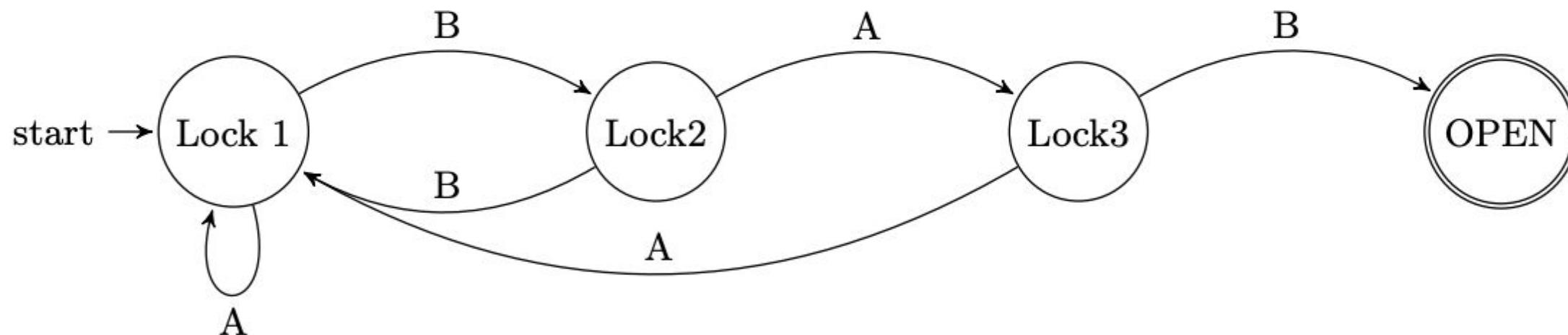
**Finite-State Machines are used in :**

- Text processing
- Compilers
- Network protocols
- Hardware design

### Switch Implementation



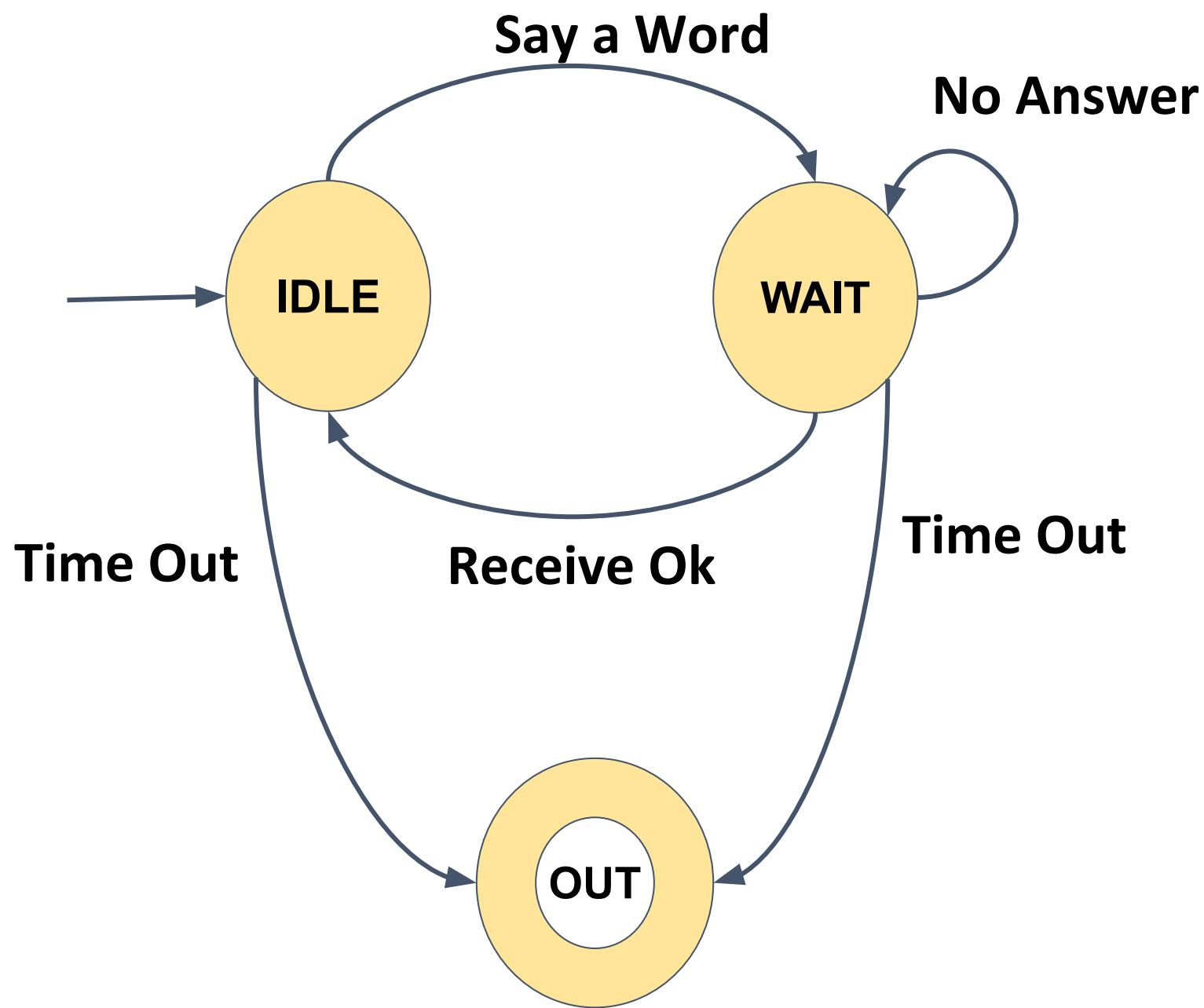
### Code Lock Implementation



# Automata Formal Languages and Logic

## Applications of Finite State Machines

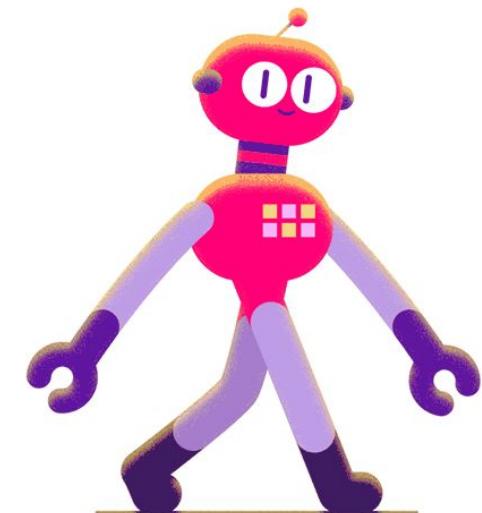
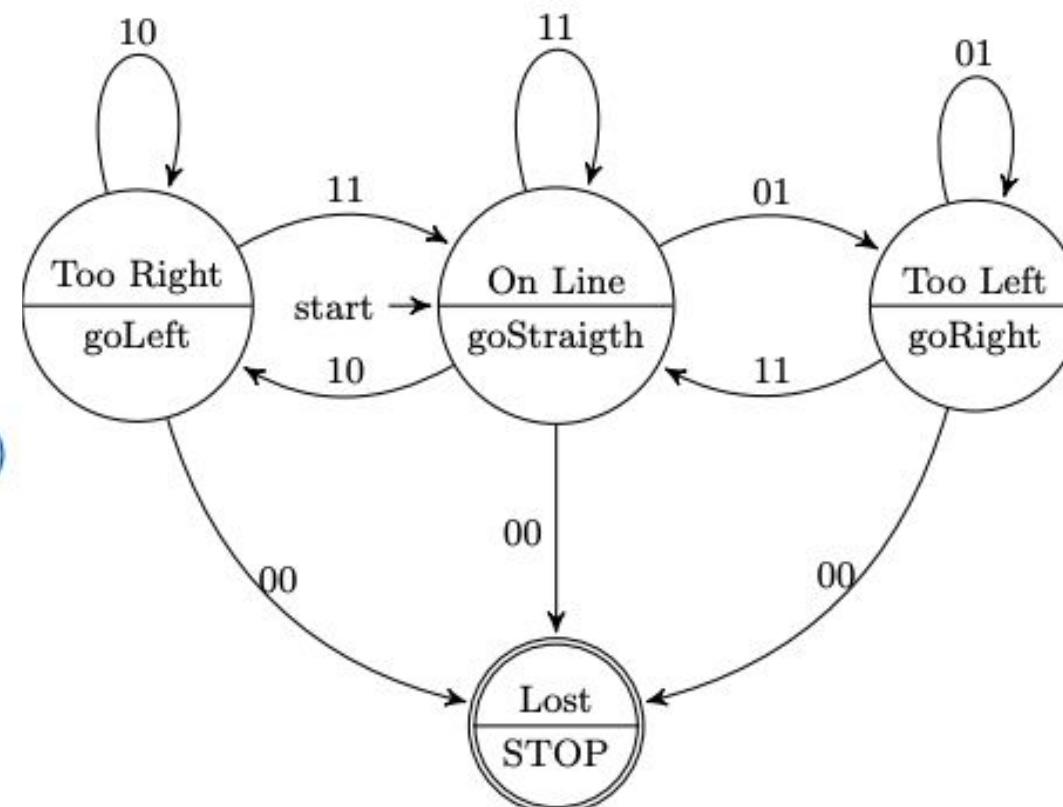
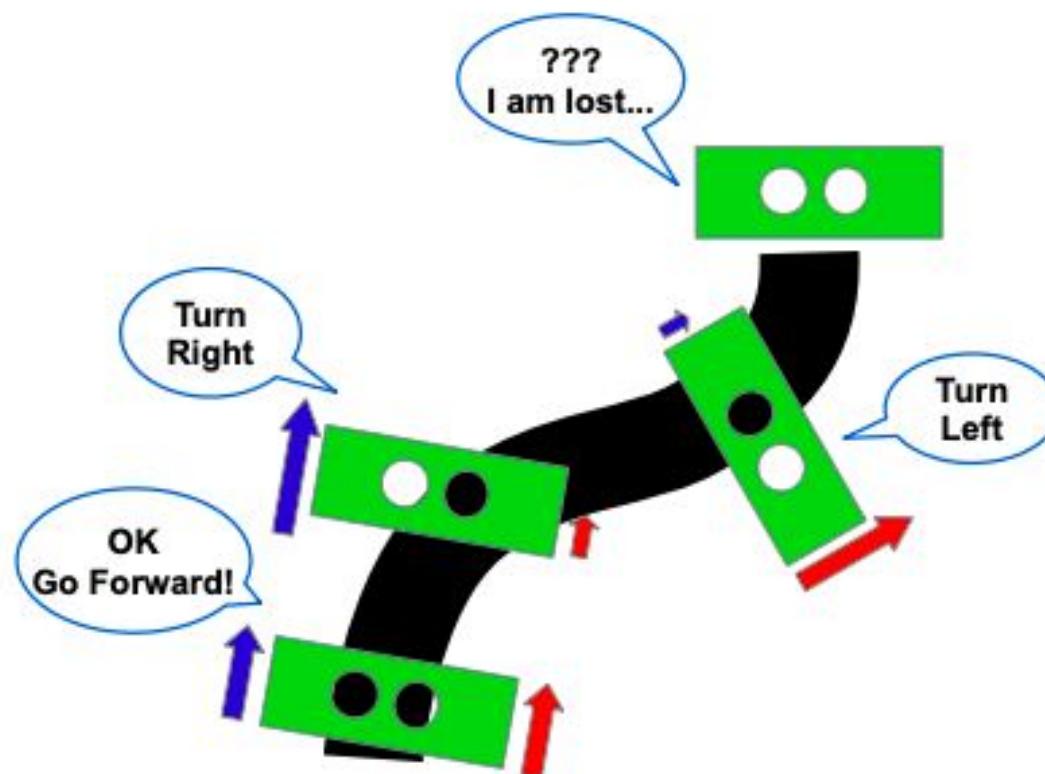
### Communication Link



# Automata Formal Languages and Logic

## Applications of Finite State Machines

### Robotics



## Text Processing / Natural Language Processing

**Spell Checker has two components :**

1. Error Detection (Detection of Misspelled word)
2. Spelling Correction or Suggestion Prediction

# **Automata Formal Languages and Logic**

## **Applications of Finite State Machines**

---



**Error Detection in Spell Checker :**

**Create FSM for all the words in the dictionary**

# Automata Formal Languages and Logic

## Applications of Finite State Machines

---



**Error Detection in Spell Checker :**

Create FSM for all the words in the dictionary.

Let's say the dictionary is made up of the following words:

Henna

Elton

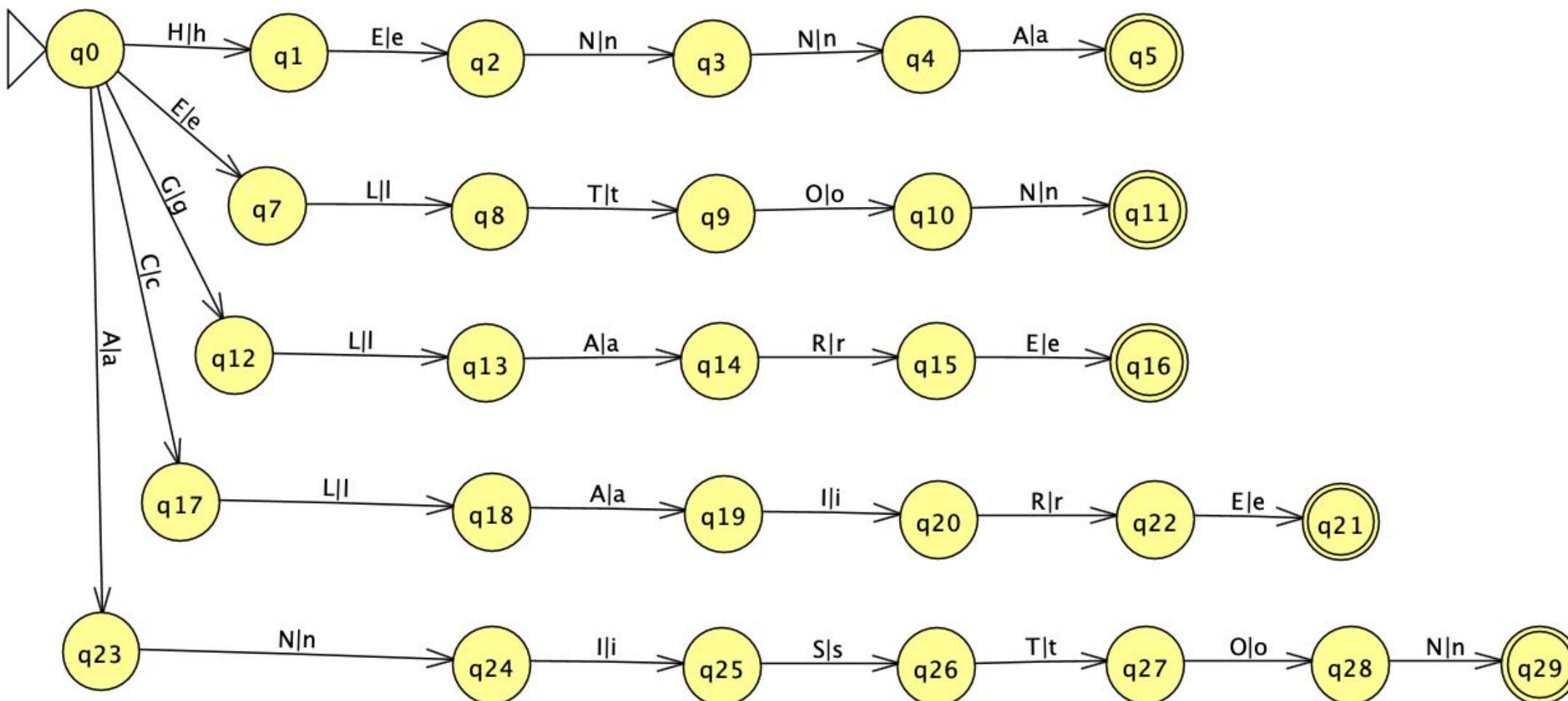
Glare

Claire

Aniston

### Error Detection in Spell Checker :

#### Our FSM will look like this:



### Spelling Correction in Spell Checker :

The word 'Helton' is misspelled as it will not lead us to a final state. We must now provide a suggestion for correction.

We make use of Edit Distance in order to provide a possible correction

### Edit distance between two strings :

- Is the number of editing operations required to transform one into the other
  - Insertion
  - Deletion
  - Substitution
- Various Edit Distance Metrics are available, we will talk about Levenshtein Distance

### Levenshtein Distance

**Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other.**

Levenshtein Distance = 0

|   |   |   |   |
|---|---|---|---|
| T | E | S | T |
| T | E | S | T |

Levenshtein Distance = 2

|   |   |   |   |
|---|---|---|---|
| T | E | S | T |
| T | E | A | M |

**S              S**

# Automata Formal Languages and Logic

## Applications of Finite State Machines



Levenshtein Distance = 2

|   |   |   |   |   |
|---|---|---|---|---|
| P | O | R | T |   |
| P | A | R | T | Y |

**S**                                   **D**

Levenshtein Distance = 2

|   |   |   |   |   |
|---|---|---|---|---|
| G | I |   | L | Y |
| G | E | E | L | Y |

|   |   |   |   |   |
|---|---|---|---|---|
| G |   | I | L | Y |
| G | E | E | L | Y |

Levenshtein Distance = 3

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| H |   | O | N | D | A |   |
| H | Y | U | N | D | A | I |

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| H | O |   | N | D | A |   |
| H | Y | U | N | D | A | I |

# Automata Formal Languages and Logic

## Applications of Finite State Machines



### Spelling Correction in Spell Checker :

Misspelled Word is “Helton”

Let's find Levenshtein Distance between “Helton” and other words in the dictionary:

| Dictionary | L.V from “Helton” |
|------------|-------------------|
| Henna      | 4                 |
| Elton      | 1                 |
| Glare      | 5                 |
| Claire     | 6                 |
| Aniston    | 4                 |

# Automata Formal Languages and Logic

## Applications of Finite State Machines

Spelling Correction in Spell Checker :

Misspelled Word is “Helton”

Hence our suggestion prediction is “Elton”

| Dictionary | L.V from “Helton” |
|------------|-------------------|
| Henna      | 4                 |
| Elton      | 1                 |
| Glare      | 5                 |
| Claire     | 6                 |
| Aniston    | 4                 |



**THANK YOU**

---

**Preet Kanwal**

Department of Computer Science & Engineering

**[preetkanwal@pes.edu](mailto:preetkanwal@pes.edu)**

**+91 80 6666 3333 Extn 724**



# Automata Formal Languages & Logic

---

**Preet Kanwal**

Department of Computer Science & Engineering

# Automata Formal Languages & Logic

---

## Unit 2

**Preet Kanwal**

Department of Computer Science & Engineering

Regex is an algebraic way to describe regular languages.

### Regular Expression

#### Atoms

- A Character/symbol
- $\phi$  ( $L = \{\}$ )
- $\lambda$  ( $L = \{\lambda\}$ )

#### Operators

- $( R )$
- $R^*$
- $R_1.R_2$  or  $R_1R_2$
- $R_1 + R_2$  or  $R_1 | R_2$

### Example

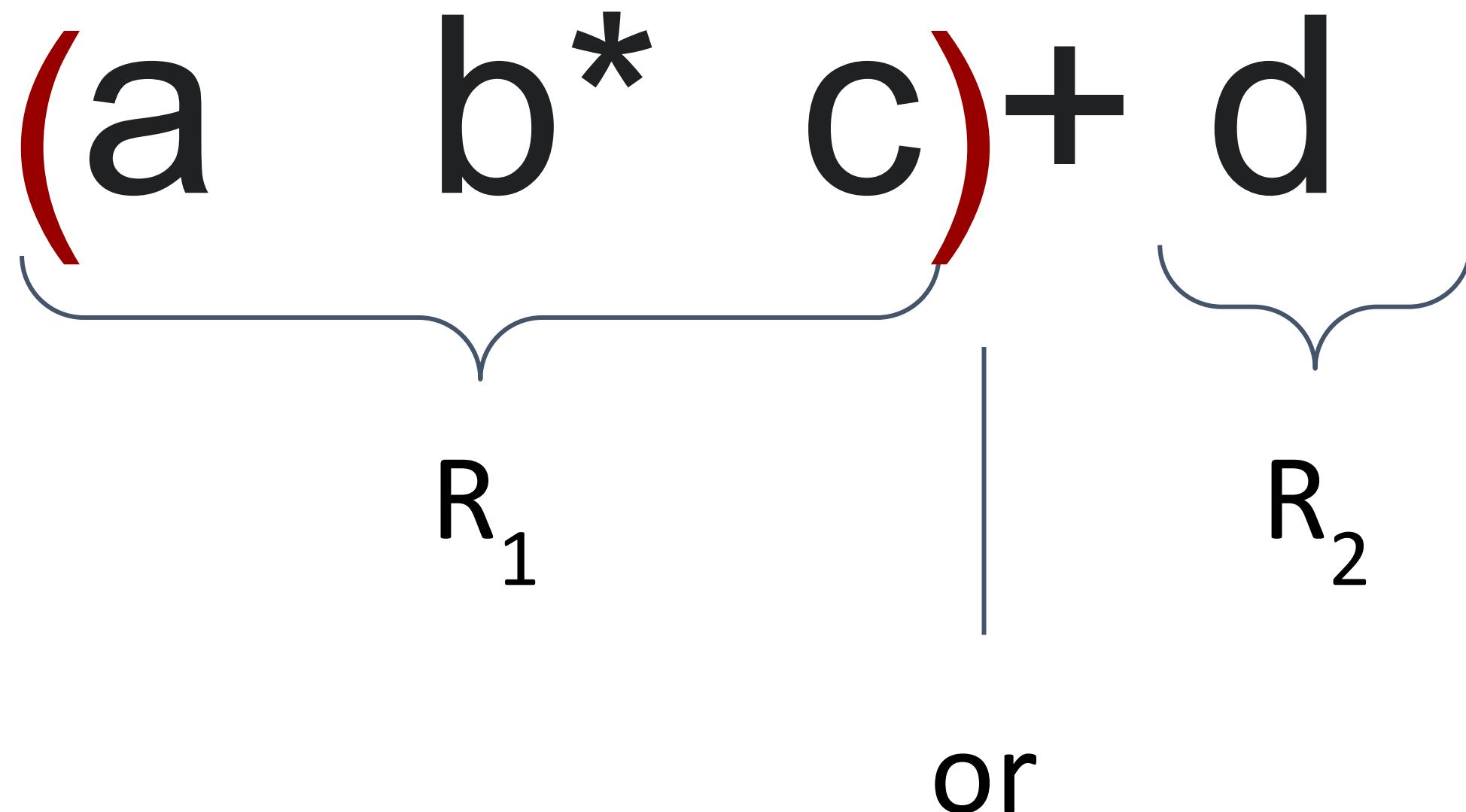
a    b\*    c + d

### Example

$$(a b^* c)^+ d$$

$R_1 \quad | \quad R_2$

or



The diagram illustrates the decomposition of the regular expression  $(a b^* c)^+ d$  into two components,  $R_1$  and  $R_2$ , separated by a vertical line. A red bracket groups the entire expression from the opening parenthesis to the closing parenthesis. Below this bracket, a blue brace labeled  $R_1$  spans from the start of the expression to the closing parenthesis. To the right of the vertical line, another blue brace labeled  $R_2$  spans from the closing parenthesis to the final character  $d$ . The word "or" is positioned below the vertical line, indicating that the expression is the union of  $R_1$  and  $R_2$ .

### Example

$$(a b^* c) + d$$

Starting with  
an 'a'

Followed by 0 or  
more no. of b's

Ending with  
a 'c'

### Example

$$(a b^* c)^+ d$$

Starting with  
an 'a'

Followed by 0 or  
more no. of b's

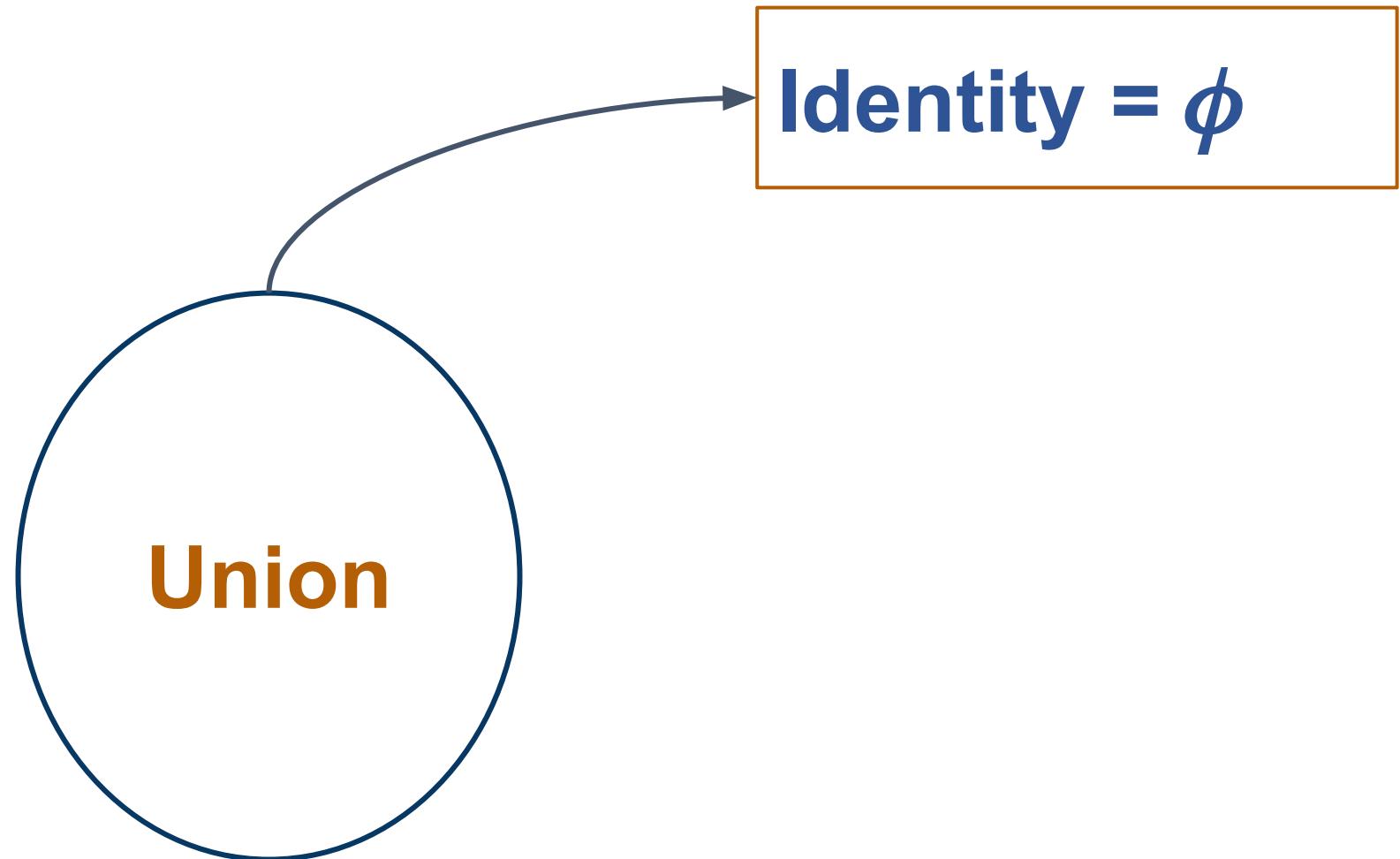
Ending with  
a 'c'

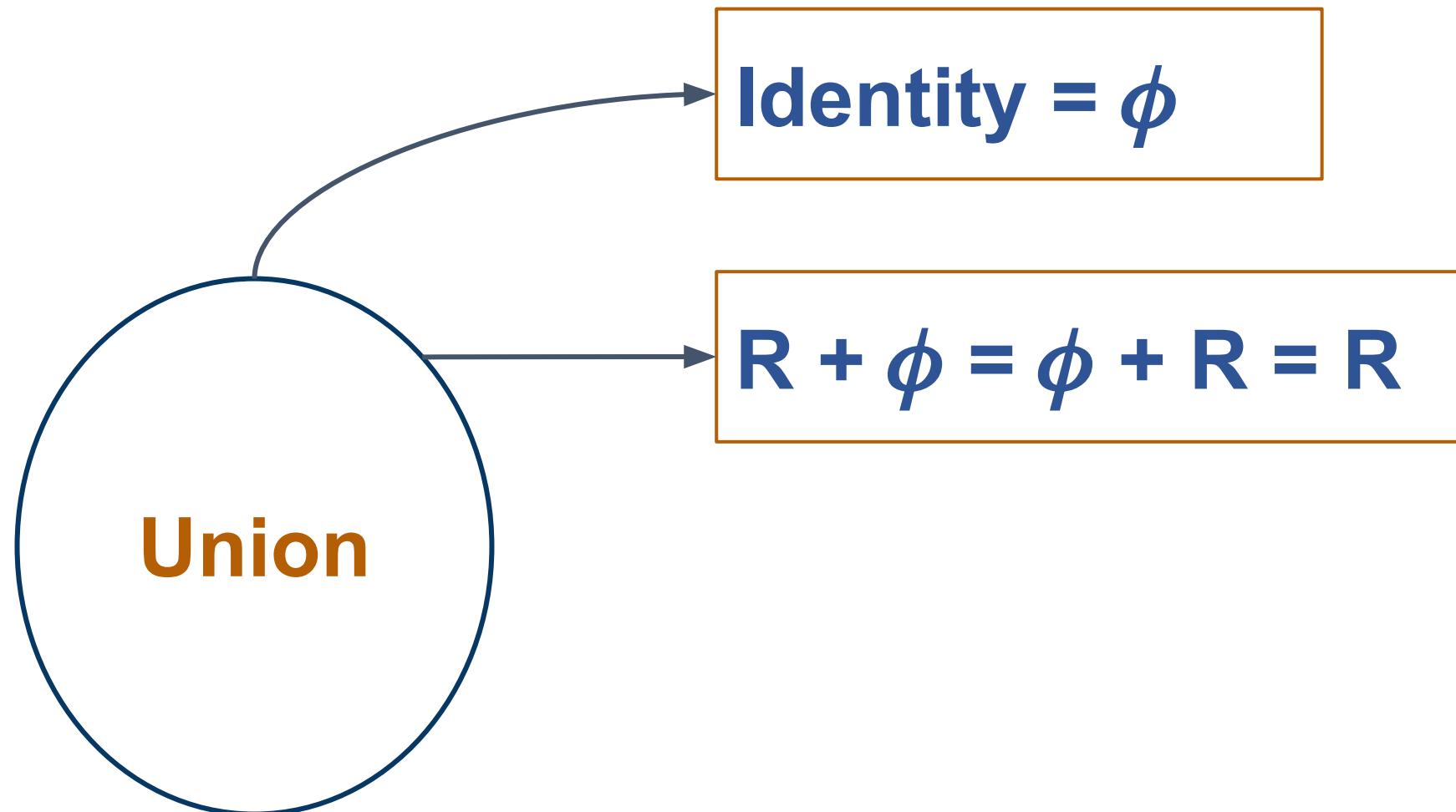
String that  
contains only d

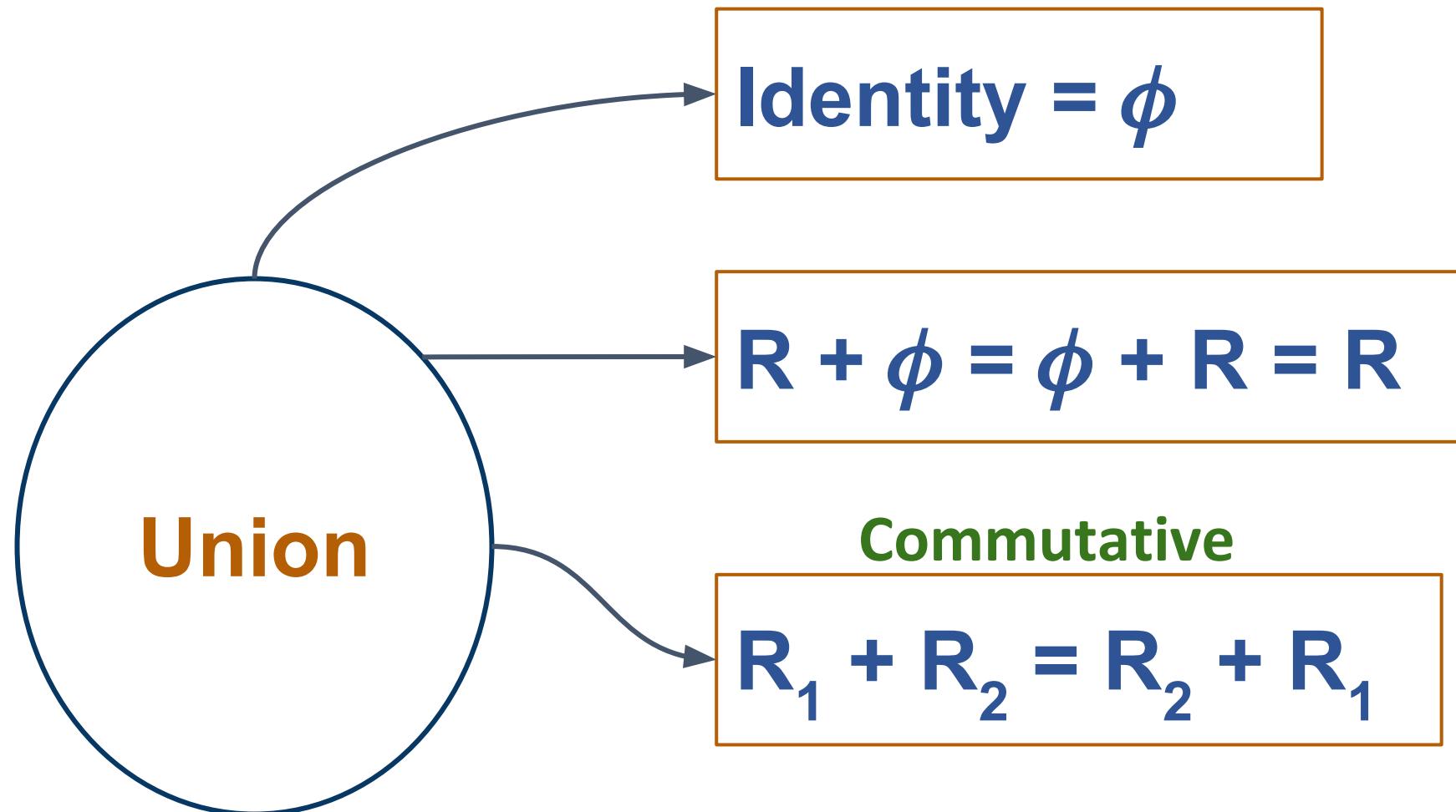
### Example

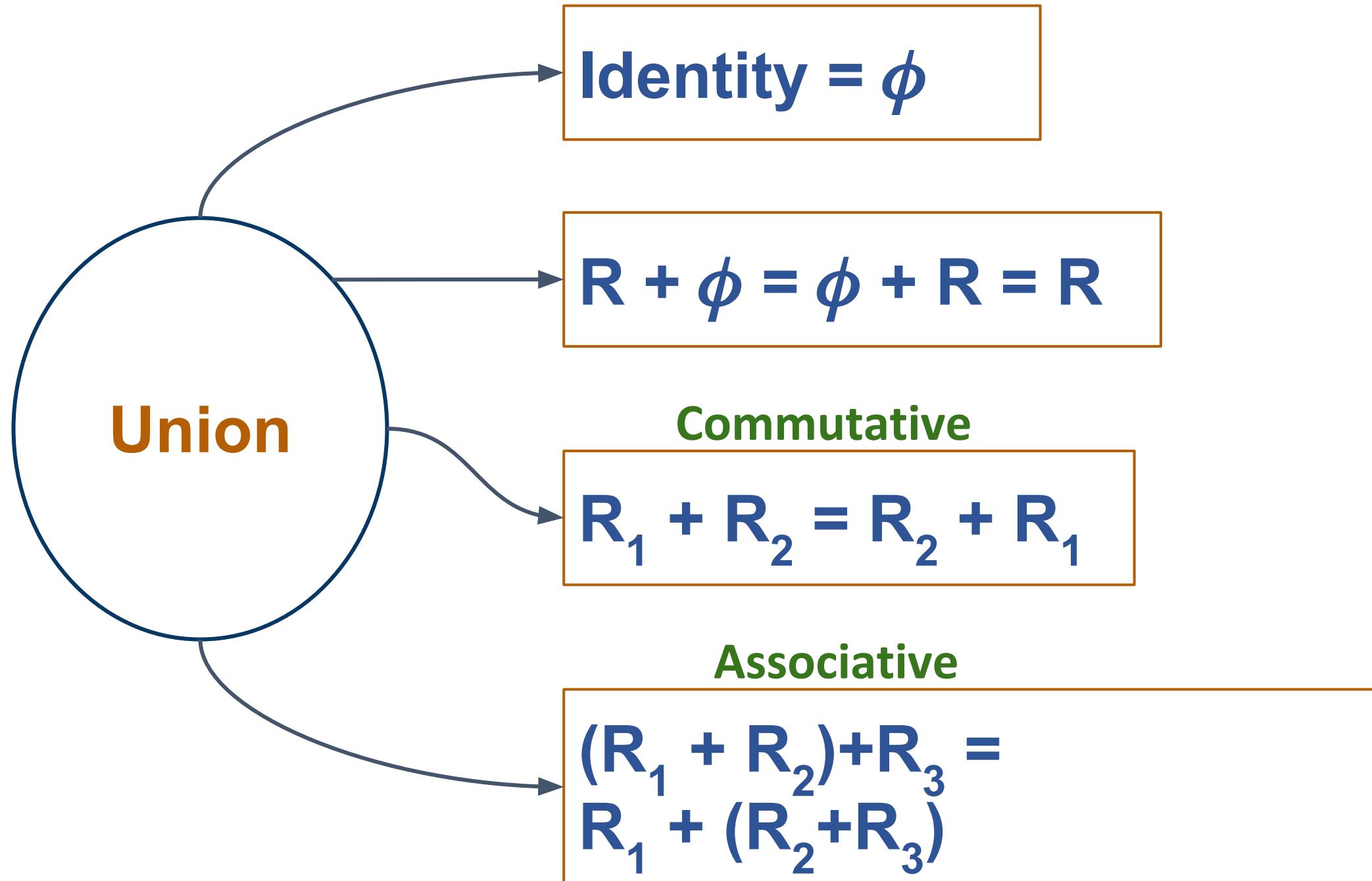
$$(a b^* c)^+ d$$

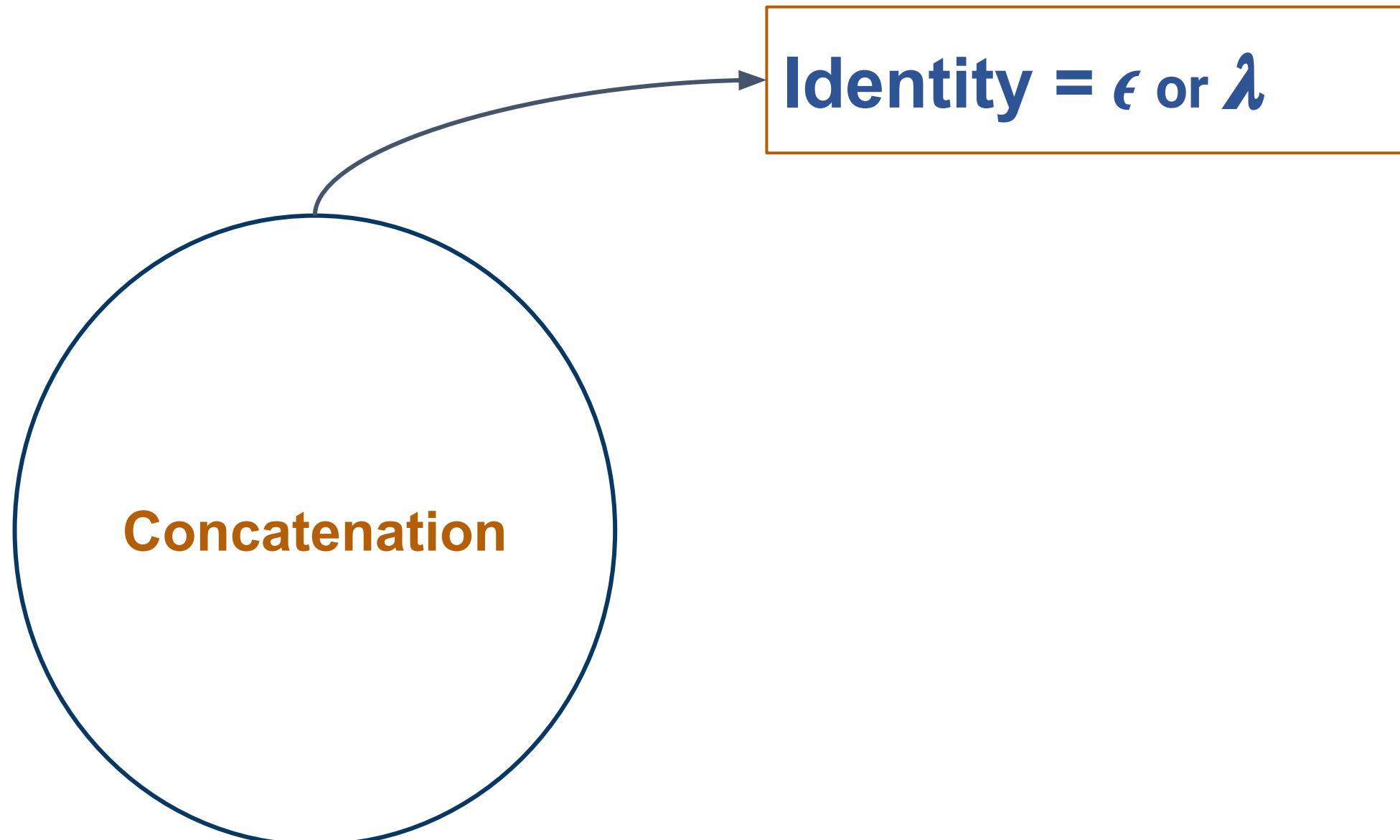
$L = \{ ac,$   
 $abc,$   
 $abbb\dots c,$   
 $d \}$

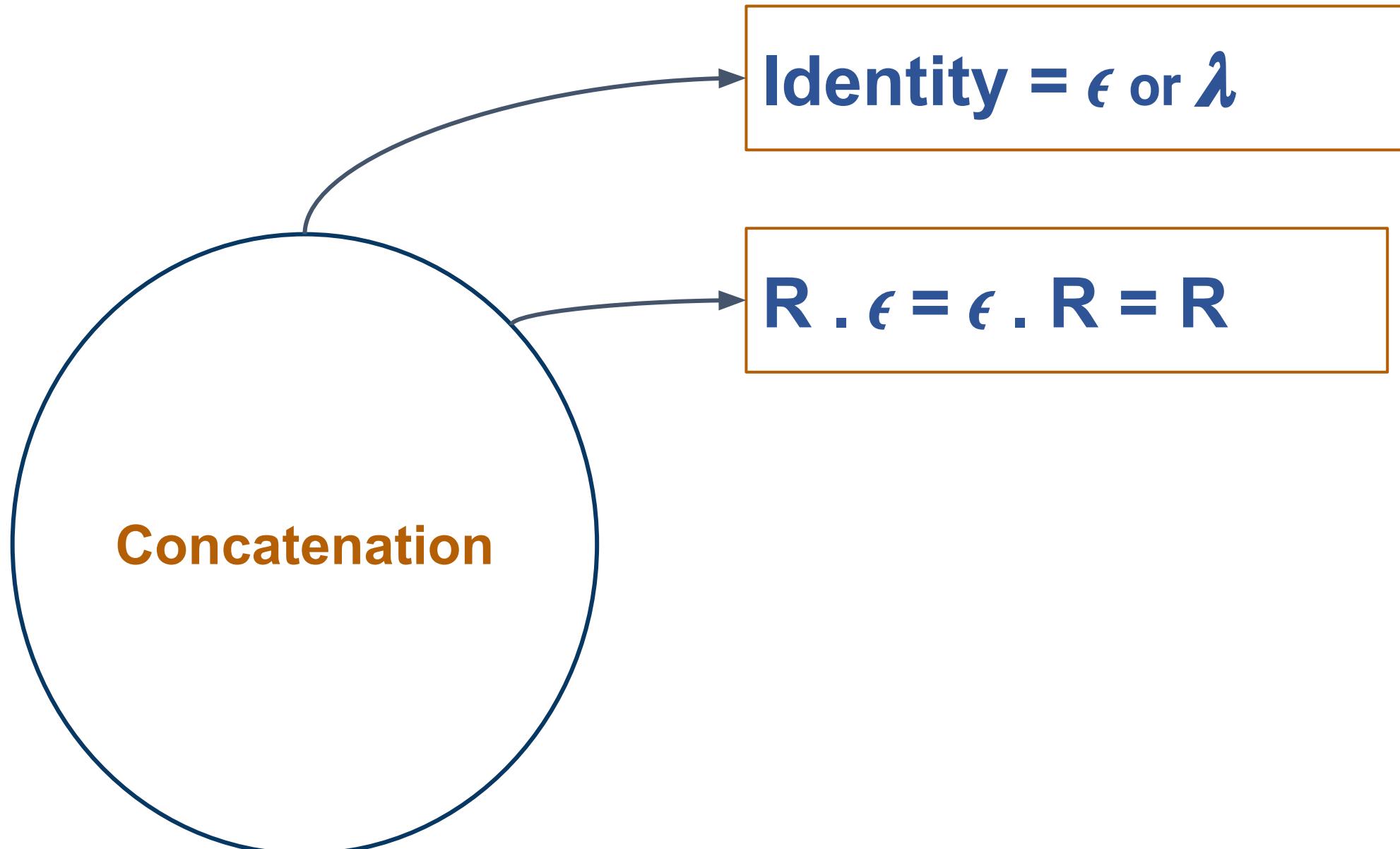






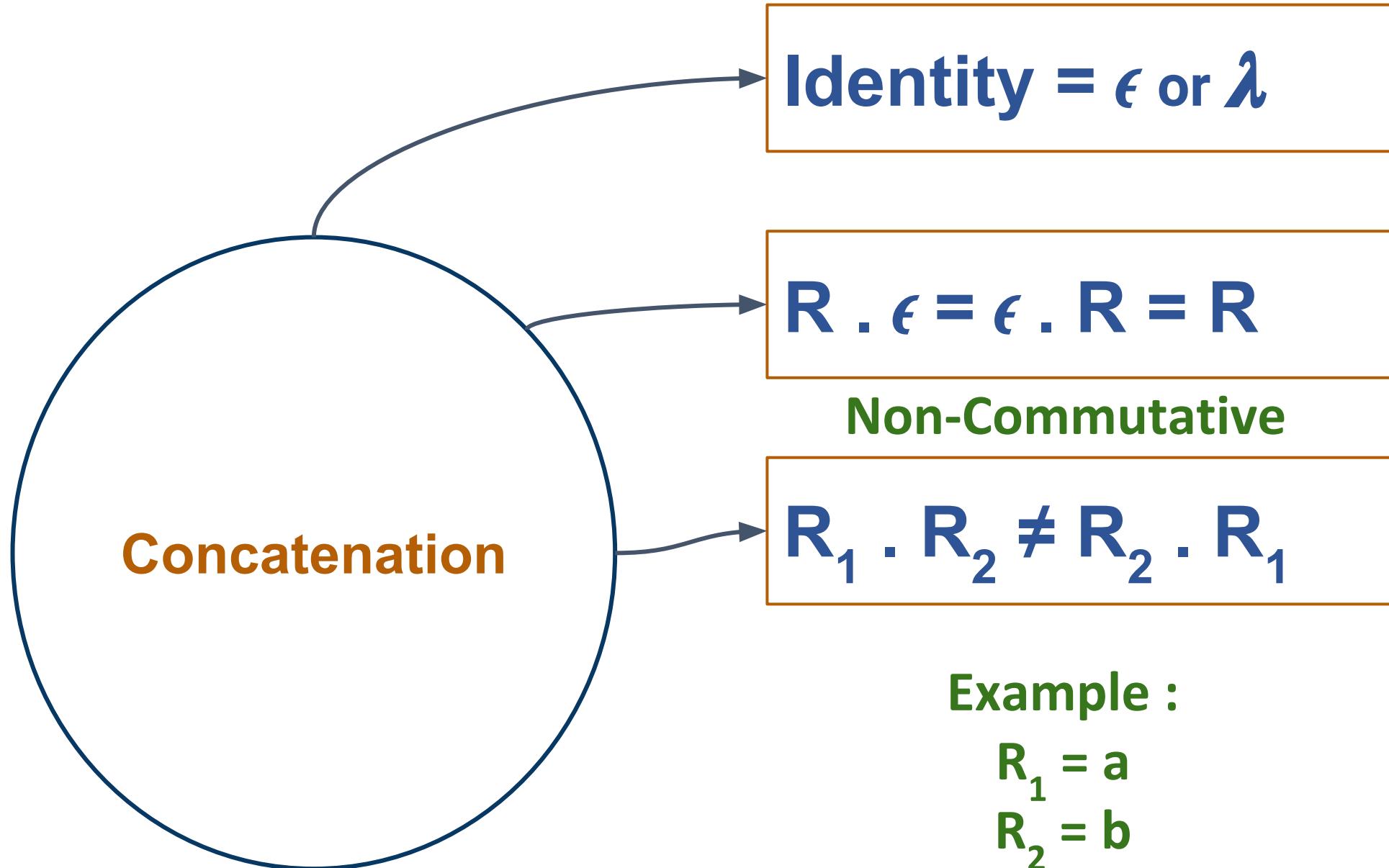






$$R \cdot \phi = \phi \cdot R = \phi$$

Let  $L_1, L_2$  be languages, then the concatenation  $L_1 \circ L_2 = \{w \mid w = xy, x \in L_1, y \in L_2\}$ . If  $L_2 = \emptyset$ , then there is no string  $y \in L_2$  and so there is no possible  $w$  such that  $w = xy$ . Thus for any  $L_1$ , we'll have  $L_1 \circ \emptyset = \emptyset$ .



Non-Commutative

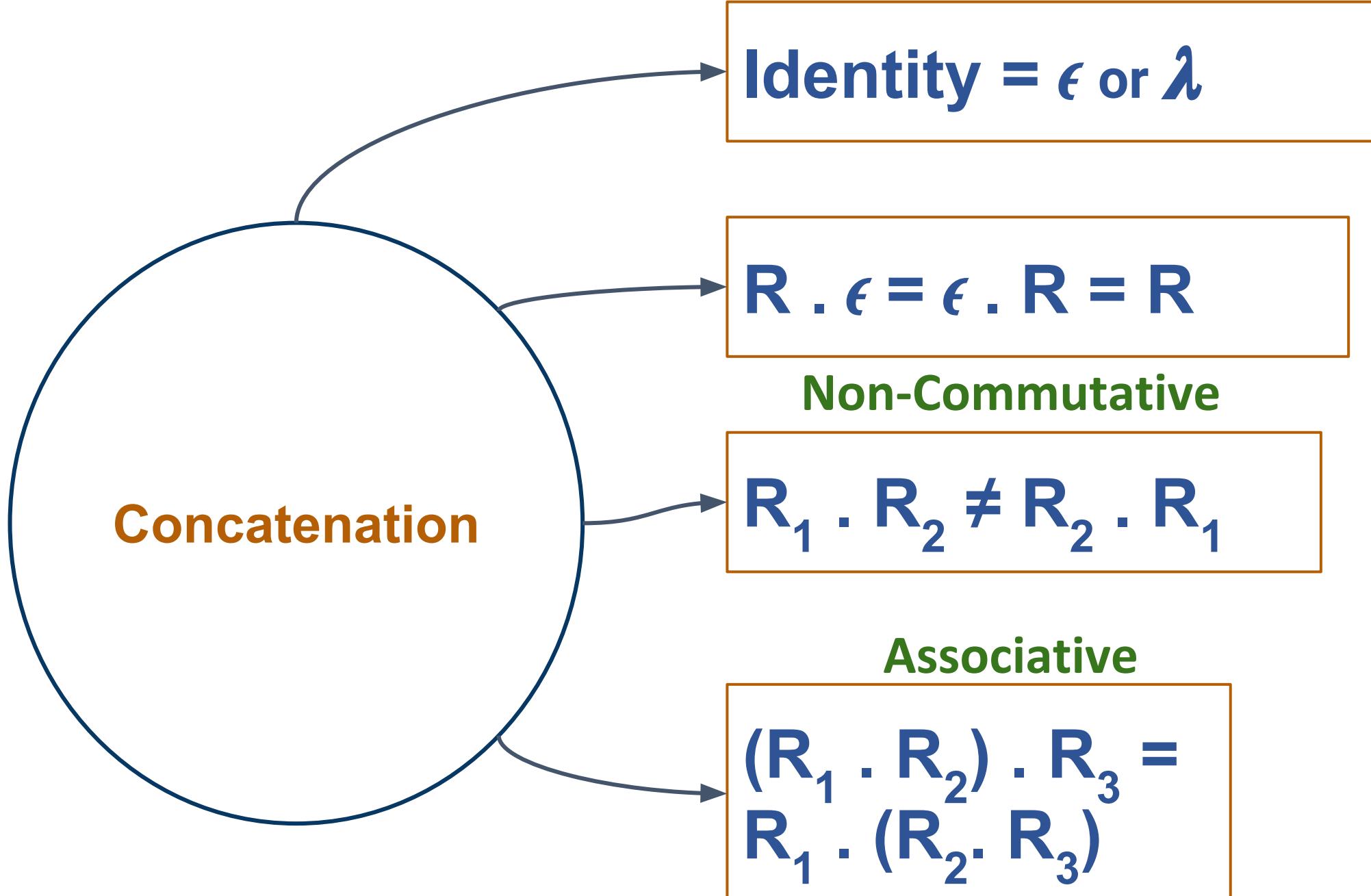
Example :

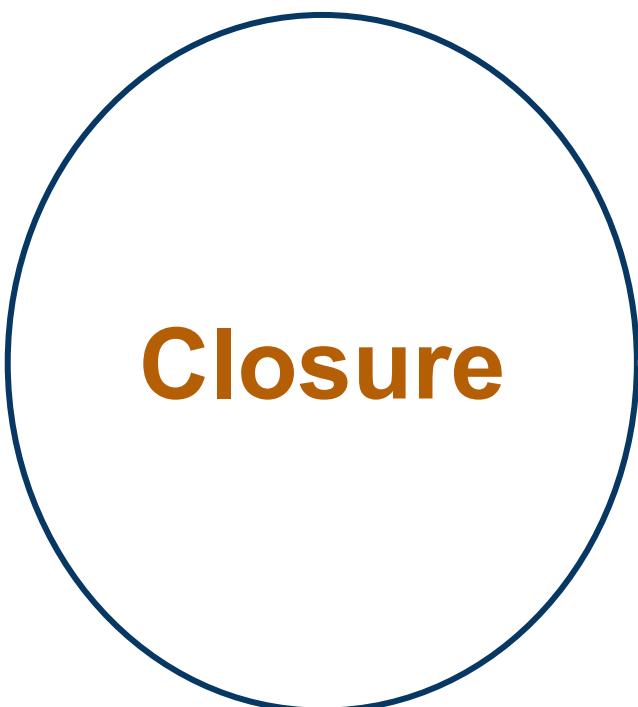
$$R_1 = a$$

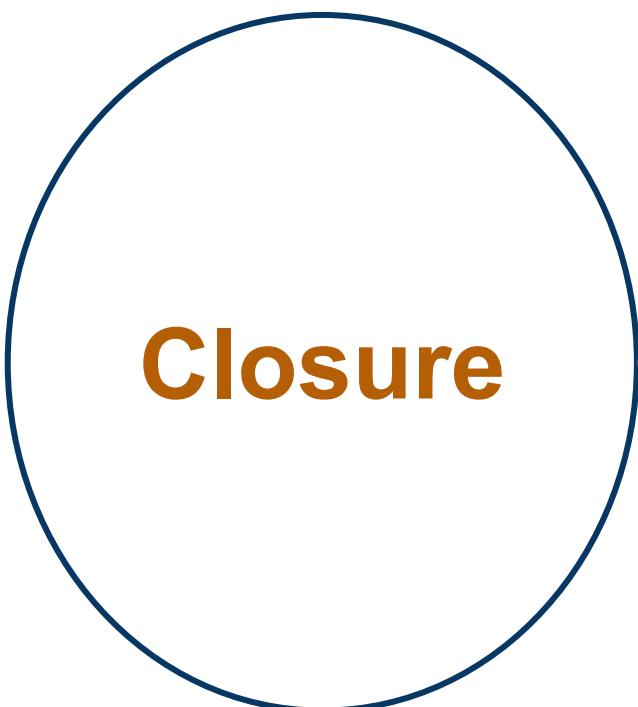
$$R_2 = b$$

then,

$$ab = ba$$




$$\epsilon^* = \epsilon$$
$$\phi^* = \epsilon$$
$$(R^*)^* = R^*$$
$$(R + \epsilon)^* = R^*$$
$$R^+ = RR^* = R^*R$$
$$(R^+ + \epsilon) = R^*$$
$$\epsilon + RR^* = \epsilon + R^* = R^*$$


$$\begin{aligned}(a + b)^* \\ &= (a^* + b^*)^* \\ &= (a^* b^*)^* \\ &= (a + b^*)^* \\ &= (a^* + b)^* \\ &= a^*(ba^*)^* \\ &= b^*(ab^*)^*\end{aligned}$$

# Automata Formal Languages and Logic

## Unit 2 - Construction of Regular Expressions

---



**Construct a Regular Expression for a given language L**



**THANK YOU**

---

**Preet Kanwal**

Department of Computer Science & Engineering

**[preetkanwal@pes.edu](mailto:preetkanwal@pes.edu)**

**+91 80 6666 3333 Extn 724**



# Automata Formal Languages & Logic

---

**Preet Kanwal**

Department of Computer Science & Engineering

# Automata Formal Languages & Logic

---

## Unit 2

**Preet Kanwal**

Department of Computer Science & Engineering

**Automata Formal Languages and Logic**

**Unit 2 - Regular Expression in Practice**

---



# **Regular Expression in Practice**

# Automata Formal Languages and Logic

## Unit 2 - Regular Expression in Practice

---



### Special characters

1. .

### Special characters

1. .

(dot)

matches any single character except newline

### Special characters

1. .
2. \*

### Special characters

1. .
  2. \*
- (Star)

0 or more repetitions of preceding regex

### Special characters

1. .
2. \*

**Example :**

a\*

**Strings matched :**

0 or more no. of a's

**Example :**  $\lambda, a, aa, aaa, aaaa, aaaaa\dots$

### Special characters

1. .
2. \*
3. +

### Special characters

1. .
2. \*
3. +

**Example :**

a+

**Strings matched :**

**1 or more no. of a's**

**Example : a,aa,aaa,aaaa,aaaaa....**

### Special characters

1. .
2. \*
3. +
4. ?

### Special characters

1. .
2. \*
3. +
4. ?

Example:

ab?c

matches abc or ac

### Special characters

1. .
2. \*
3. +
4. ?
5. [ ]

### Character Class

Matches any single character

### Special characters

1. .
2. \*
3. +
4. ?
5. [ ]

Example

[cat]

String matched :

c

or a

or t

### Special characters

1. .
2. \*
3. +
4. ?
5. [ ]
6. ^

**^ (caret) :**

**Example:**

**Example: ^abc matches "a" at the start of the string.**

**[^abc]: This pattern matches any character except a or b or c.**

### Special characters

1. .
2. \*
3. +
4. ?
5. [ ]
6. ^
7. \$

**\$ (dollar) :**

**Example :**

- > abc\$" matches "c" at the end of a line.
- > ^\$ matches the empty string.

### Special characters

1. .
2. \*
3. +
4. ?
5. [ ]
6. ^
7. \$
8. { } (Syntax - R{m,n})

### Example of { }

$a\{0, \}$  - same as  $a^*$

$a\{1, \}$  - same as  $a^+$

$a\{2,3\}$  matches "aa" or "aaa".

$a\{3\}$  matches aaa only

### Special characters

1. .
2. \*
3. +
4. ?
5. [ ]
6. ^
7. \$
8. { } (Syntax - R{m,n})
9. \d - matches any digit between [0-9]
  - a. \d{9} matches any 9 digits
  - b. \d{2,3} matches 2 or 3 digits

# Automata Formal Languages and Logic

## Unit 2 - Regular Expression in Practice

---



We will discuss how to construct regex for validating the following :

- PAN Card
- Adhaar Card
- Mobile Number
- Date
- Email Address

- **PAN CARD:**

**The valid PAN Card number must satisfy the following conditions:**

1. It should be 10 characters long.
2. The first five characters should be any upper case alphabets.
3. The next four-characters should be any number from 0 to 9.
4. The last(tenth) character should be any upper case alphabet.
5. It should not contain any white spaces.

### PAN CARD Regex - $^{\text{A-Z}}\{\text{5}\}[\text{0-9}]\{\text{4}\}[\text{A-Z}]\$$

*Input: str = “BNZAA2318J”*

*Output: true*

*Explanation:*

*The given string satisfies all the above mentioned conditions.*

*Input: str = “23ZAABN18J”*

*Output: false*

*Explanation:*

*The given string does not start with upper case alphabets, therefore it is not a valid PAN Card number.*

- **AADHAR NUMBER:**

The valid Aadhar number must satisfy the following conditions:

1. It should have 12 digits.
2. It should not start with 0 and 1.
3. It should not contains any alphabet and special characters.
4. It should have white space after every 4 digits.

# Automata Formal Languages and Logic

## Unit 2 - Regular Expression in Practice

---



**AADHAR Number Regex - `^[2-9]{1}\d{3}\s{4}[" "]|\s{4}$`**

***Input: str = “3675 9834 6012”***

***Output: true***

***Explanation:***

***The given string satisfies all the above mentioned conditions. Therefore it is a valid Aadhar number.***

***Input: str = “3675 9834 6012 8”***

***Output: false***

***Explanation:***

***The given string contains 13 digits. Therefore it is not a valid Aadhar number.***

- **INDIAN MOBILE NUMBER:**

The valid Mobile number must satisfy the following conditions:

- It is a 10 digits number
- The first digit should contain number between 6 to 9.
- The rest 9 digit can contain any number between 0 to 9.
- The mobile number can have 11 digits also by including 0 at the starting.
- The mobile number can be of 13 digits also by including +91 at the starting

# Automata Formal Languages and Logic

## Unit 2 - Regular Expression in Practice

---



**MOBILE Number Regex - `^(0|"+91")?[6-9]\d{9}$`**

### Dates in the year 2020

- It is a leap year
- Let the format be DD-MM-YY

Months with 31 days are : Jan, Mar, May, July, Aug, Oct, Dec  
: [1, 3, 5, 7, 8, 10, 12]

Months with 30 days are : Apr, June, Sep, Nov  
: [4, 6, 9, 11]

Month with 29 days : Feb  
: [2]

- Year : 2020

### Dates in the year 2020

Regex:

```
(  
(0[0-9])|[10-31]"-(0[13578]|1[0|2])  
|  
(0[0-9])|[10-30]"-(0[469]|11)  
|  
(0[0-9])|[10-29]"-""02"  
)""2020"
```

Email Address:

An email is a string (a subset of ASCII characters) separated into two parts by @ symbol. a "personal\_info" and a domain, that is personal\_info@domain.

The length of the personal\_info part may be up to 64 characters long and domain name may be up to 253 characters.

The personal\_info part contains the following ASCII characters.

- Uppercase (A-Z) and lowercase (a-z) English letters.
- Digits (0-9).
- Characters ! # \$ % & ' \* + - / = ? ^ \_ ` { | } ~
- Character . ( period, dot or fullstop) provided that it is not the first or last character and it will not come one after the other.

The domain name [for example com, org, net, in, us, info] part contains letters, digits, hyphens, and dots.

Email Address:

Example of valid email id

- mysite@ourearth.com
- my.ownsite@ourearth.org
- mysite@you.net

Example of invalid email id

- mysite.ourearth.com [ @ is not present ]
- mysite@.com.my [ tld (Top Level domain) can not start with dot ". " ]
- @you.me.net [ No character before @ ]

`^[_a-zA-Z0-9-]+(\.[_a-zA-Z0-9-]+)*@[a-zA-Z0-9-]+(\.[a-zA-Z0-9-]+)*\.(0-9){1,3}|([a-zA-Z]{2,3})|(com|edu|info|museum|name))$`

### Regular Expression in practice: Examples

#### Example 2: Simple URL Validator

### Regular Expression in practice: Examples

#### Example 2: Simple URL Validator

`^((ht|f)tp(s?))\://([0-9a-zA-Z\-\-]+\.)+[a-zA-Z]{2,6}(\:[0-9]+)?(\:\S*)?$/`

### Regular Expression in practice: Examples

#### Example 5: -mail addresses Validation

### Regular Expression in practice: Examples

#### Example 5: -mail addresses Validation

```
^[_a-zA-Z0-9-]+(\.[_a-zA-Z0-9-]+)*@[a-zA-Z0-9-]+(\.[a-zA-Z0-9-]+)*\.
(([0-9]{1,3})|([a-zA-Z]{2,3})|(com|edu|info|museum|name))$
```

### Regular Expression in practice: Examples

**Example 6: Ip Address validation:**

### Regular Expression in practice: Examples

#### Example 6: Ip Address validation:

```
^(25[0-5]|2[0-4][0-9]| [0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9])\.(25[0-5]|2[0-4][0-9]| [0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9]|0)\.(25[0-5]|2[0-4][0-9]| [0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9]|0)\.(25[0-5]|2[0-4][0-9]| [0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[0-9])$
```



**THANK YOU**

---

**Preet Kanwal**

Department of Computer Science & Engineering

**[preetkanwal@pes.edu](mailto:preetkanwal@pes.edu)**

**+91 80 6666 3333 Extn 724**



# Automata Formal Languages & Logic

---

**Preet Kanwal**

Department of Computer Science & Engineering

# Automata Formal Languages & Logic

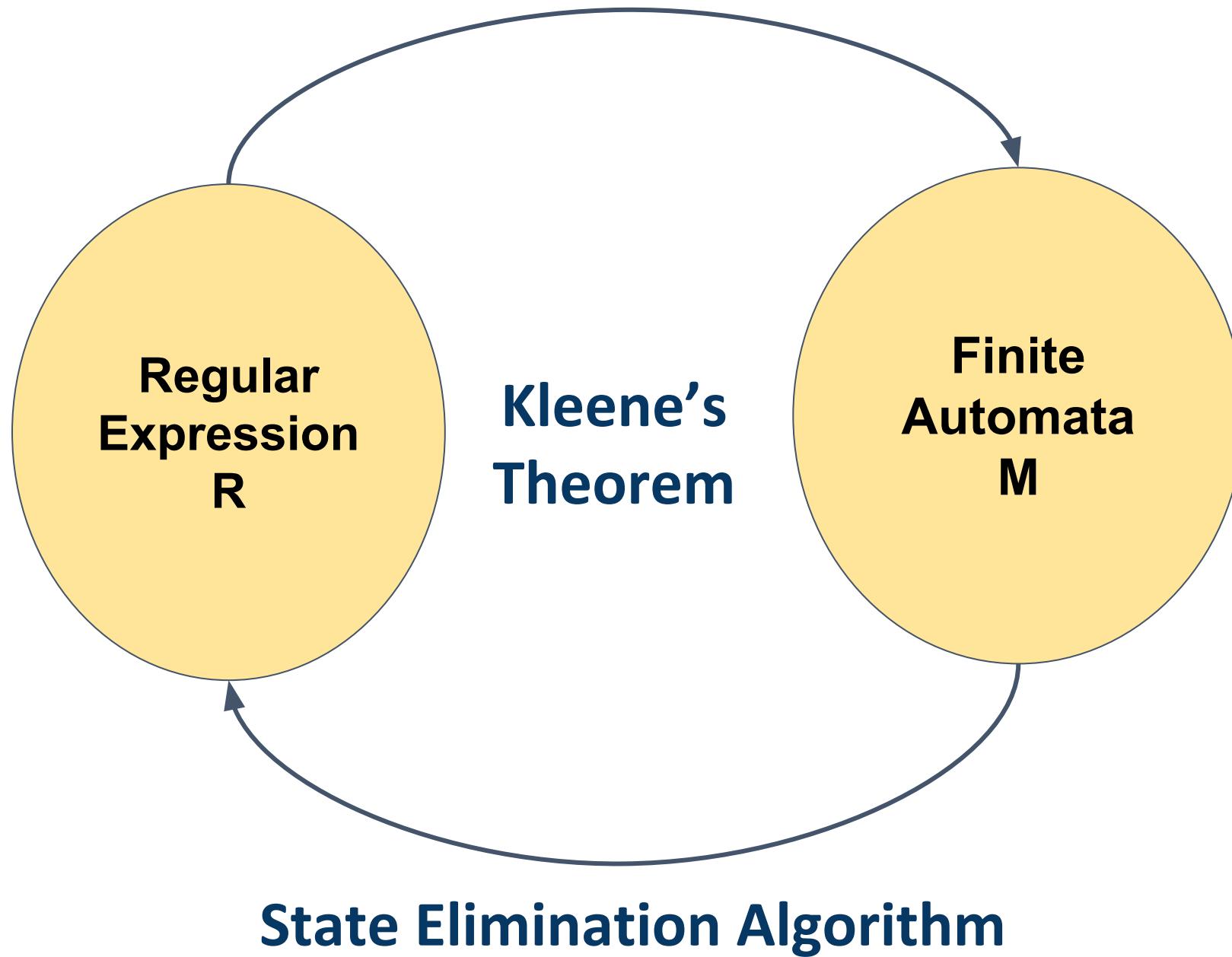
---

## Unit 2

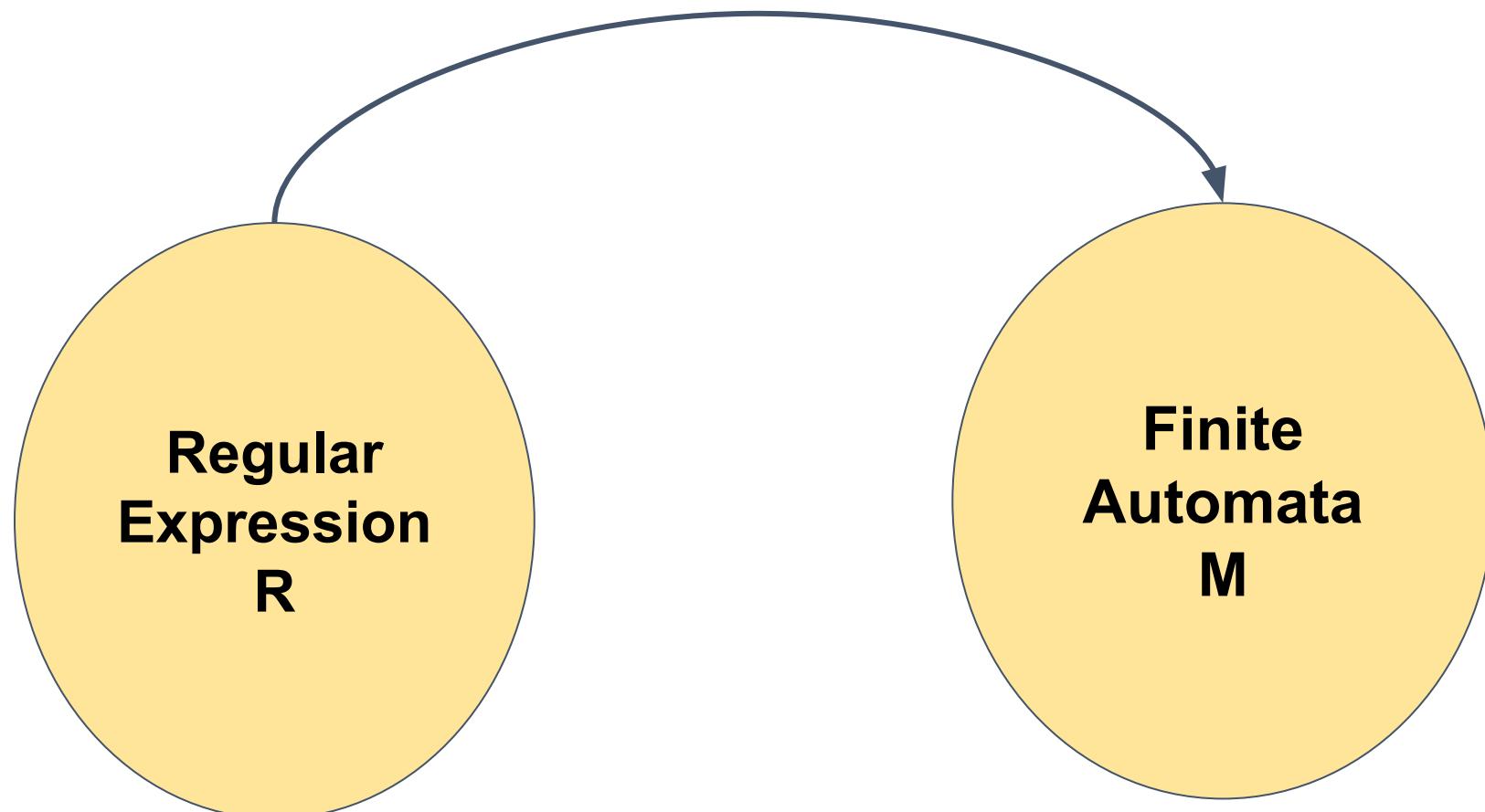
**Preet Kanwal**

Department of Computer Science & Engineering

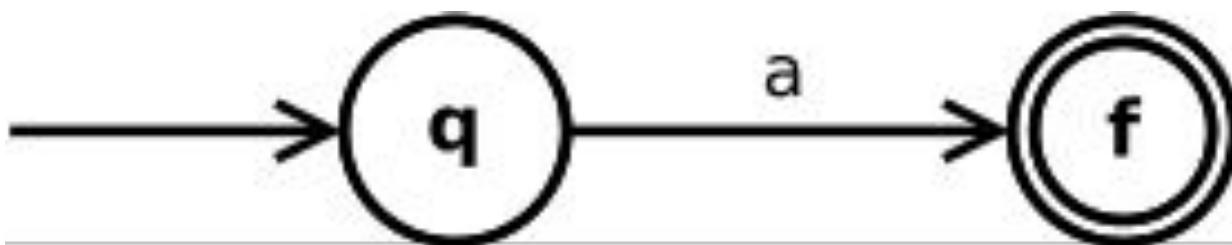
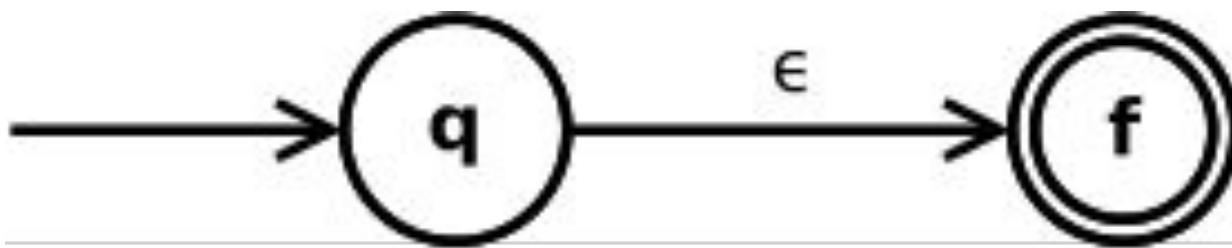
### Thompson's Construction Algorithm



### Thompson's Construction Algorithm

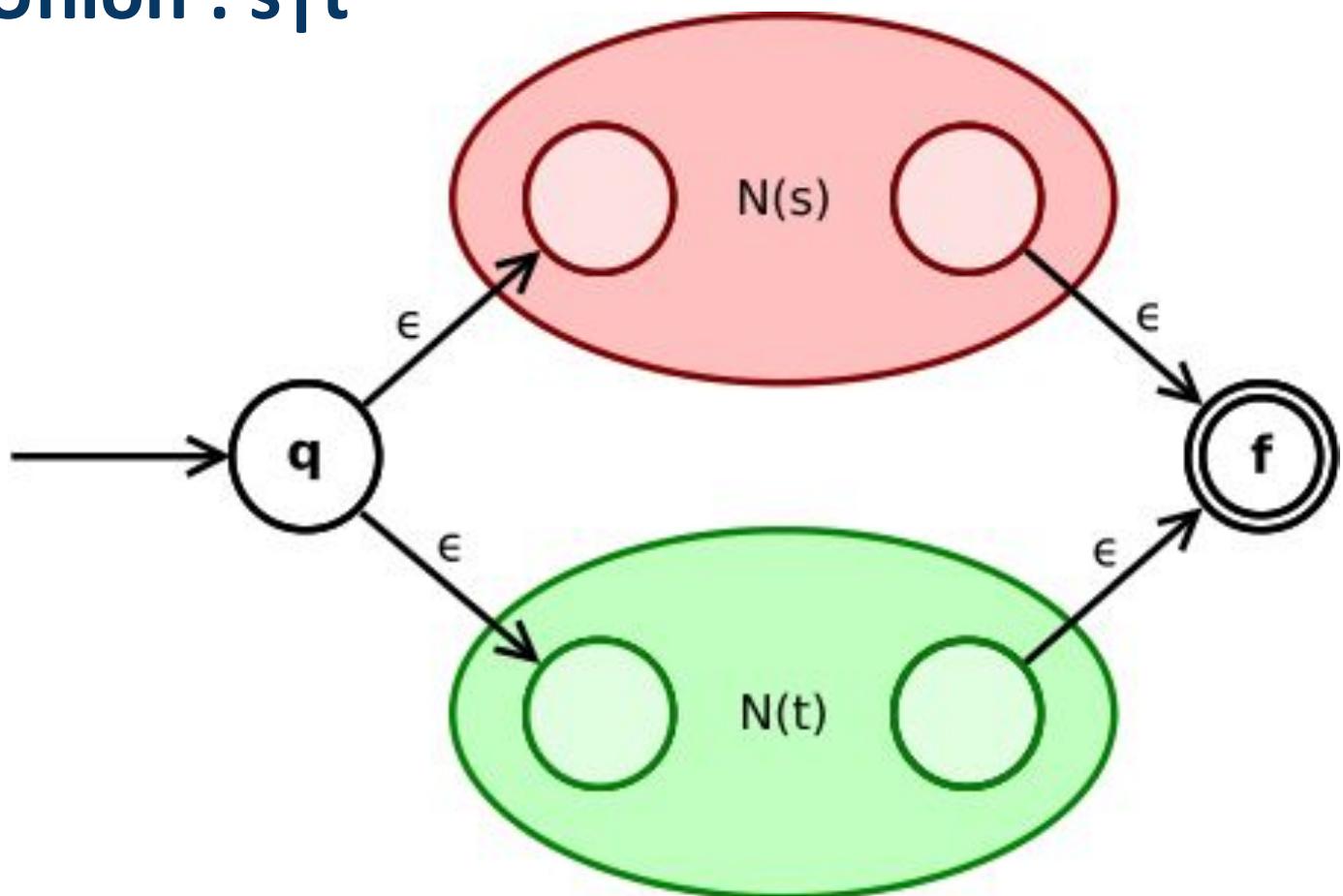


### Thompson's Construction Algorithm



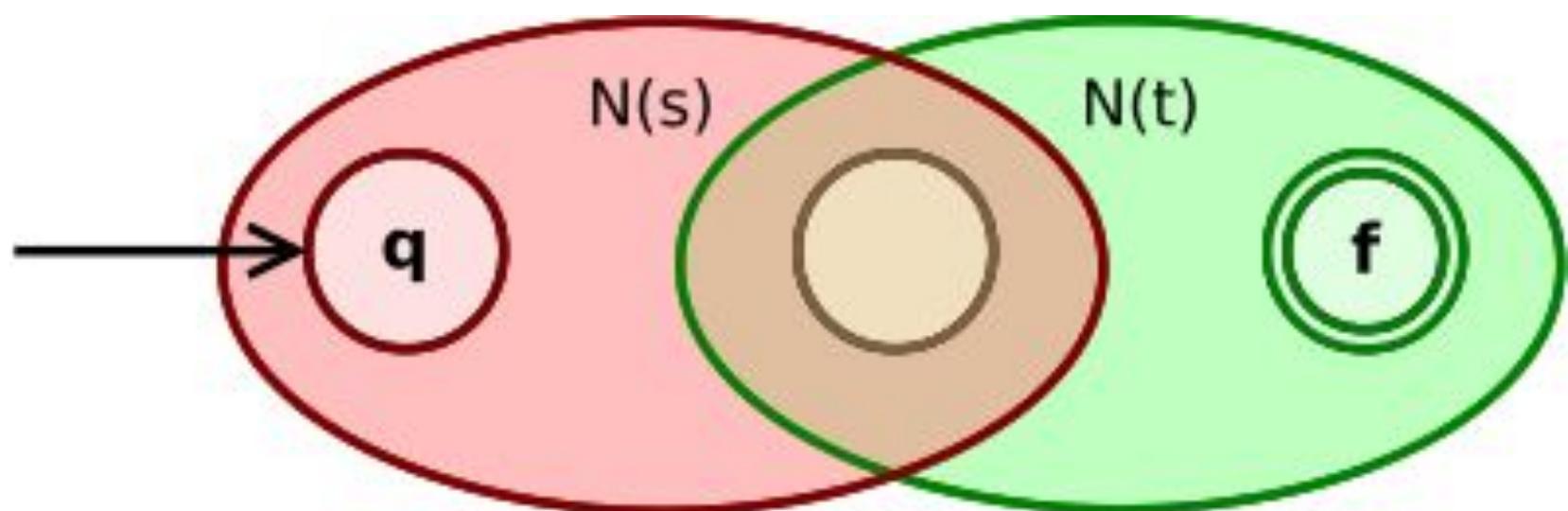
### Thompson's Construction Algorithm

Union :  $s|t$



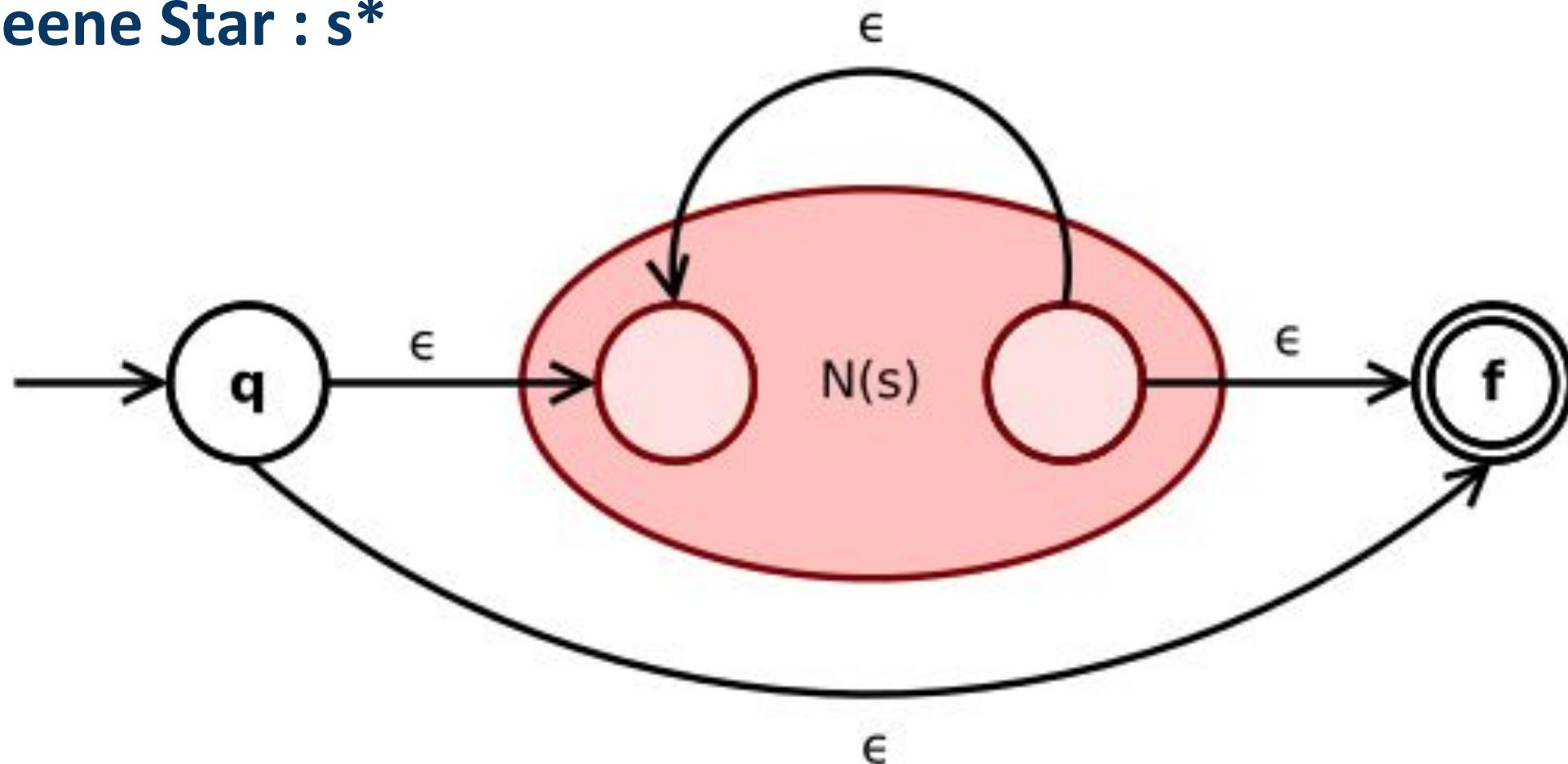
### Thompson's Construction Algorithm

Concatenation : s.t



### Thompson's Construction Algorithm

Kleene Star :  $s^*$



# Automata Formal Languages and Logic

## Unit 2 - Regular Expression to Finite Automata

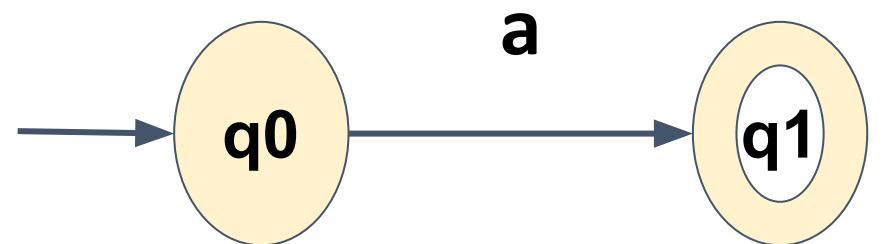
---



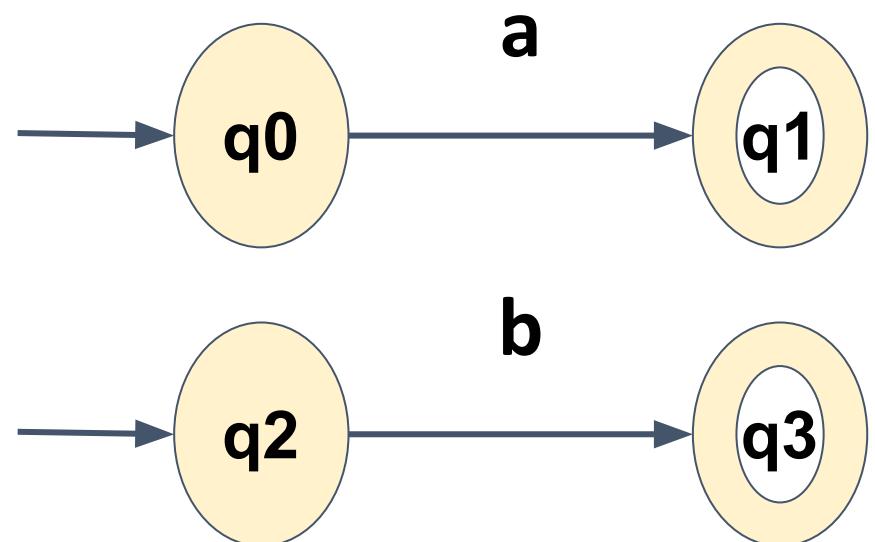
Construct Finite Automata for the following Regex :

$$(ab+c)^*$$

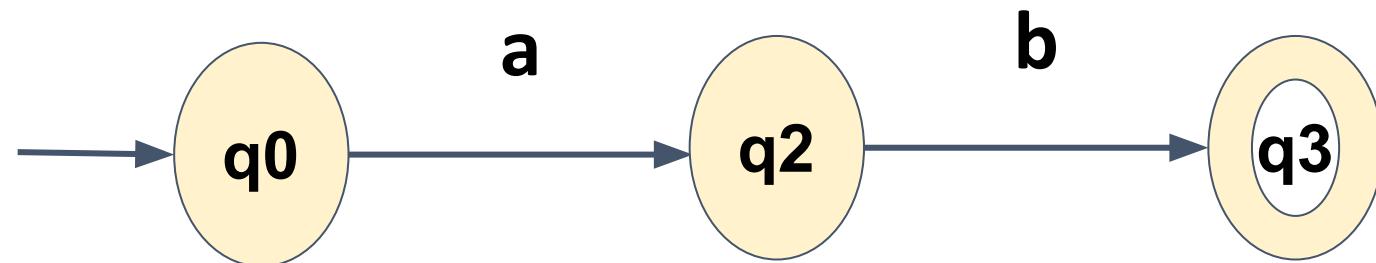
Construct Finite Automata for the following Regex :

$$(ab+c)^*$$


Construct Finite Automata for the following Regex :

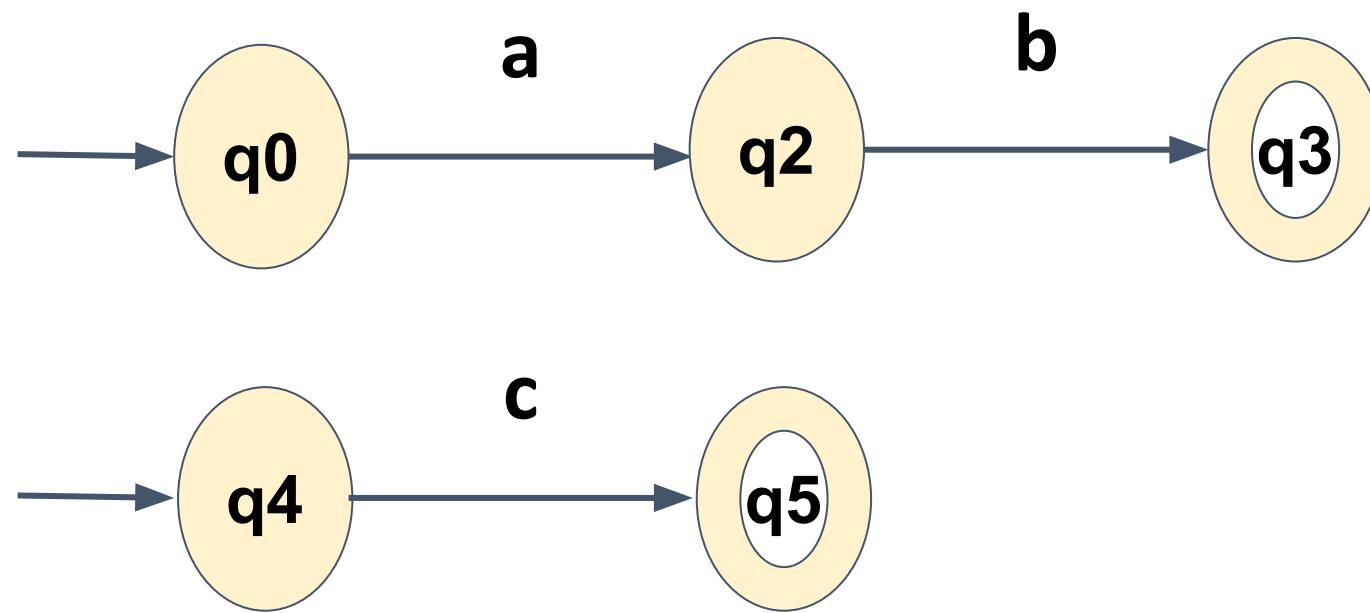
$$(ab+c)^*$$


Construct Finite Automata for the following Regex :

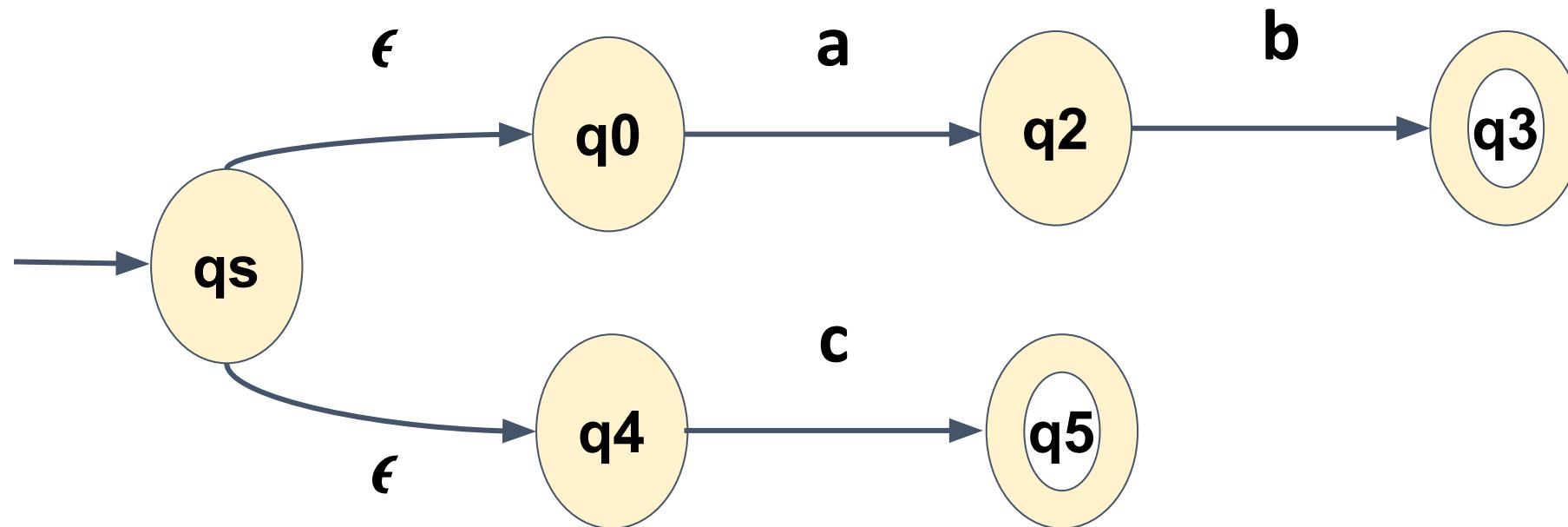
$$(ab+c)^*$$


Construct Finite Automata for the following Regex :

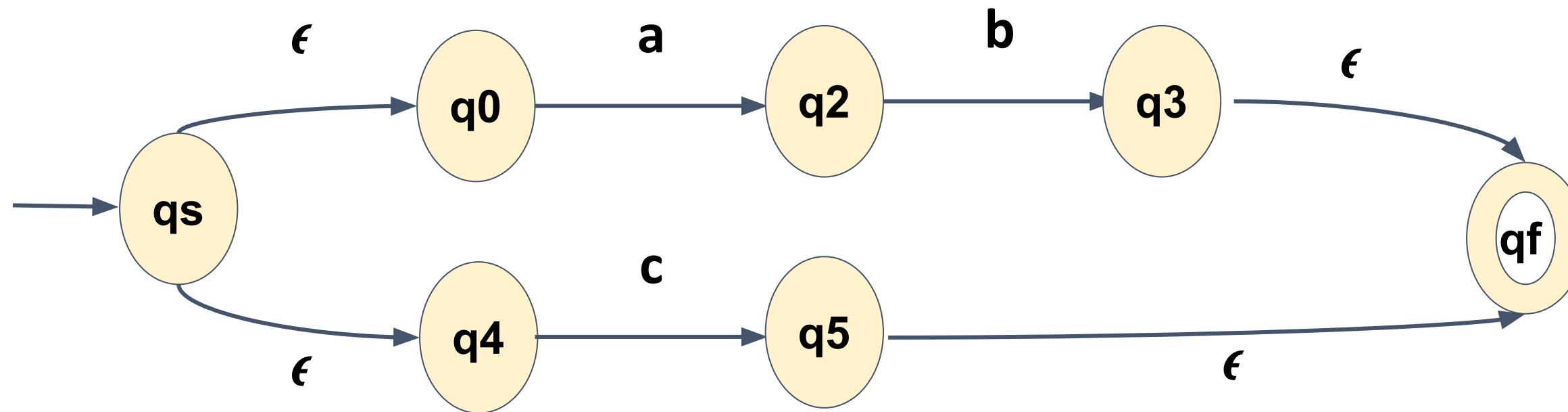
$$(ab+c)^*$$



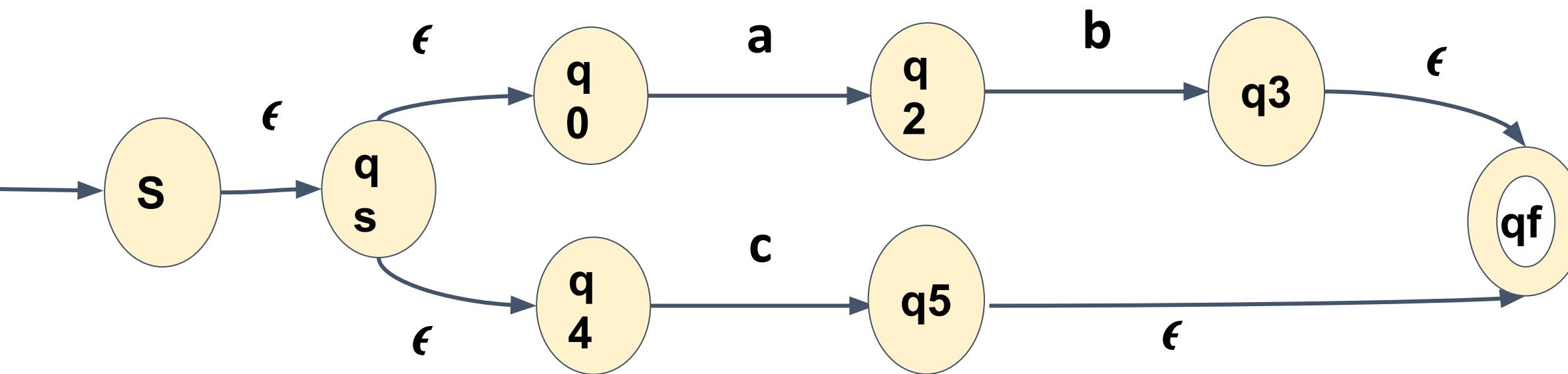
Construct Finite Automata for the following Regex :

$$(ab+c)^*$$


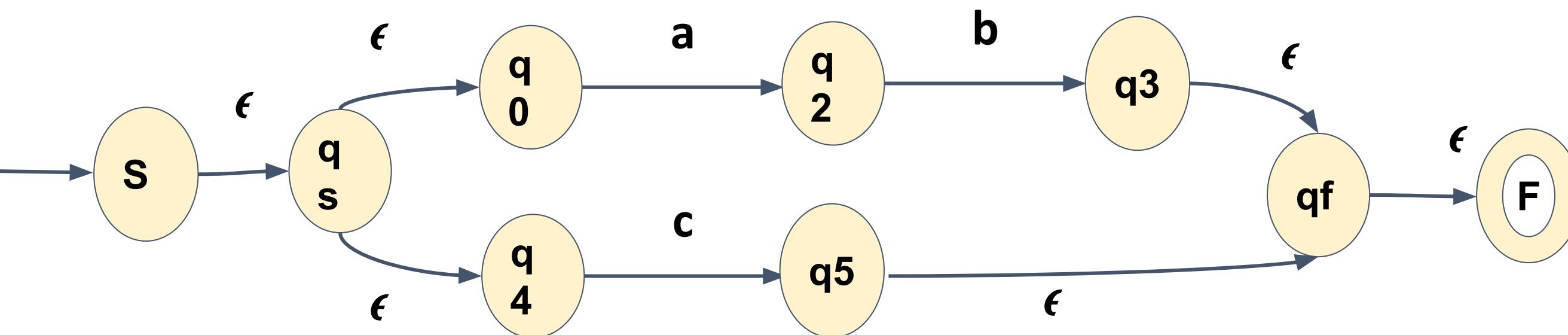
Construct Finite Automata for the following Regex :

$$(ab+c)^*$$


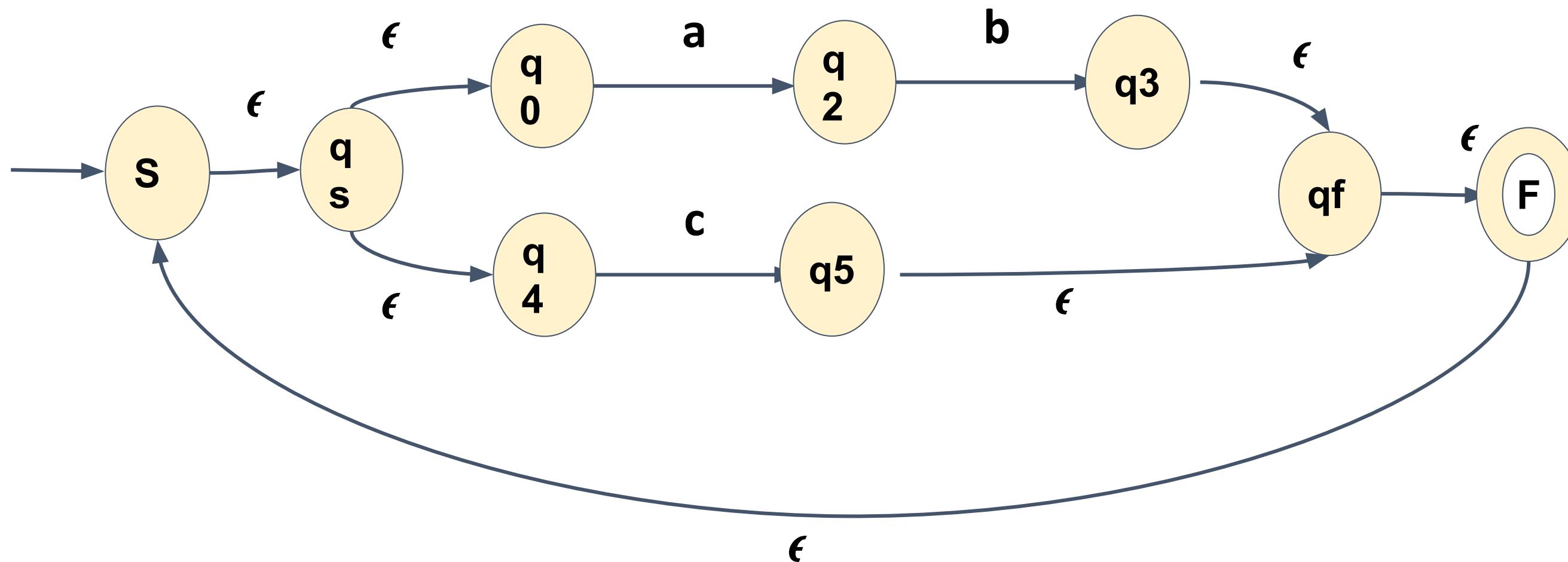
$(ab+c)^*$



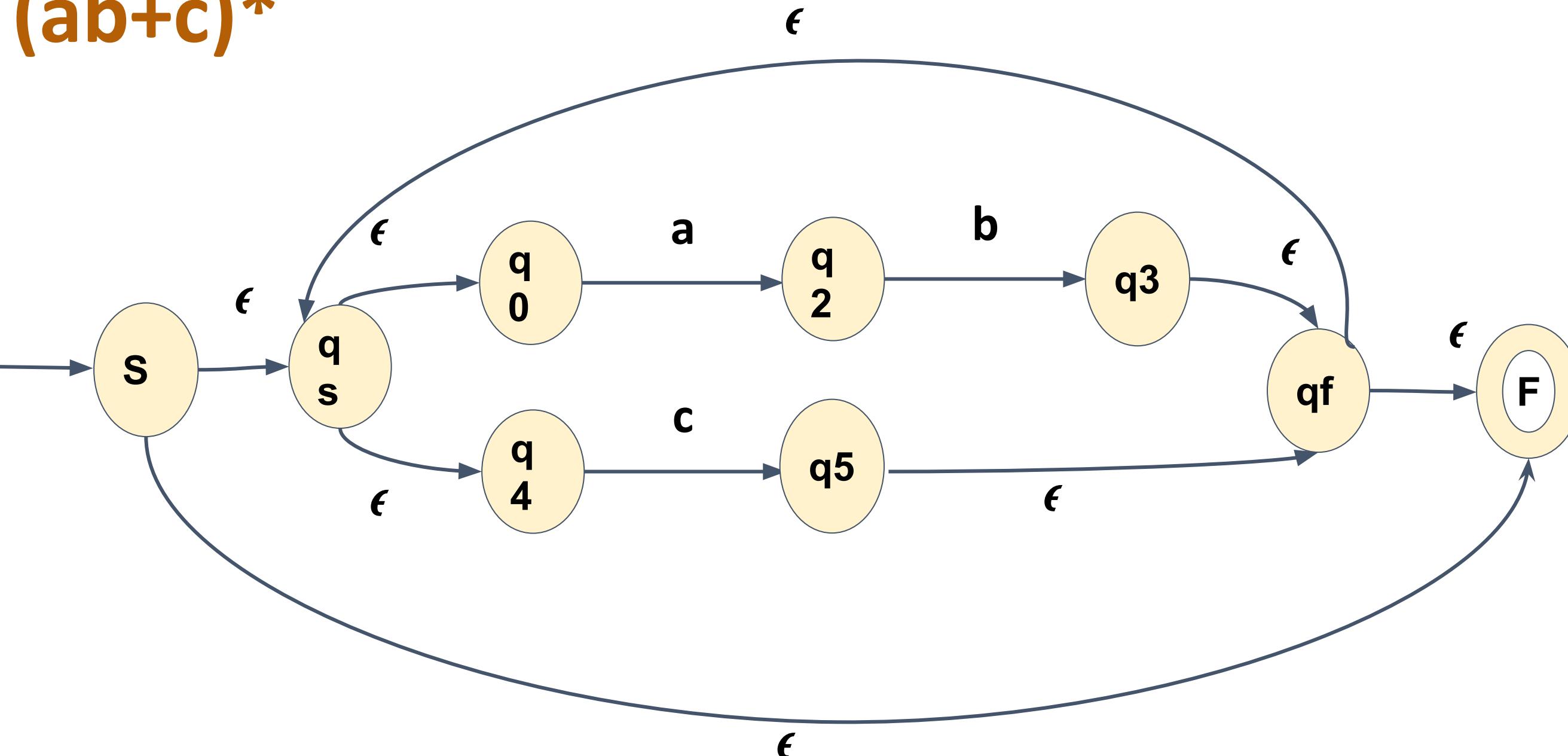
$(ab+c)^*$



$(ab+c)^*$



$(ab+c)^*$



# Automata Formal Languages and Logic

## Unit 2 - Regular Expression to Finite Automata

---



### Construct Finite Automata for given Regex



**THANK YOU**

---

**Preet Kanwal**

Department of Computer Science & Engineering

**[preetkanwal@pes.edu](mailto:preetkanwal@pes.edu)**

**+91 80 6666 3333 Extn 724**



# Automata Formal Languages & Logic

---

**Preet Kanwal**

Department of Computer Science & Engineering

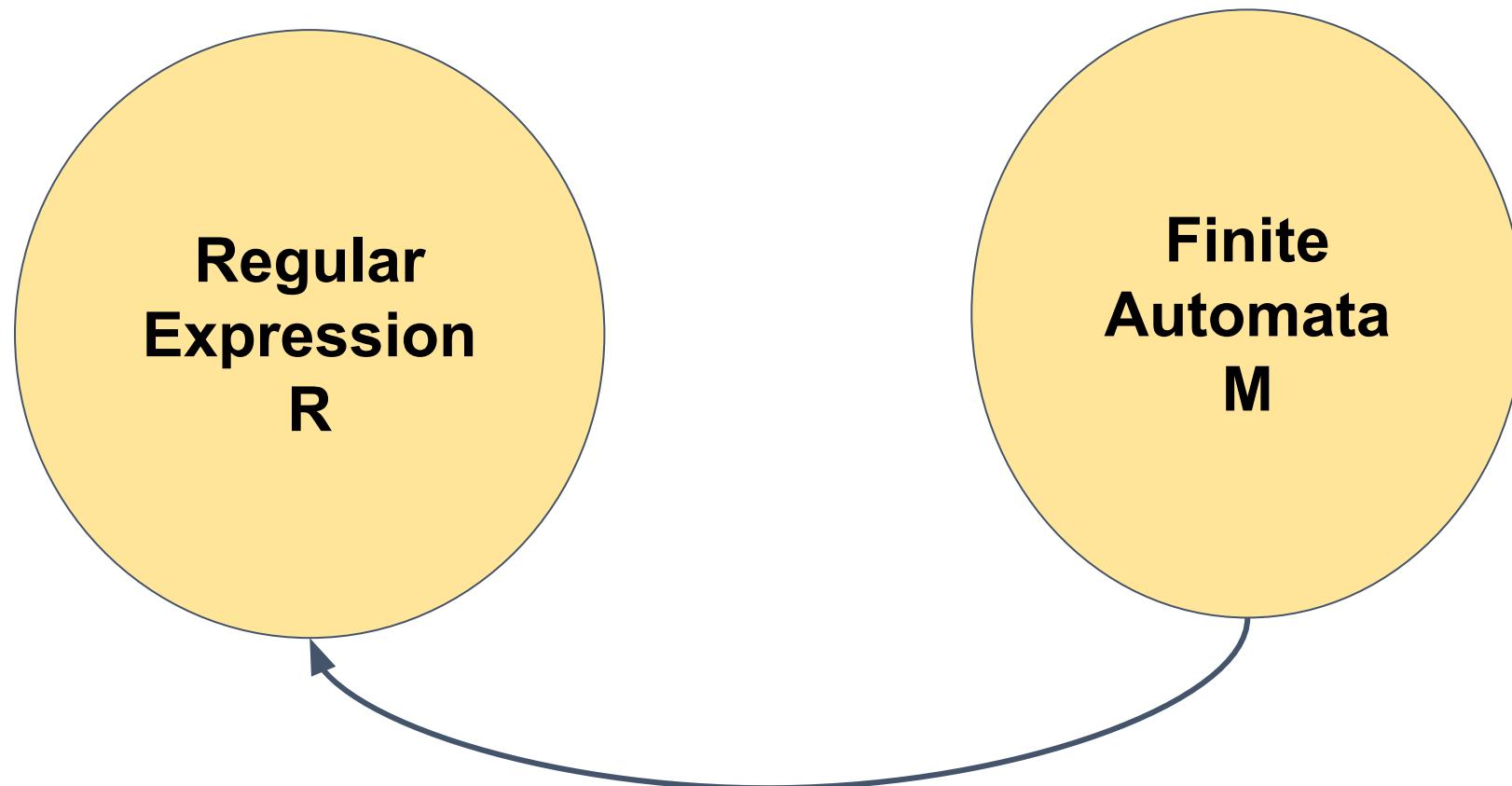
# Automata Formal Languages & Logic

---

## Unit 2

**Preet Kanwal**

Department of Mechanical Engineering



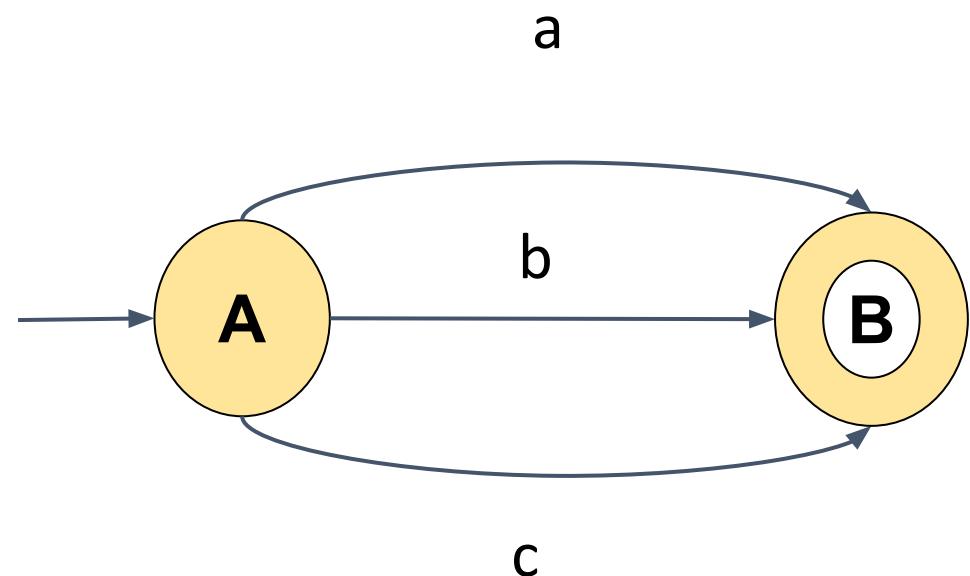
**State Elimination Algorithm**

### State Elimination Method

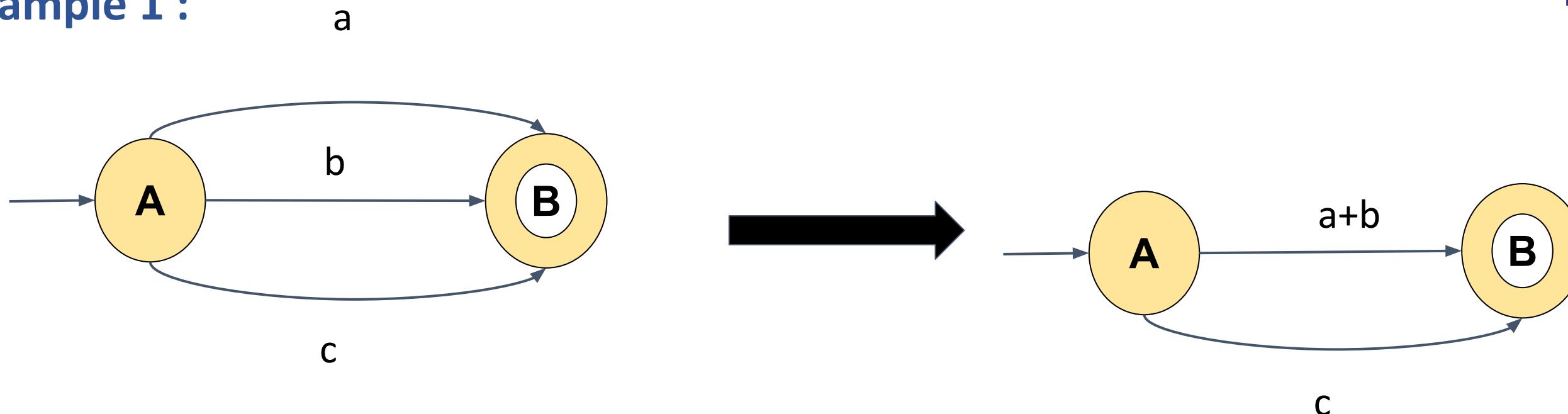
Start with an FA for the language L.

- Add a new start state  $q_s$  and accept state  $q_f$  to the FA.
- Add  $\epsilon$ -transitions from each original accepting state to  $q_f$  , then mark them as not accepting.
  - Repeatedly remove states other than  $q_s$  and  $q_f$  from the FA by “shortcutting” them until only two states remain:  $q_s$  and  $q_f$  .
  - The transition from  $q_s$  to  $q_f$  is then a regular expression for the FA.

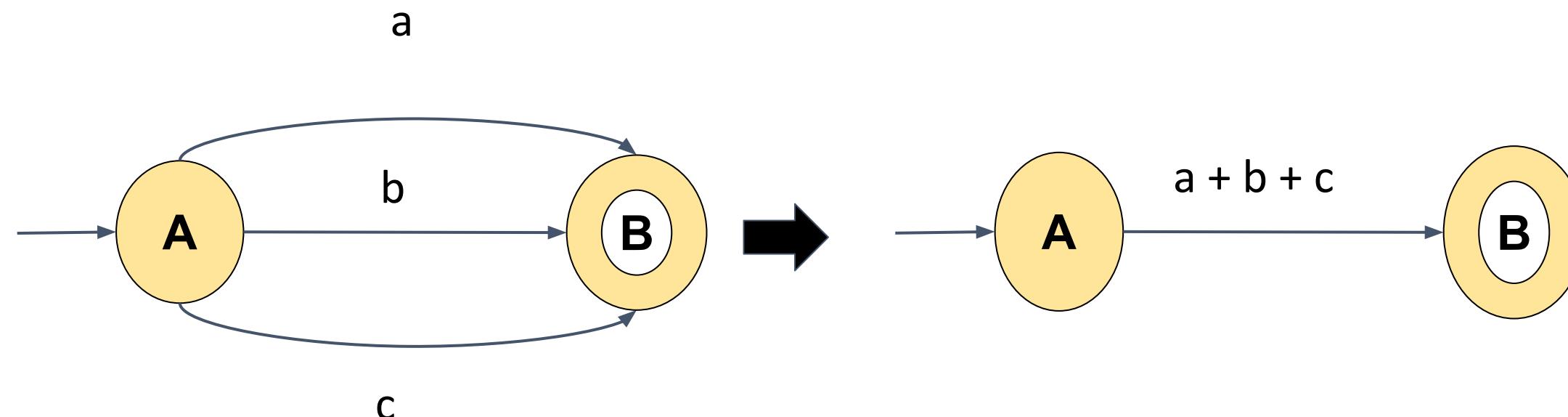
### Example 1 :



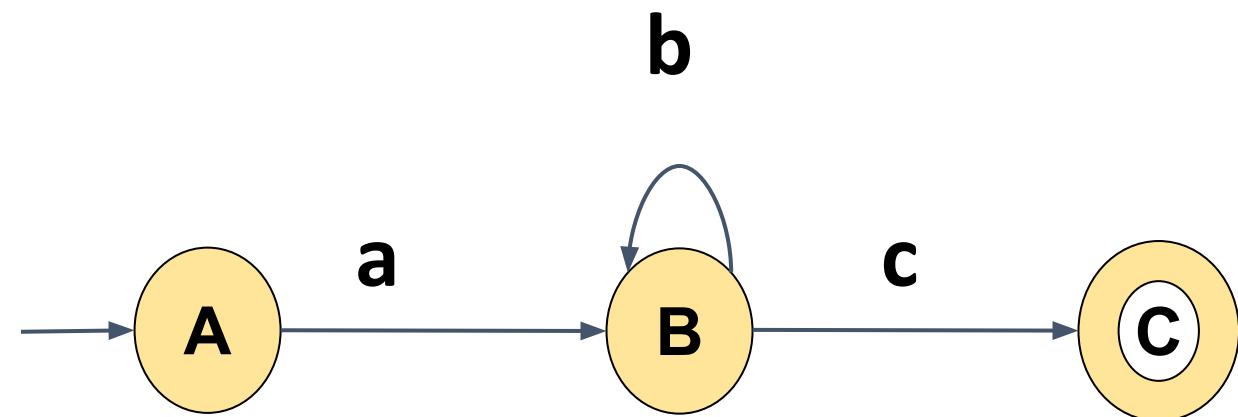
### Example 1 :



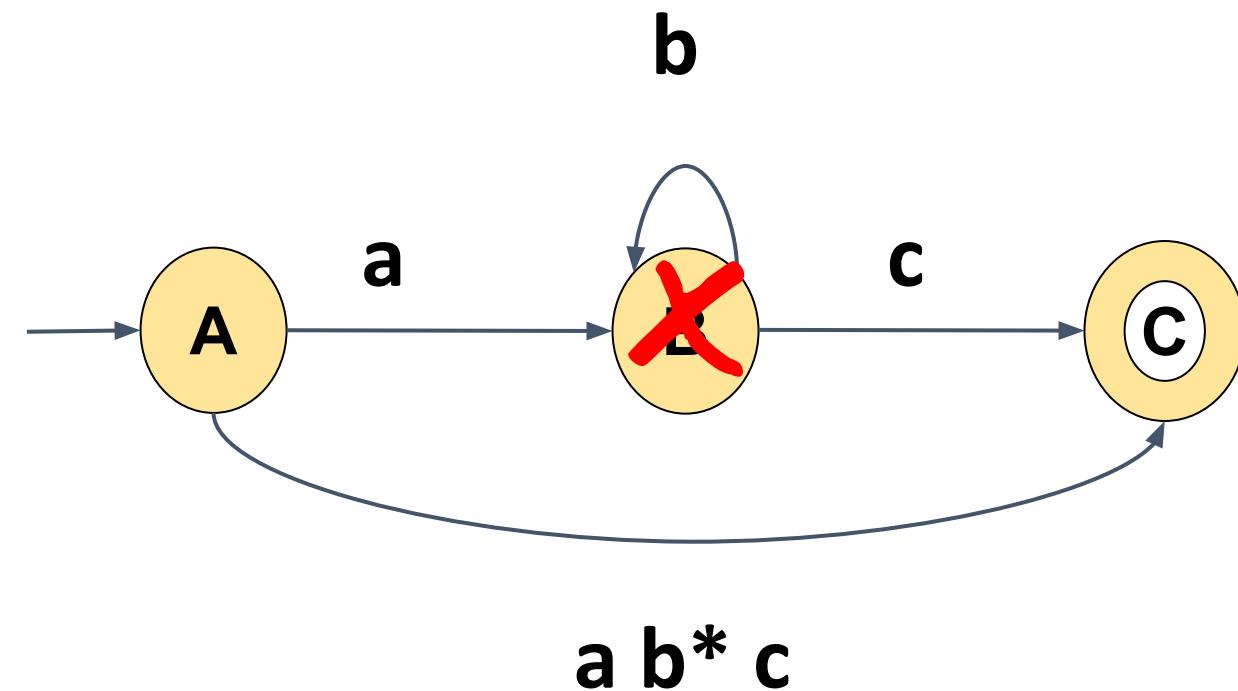
### Example 1 :



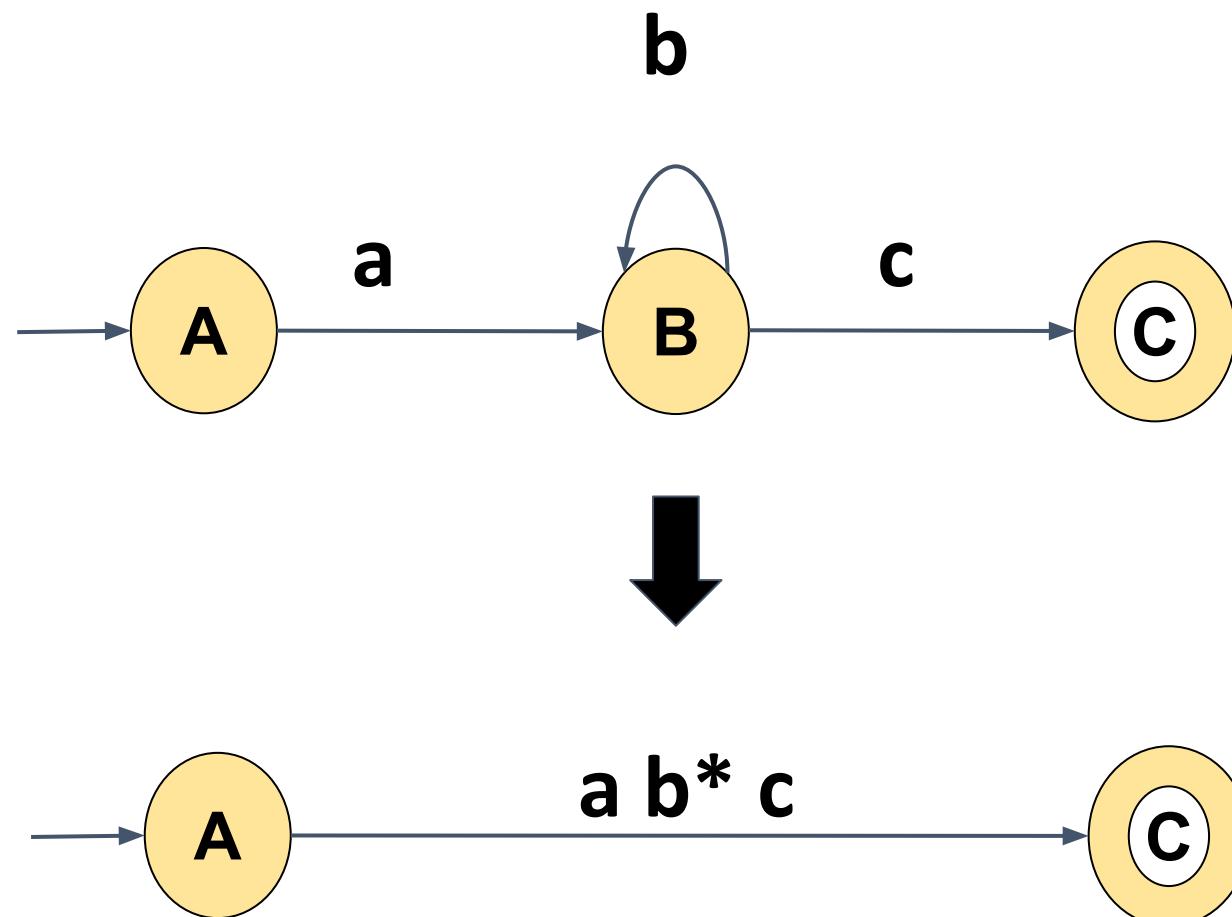
### Example 2 :



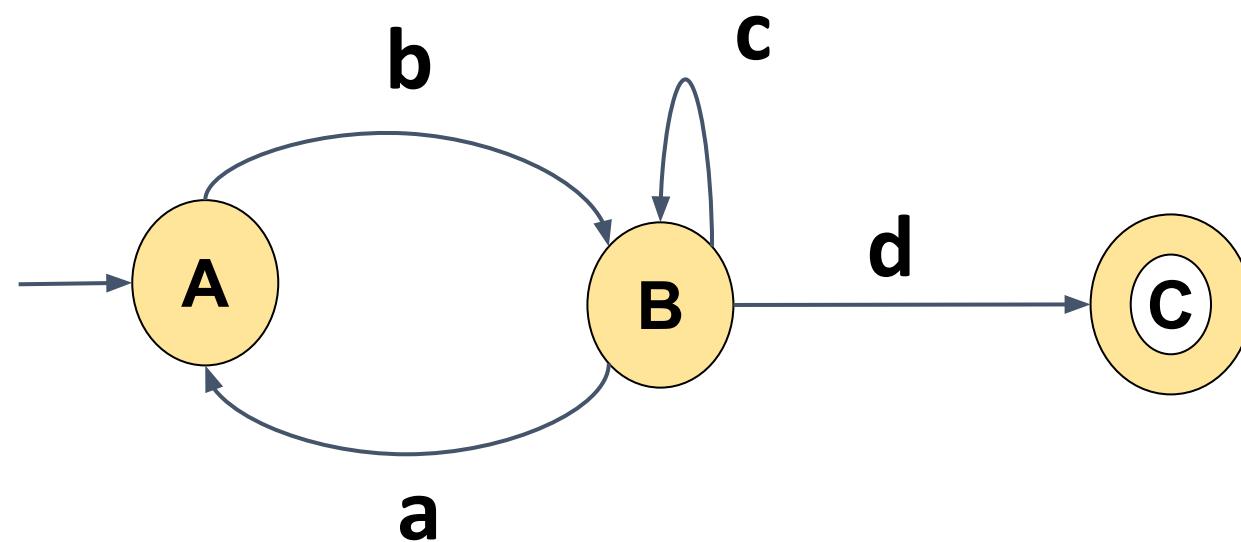
Example 2 :



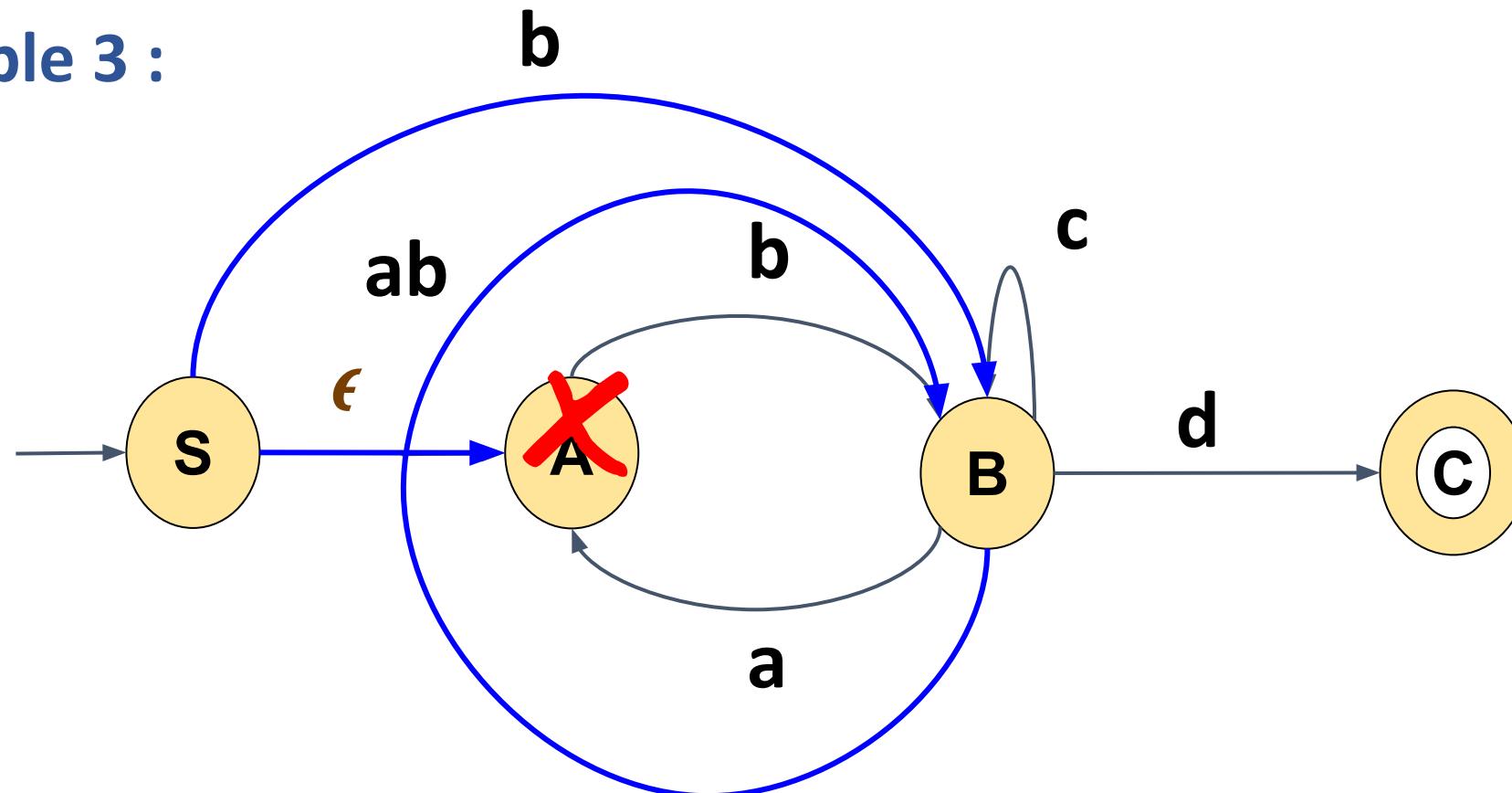
Example 2 :



### Example 3 :

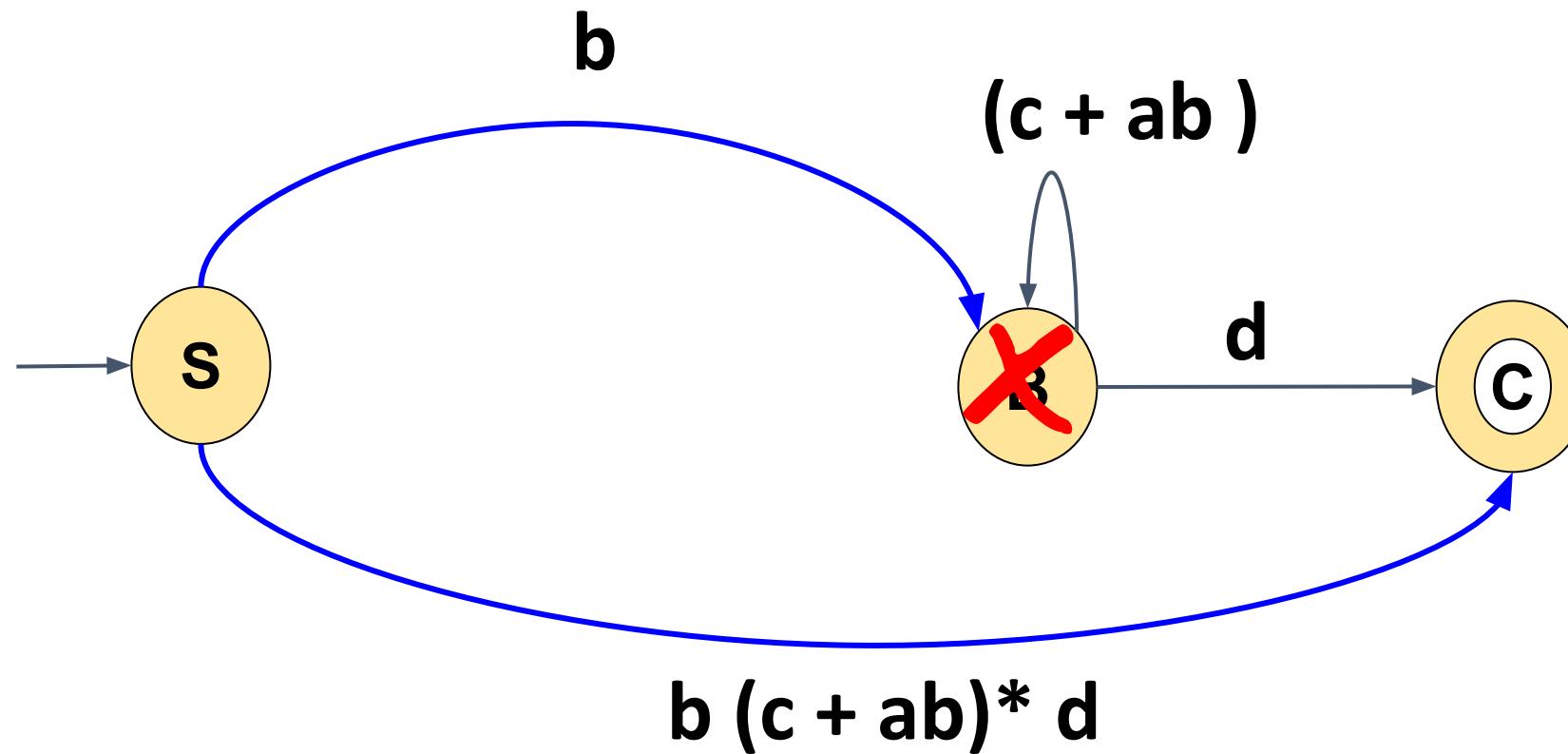


Example 3 :



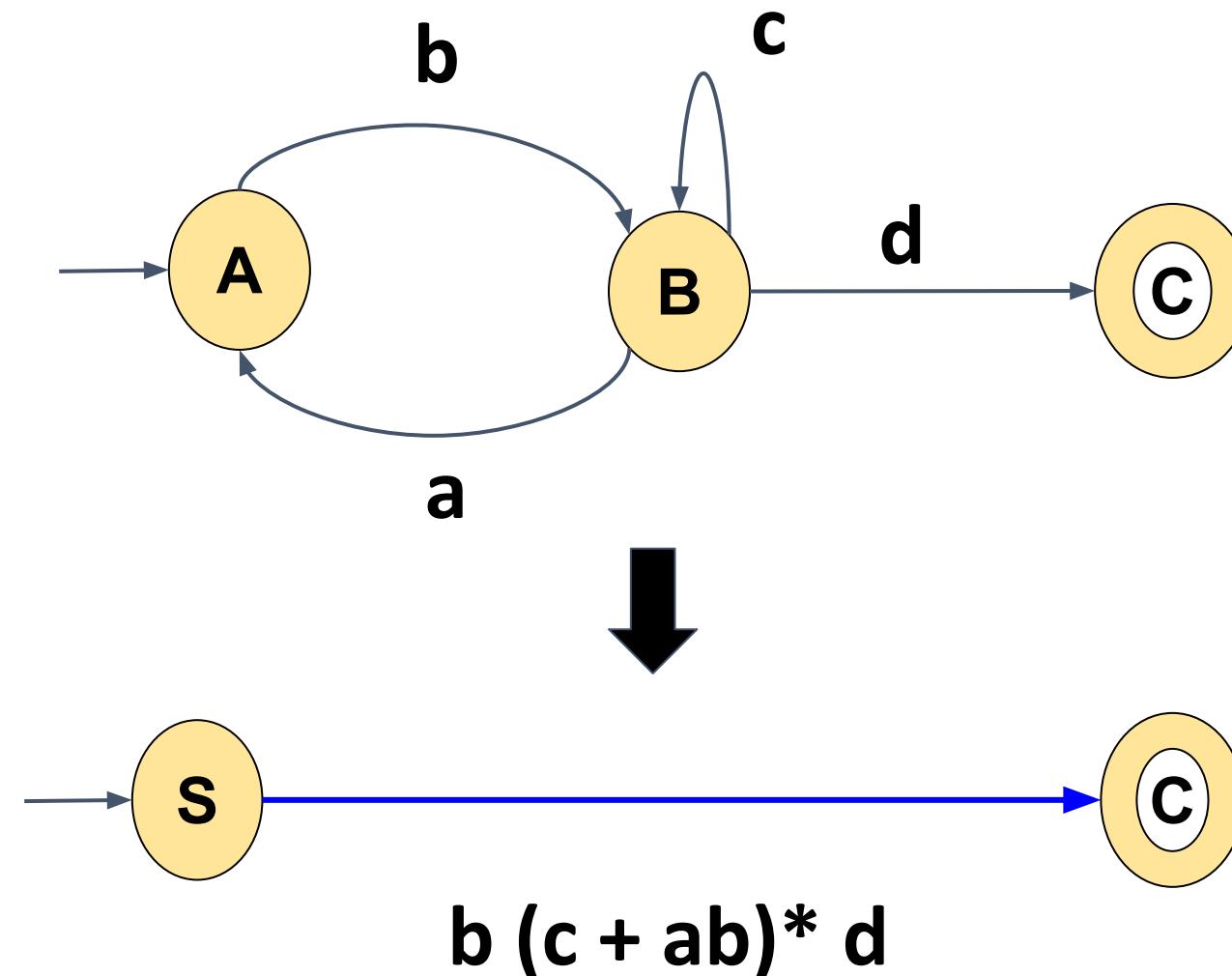
1. Eliminate A

### Example 3 :

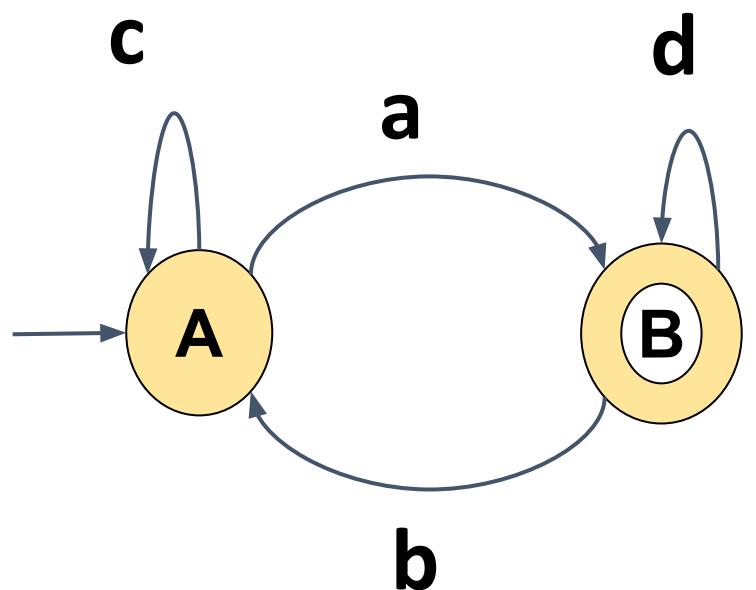


1. Eliminate A
2. Eliminate B

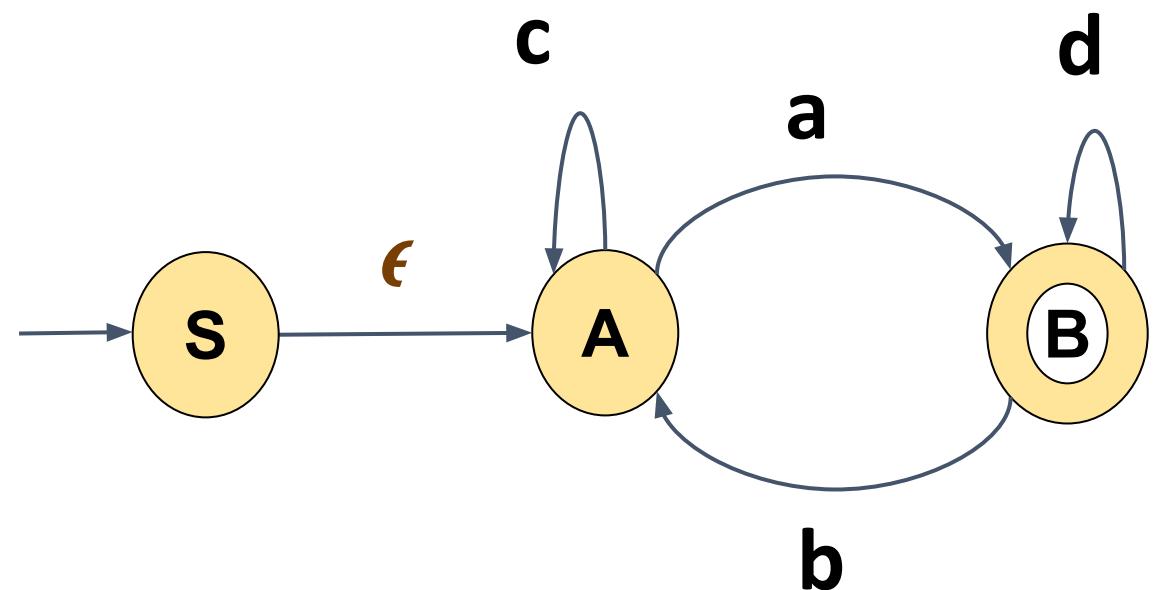
### Example 3 :



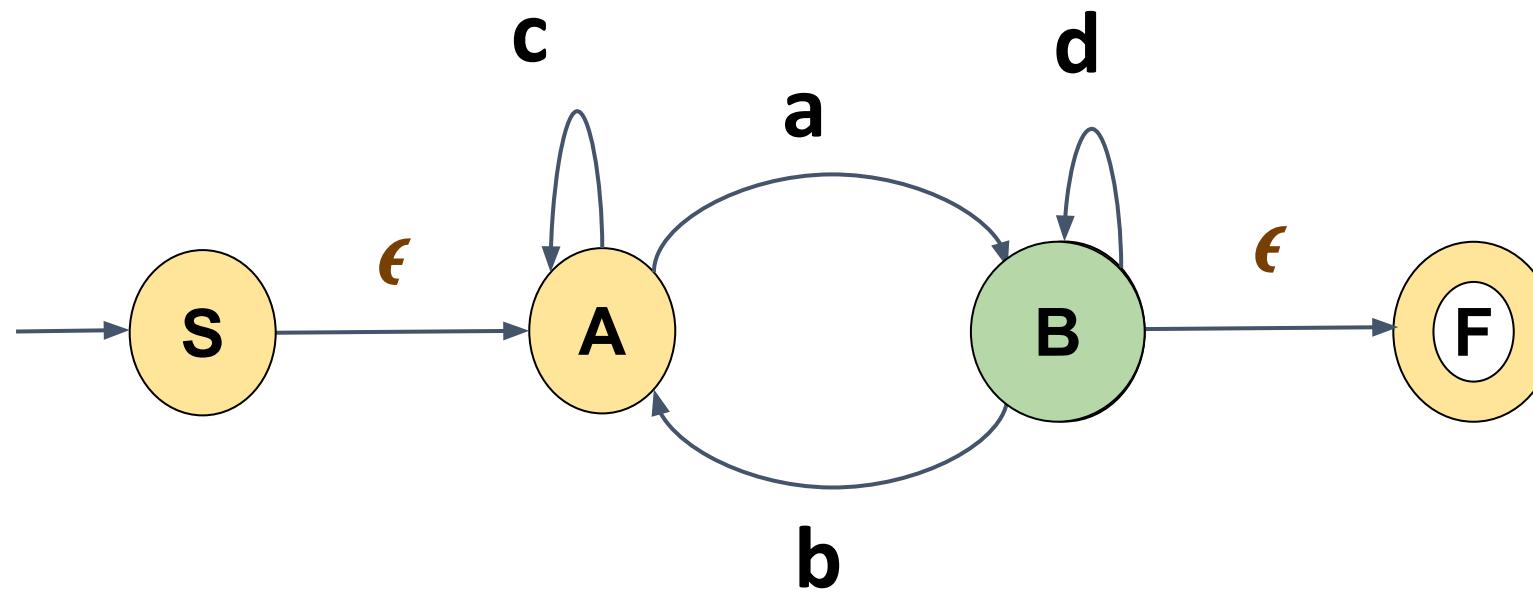
### Example 4 :



### Example 4 :

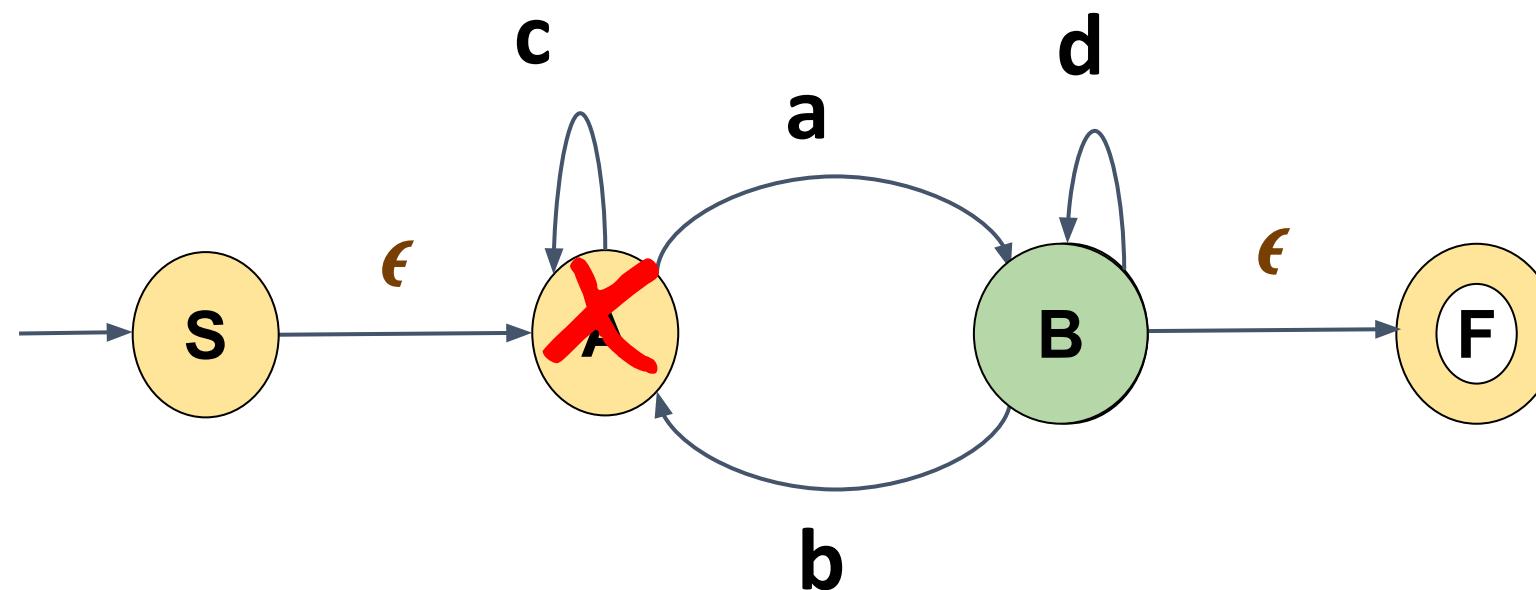


### Example 4 :



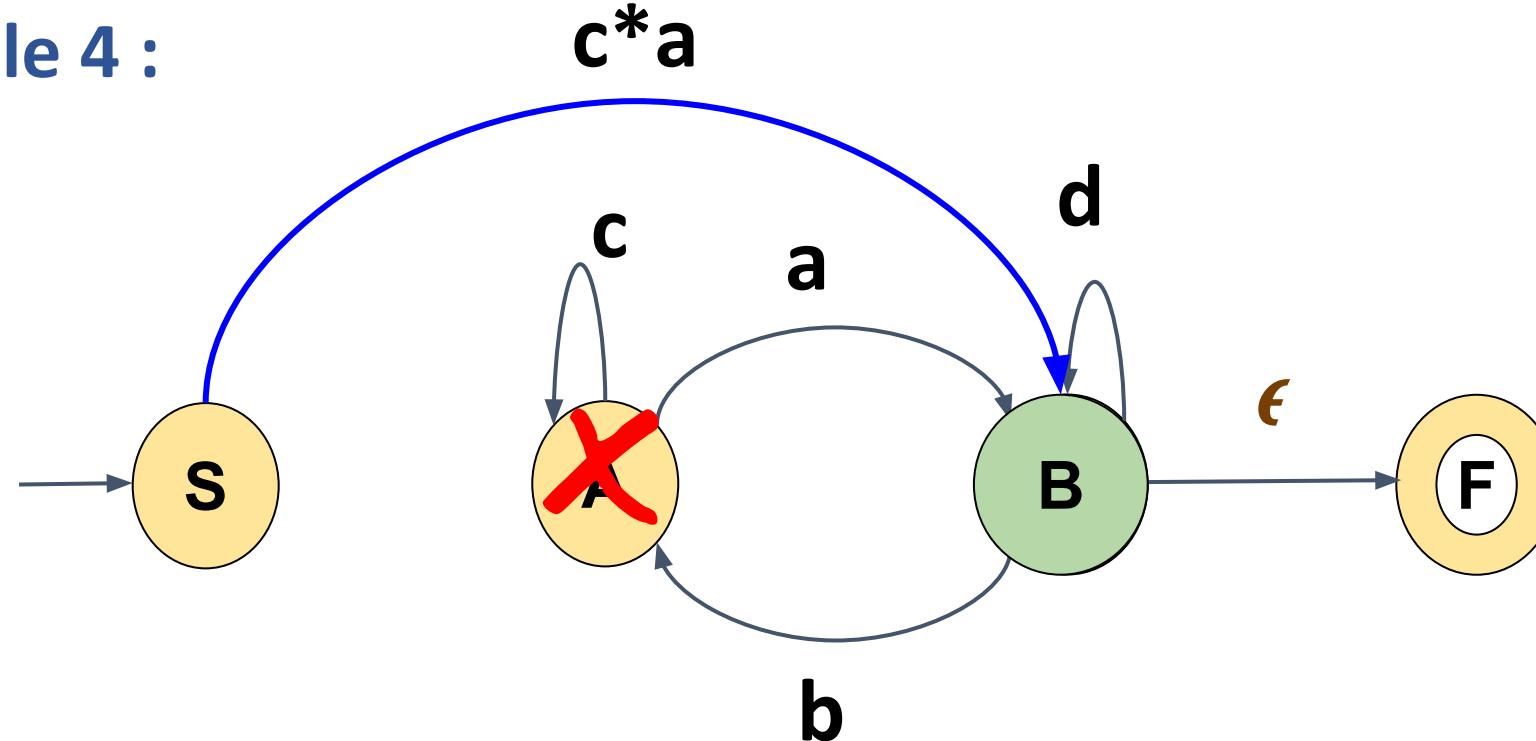
A new Final state F is introduced as there is an Outgoing edge to the existing Final State B.

### Example 4 :



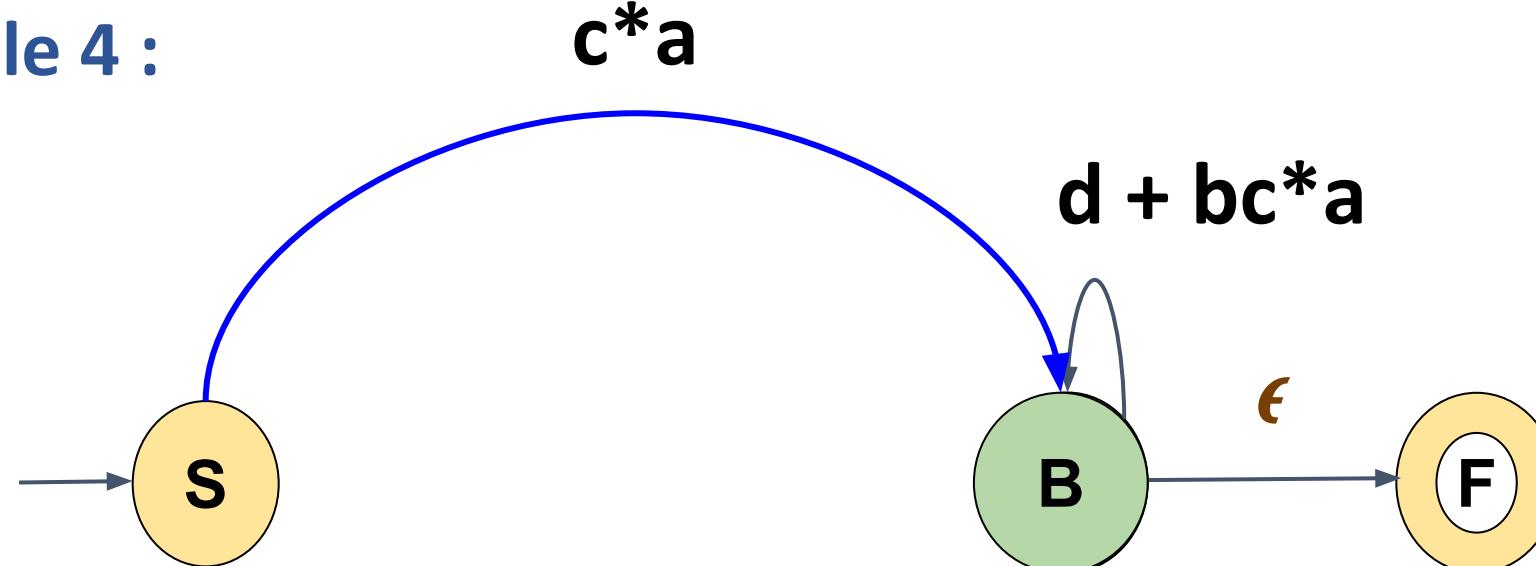
1. Eliminate A

Example 4 :



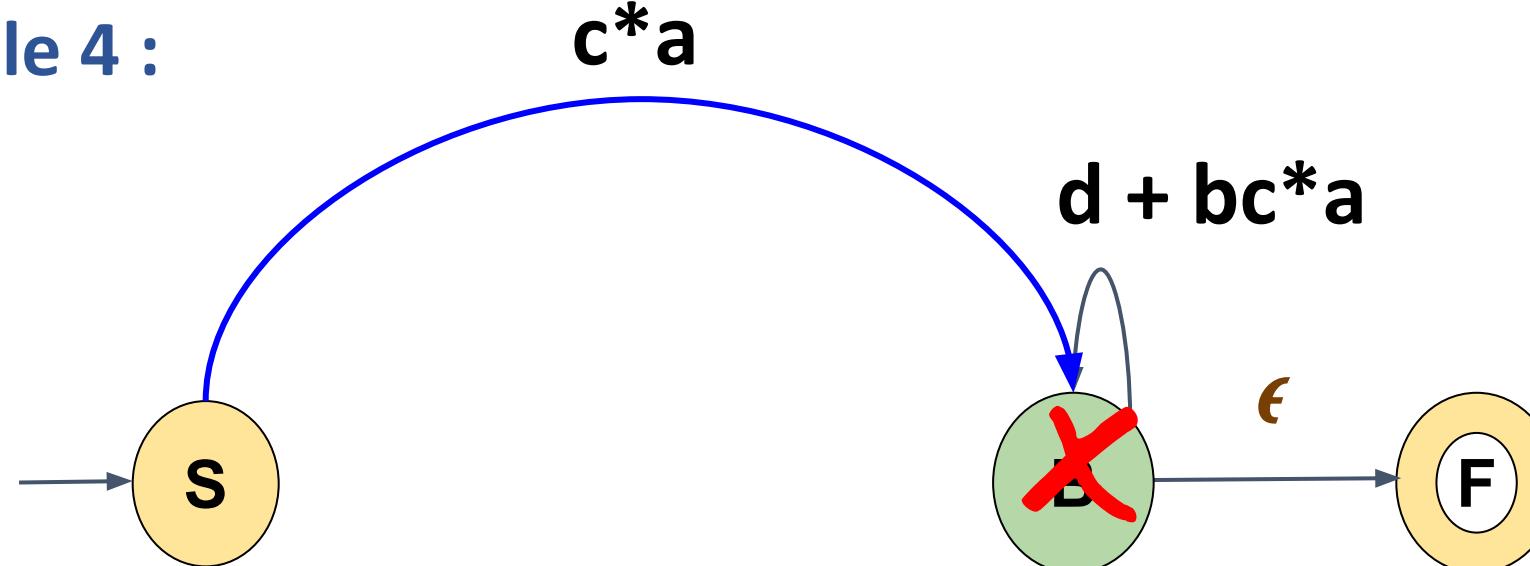
1. Eliminate A

Example 4 :



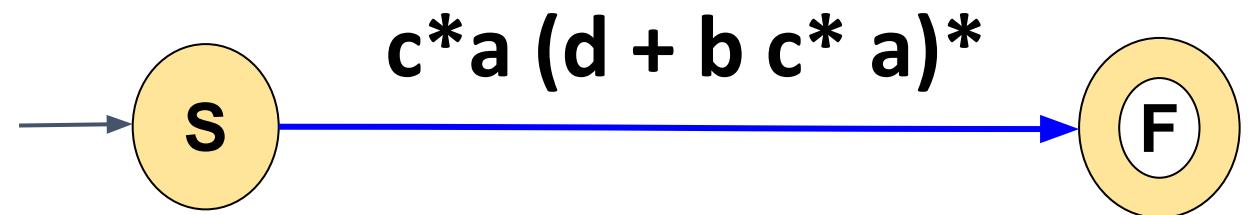
1. Eliminate A

Example 4 :



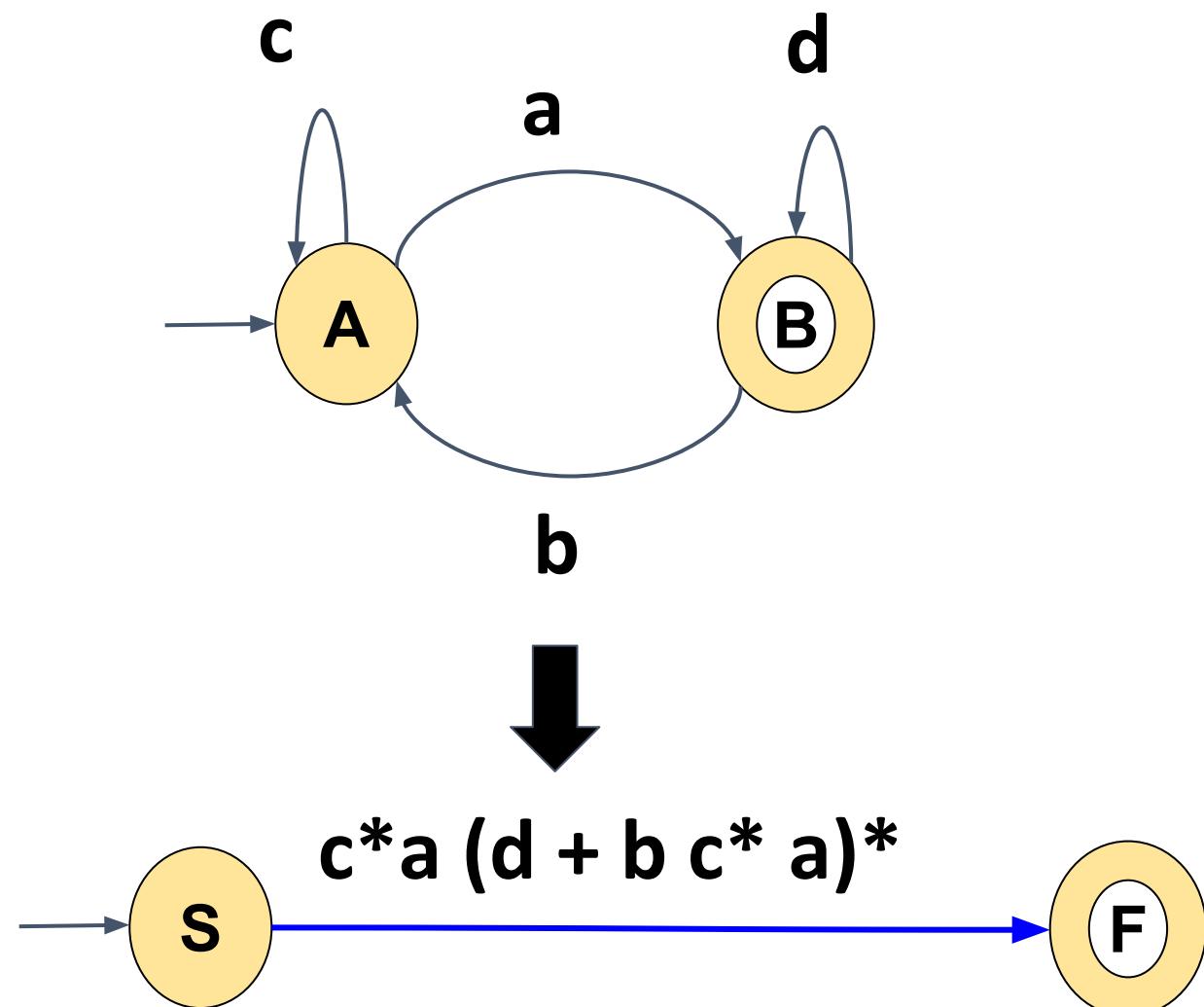
1. Eliminate A
2. Eliminate B

### Example 4 :

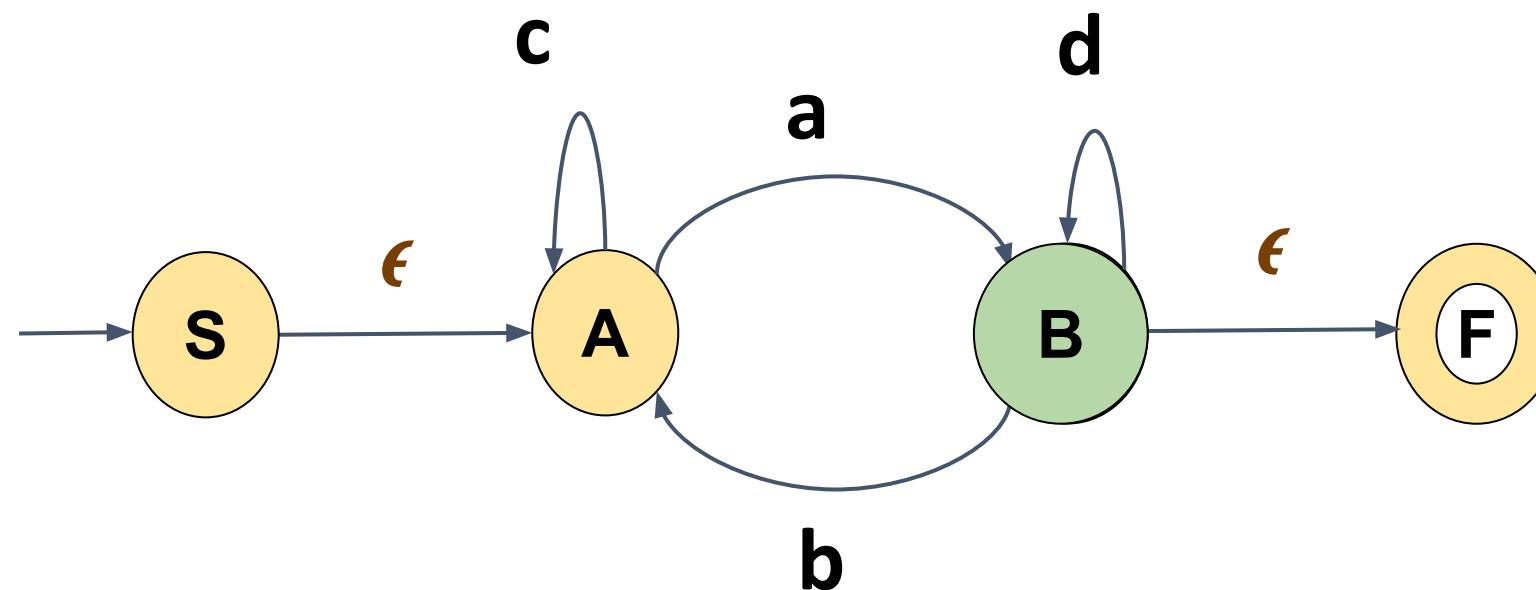


1. Eliminate A
2. Eliminate B

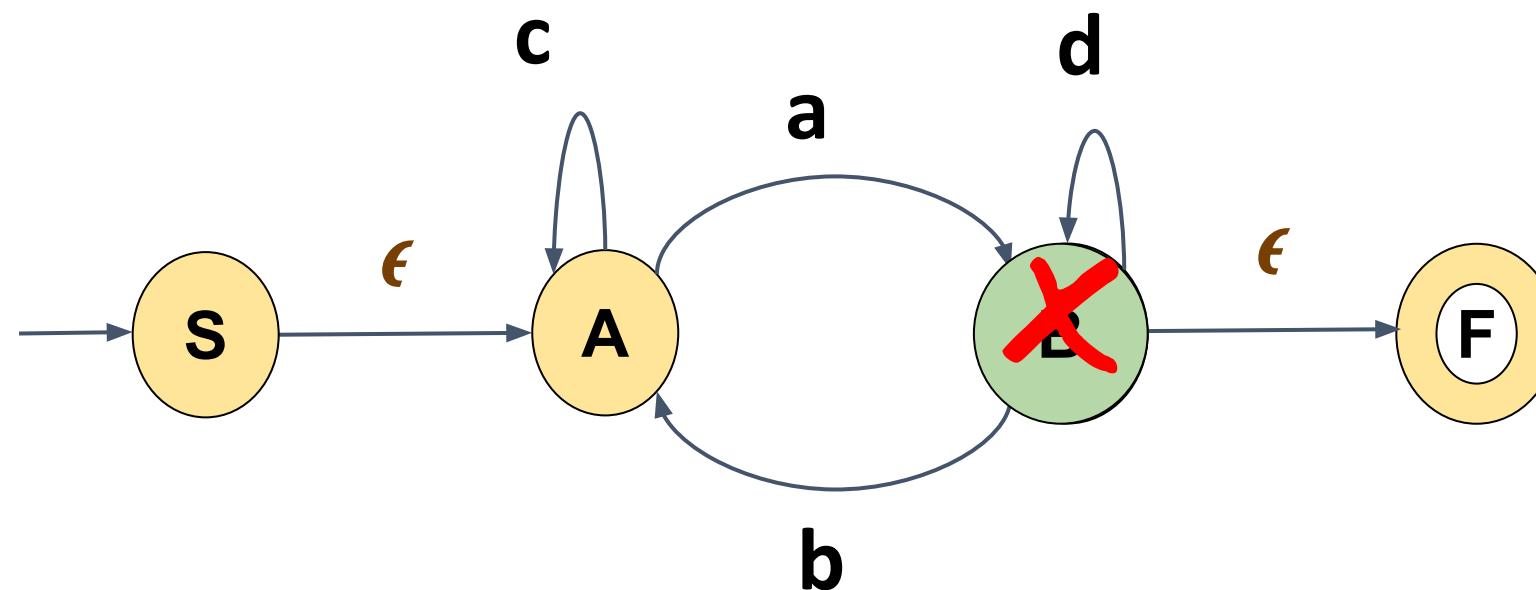
### Example 4 :



### Example 4 :

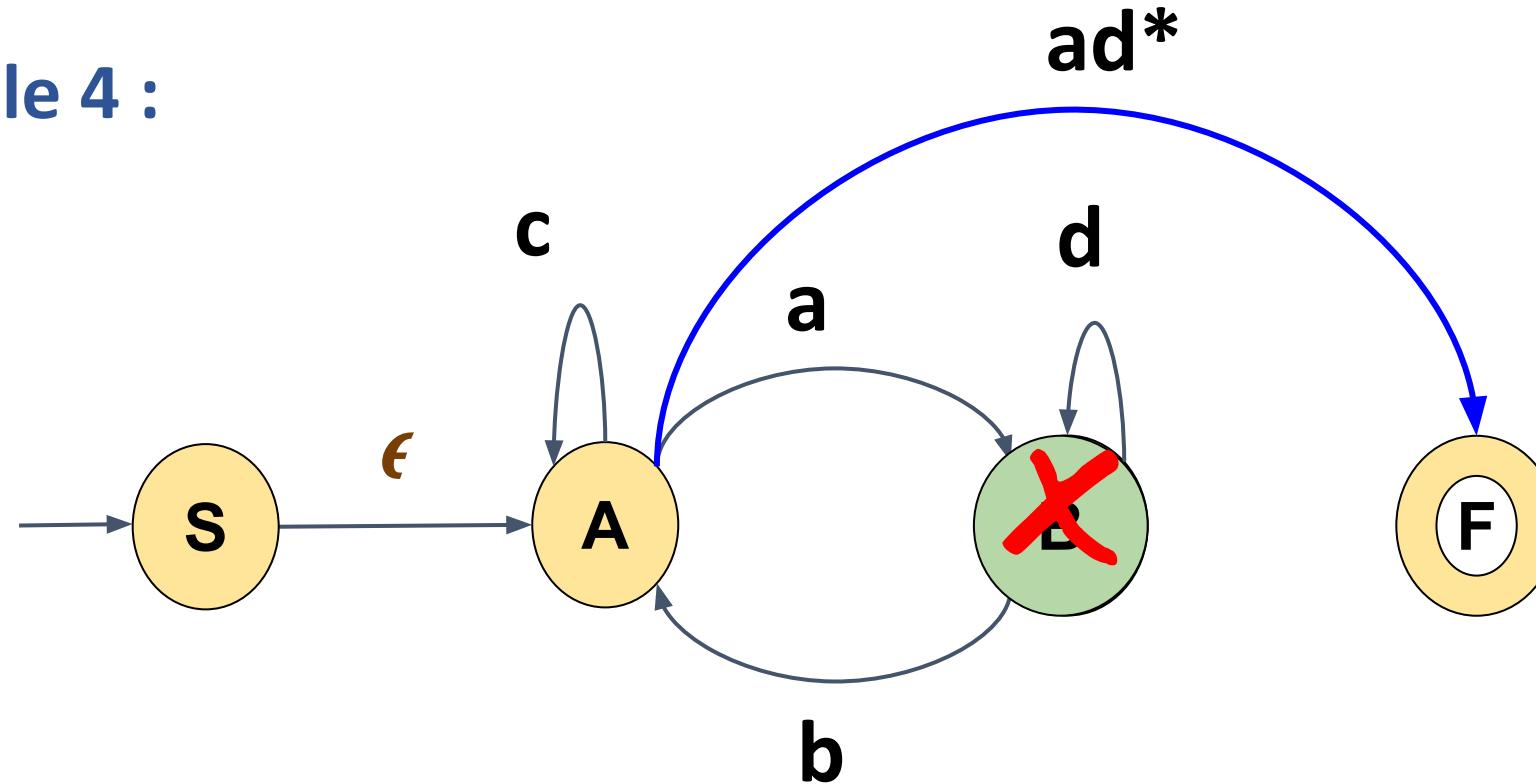


### Example 4 :



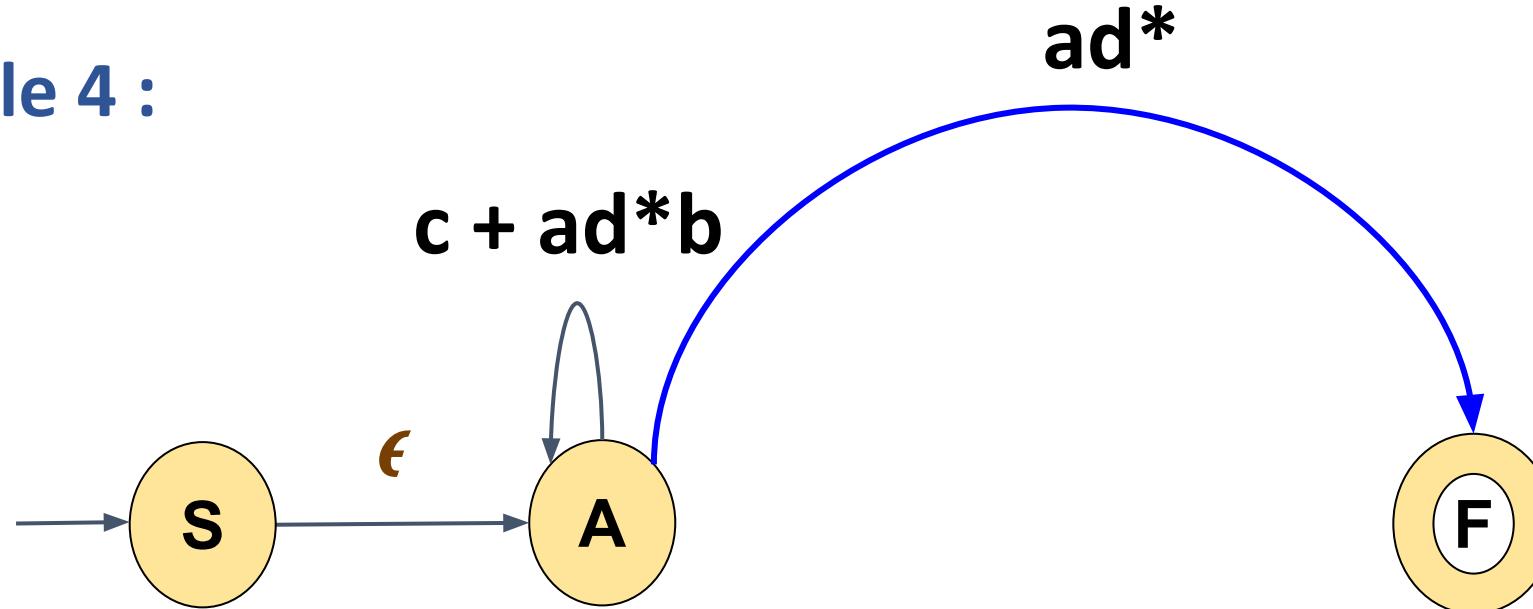
1. Eliminate B

Example 4 :



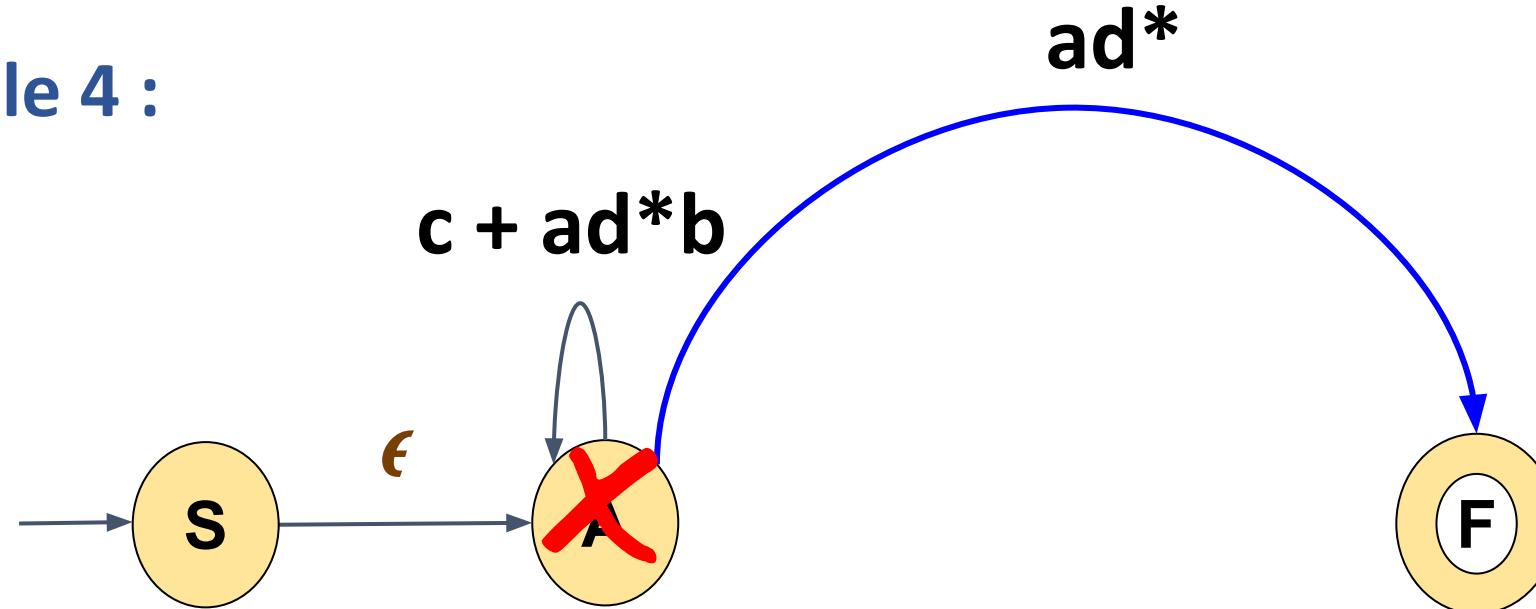
1. Eliminate B

Example 4 :



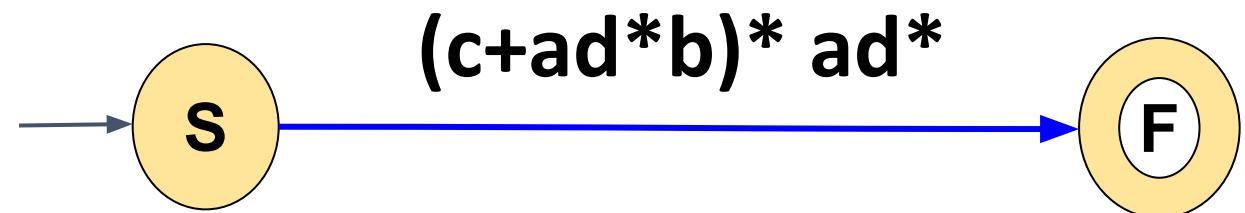
1. Eliminate B

Example 4 :



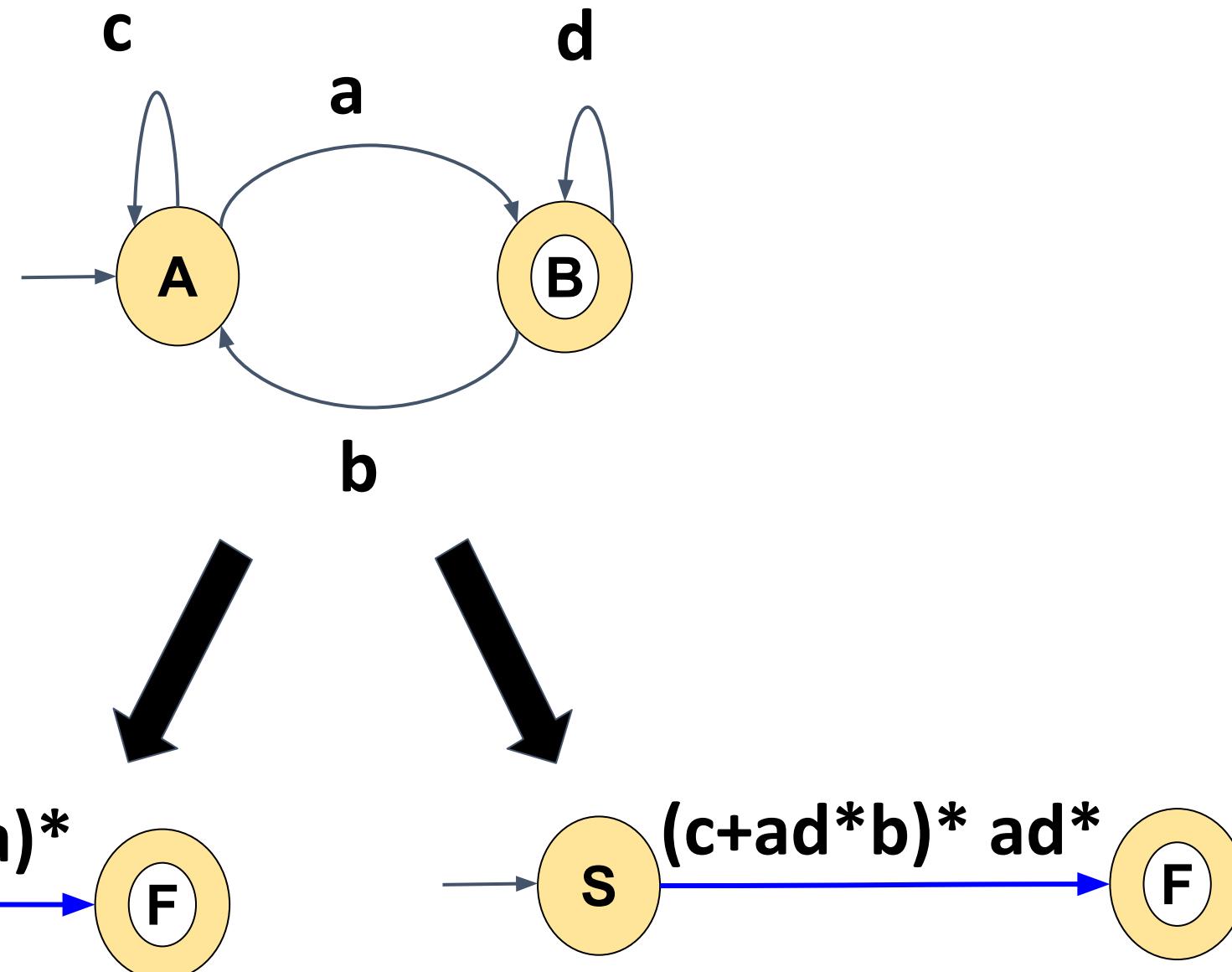
1. Eliminate B
2. Eliminate A

### Example 4 :



1. Eliminate B
2. Eliminate A

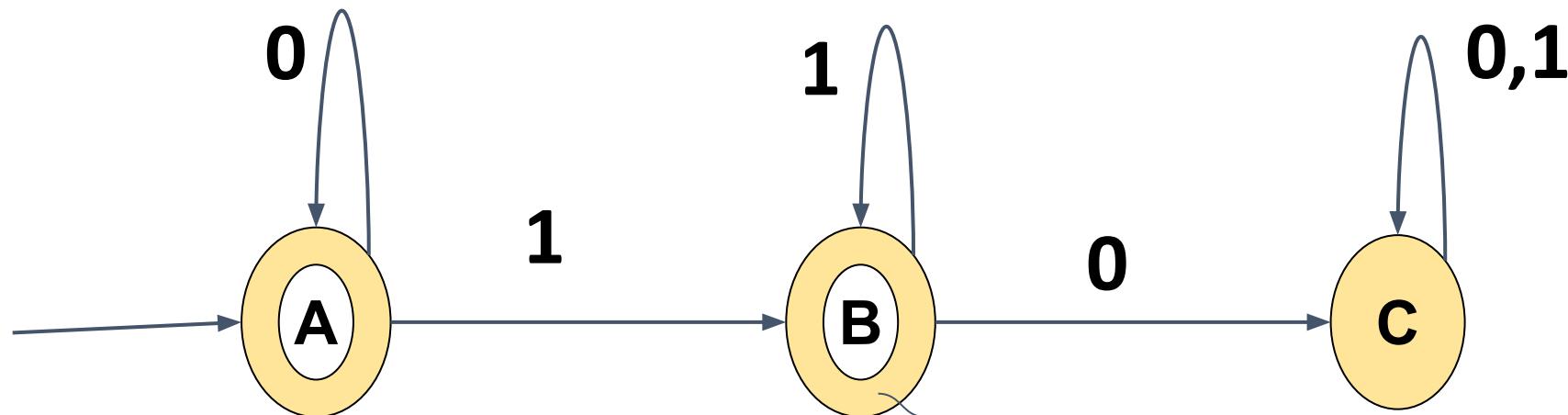
Example 4 :



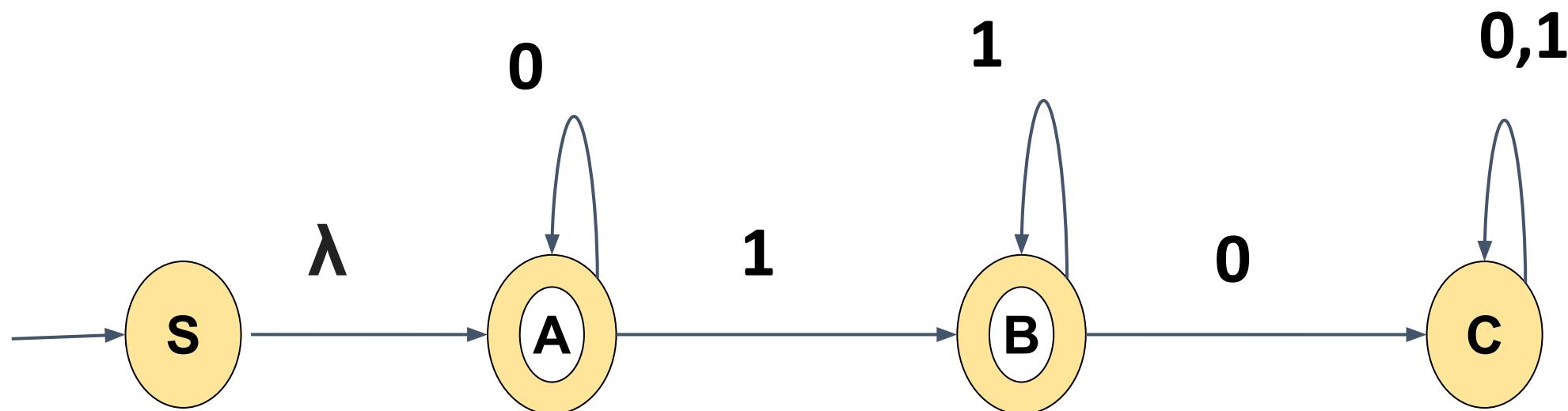
1. Eliminate A
2. Eliminate B

1. Eliminate B
2. Eliminate A

### Example 5 :

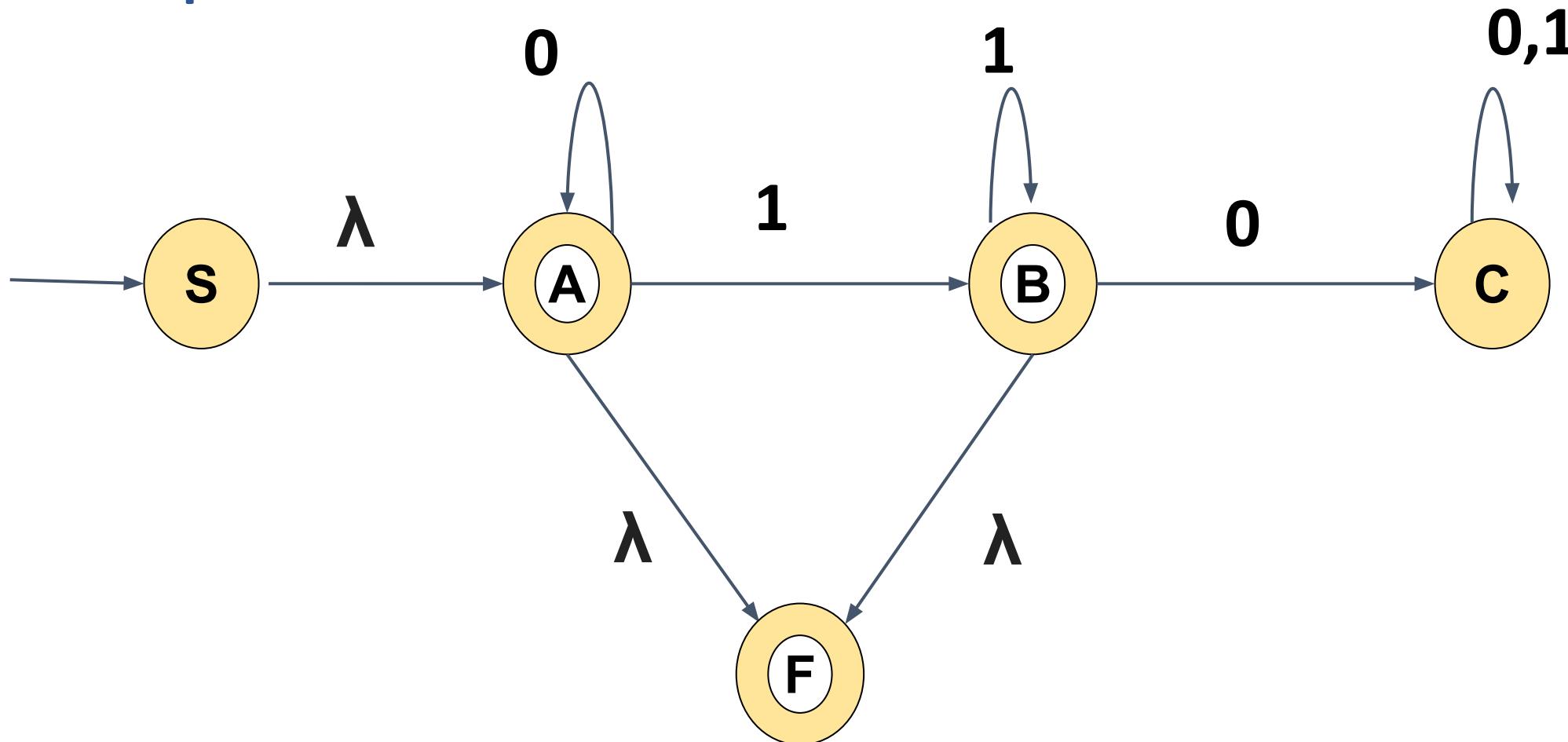


### Example 5 :



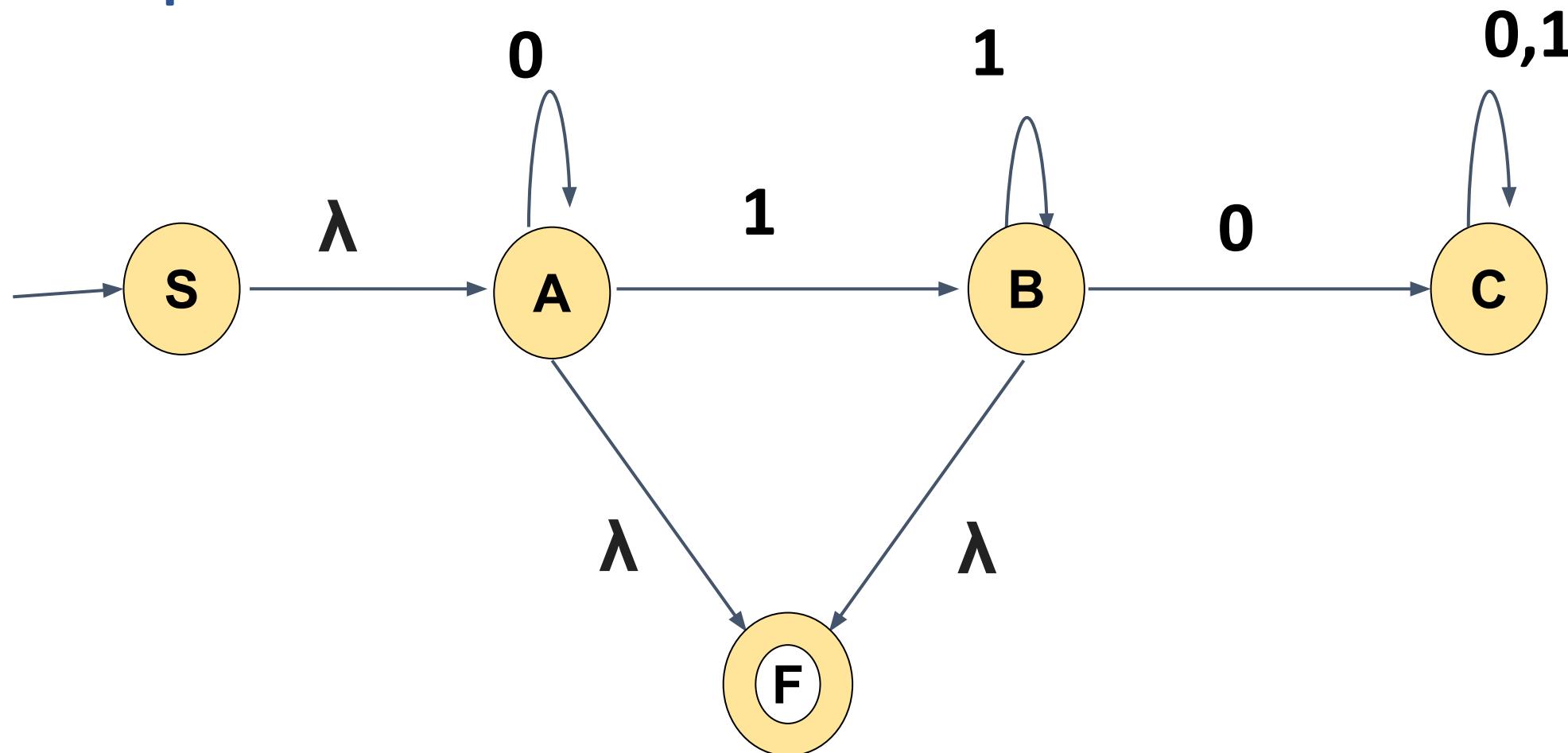
A new start state (S) is introduced as there is an incoming edge to the existing start state

### Example 5 :



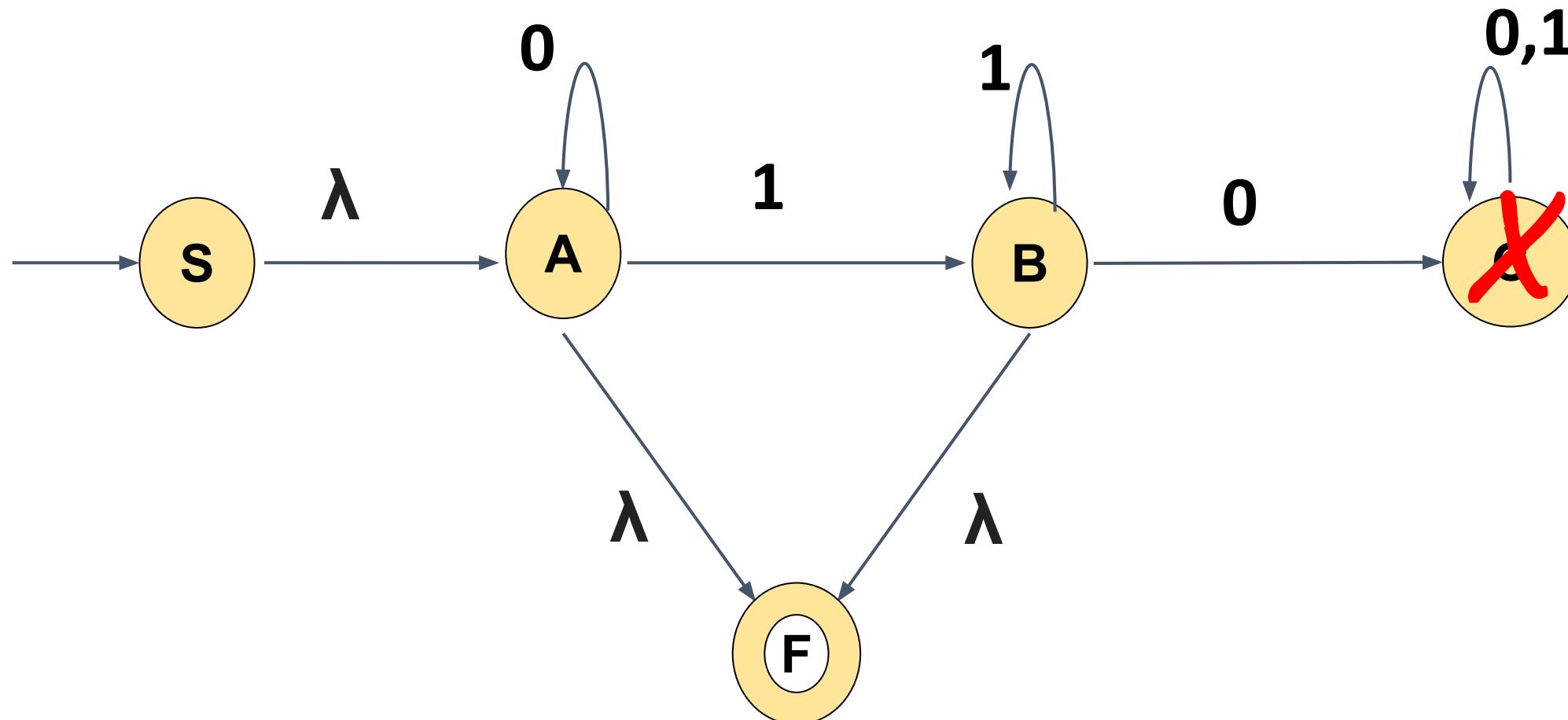
A new final state (F) is introduced as there is an outgoing edge from the existing final state and we must have a single accepting state

Example 5 :



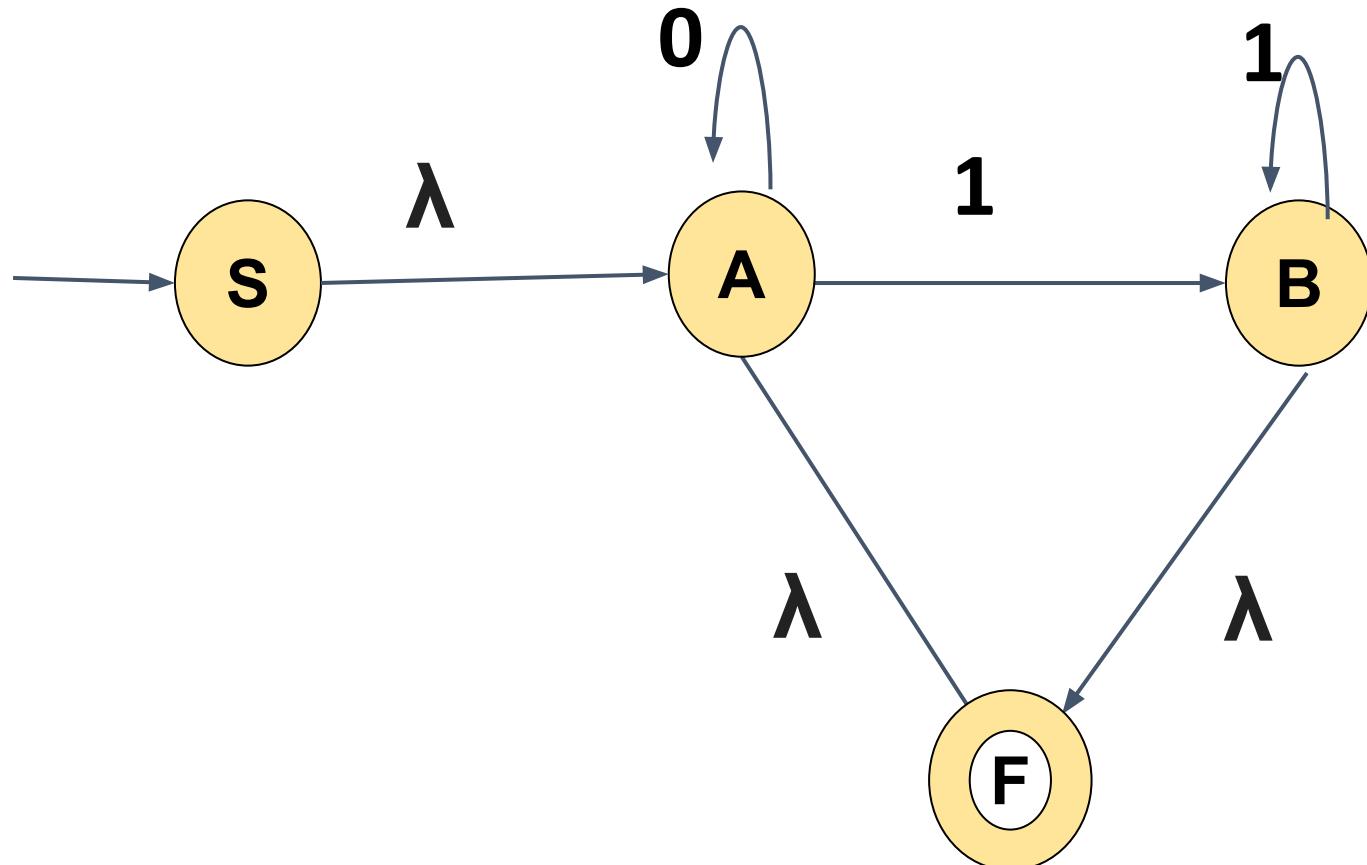
Previous final state is made as non final state

### Example 5 :



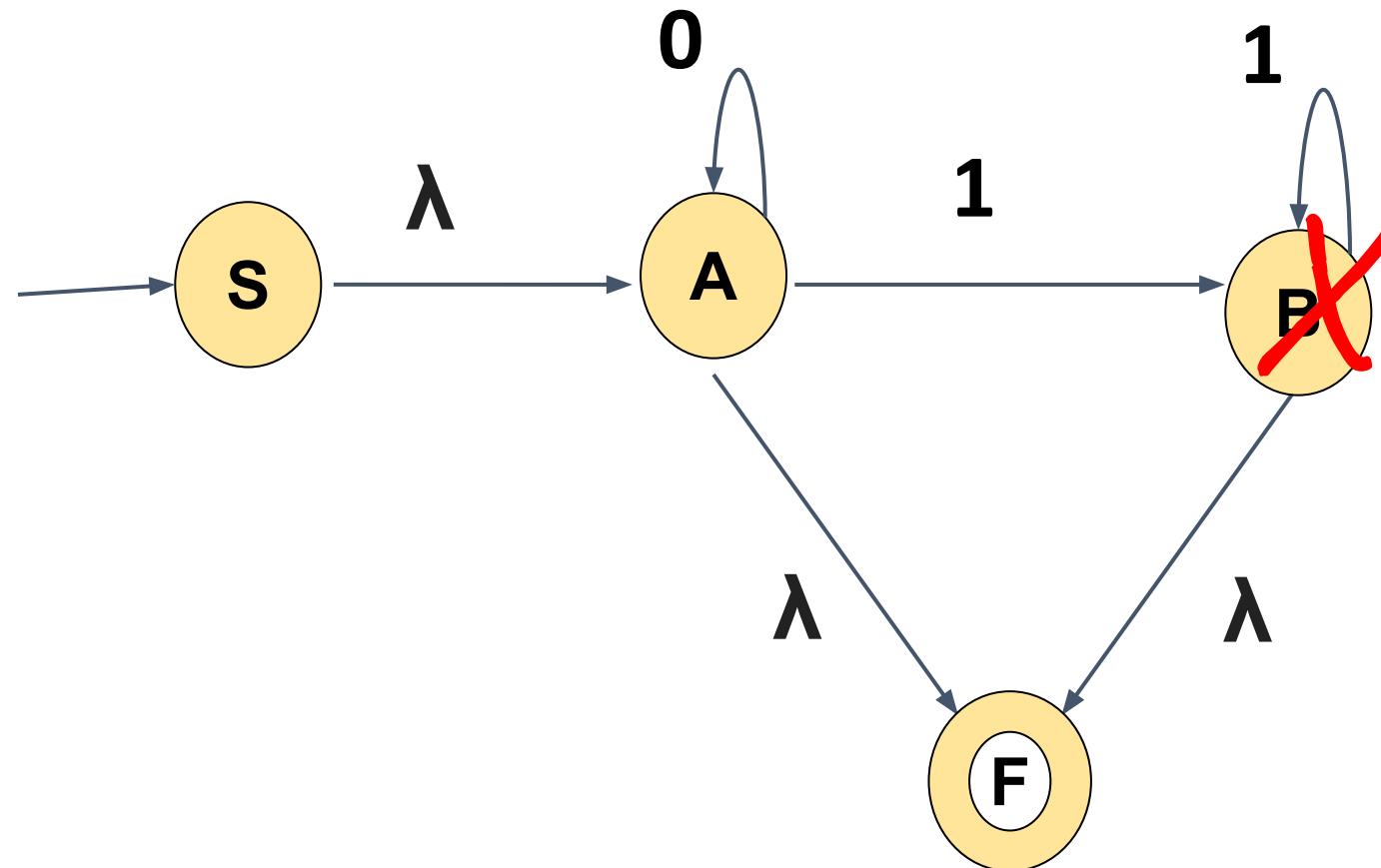
1. Eliminate C

### Example 5 :



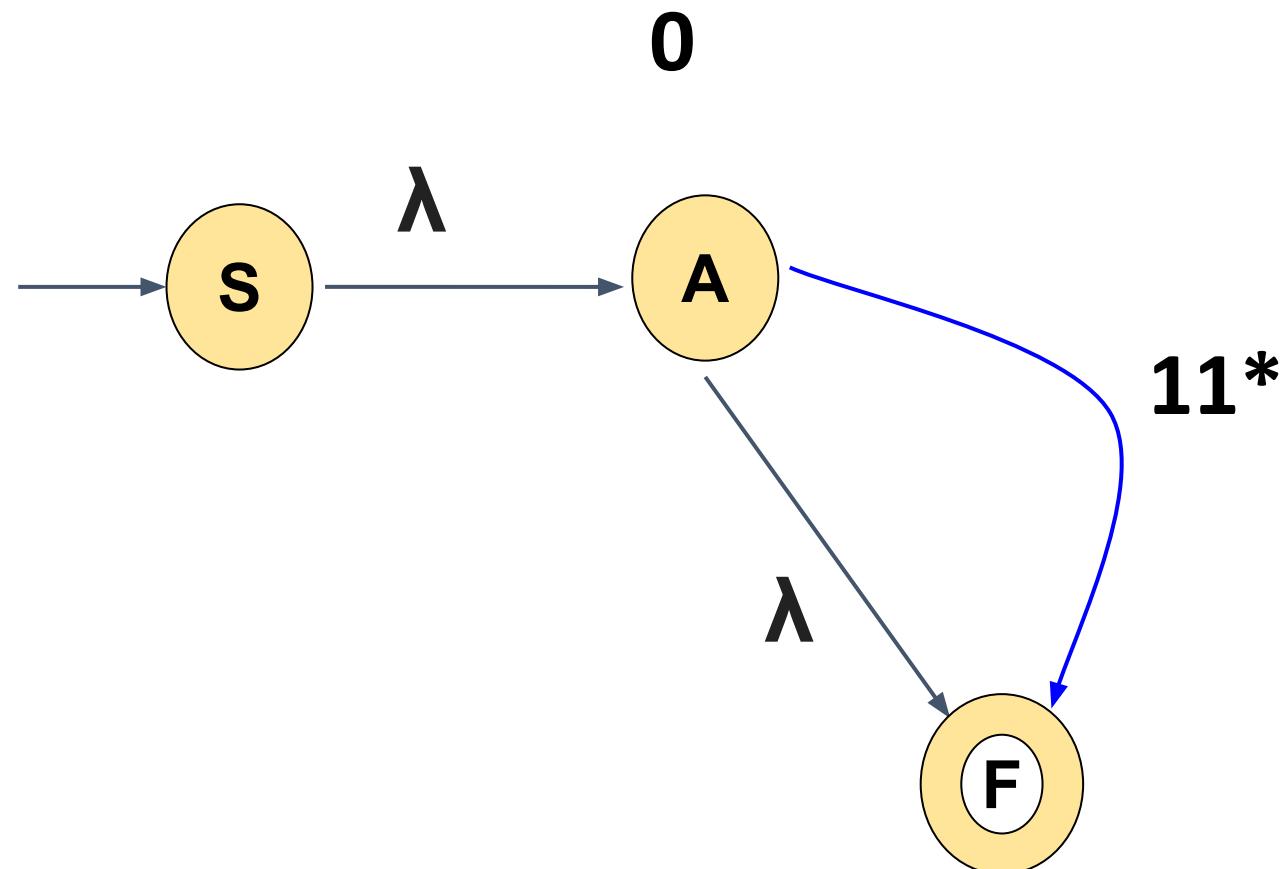
1. Eliminate C

### Example 5 :



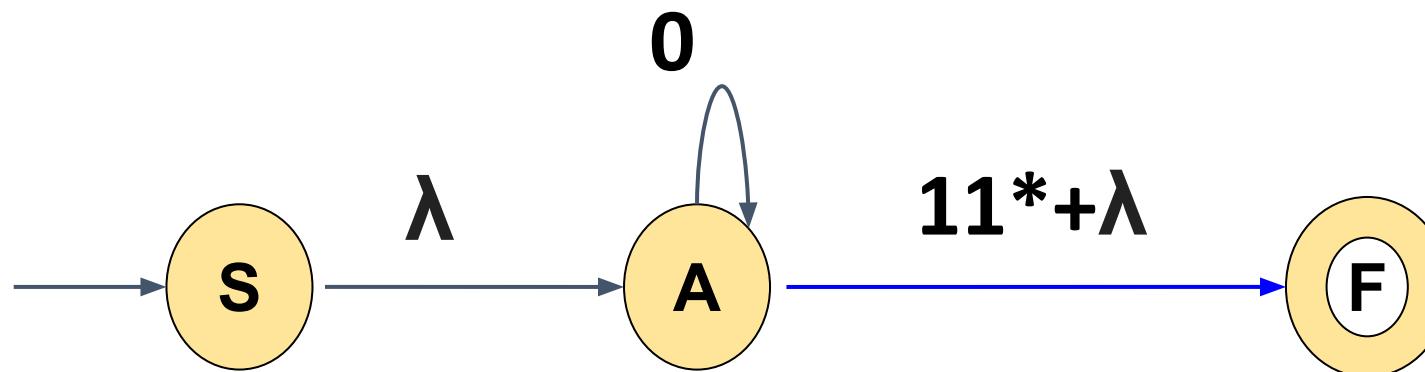
1. Eliminate C
2. Eliminate B

### Example 5 :



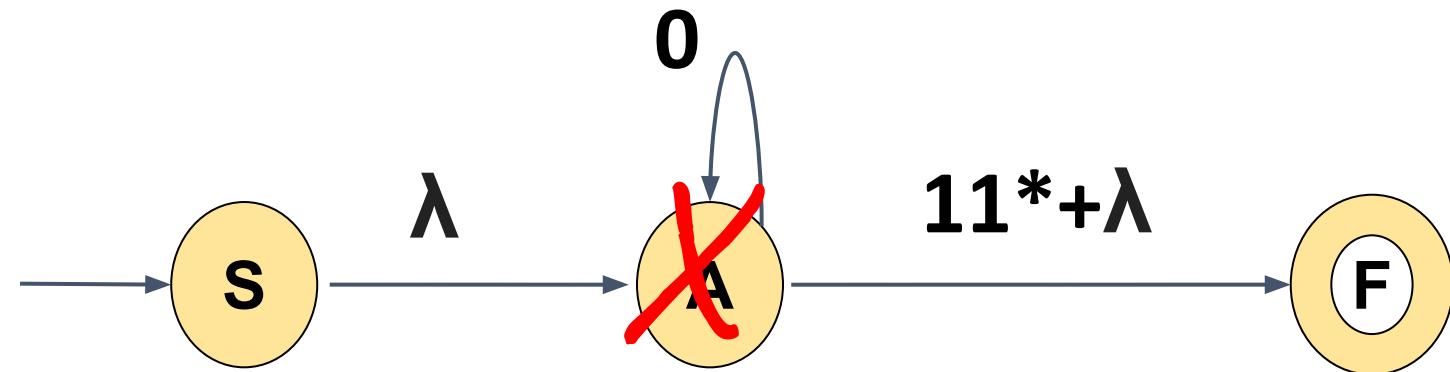
1. Eliminate C
2. Eliminate B

### Example 5 :



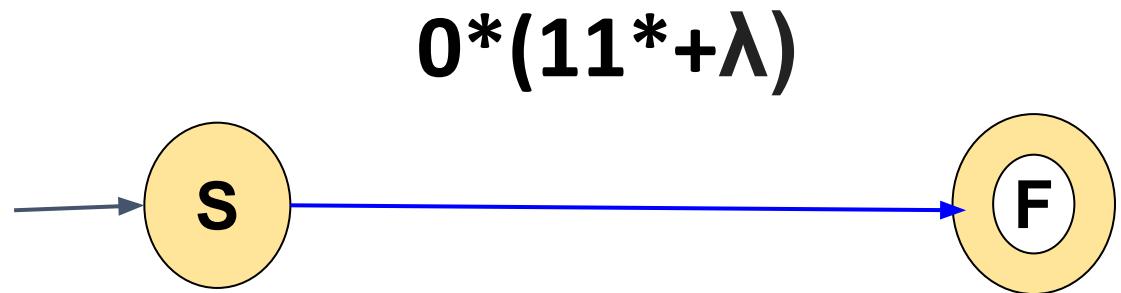
1. Eliminate C
2. Eliminate B

### Example 5 :



1. Eliminate C
2. Eliminate B
3. Eliminate A

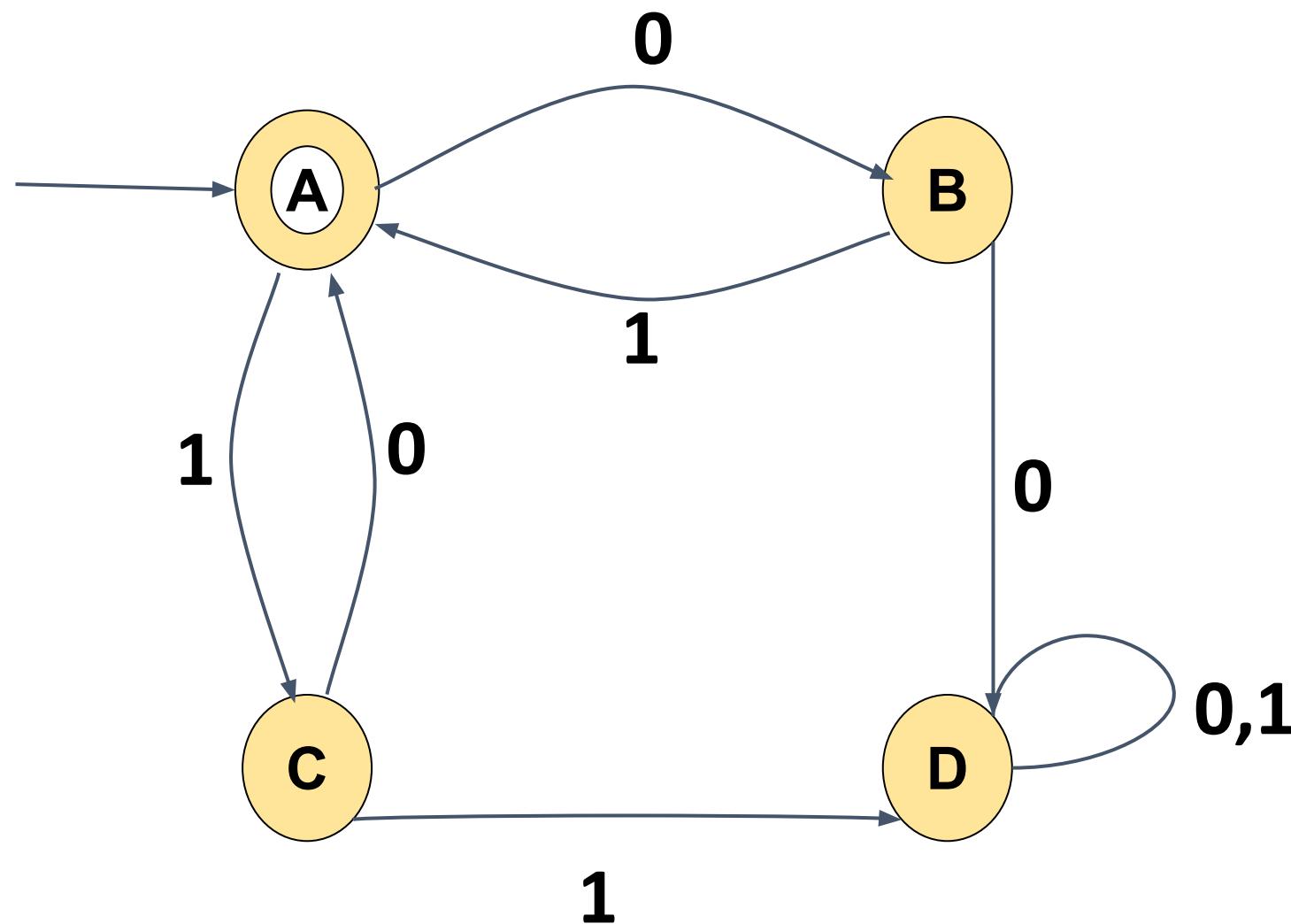
### Example 5 :



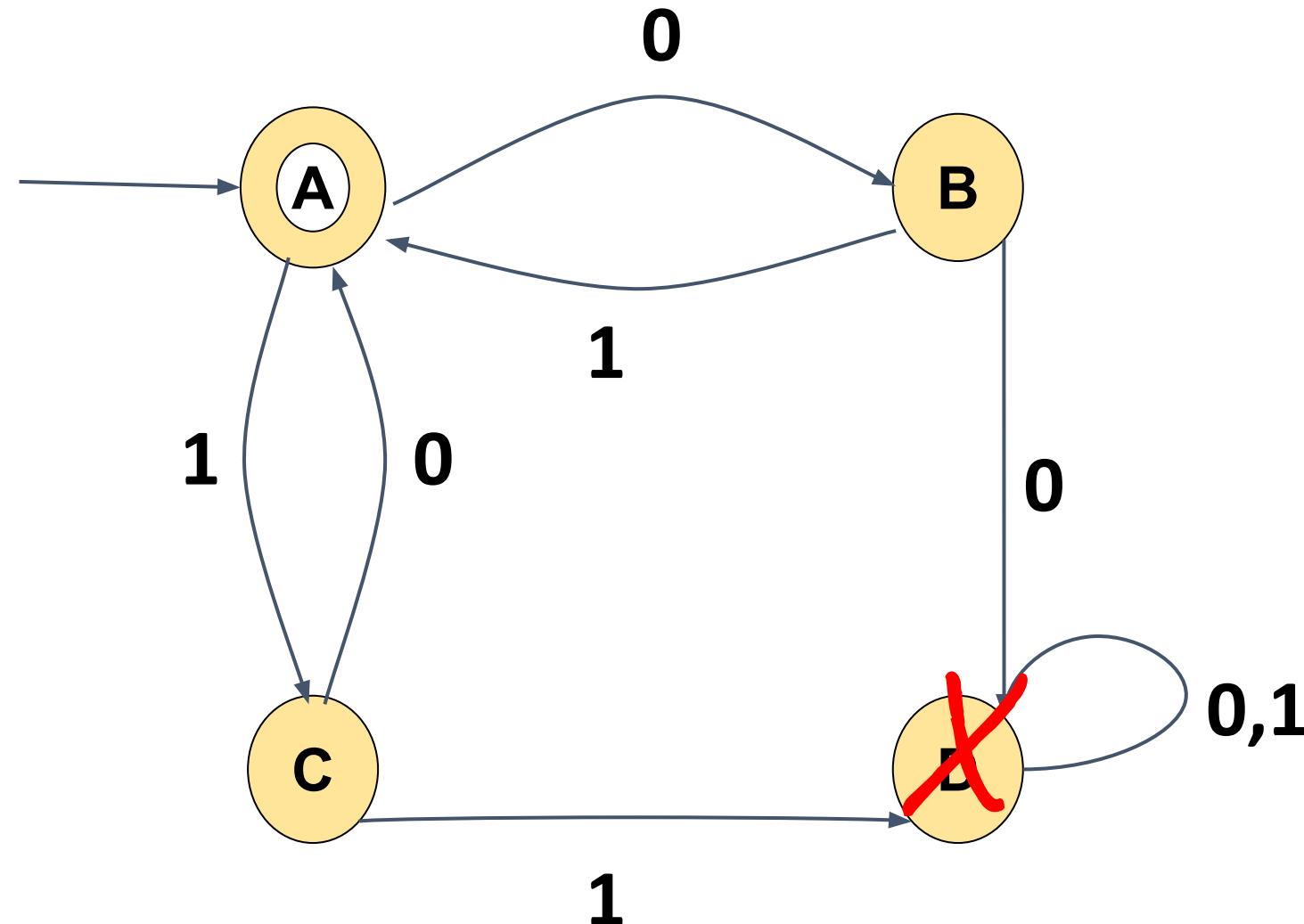
$$\begin{aligned} &= 0(11^* + \lambda) \\ &= 0^*(1^* + \lambda) \end{aligned}$$

$$RE = 0^*1^*$$

### Example 6 :

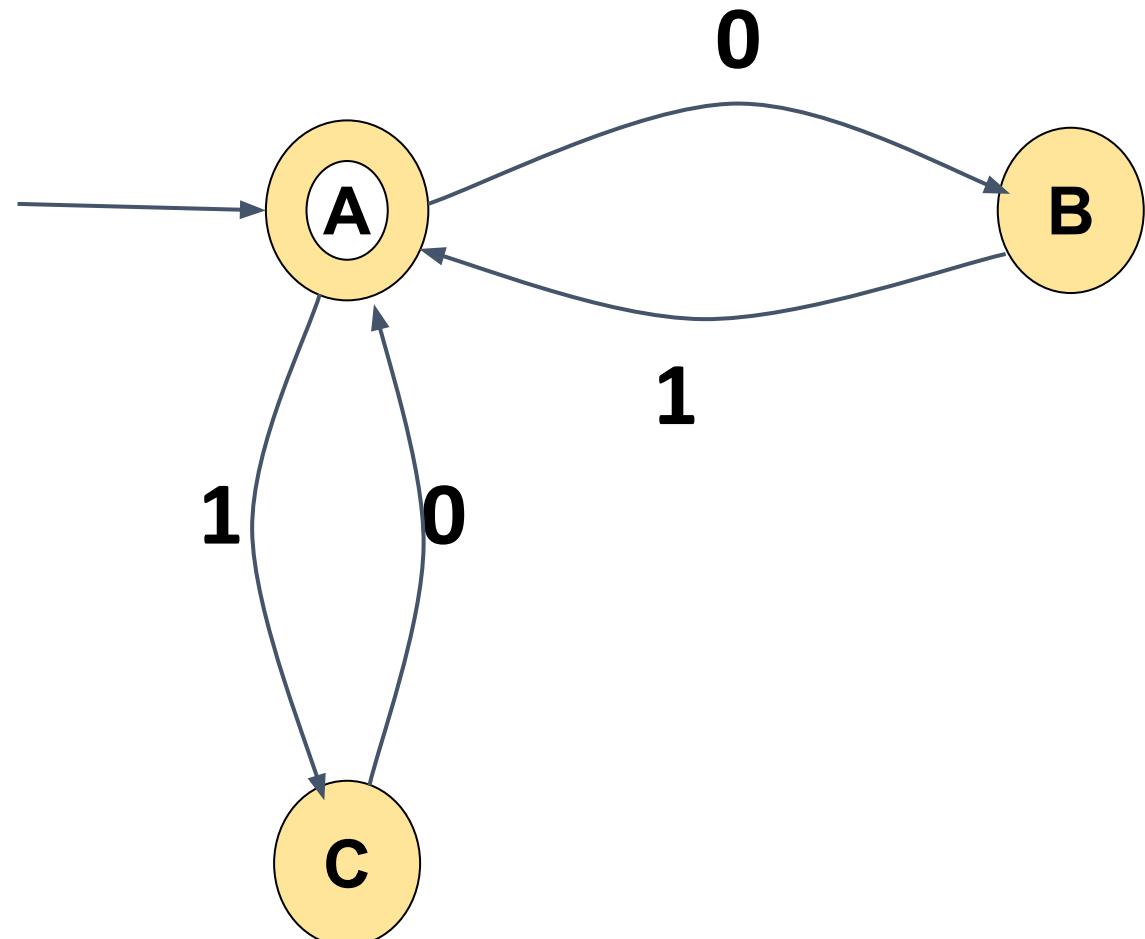


### Example 6 :

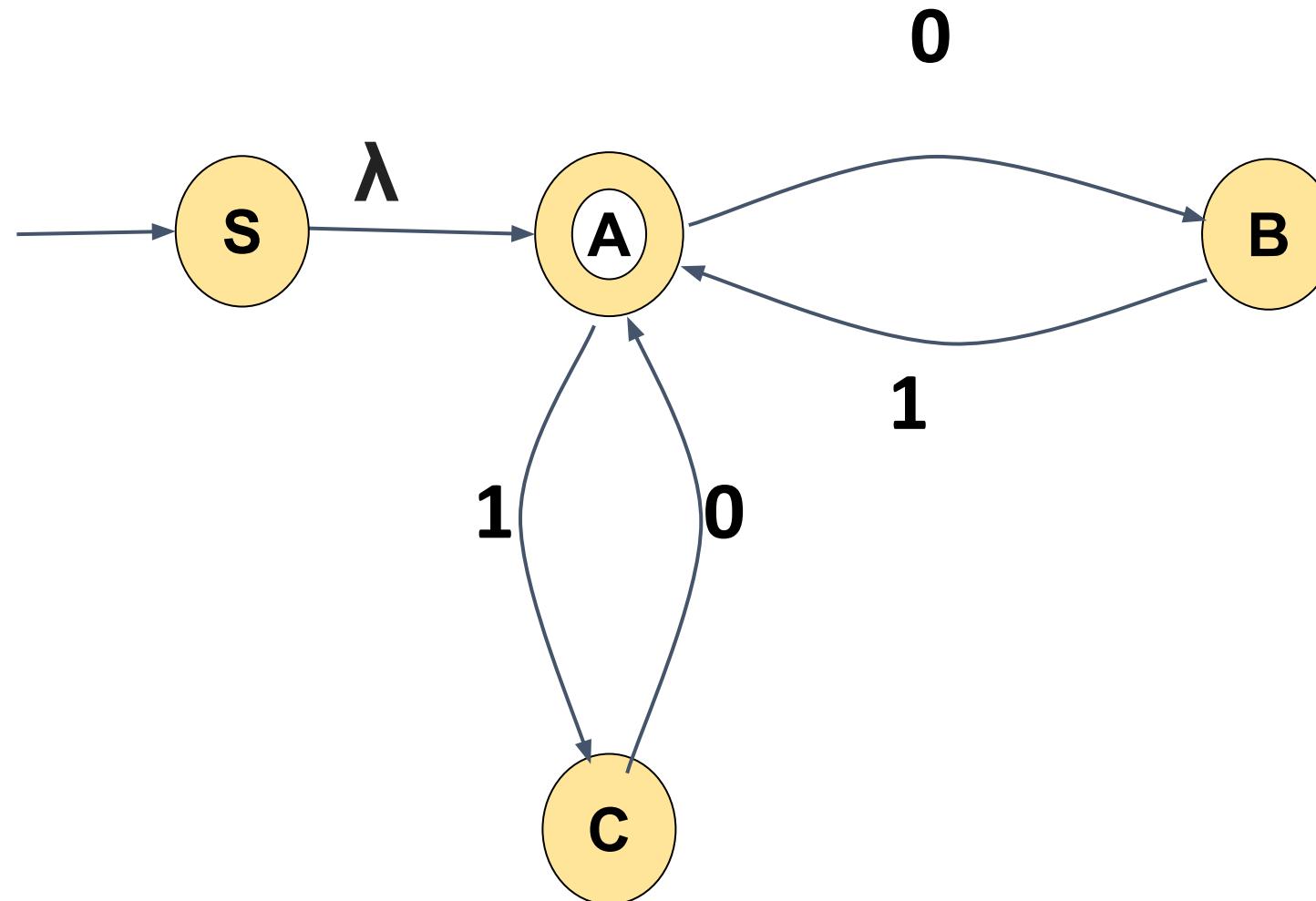


1. Eliminate D  
(Dead state)

### Example 6 :

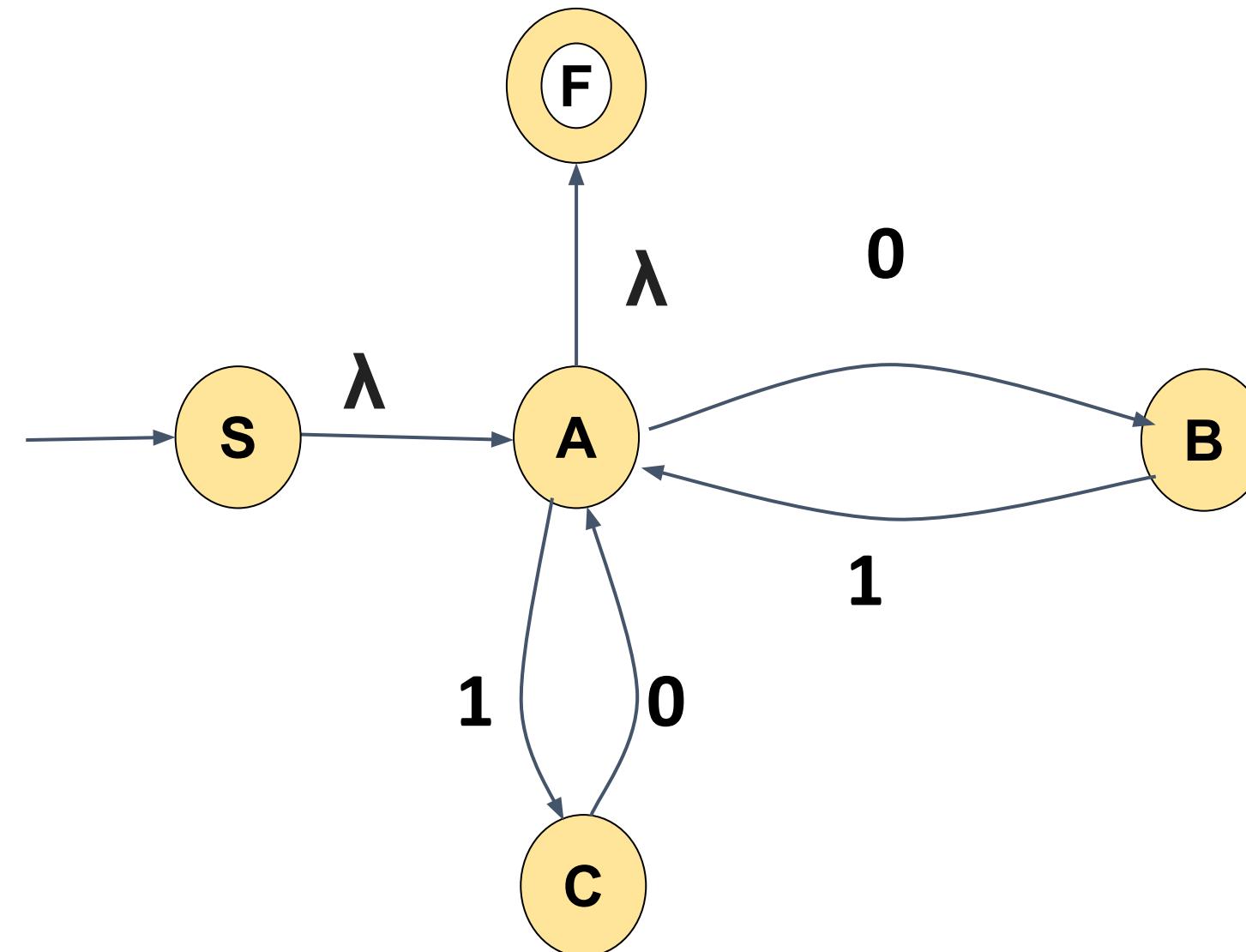


### Example 6 :



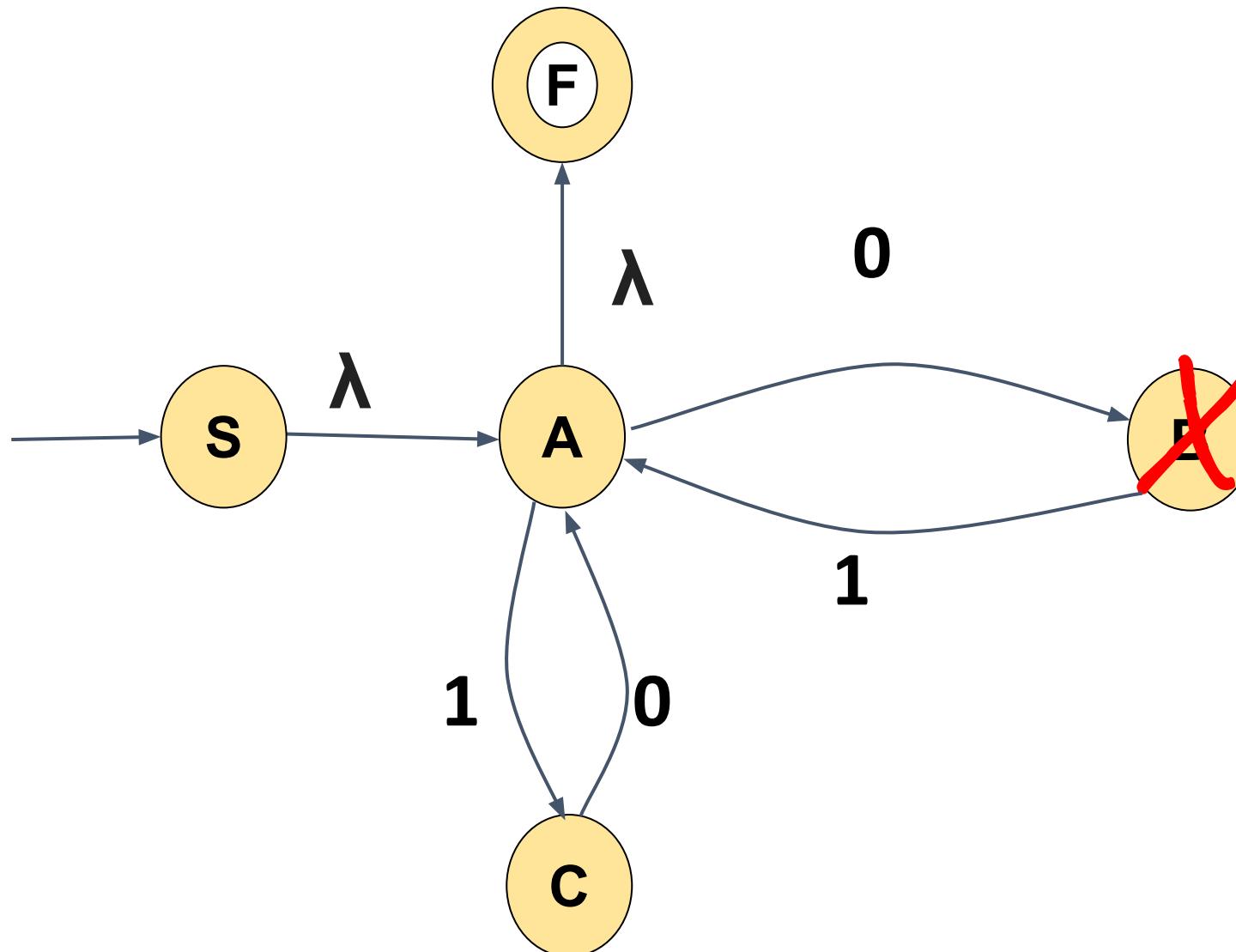
A new start state (S) is introduced as there is an incoming edge to the existing start state

### Example 6 :



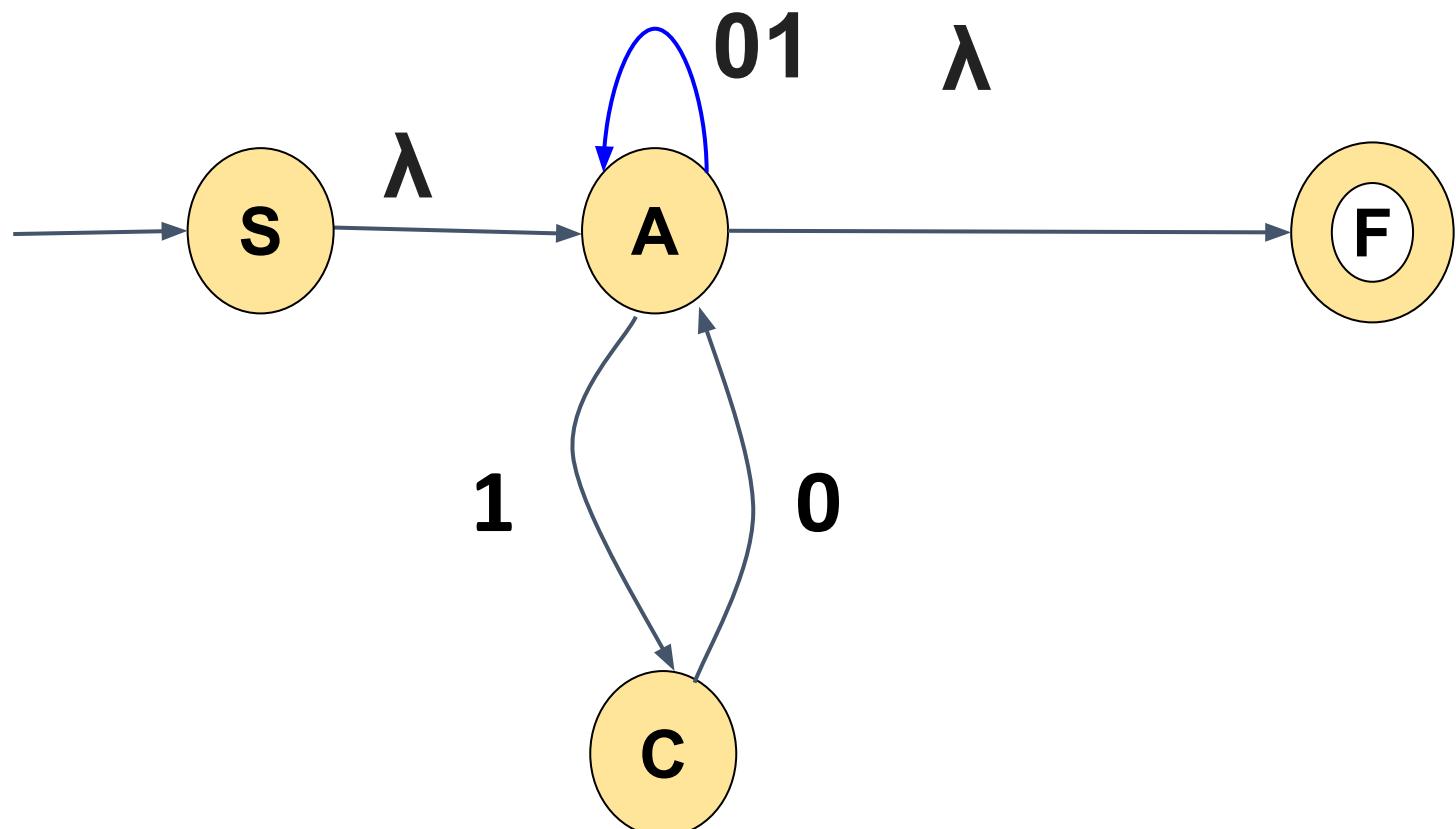
A new final state (F) is introduced as there is an outgoing edge from the existing final state

### Example 6 :



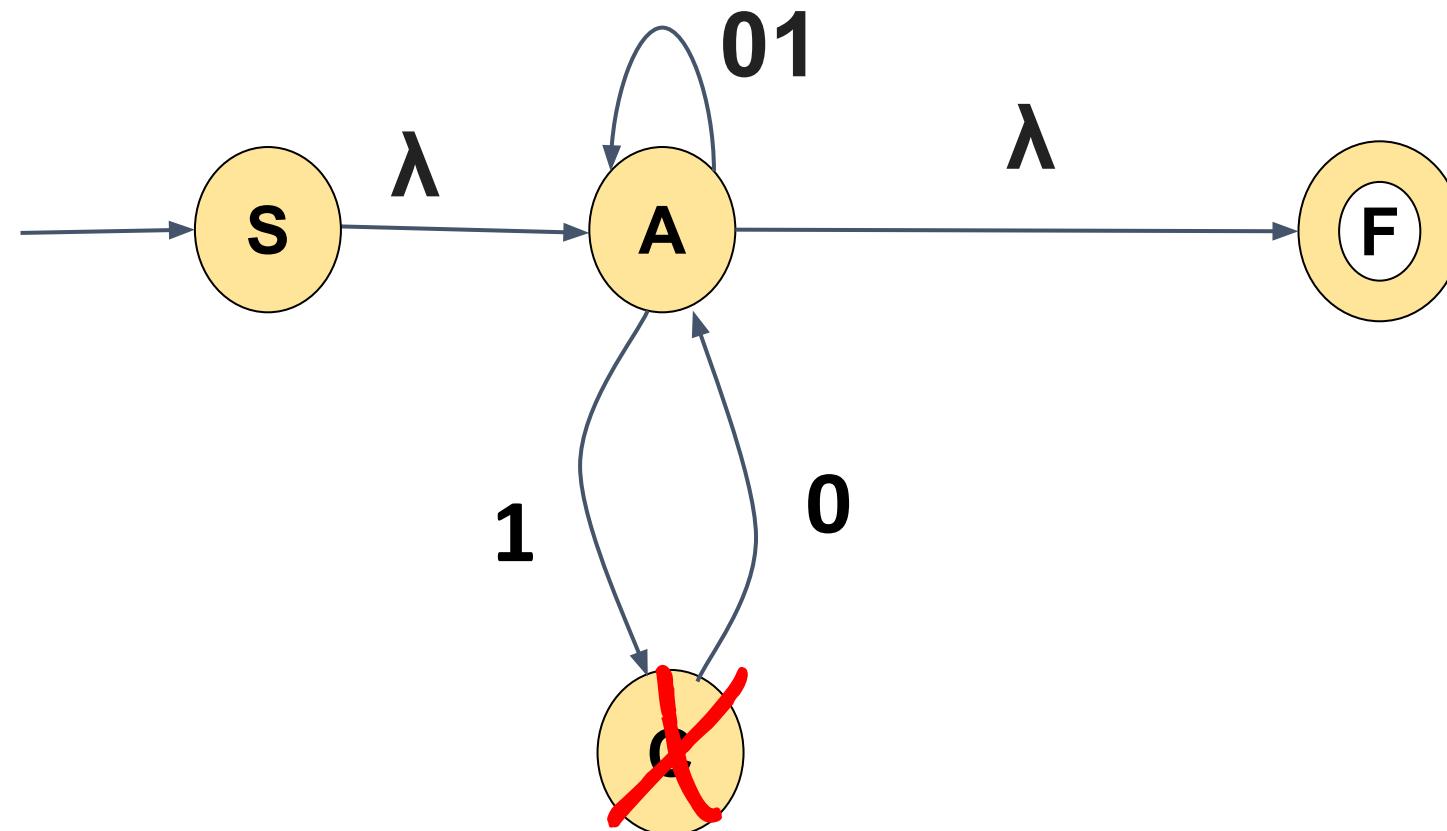
1. Eliminate D
2. Eliminate B

### Example 6 :



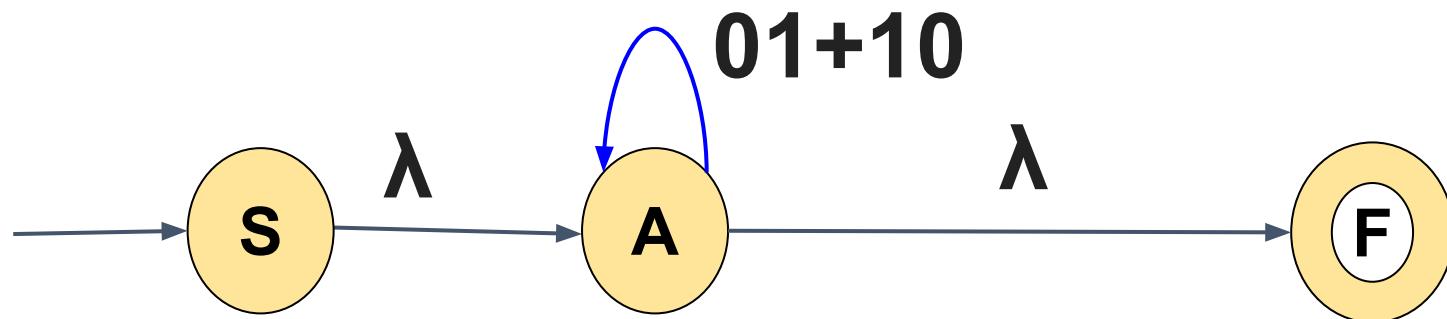
1. Eliminate D
2. Eliminate B

### Example 6 :



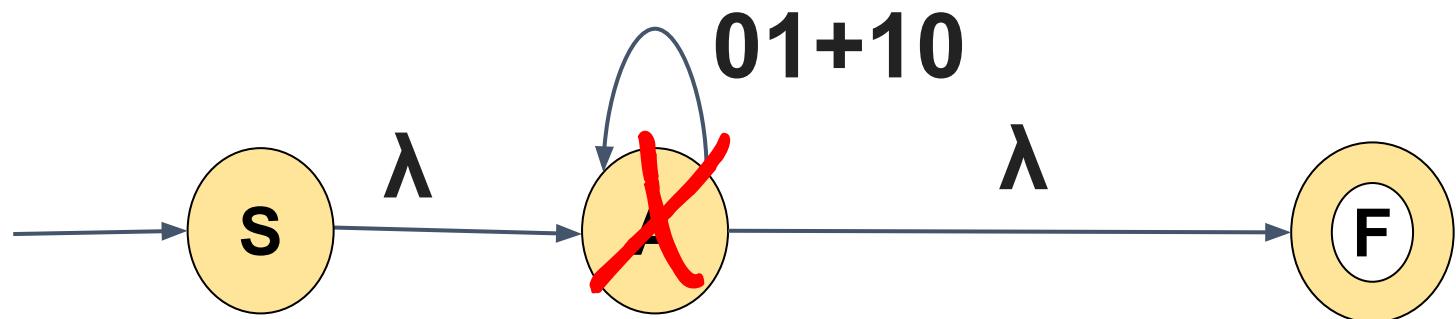
1. Eliminate D
2. Eliminate B
3. Eliminate C

### Example 6 :



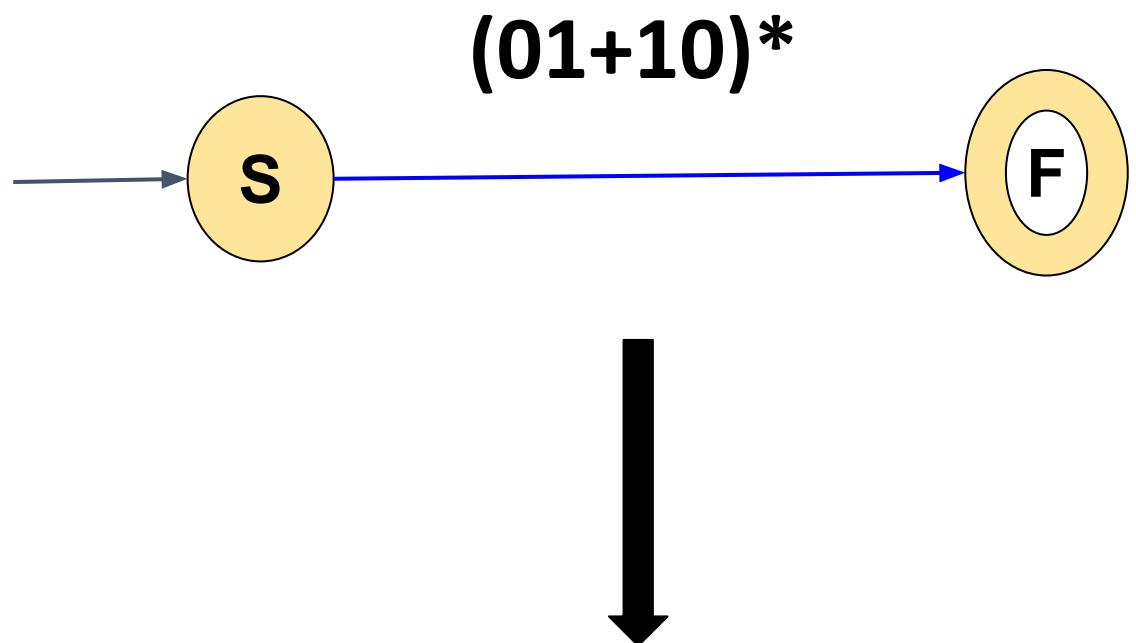
1. Eliminate D
2. Eliminate B
3. Eliminate C

### Example 6 :



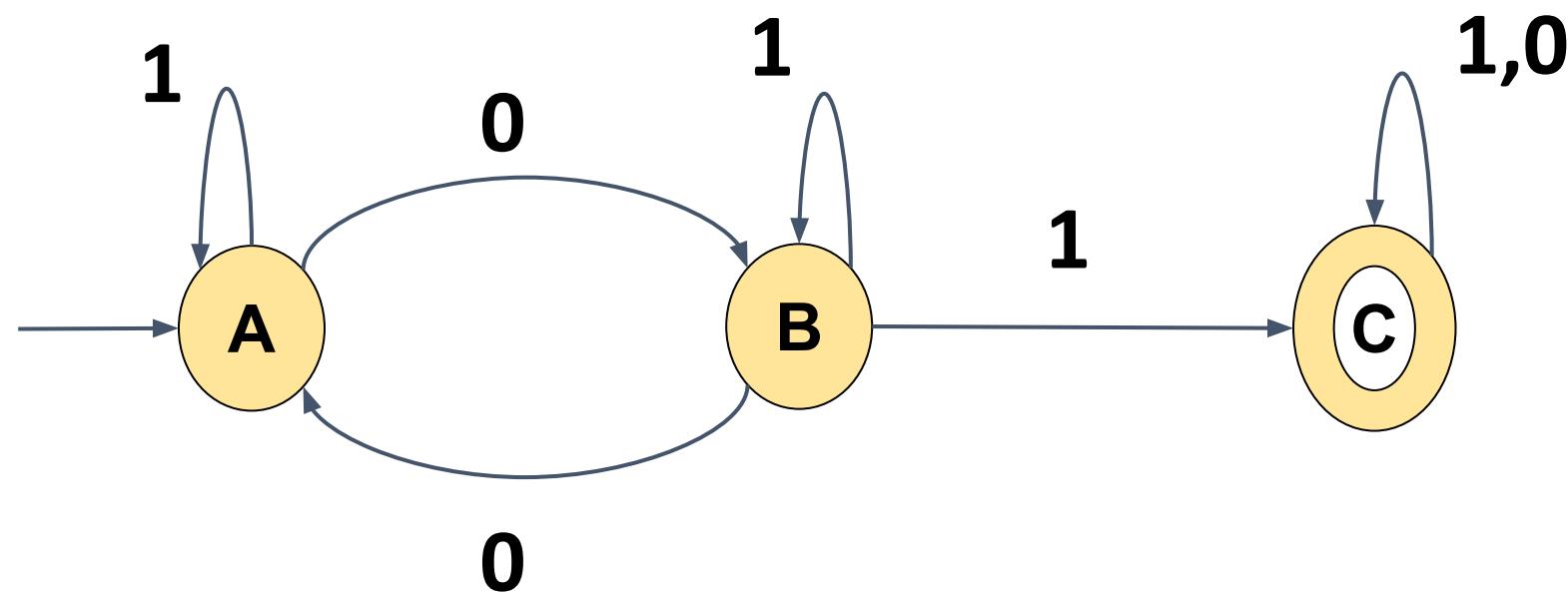
1. Eliminate D
2. Eliminate B
3. Eliminate C
4. Eliminate A

### Example 6 :

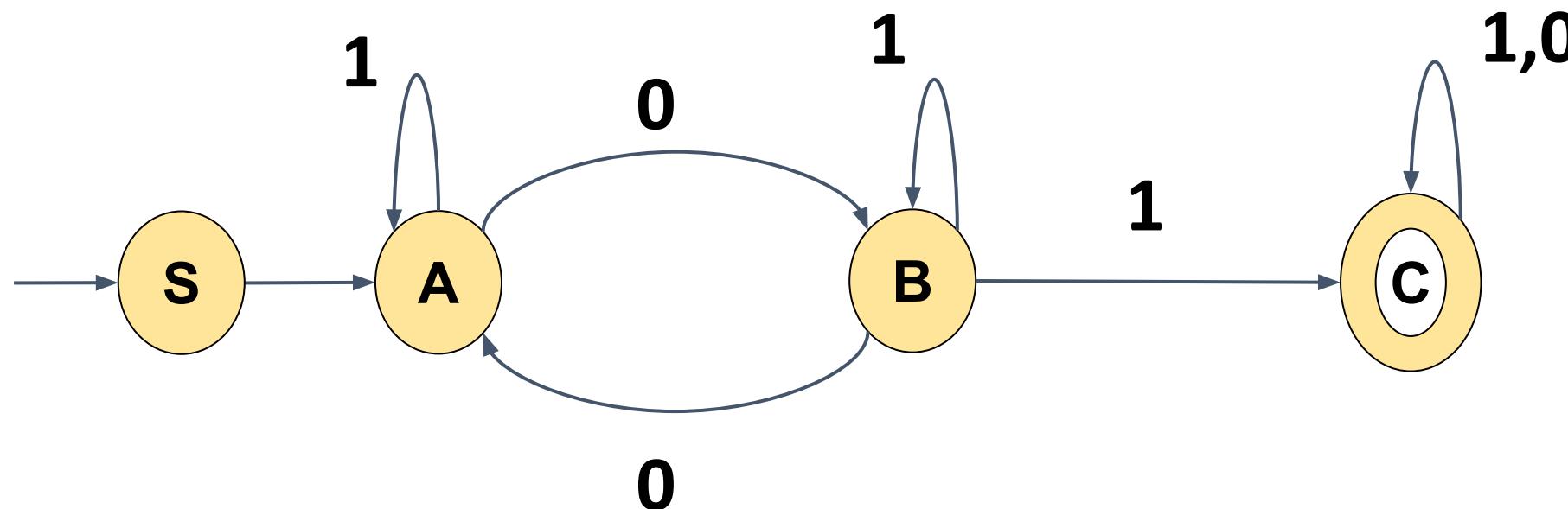


$$RE = (01+10)^*$$

### Example 7 :

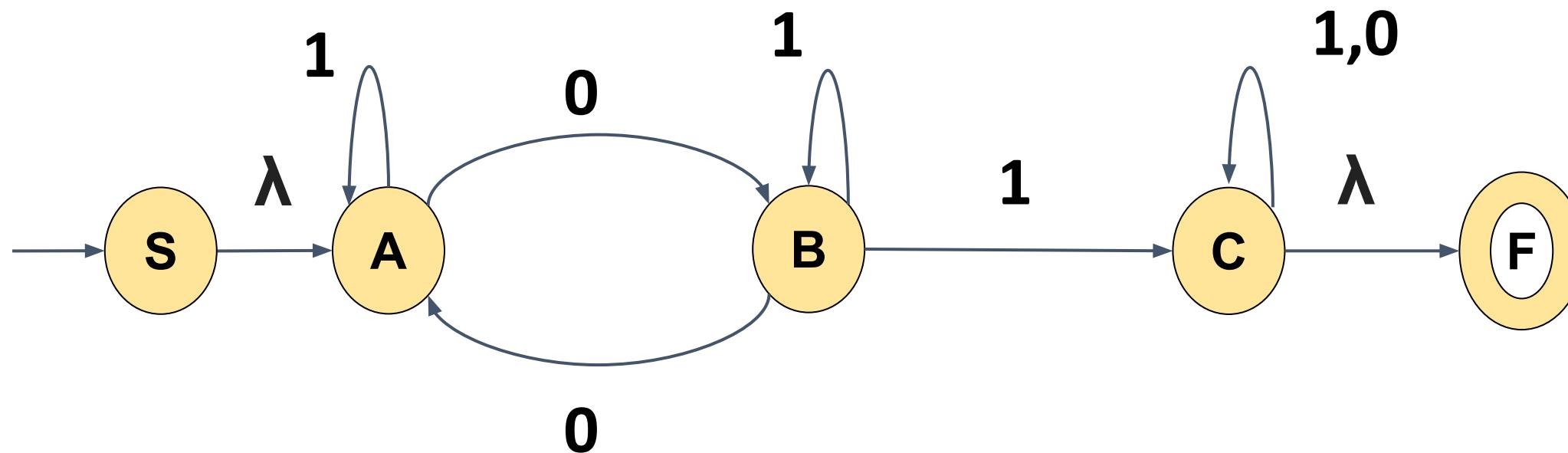


### Example 7 :



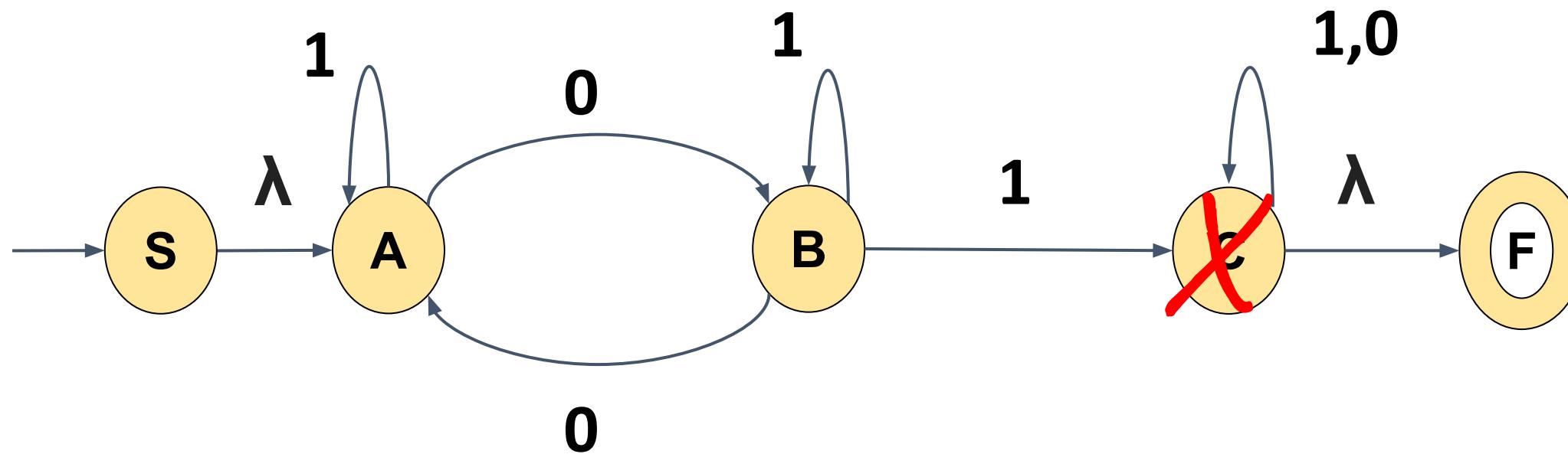
A new start state (S) is introduced as there is an incoming edge to the existing start state

### Example 7 :



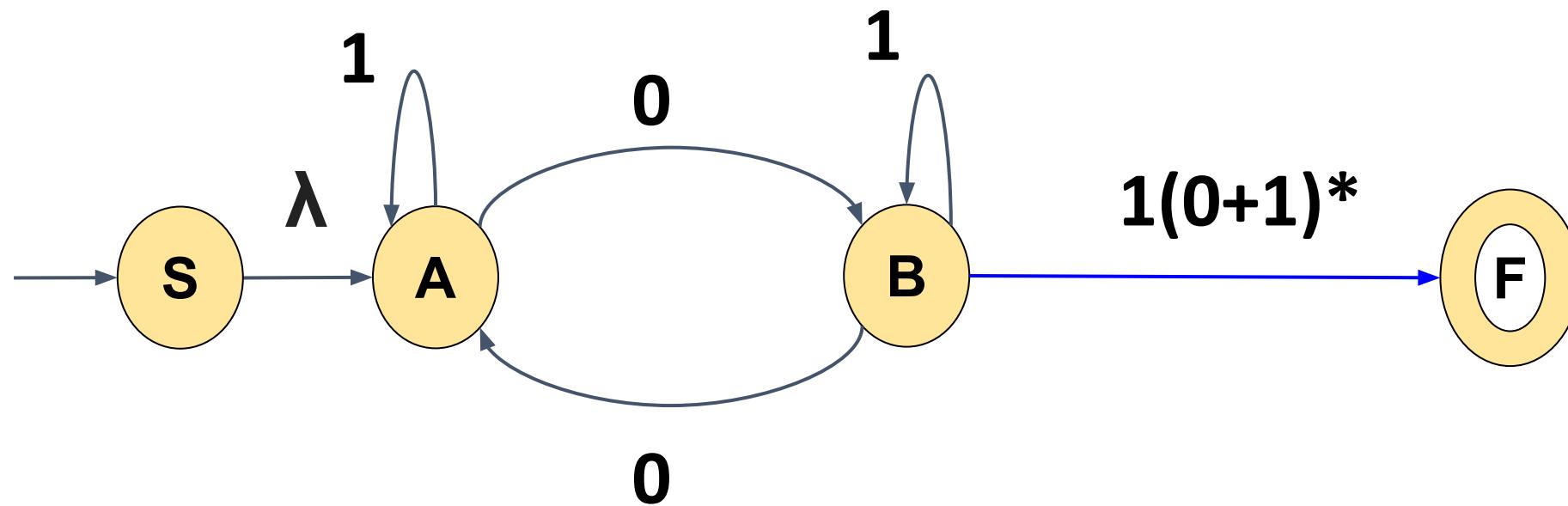
A new final state (F) is introduced as there is an outgoing edge from the existing final state

### Example 7 :



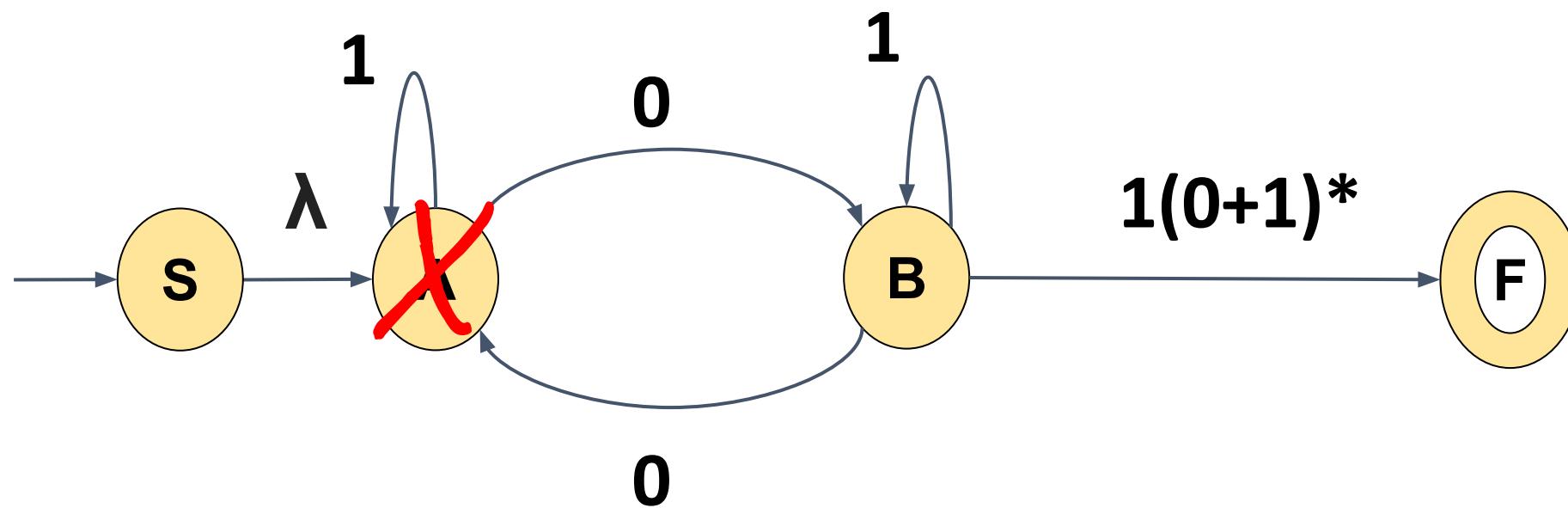
1. Eliminate C

### Example 7 :



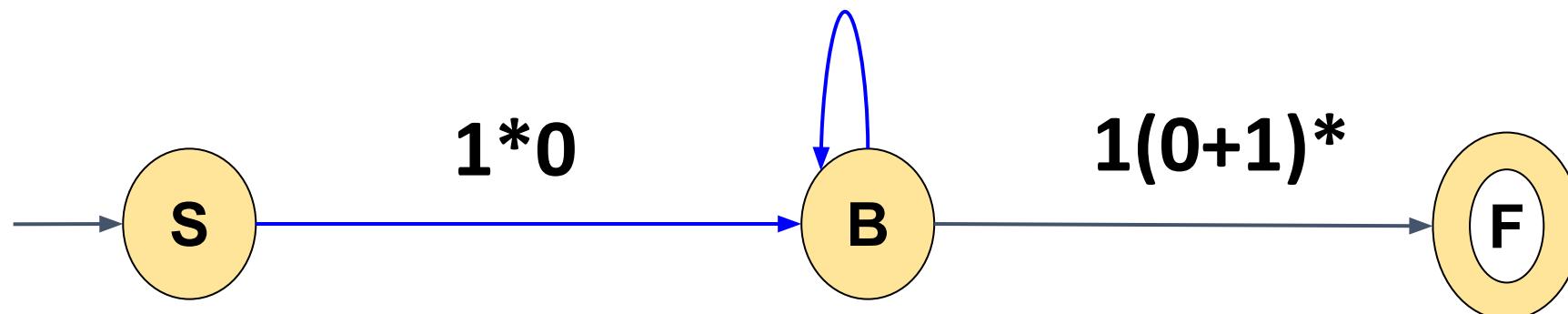
1. Eliminate C

### Example 7 :



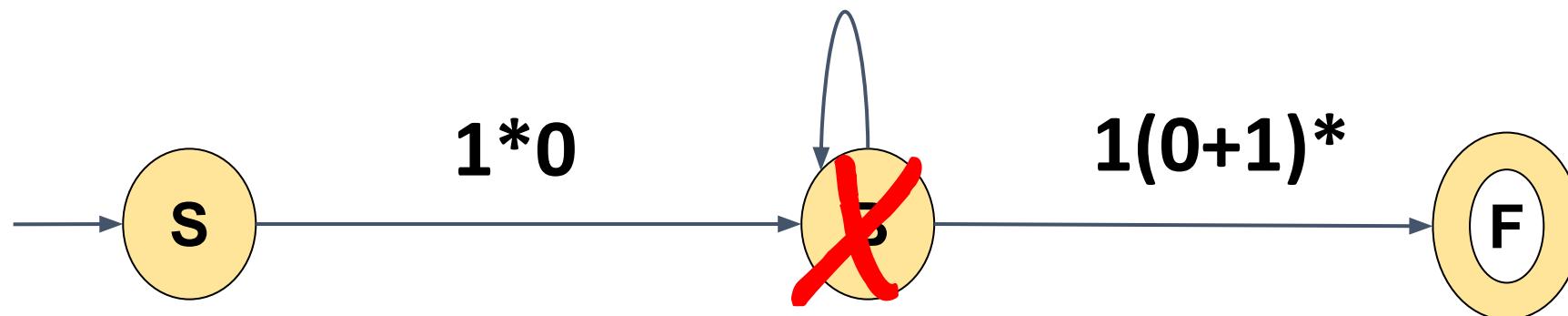
1. Eliminate C
2. Eliminate A

### Example 7 :

$$(1+01^*0)$$


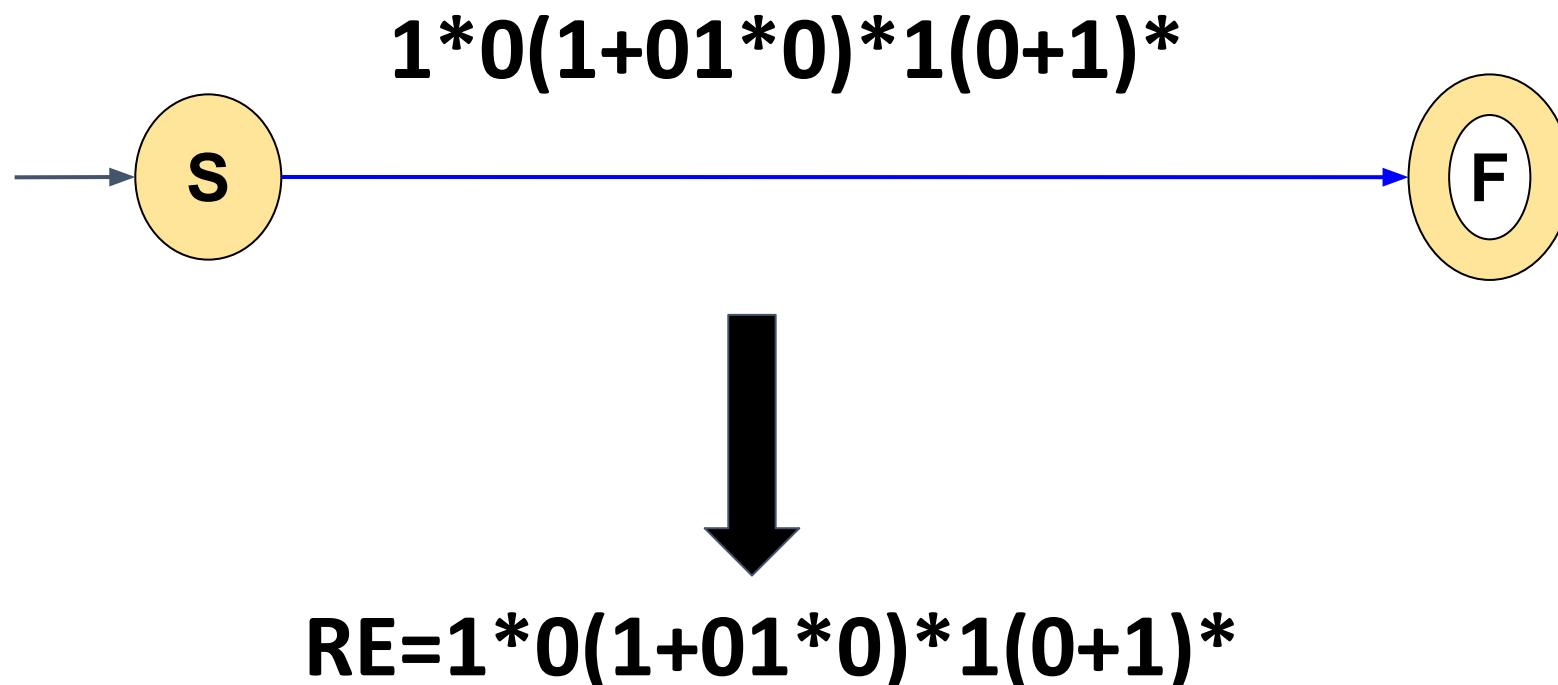
1. Eliminate C
2. Eliminate A

### Example 7 :

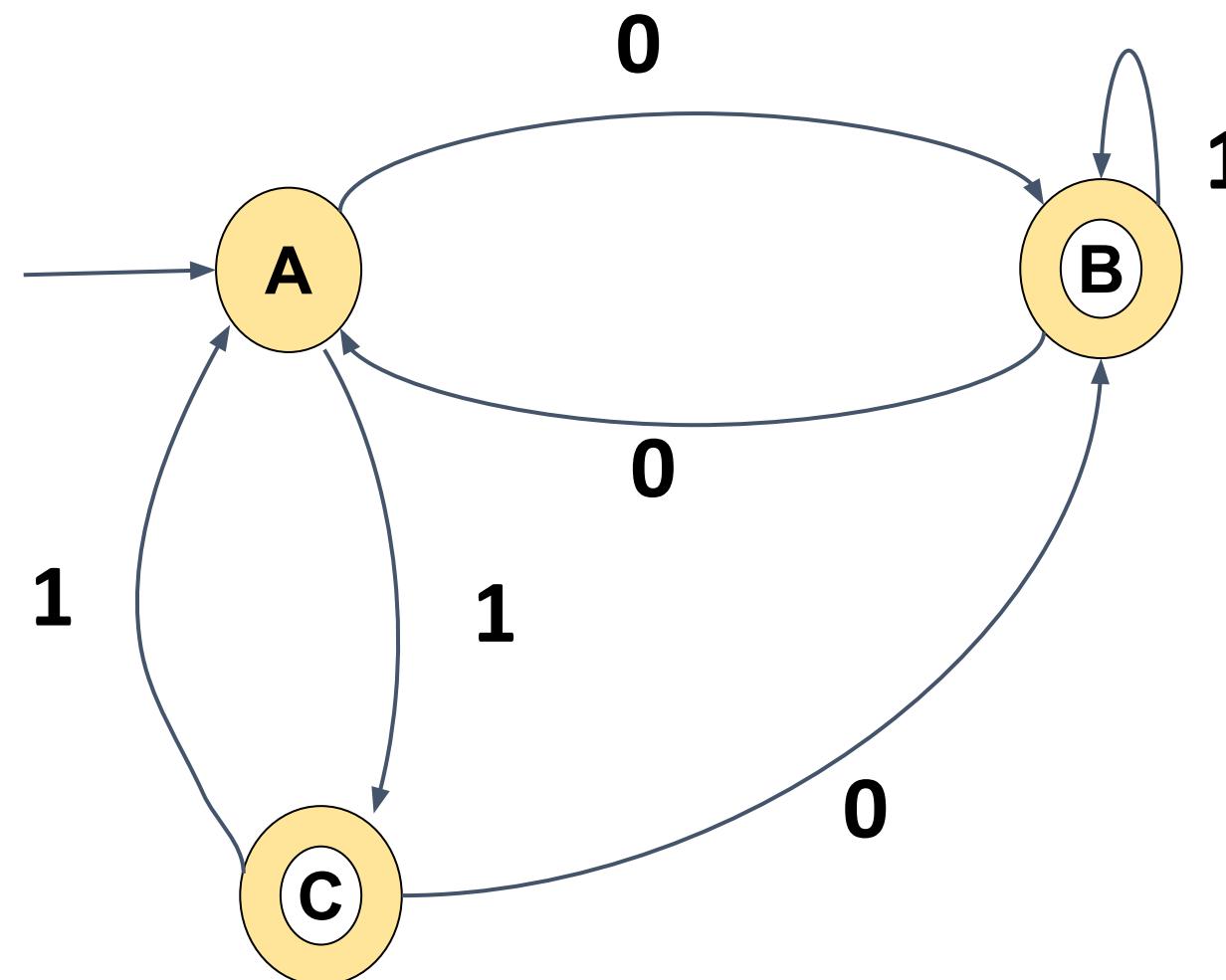
$$(1+01^*0)$$


1. Eliminate C
2. Eliminate A
3. Eliminate B

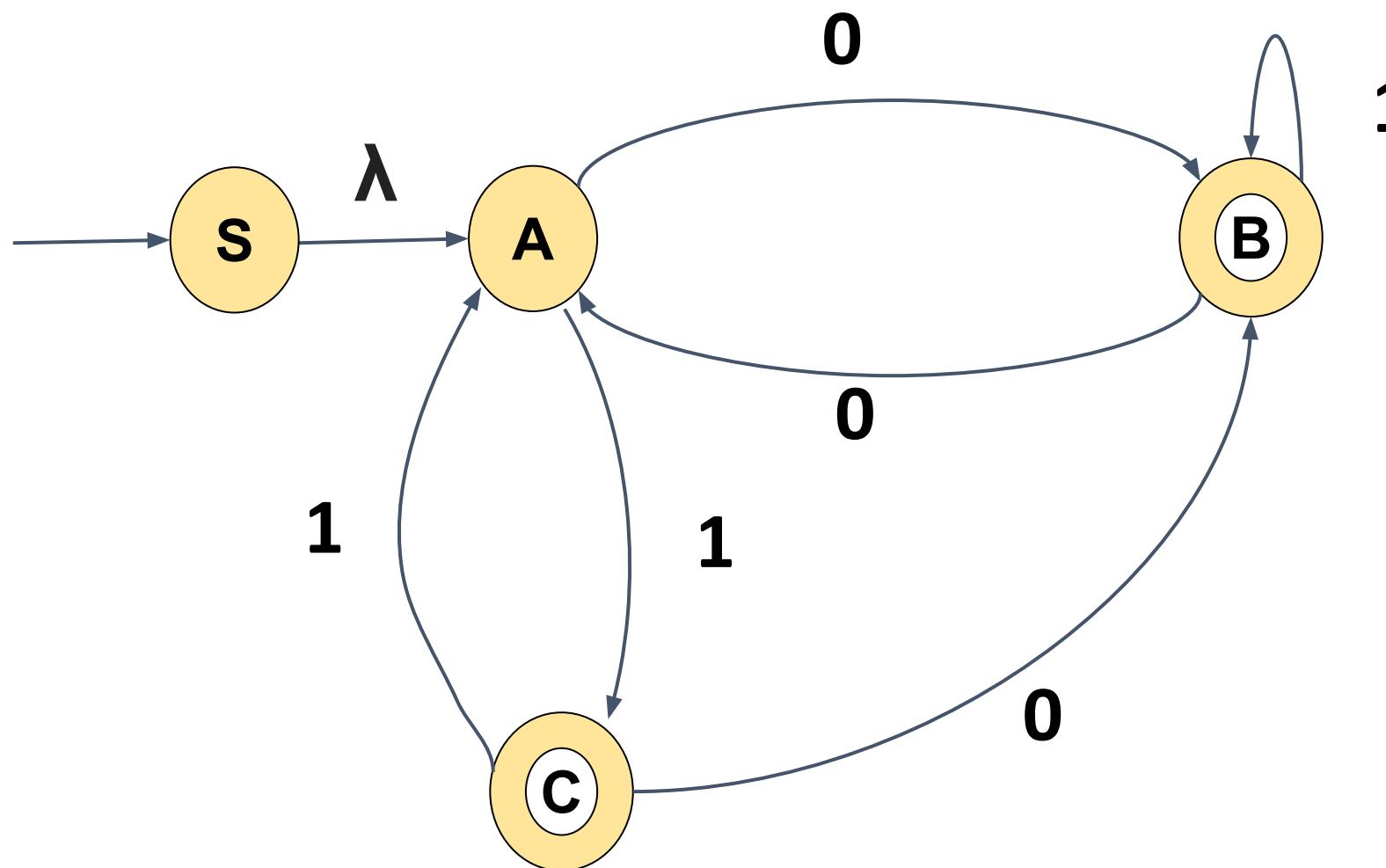
### Example 7 :



### Example 8 :

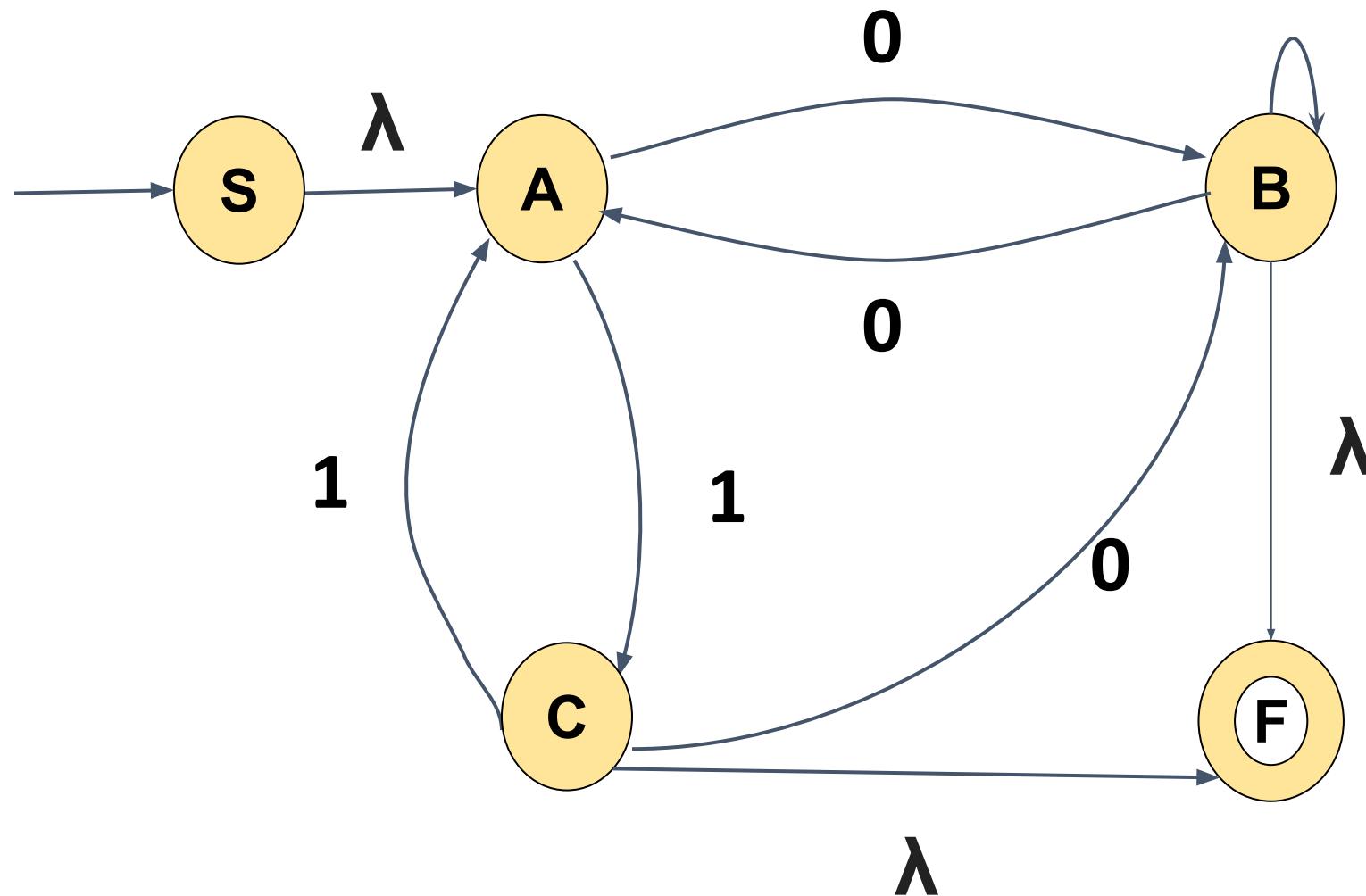


### Example 8 :



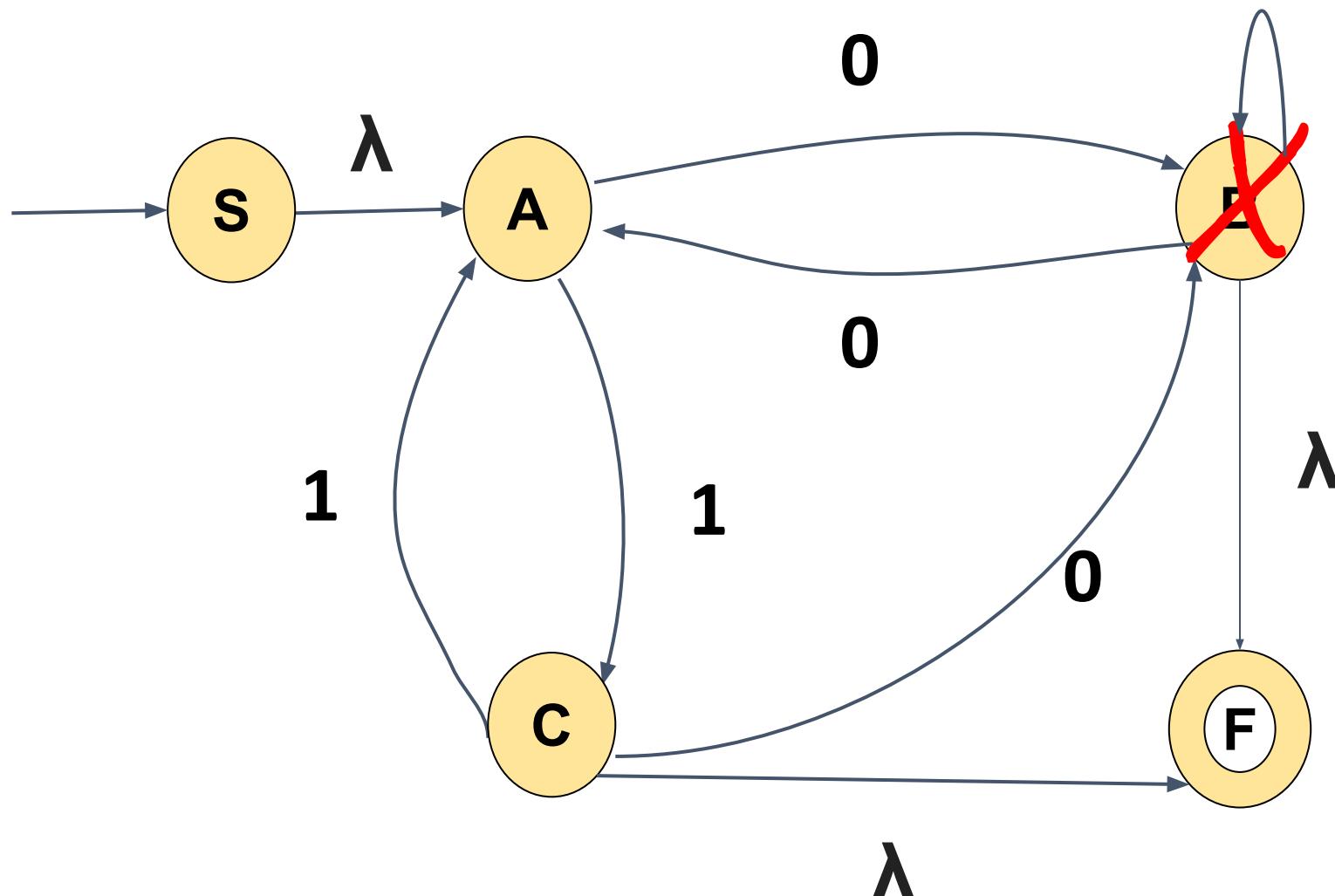
A new start state (S) is introduced as there is an incoming edge to the existing start state

### Example 8 :



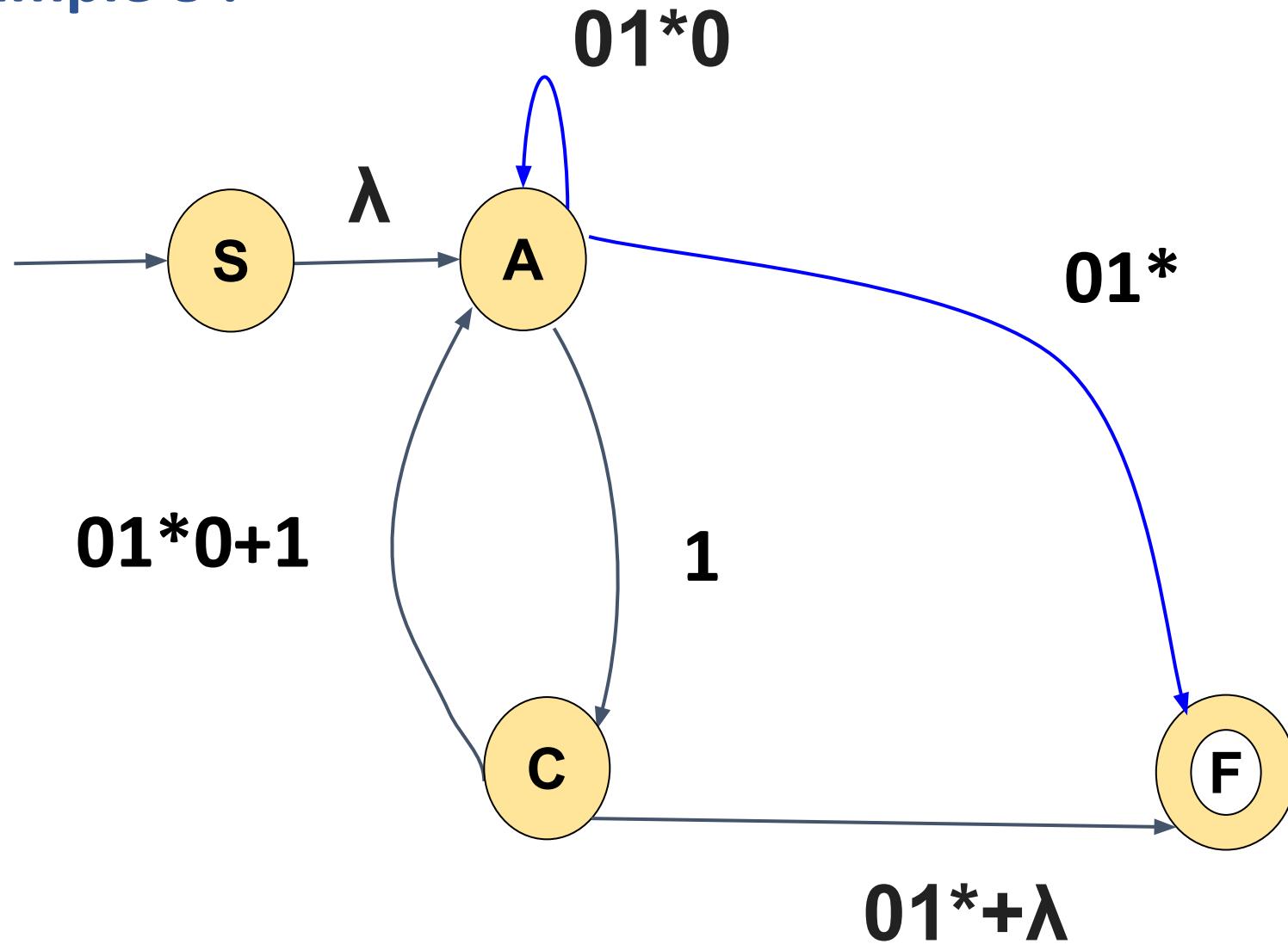
A new final state (F) is introduced as there is an outgoing edge from the existing final state

### Example 8 :



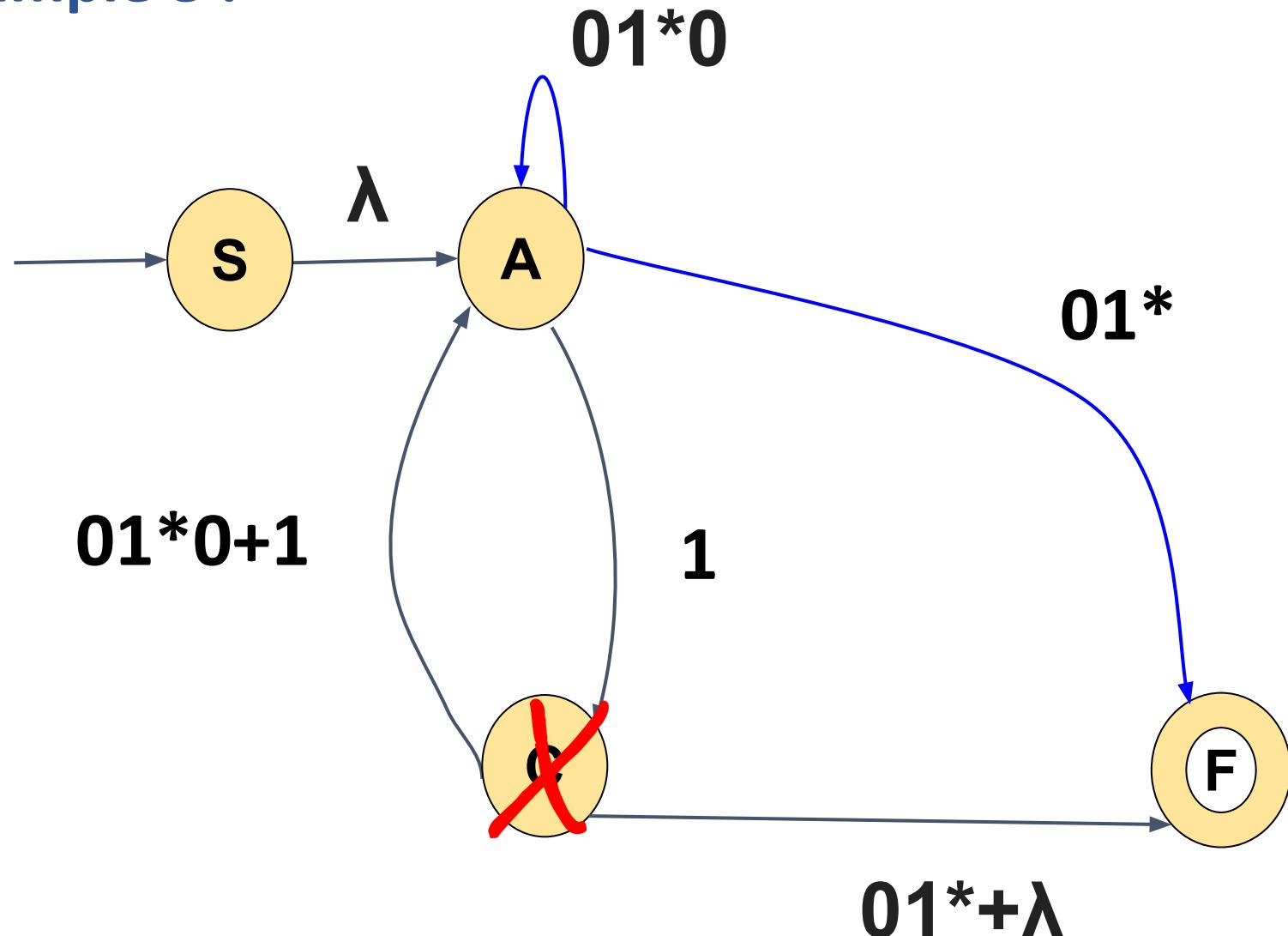
1. Eliminate B

### Example 8 :



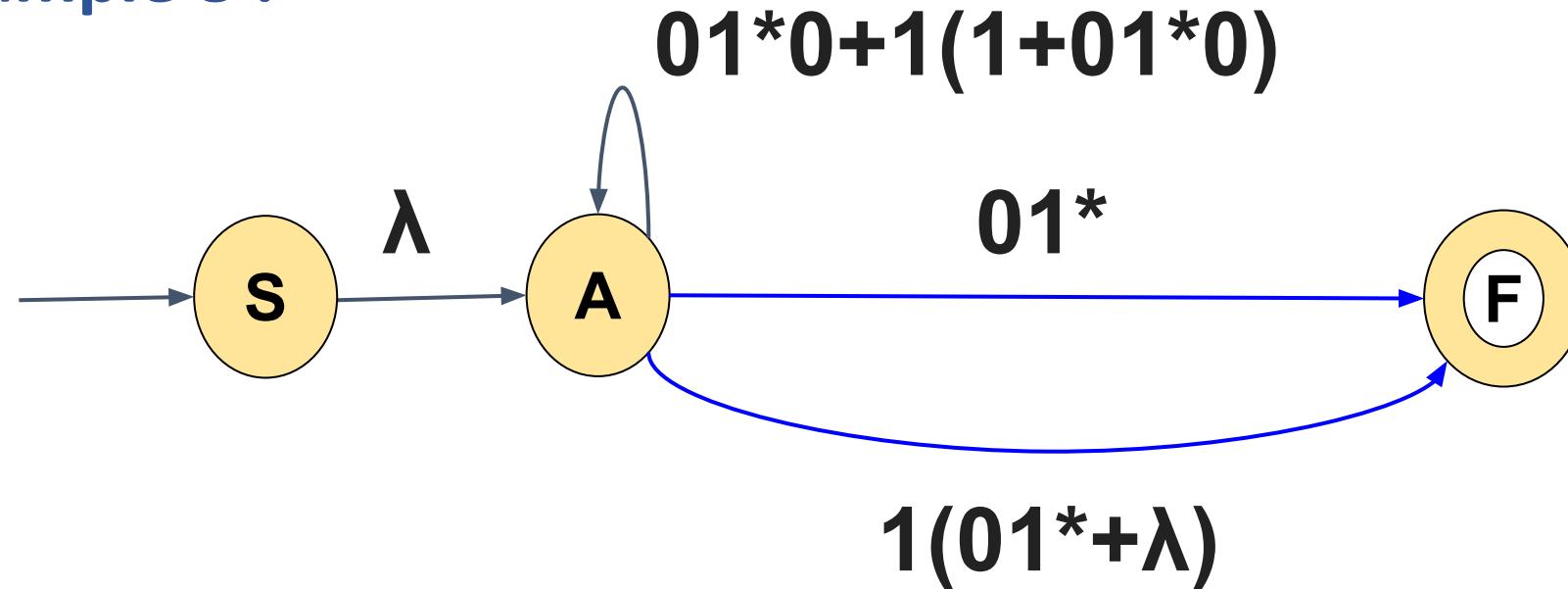
1. Eliminate B

### Example 8 :



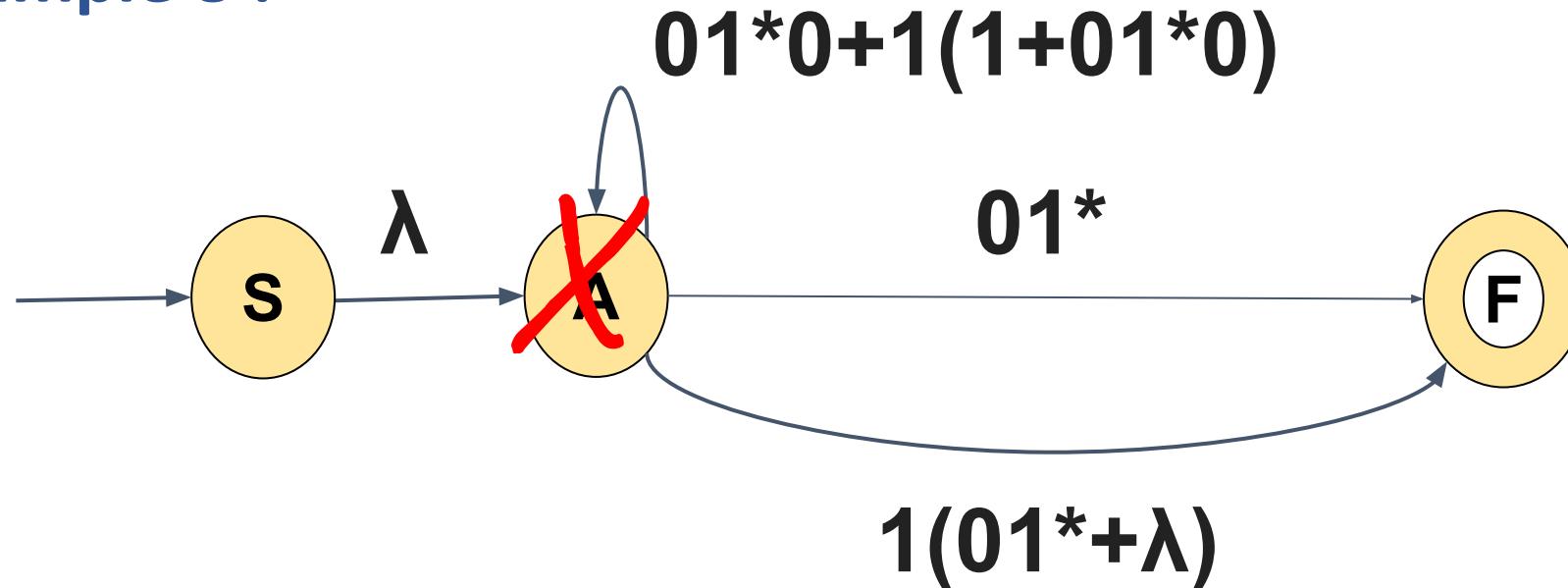
1. Eliminate B
2. Eliminate C

### Example 8 :



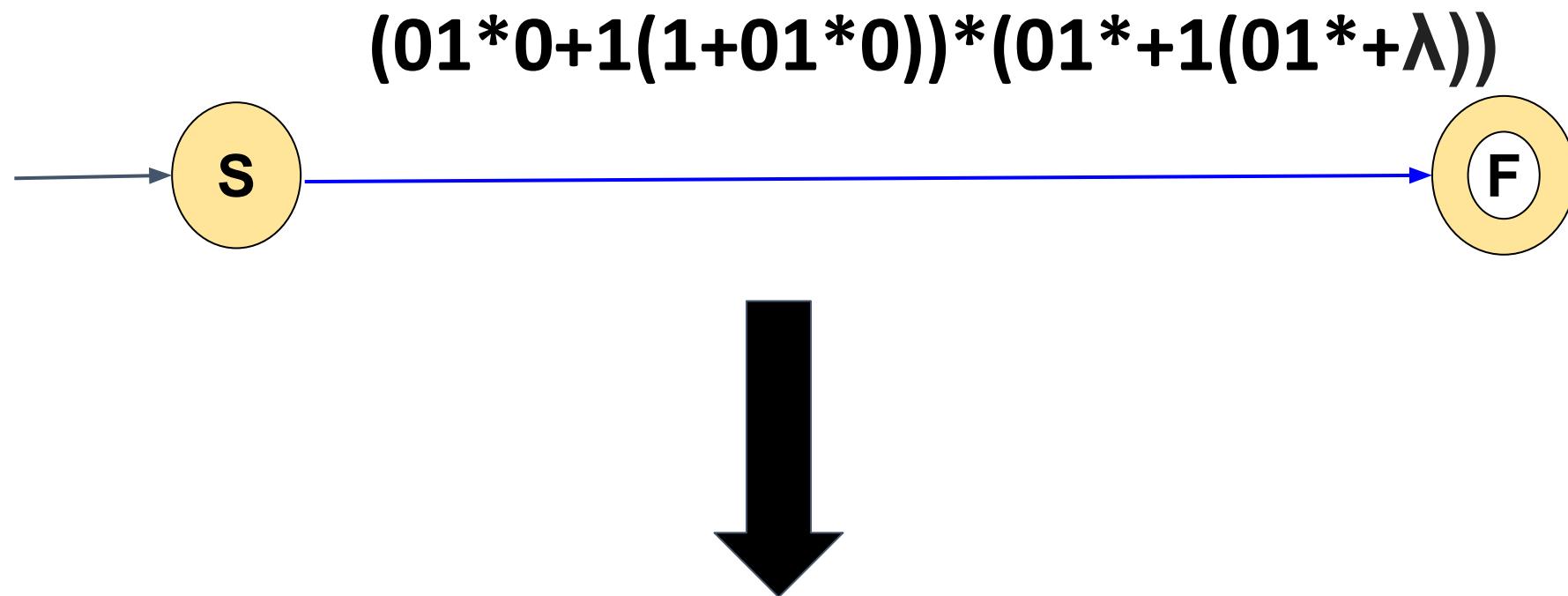
1. Eliminate B
2. Eliminate C

### Example 8 :



1. Eliminate B
2. Eliminate C
3. Eliminate A

### Example 8 :



$$RE = (01^*0 + 1(1 + 01^*0))^*(01^* + 1(01^* + \lambda))$$

# Automata Formal Languages and Logic

## Unit 2 - Finite Automata to Regular Expression

---

Example 8 :

$$RE = (01^*0 + 1(1 + 01^*0))^* (01^* + 1(01^* + \lambda))$$

$$= (01^*0 + 11 + 101^*0)^* (01^* + (101^* + 1))$$



### Example 8 :

$$RE = (01^*0 + 1(1 + 01^*0))^* (01^* + 1(01^* + \lambda))$$

$$= (01^*0 + 11 + 101^*0)^* (01^* + (101^* + 1))$$

$$= (101^*0 + 01^*0 + 11)^* (01^*(\lambda + 1) + 1)$$

### Example 8 :

$$RE = (01^*0 + 1(1 + 01^*0))^* (01^* + 1(01^* + \lambda))$$

$$= (01^*0 + 11 + 101^*0)^* (01^* + (101^* + 1))$$

$$= (101^*0 + 01^*0 + 11)^* (01^*(\lambda + 1) + 1)$$

$$= (01^*0(1 + \lambda) + 11)^* (01^*(\lambda + 1) + 1)$$

### Example 8 :

$$RE = (01^*0 + 1(1 + 01^*0))^* (01^* + 1(01^* + \lambda))$$

$$= (01^*0 + 11 + 101^*0)^* (01^* + (101^* + 1))$$

$$= (101^*0 + 01^*0 + 11)^* (01^*(\lambda + 1) + 1)$$

$$= (01^*0(1 + \lambda) + 11)^* (01^*(\lambda + 1) + 1)$$

$$= (01^*0(1 + \lambda))^* (11)^* (01^*(\lambda + 1) + 1)$$

### Example 8 :

$$RE = (01^*0 + 1(1 + 01^*0))^* (01^* + 1(01^* + \lambda))$$

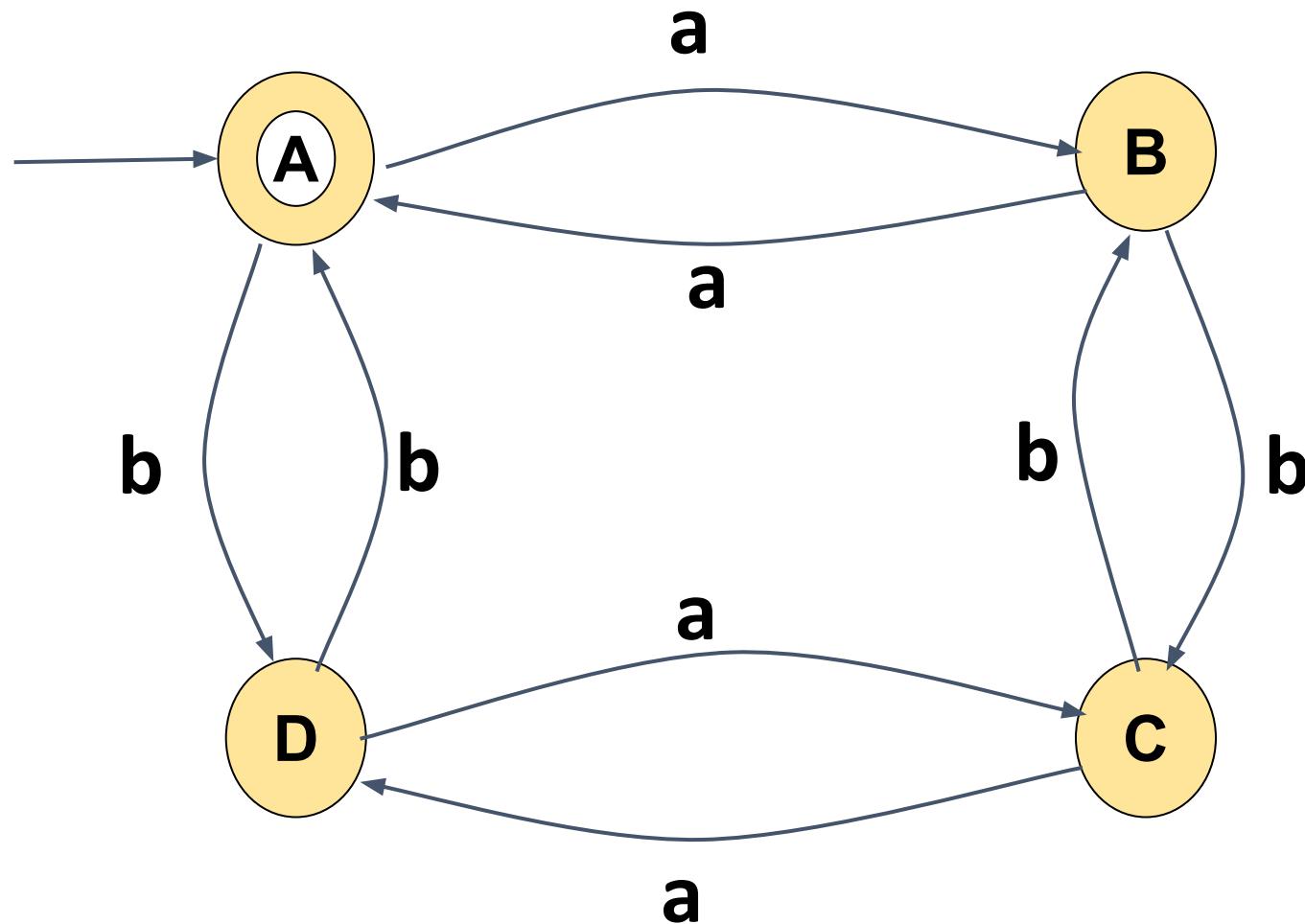
$$= (01^*0 + 11 + 101^*0)^* (01^* + (101^* + 1))$$

$$= (101^*0 + 01^*0 + 11)^* (01^*(\lambda + 1) + 1)$$

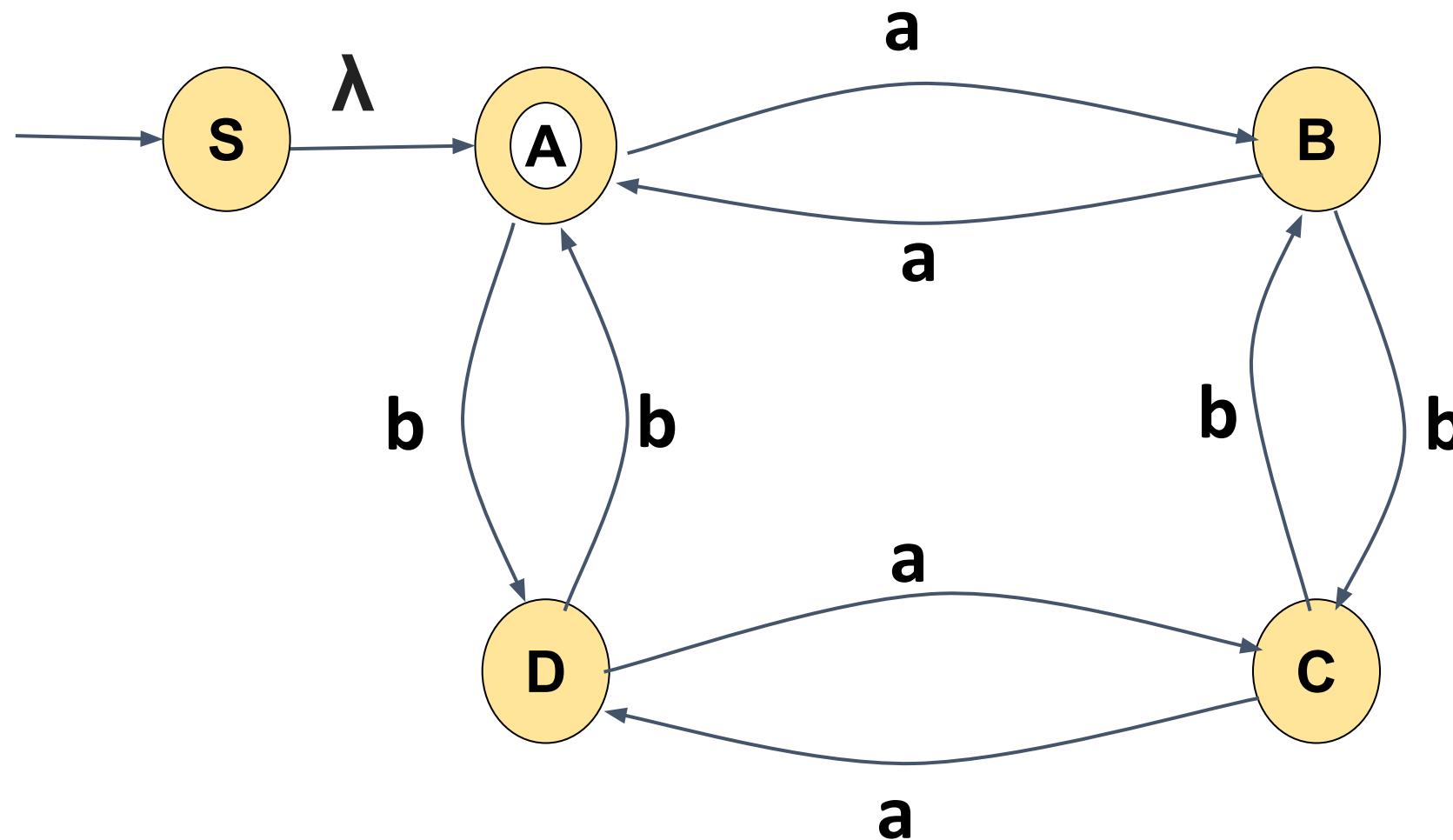
$$= (01^*0(1 + \lambda) + 11)^* (01^*(\lambda + 1) + 1)$$

$$RE = (01^*0(1 + \lambda))^* (11)^* (01^*(\lambda + 1) + 1)$$

### Example 9 :order of elimination( B,D,C,A)

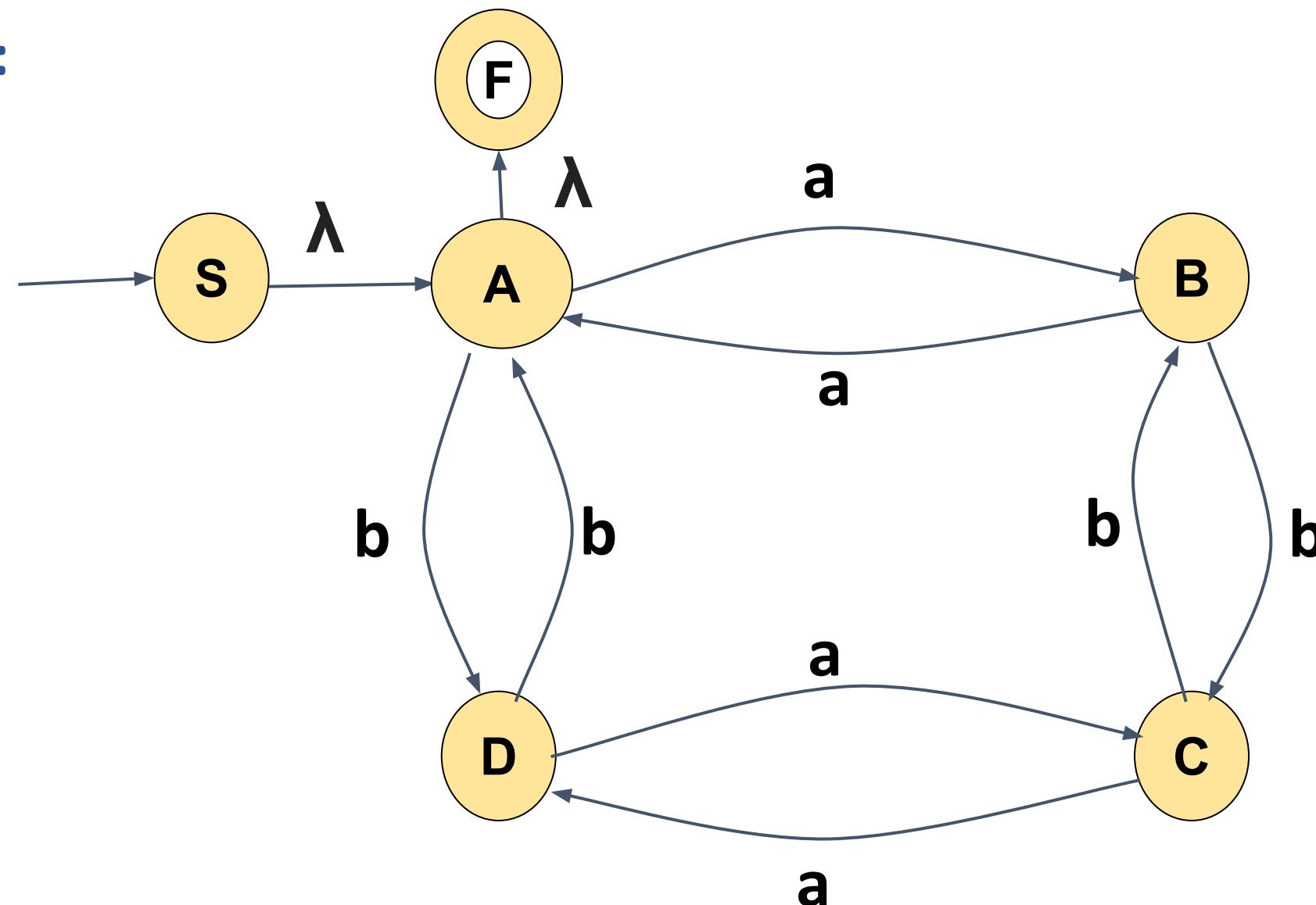


### Example 9 :



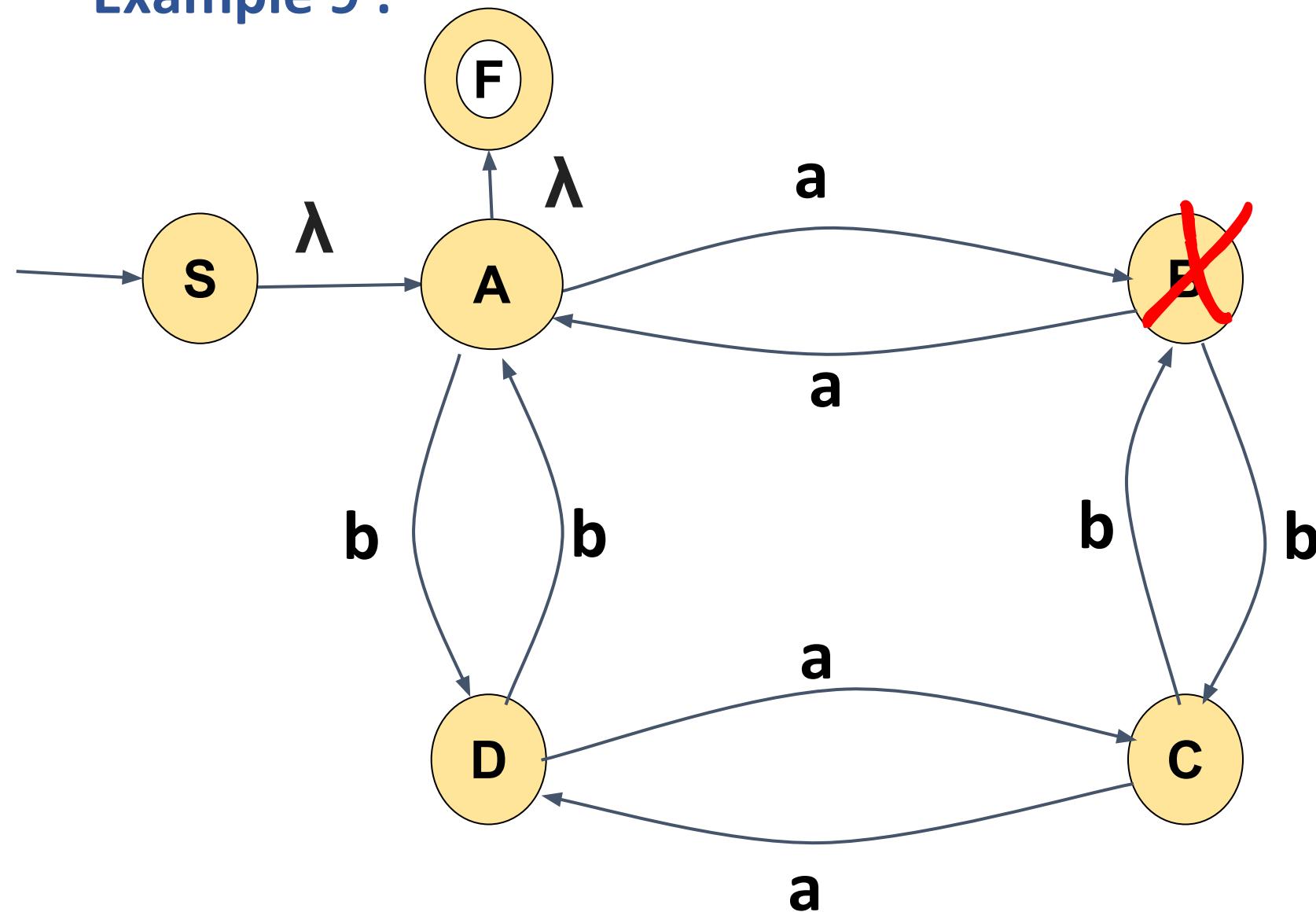
A new start state (S) is introduced as there is an incoming edge to the existing start state

### Example 9 :



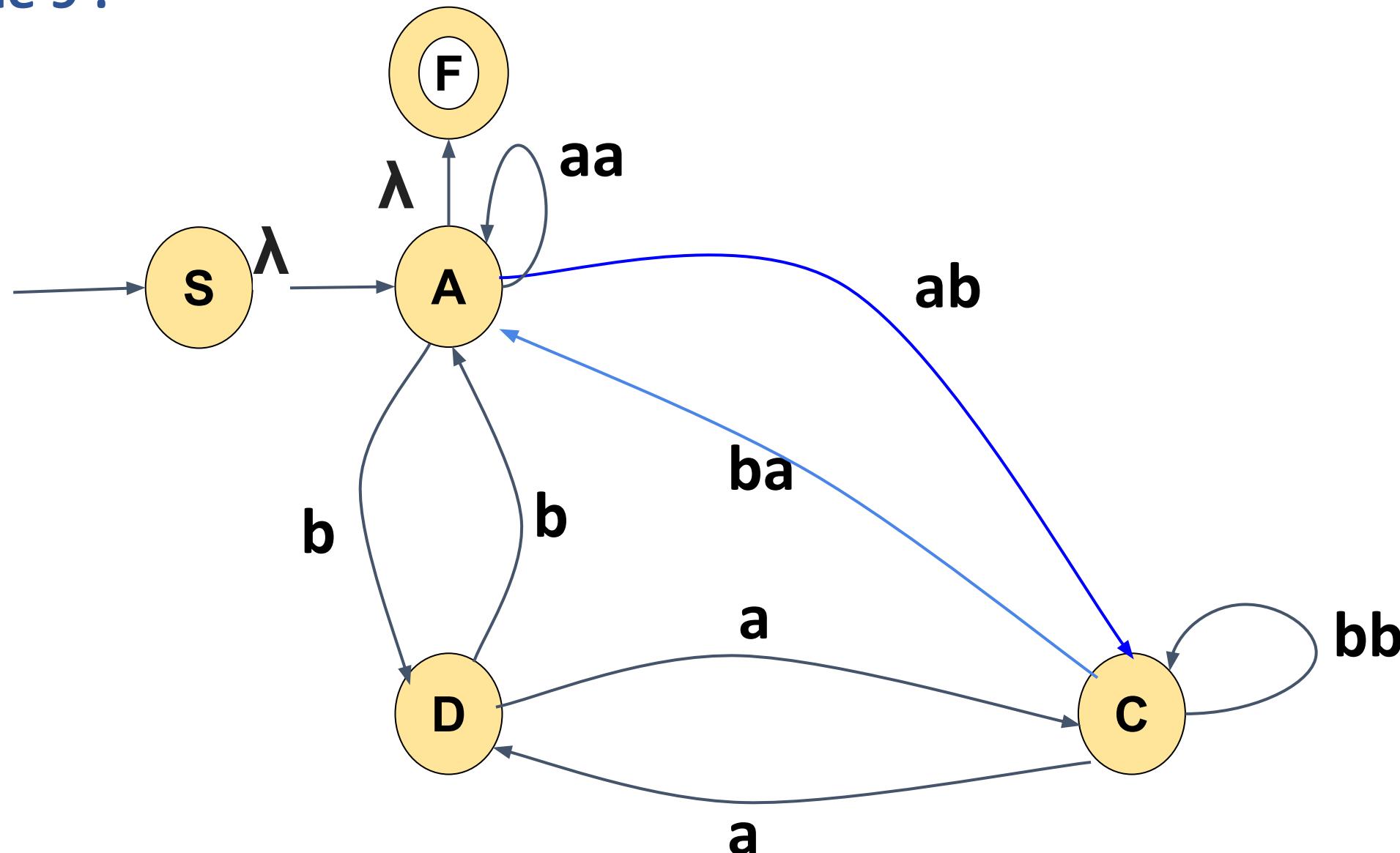
A new final state (F) is introduced as there is an outgoing edge from the existing final state

### Example 9 :



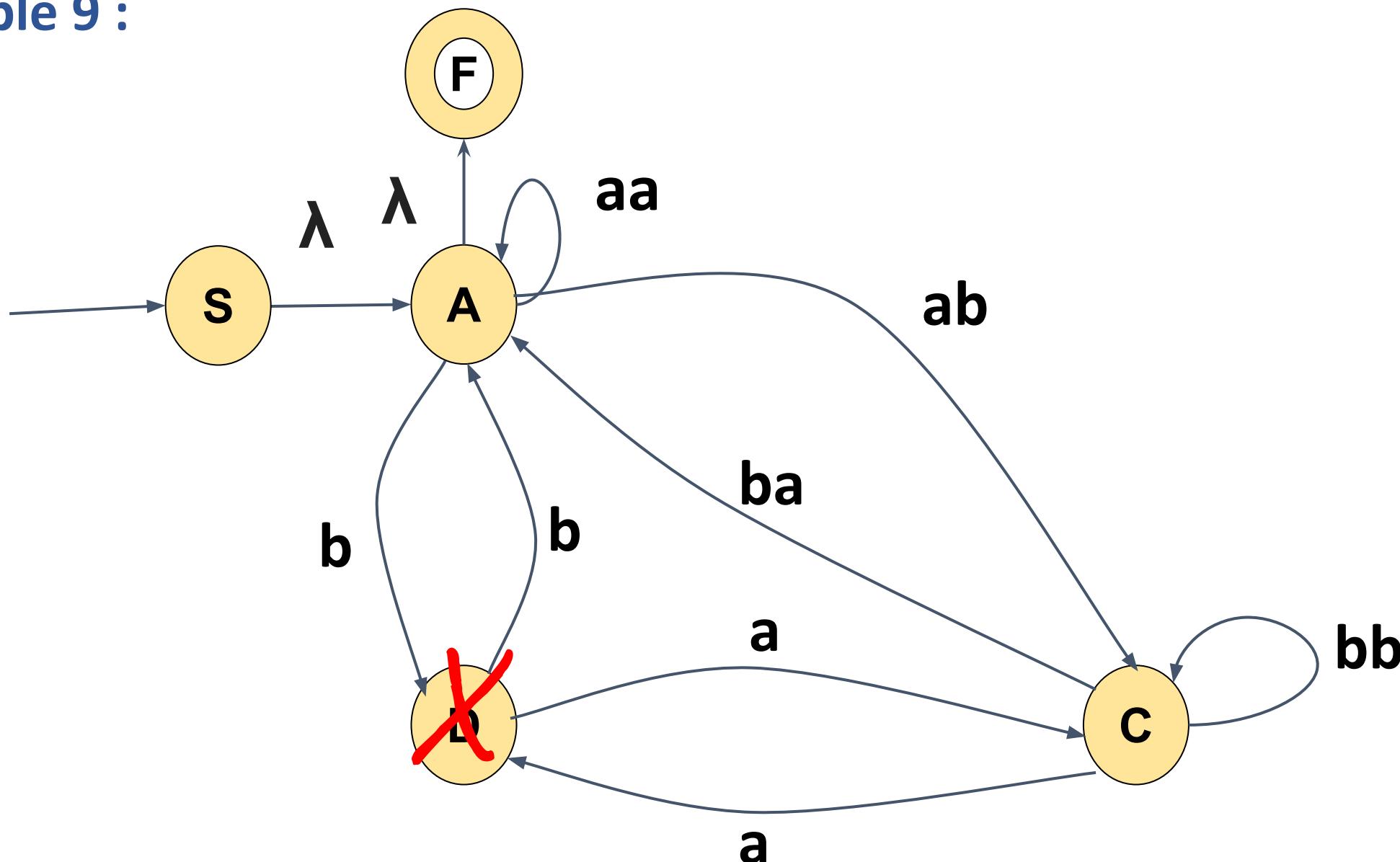
1. Eliminate B

### Example 9 :



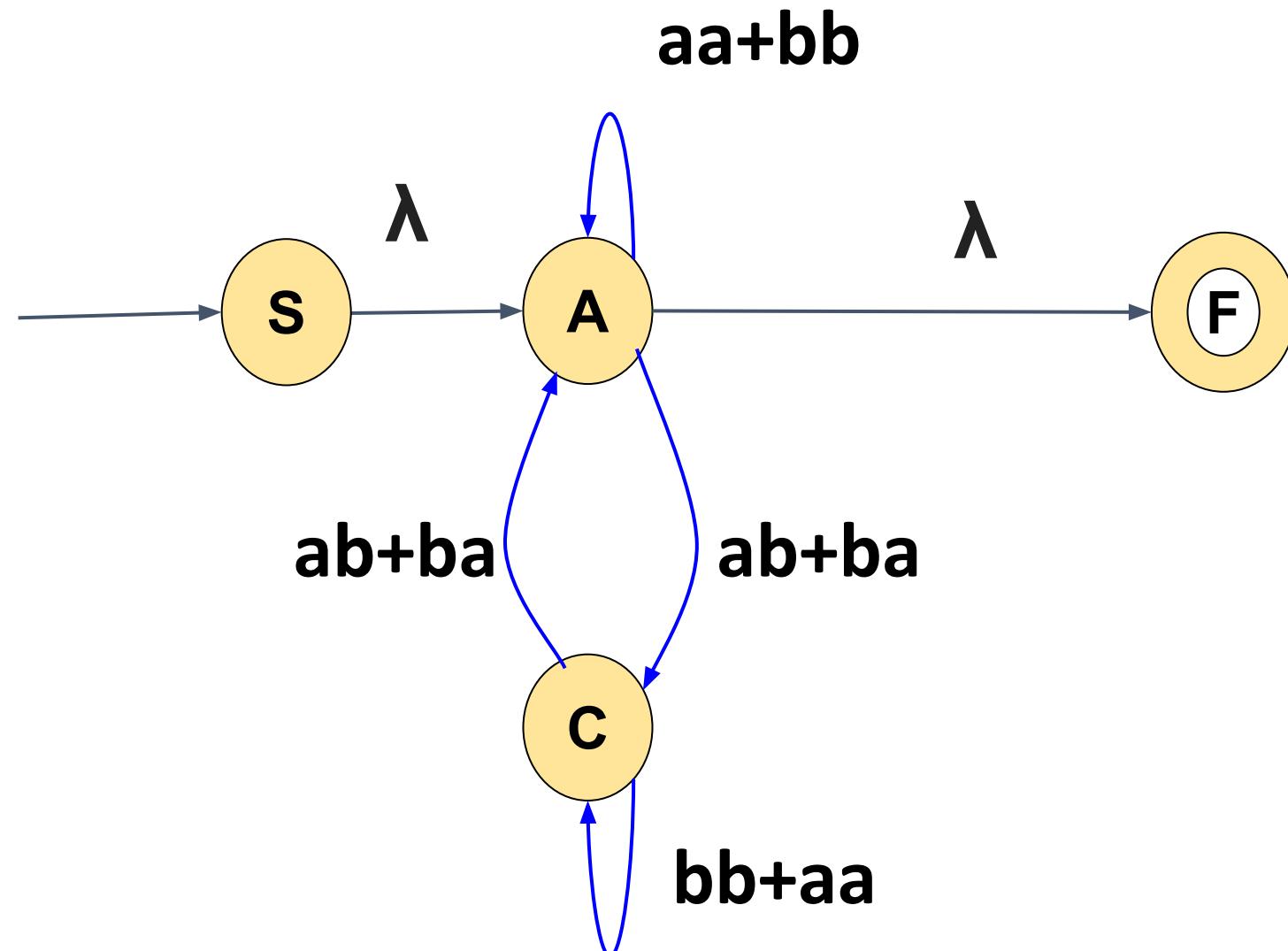
1. Eliminate B

### Example 9 :



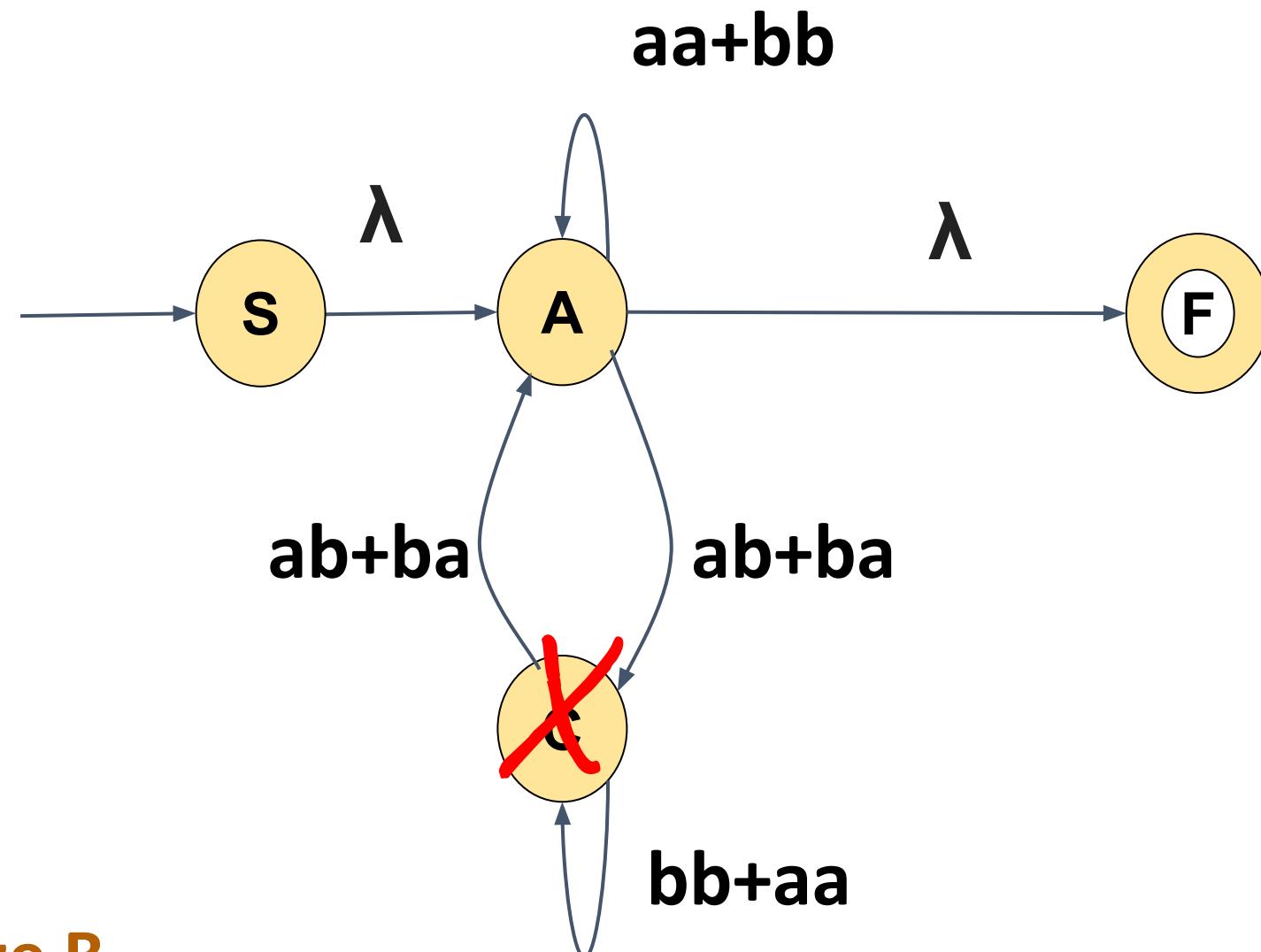
1. Eliminate B
2. Eliminate D

### Example 9 :



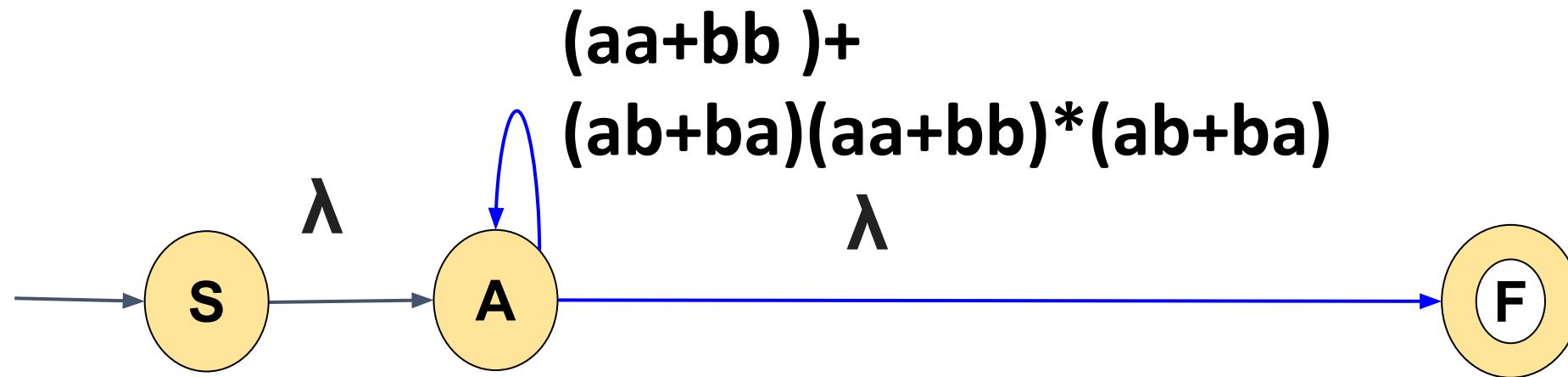
1. Eliminate B
2. Eliminate D

### Example 9 :



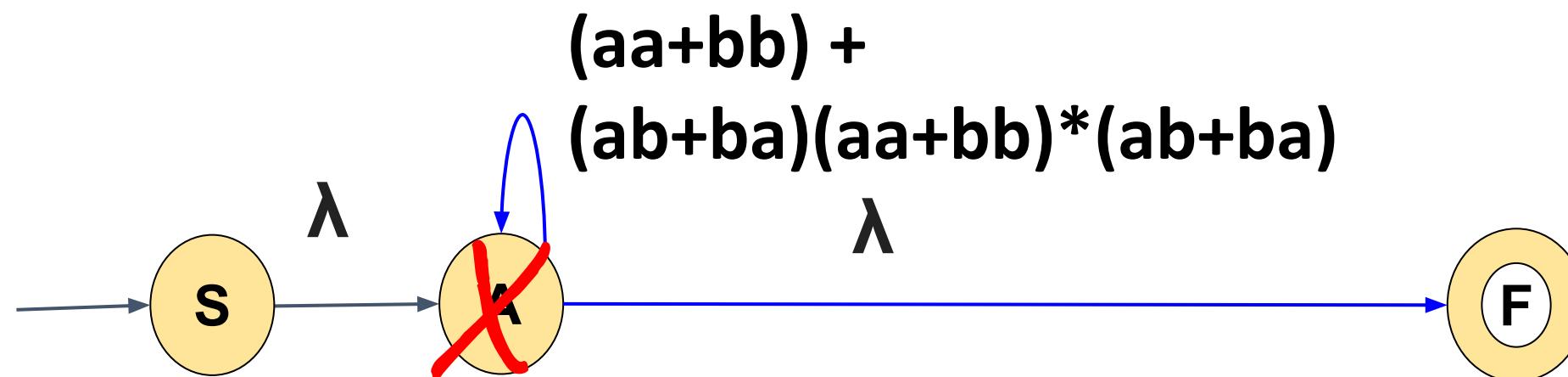
1. Eliminate B
2. Eliminate D
3. Eliminate C

### Example 9 :



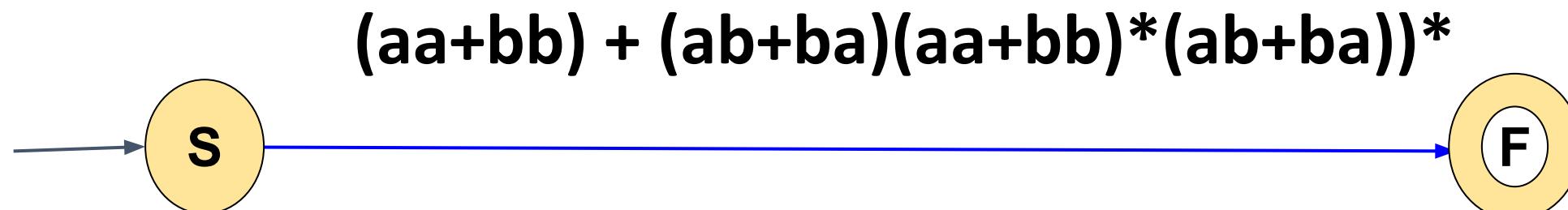
1. Eliminate B
2. Eliminate D
3. Eliminate C

Example 9 :



1. Eliminate B
2. Eliminate D
3. Eliminate C
4. Eliminate A

Example 9 :

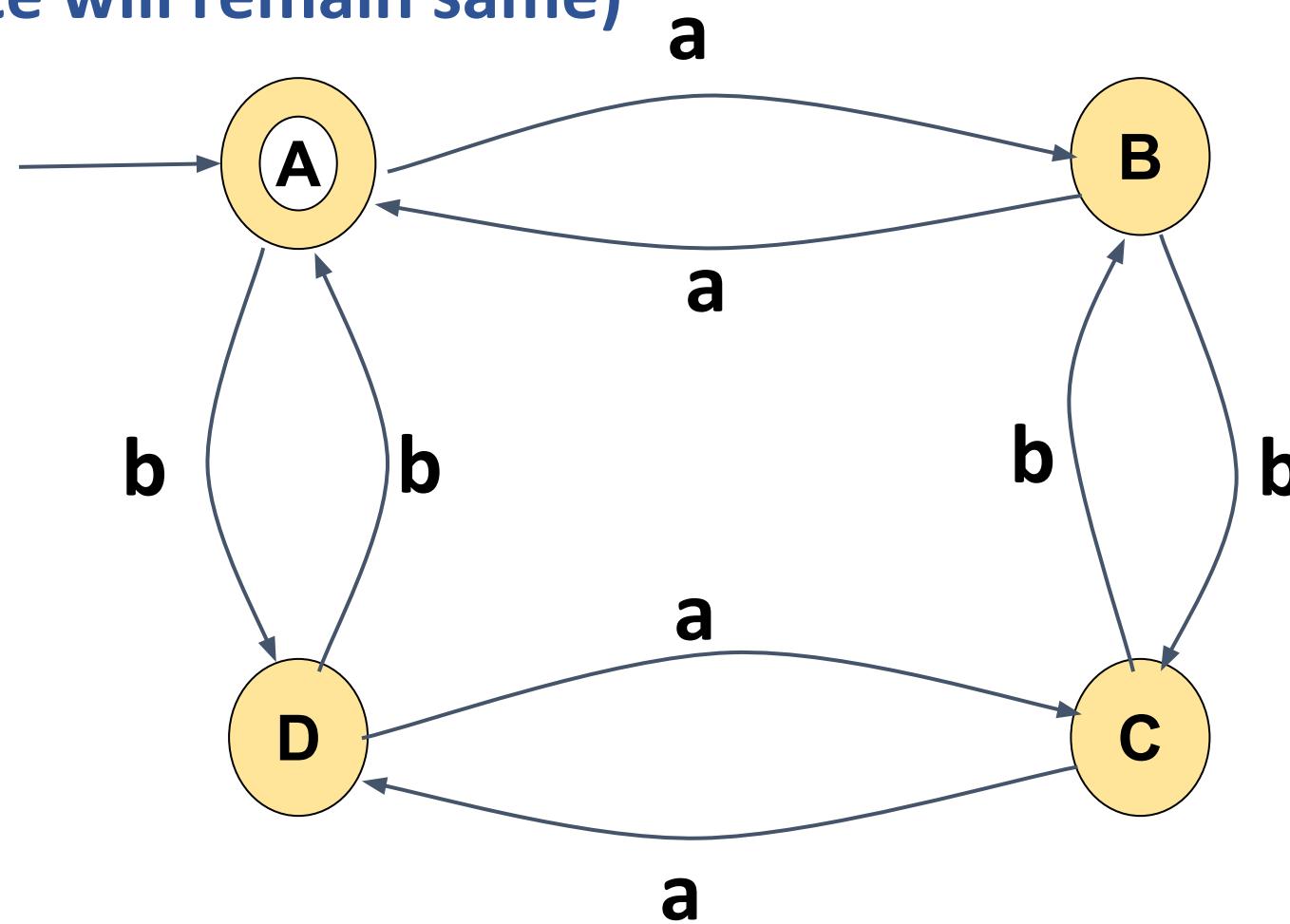


$RE = ((aa+bb) + (ab+ba)(aa+bb)^*(ab+ba))^*$

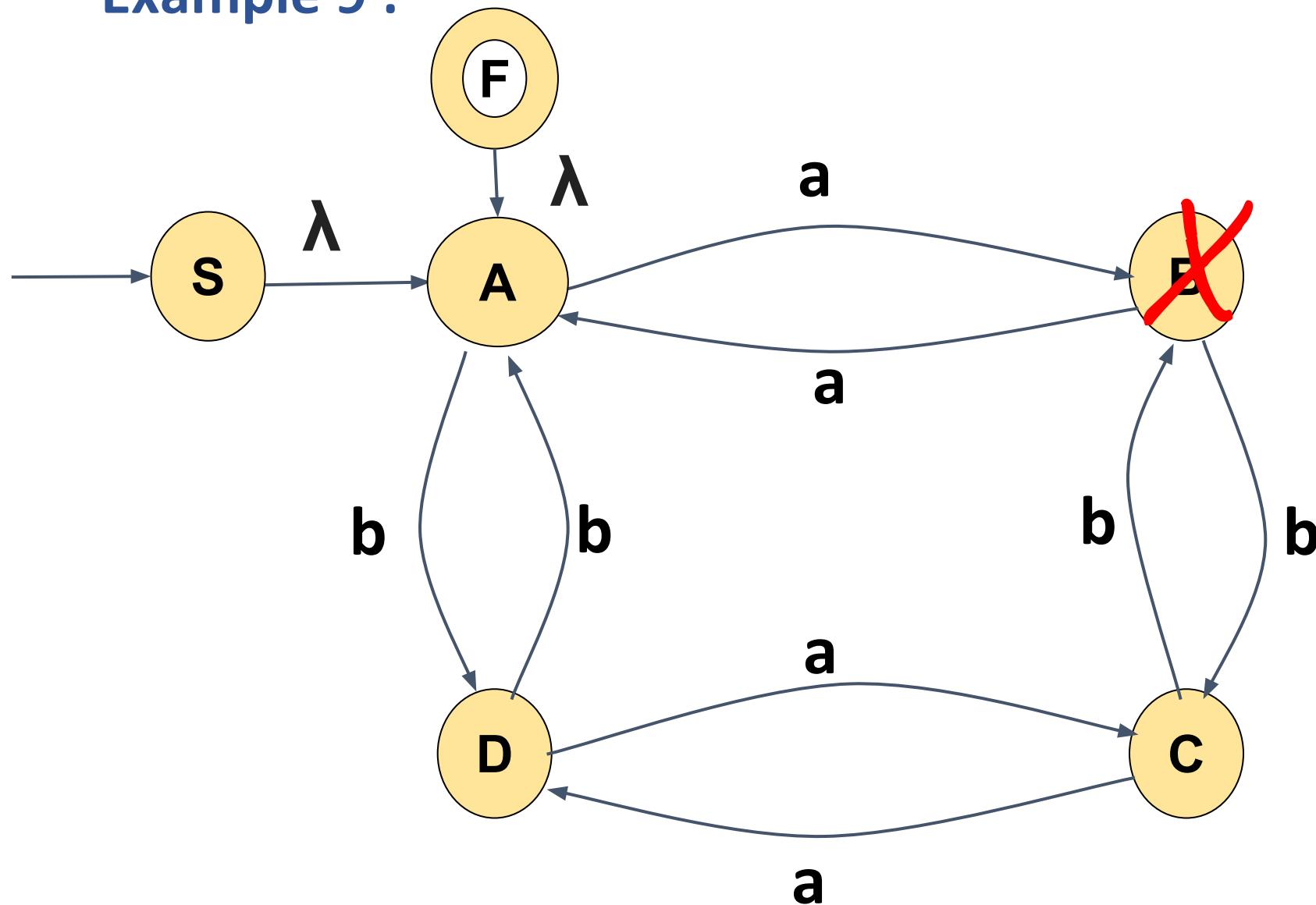
### Example 9 :

Consider the same example: Order of elimination: B,C, D ,A (Adding new start state and

new final state will remain same)

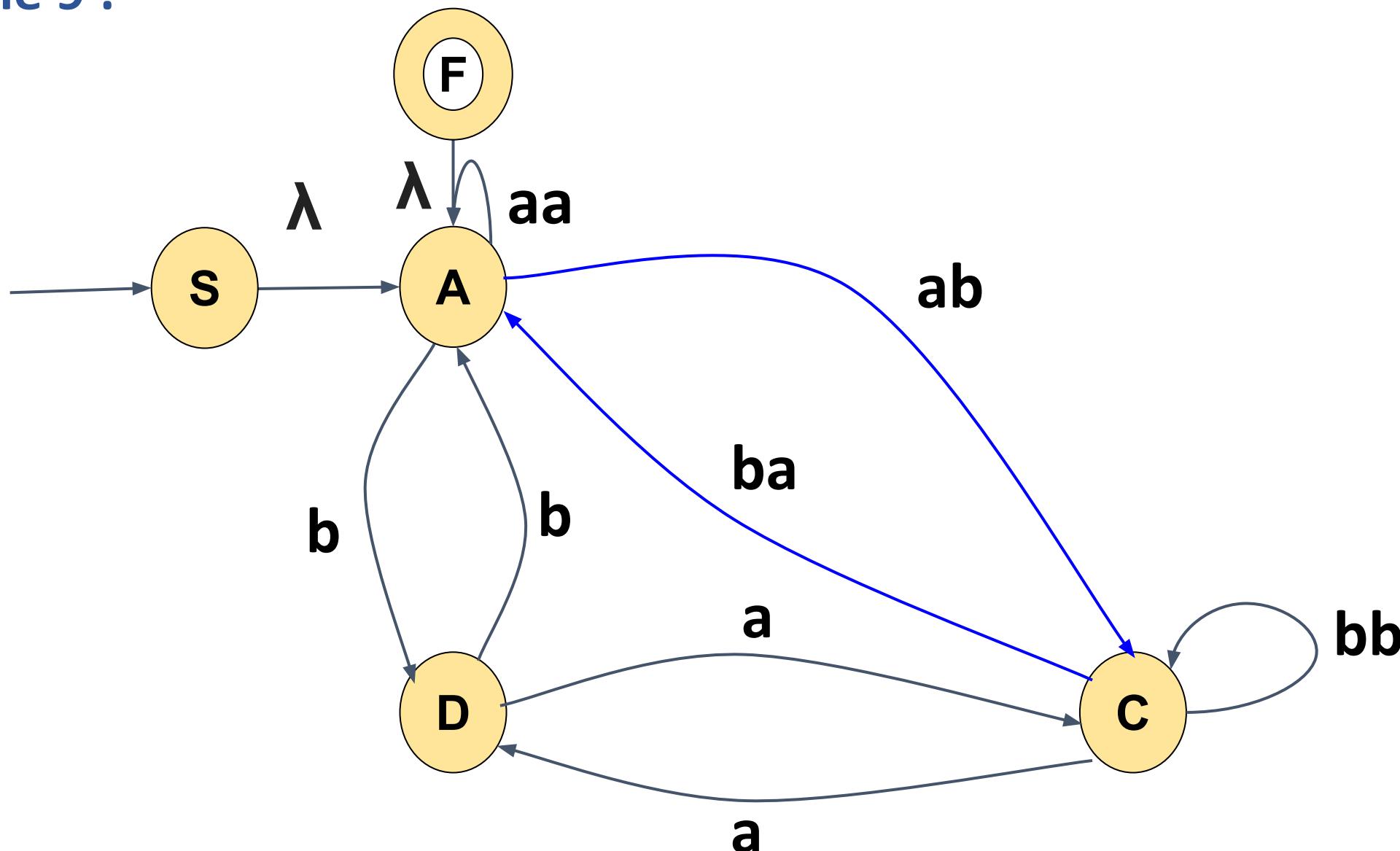


### Example 9 :



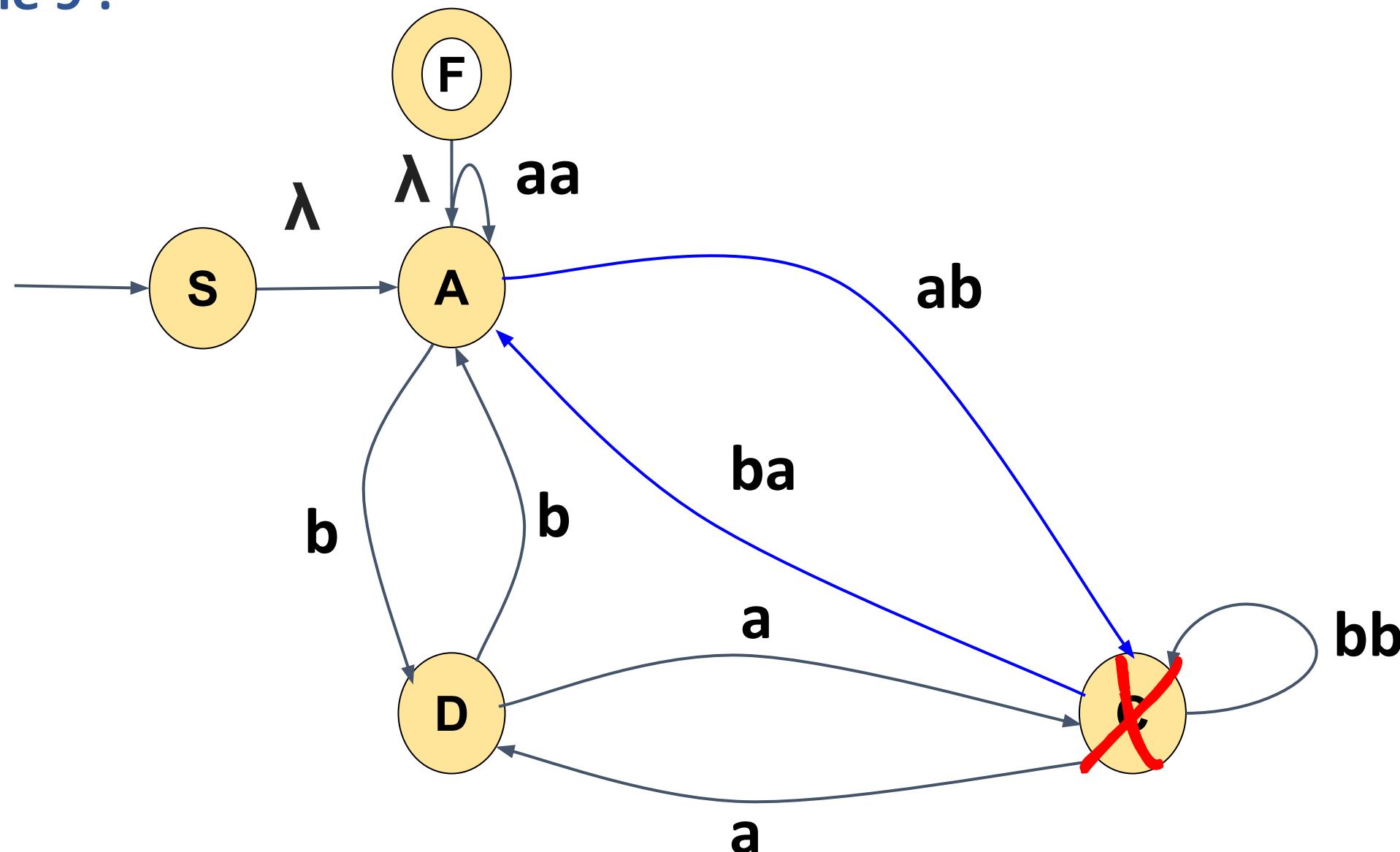
1. Eliminate B

### Example 9 :



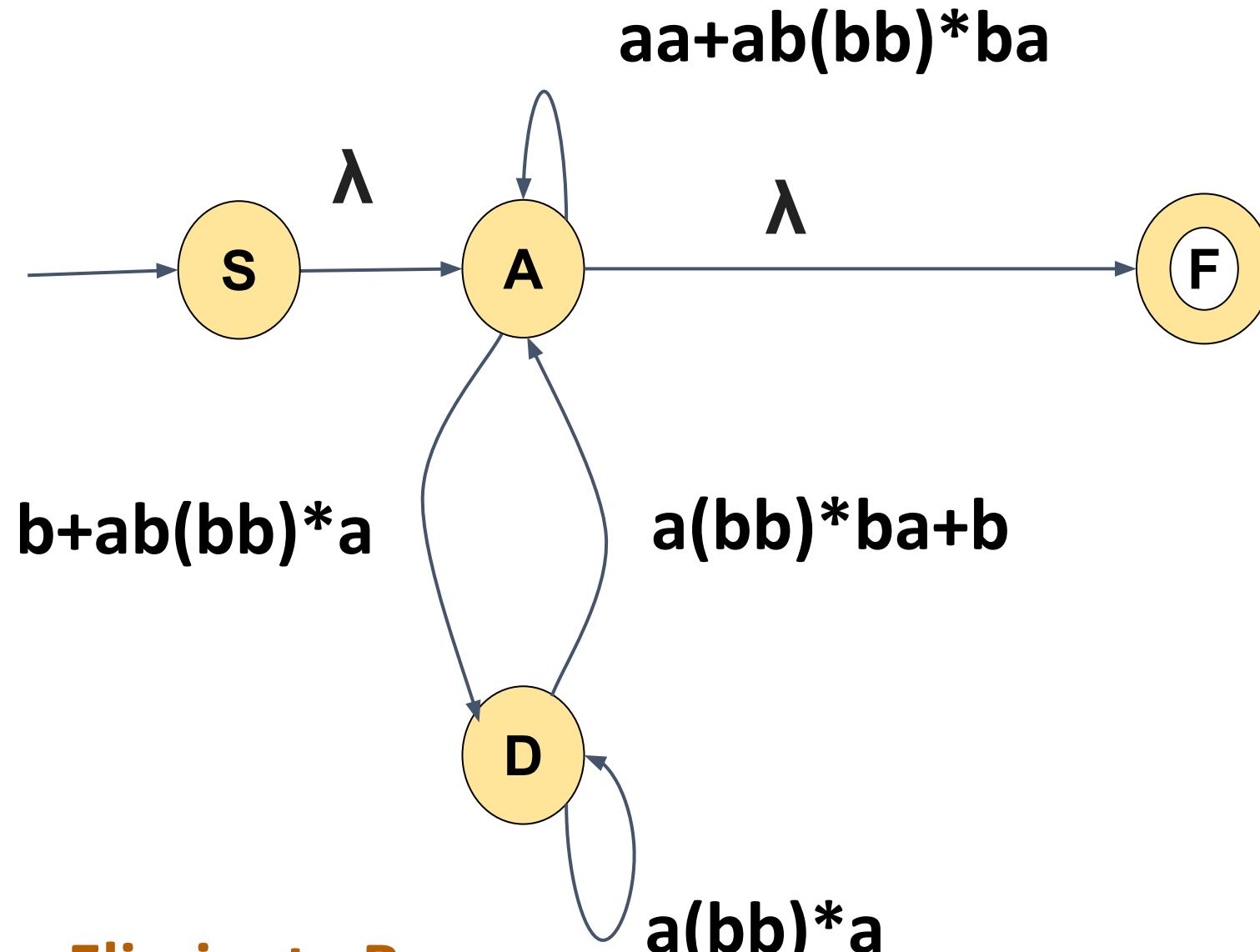
1. Eliminate B

### Example 9 :



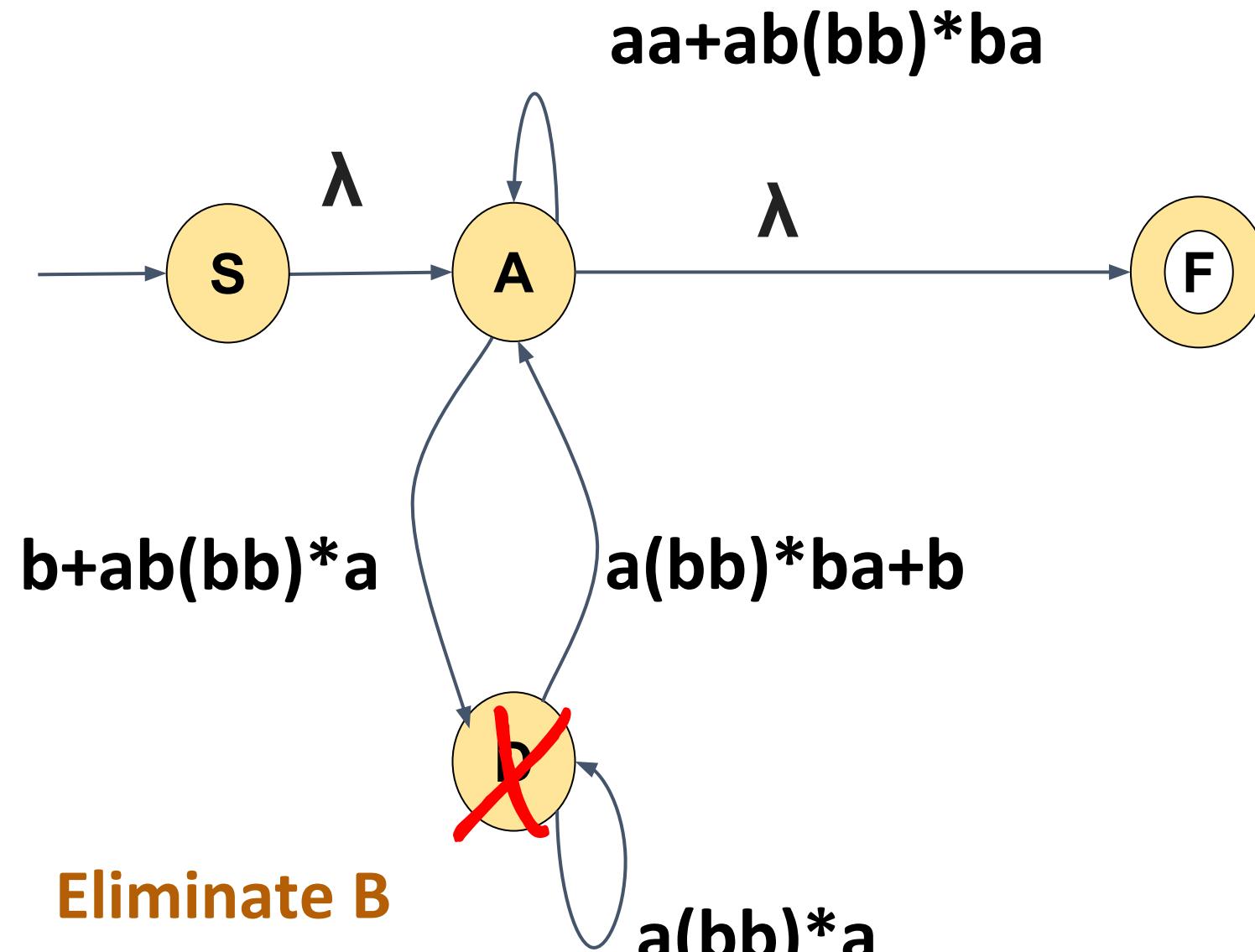
1. Eliminate B
2. Eliminate C

### Example 9 :



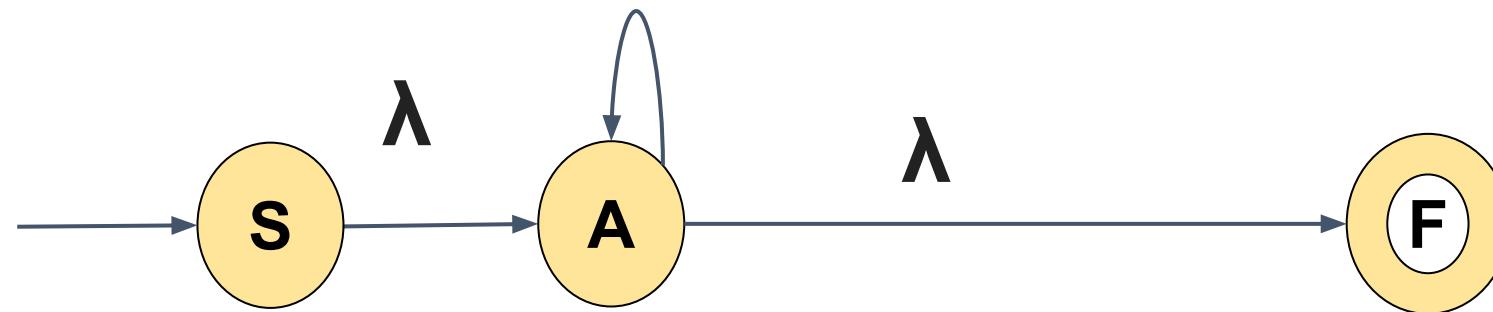
1. Eliminate B
2. Eliminate C

### Example 9 :



1. Eliminate B
2. Eliminate C
3. Eliminate D

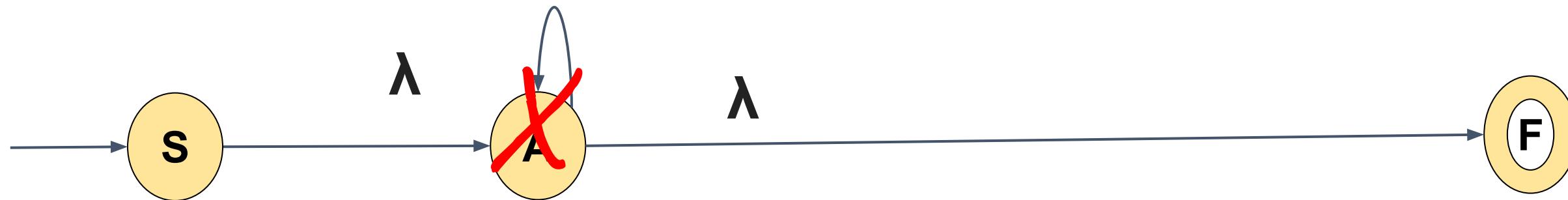
### Example 9 :

$$aa + ab(bb)^*ba + (b + ab(bb)^*a)(a(bb)^*a)^*(a(bb)^*ba + b)$$


1. Eliminate B
2. Eliminate C
3. Eliminate D

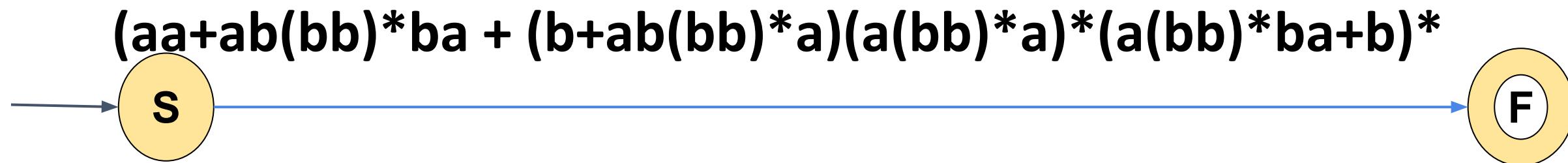
### Example 9 :

$$(aa+ab(bb)^*ba + (b+ab(bb)^*a)(a(bb)^*a)^*(a(bb)^*ba+b))$$



1. Eliminate B
2. Eliminate C
3. Eliminate D
4. Eliminate A

### Example 9 :



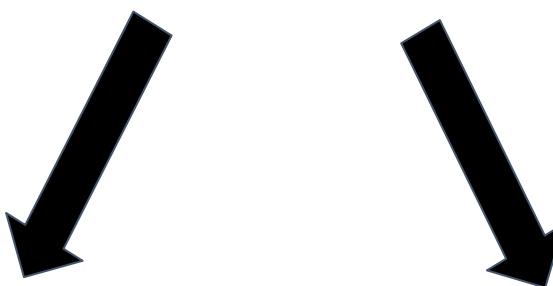
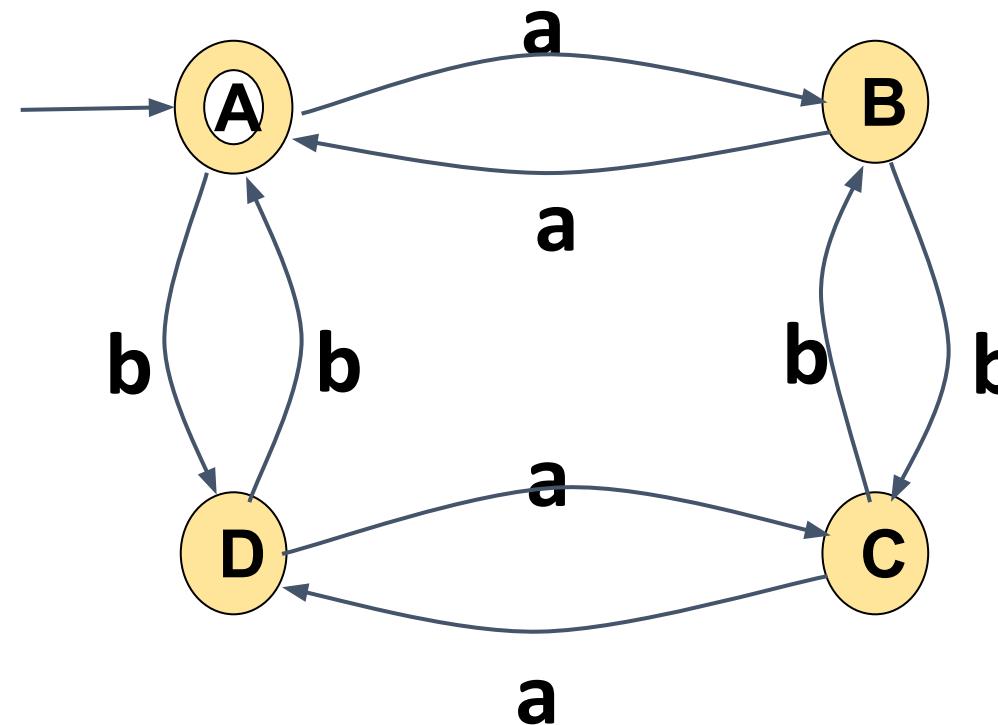
1. Eliminate B
2. Eliminate C
3. Eliminate D
4. Eliminate A

### Example 9 :



$RE = ((aa+ab(bb)^*ba) + (b+ab(bb)^*a)(a(bb)^*a)^*(a(bb)^*ba+b))^*$

### Example 9 :



$(aa+bb + (ab+ba+(aa+bb)*(ab+ba))*$

Eliminate : B,D,C,A

$((aa+ab(bb)*ba)^* +$   
 $(b+ab(bb)*a)(a(bb)*a)^*(a(bb)*ba+b))^*$

Eliminate B,C,D,A



**THANK YOU**

---

**Preet Kanwal**

Department of Computer Science & Engineering

**[preetkanwal@pes.edu](mailto:preetkanwal@pes.edu)**

**+91 80 6666 3333 Extn 724**



# Automata Formal Languages & Logic

---

**Preet Kanwal**

Department of Computer Science & Engineering

# Automata Formal Languages & Logic

---

## Unit 2

**Preet Kanwal**

Department of Computer Science & Engineering

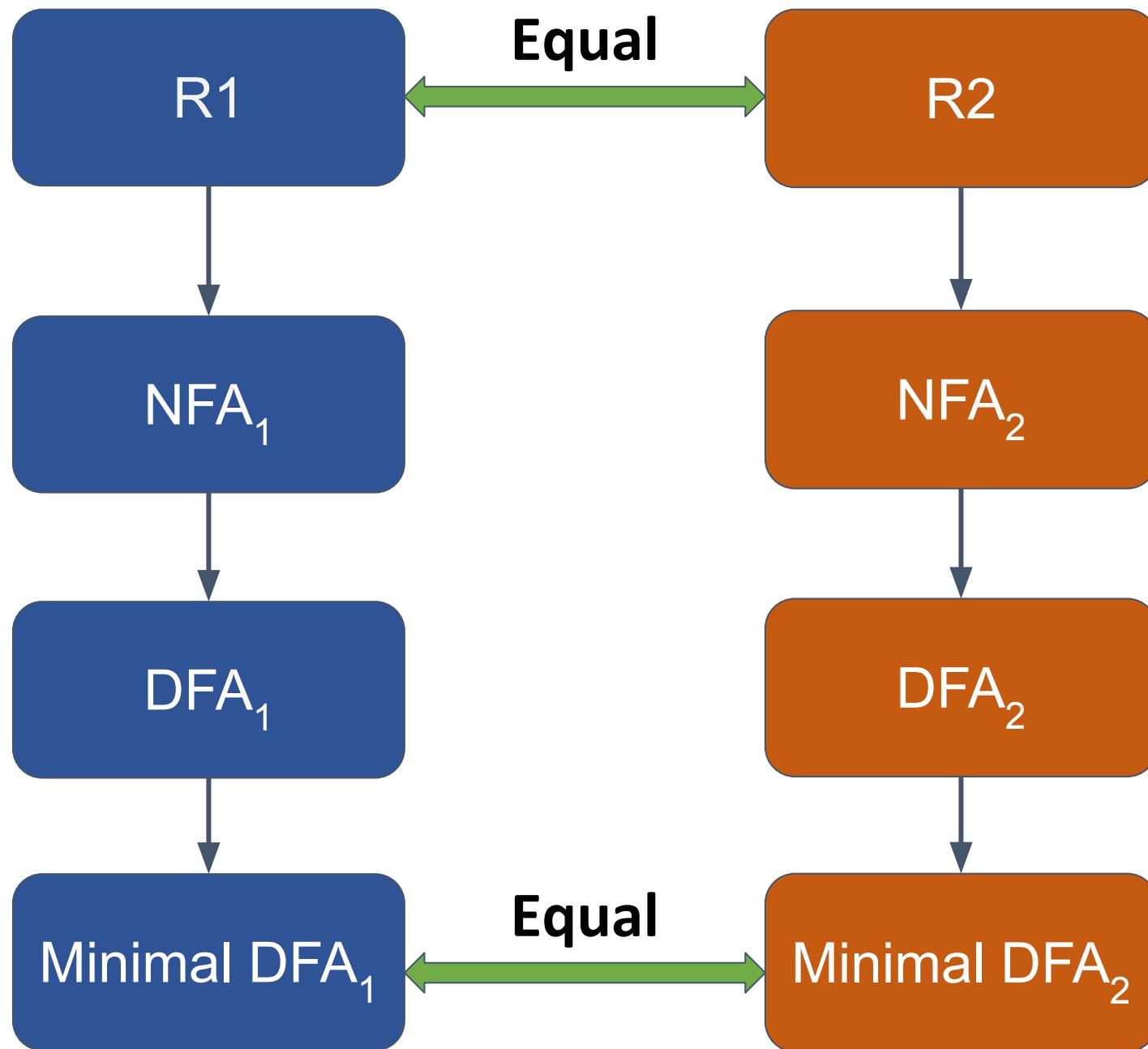
Two regular expressions ( $R_1$  and  $R_2$ ) are equivalent ( $R_1 = R_2$ ) iff:

$$L(R_1) = L(R_2)$$

We can determine the equivalence using:

- 1) A Formal method
- 2) An Informal method:

### Formal method to prove Equivalence of two Regex



### Informal Method -

We try proving  $R_1 \neq R_2$

- Find a string that can be matched with only one of the regex hence proving that the two regex are not equivalent.
- Faster
- But, based on hit and trail.
- Can only be used to prove inequality!!

**Let us look at examples and find out whether two regex are equivalent or not??**

### Example 1:

$$(0+1)^*(0+\lambda)$$
$$(1+\lambda)(1+0)^*$$

Example 1:

Formal  
Method

$$(0+1)^*(0+\lambda)$$

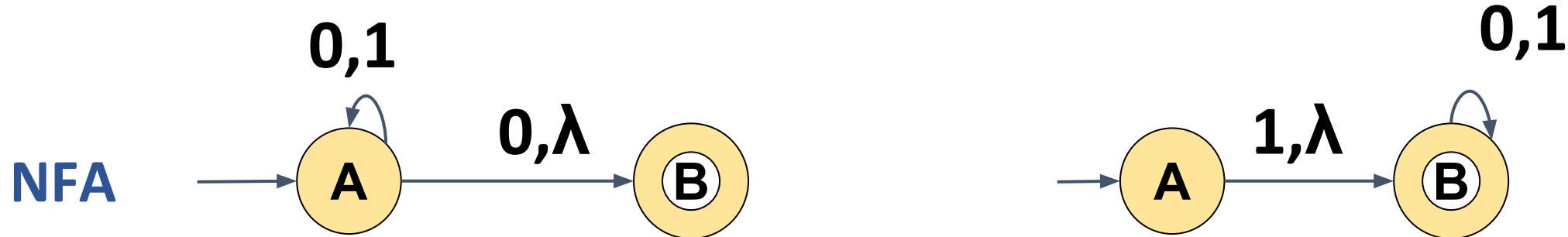
$$(1+\lambda)(1+0)^*$$

### Example 1:

Formal  
Method

$$(0+1)^*(0+\lambda)$$

$$(1+\lambda)(1+0)^*$$

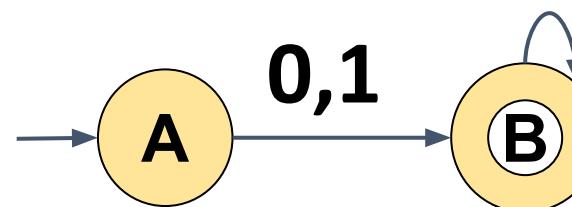
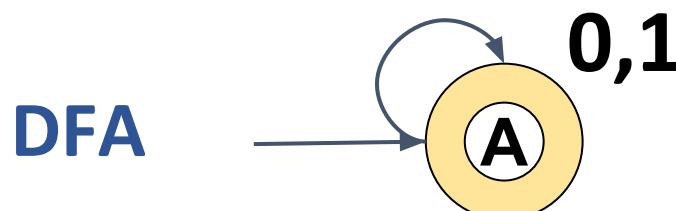
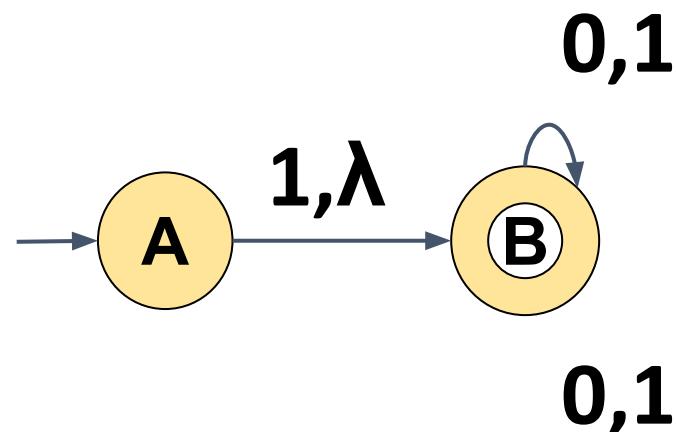
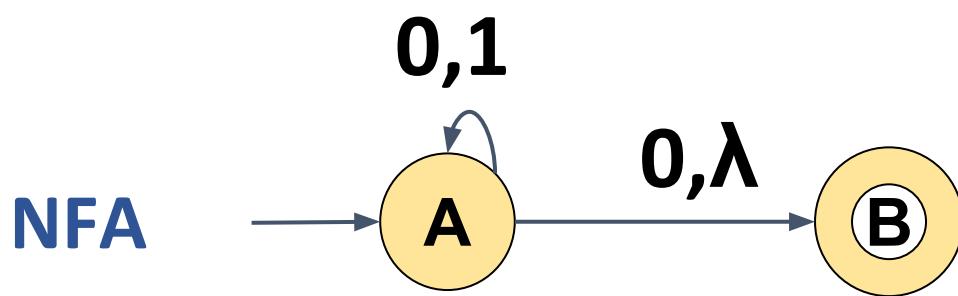


### Example 1:

Formal  
Method

$$(0+1)^*(0+\lambda)$$

$$(1+\lambda)(1+0)^*$$

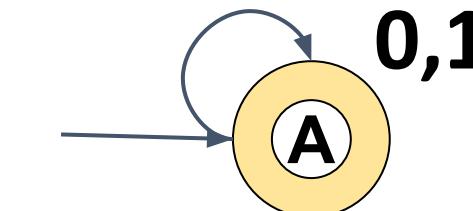
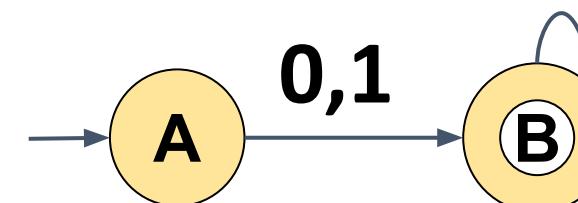
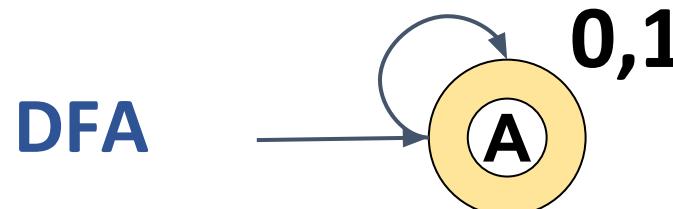
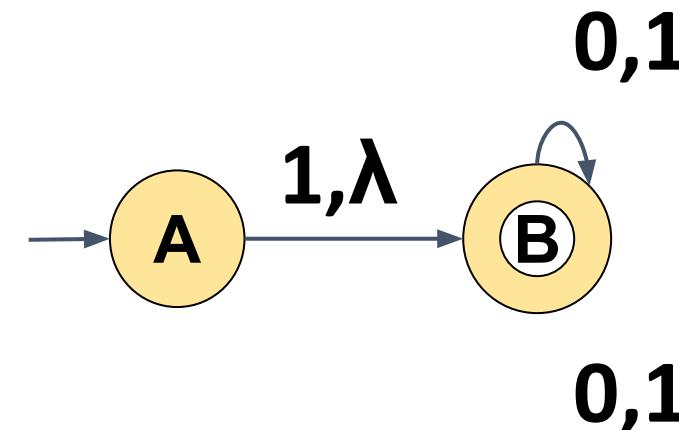
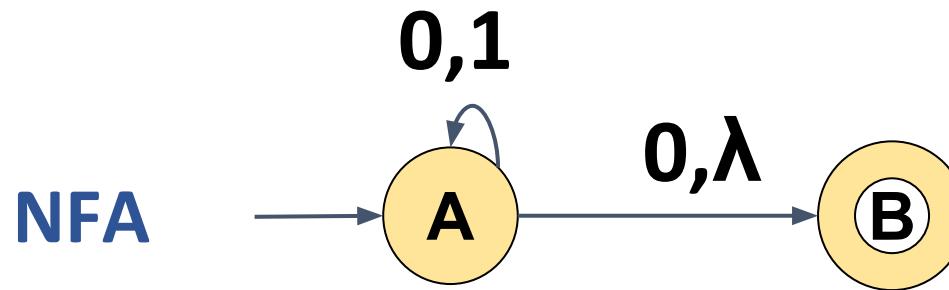


### Example 1:

Formal  
Method

$$(0+1)^*(0+\lambda)$$

$$(1+\lambda)(1+0)^*$$

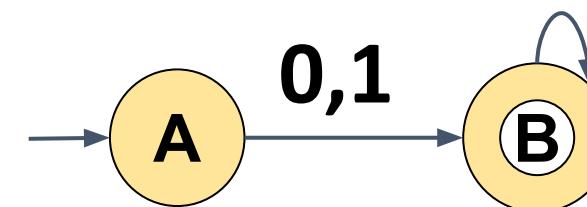
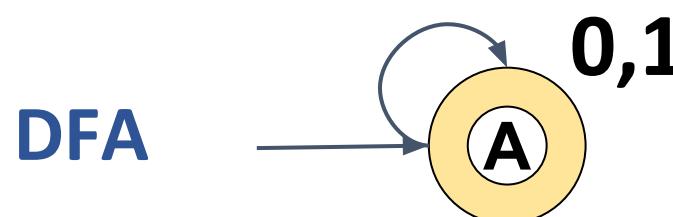
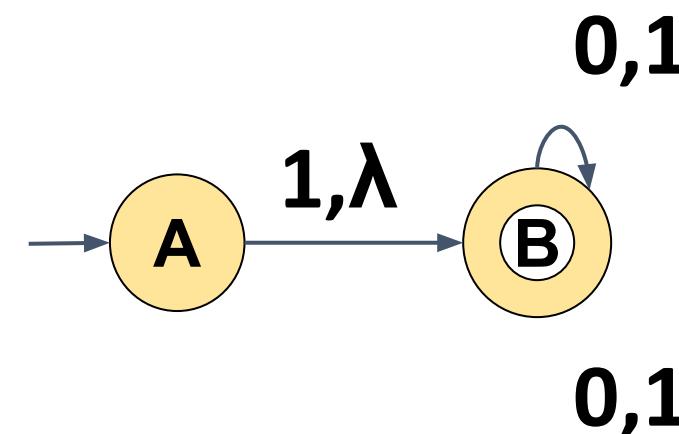
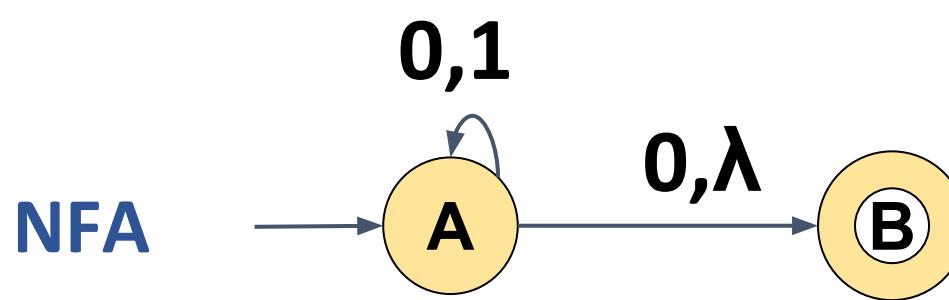


### Example 1:

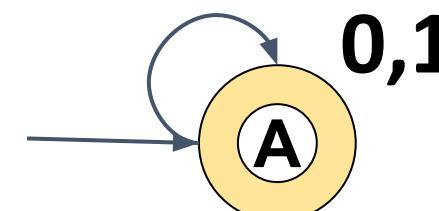
Formal  
Method

$$(0+1)^*(0+\lambda)$$

$$(1+\lambda)(1+0)^*$$

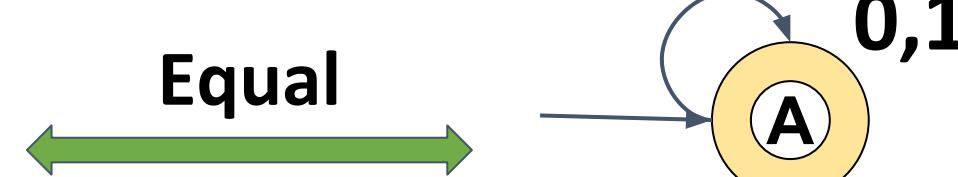
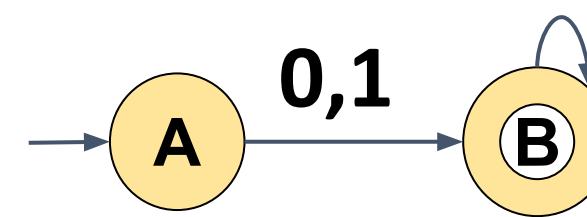
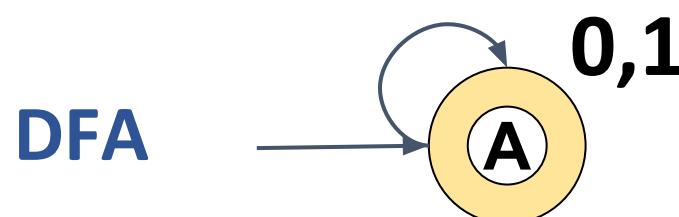
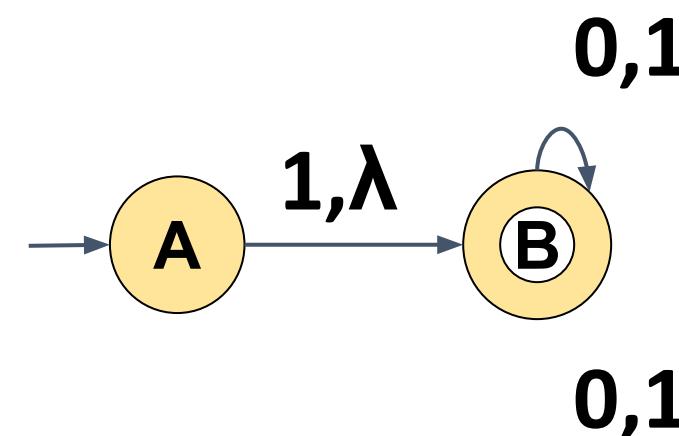
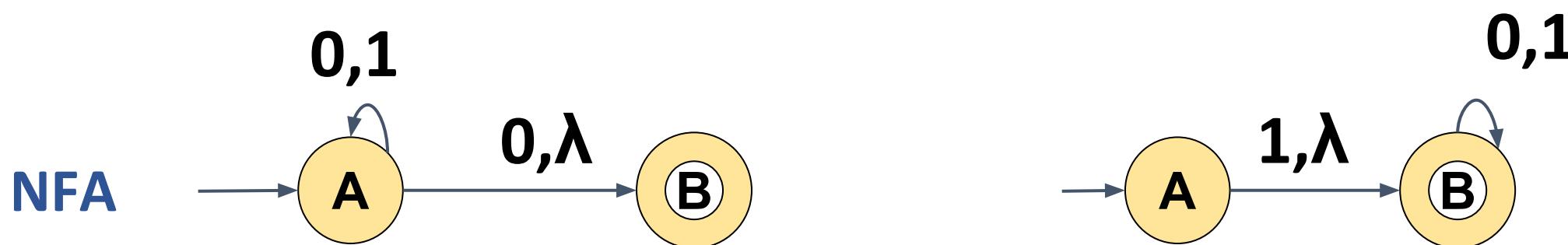


Equal



## Example: Equivalence of two regular expression

Formal  
Method



### Example 2:

$$(0+\lambda)(11^*0)^*(1+\lambda)$$
$$(1+\lambda)(011^*)(0+\lambda)$$

# Automata Formal Languages and Logic

## Unit 2 - Equivalence of two regular expression



Example 2:

$$(0+\lambda)(11^*0)^*(1+\lambda)$$

Informal  
Method

$$(1+\lambda)(011^*)(0+\lambda)$$

Example 2:

$$(0+\lambda)(11^*0)^*(1+\lambda)$$

Informal  
Method

$$(1+\lambda)(011^*)(0+\lambda)$$

Find a string belongs to only one of the regex

Example 2:

$(0+\lambda)(11^*0)^*(1+\lambda)$

Informal  
Method

$(1+\lambda)(011^*)(0+\lambda)$

Find a string belongs to only one of the regex

$(1+\lambda)(011^*)(0+\lambda)$

Example 2:

$(0+\lambda)(11^*0)^*(1+\lambda)$

Informal  
Method

$(1+\lambda)(011^*)(0+\lambda)$

Find a string belongs to only one of the regex

$(1+\lambda)(011^*)(0+\lambda)$

011

Example 2:

$$(0+\lambda)(11^*0)^*(1+\lambda)$$

Informal  
Method

$$(1+\lambda)(011^*)(0+\lambda)$$

Find a string belongs to only one of the regex

$$(0+\lambda)(\textcolor{yellow}{11^*0})^*(1+\lambda)$$
$$(\textcolor{yellow}{1+\lambda})(\textcolor{yellow}{011^*})(0+\lambda)$$

011

Example 2:

$$(0+\lambda)(11^*0)^*(1+\lambda)$$

Informal  
Method

$$(1+\lambda)(011^*)(0+\lambda)$$

Find a string belongs to only one of the regex

$$(0+\lambda)(11^*0)^*(1+\lambda)$$

011 cannot be  
generated

$$(1+\lambda)(011^*)(0+\lambda)$$

011

## Example Equivalence of two regular expression Informal Method

$$(0+\lambda)(11^*0)^*(1+\lambda)$$
 $\neq$ 
$$(1+\lambda)(011^*)(0+\lambda)$$

Find a string belongs to only one of the regex

$$(0+\lambda)(11^*0)^*(1+\lambda)$$

011 cannot be generated

$$(1+\lambda)(011^*)(0+\lambda)$$

011

Can you answer whether the two regex are equivalent or not??

1)  $(1+\lambda)(00^*1)^* 0^*$  and  $(0+\lambda)(11^*0)1^*$

2)  $0^*(10^*)^*$  and  $(1^*0)^*1^*$



**THANK YOU**

---

**Preet Kanwal**

Department of Computer Science & Engineering

**[preetkanwal@pes.edu](mailto:preetkanwal@pes.edu)**

**+91 80 6666 3333 Extn 724**

# **AUTOMATA FORMAL LANGUAGES AND LOGIC**

## **Lecture notes on Regular grammar and Parsing**



**Prepared by:**

**Prof.Sangeeta V I**

**Assistant Professor**

**Department of Computer Science & Engineering**

**PES UNIVERSITY**

**(Established under Karnataka Act No.16 of 2013)**

**100-ft Ring Road, BSK III Stage, Bangalore - 560 085**

## Table of contents

| <b>Section</b> | <b>Topic</b>   | <b>Page number</b> |
|----------------|--|--------------------|
| <b>1</b>       | <b>Introduction to Grammar</b>                                       | <b>7</b>           |
| <b>2</b>       | <b>Grammar definition</b>  | <b>7</b>           |
| <b>3</b>       | <b>Sentence and Sentential form</b>                                  | <b>8</b>           |
| <b>4</b>       | <b>Chomsky Hierarchy</b>   | <b>8</b>           |
| <b>5</b>       | <b>Linear and Non-Linear grammar</b>                                 | <b>8</b>           |
| <b>5.1</b>     | <b>Right Linear grammar and Left Linear Grammar</b>                  | <b>9</b>           |
| <b>6</b>       | <b>Constructing regular grammar for given language descriptions.</b> | <b>10</b>          |
| <b>7</b>       | <b>Aspects of a Grammar</b>  | <b>16</b>          |
| <b>8</b>       | <b>Parse tree/Derivation tree</b>                                    | <b>16</b>          |
| <b>9.1</b>     | <b>Tree and its representation</b>                                   | <b>17</b>          |
| <b>9.2</b>     | <b>Parsing</b>   | <b>18</b>          |
| <b>9.2.1</b>   | <b>Top-down Parsing and bottom up Parsing</b>                        | <b>18</b>          |
| <b>9.2.2</b>   | <b>Generating a parse tree for a given String w and Grammar G.</b>   | <b>19</b>          |

| <b>Table of contents</b> |   |                    |
|--------------------------|---|--------------------|
| <b>Section</b>           | <b>Topic</b>  | <b>Page number</b> |
| 9.3                      | <b>Constructing a left linear Grammar</b>                           | 21                 |
| 9.4                      | <b>Constructing a left linear grammar from the finite automata.</b> | 21                 |
| 9.5                      | <b>Converting to left linear grammar to finite automata</b>         | 23                 |
| 9.6                      | <b>Which one is easier left linear or right linear?</b>             | 24                 |

### **Examples Solved:**

| # | <b>Constructing regular grammar for given language descriptions.</b>                    | <b>Page number</b> |
|---|---|--------------------|
| 1 | <b>Construct a regular grammar for the language <math>L = \{a\}</math>.</b>             | 9                  |
| 2 | <b>Construct a regular grammar for the language <math>L = \{a, b\}</math>.</b>          | 9                  |
| 3 | <b>Construct a regular grammar for the language <math>L = \{ab\}</math>.</b>            | 10                 |
| 4 | <b>Construct a regular grammar for the language <math>L = \{ab, ba\}</math>.</b>        | 10                 |
| 5 | <b>Construct a regular grammar for the language <math>L = \{a^n   n \geq 0\}</math></b> | 10                 |

| #  | Constructing regular grammar for given language descriptions.   | Page number |
|----|---|-------------|
| 6  | Construct a regular grammar for the language $L=\{a^n   n \geq 1\}$   | 11          |
| 7  | Construct a regular grammar for the regular expression $(a+b)^*$  | 11          |
| 8  | Construct a regular grammar for the regular expression $(a+b)^+$  | 12          |
| 9  | Construct a regular grammar for the regular expression $(ab)^*$   | 12          |
| 10 | Construct a regular grammar for the regular expression $(ab+ba)^*$  | 13          |
| 11 | Construct a regular grammar for the language $L=\{a^m b^n   m,n \geq 0\}$   | 13          |
| 12 | Construct a regular grammar for the given language with an even number of a's. $L=\{a^{2n}   n \geq 0\}$                    | 14          |
| 13 | Construct a regular grammar for the given language with an odd number of a's. $L=\{a^{2n+1}   n \geq 0\}$ .                 | 14          |
| 14 | Construct a regular grammar for the given language with a number of a's as multiples of 4. $L=\{a^{4n}   n \geq 0\}$ .      | 15          |
| 15 | Convert finite automata to regular grammar for the language to accept at least one 'a' over the alphabet $\Sigma=\{a,b\}$ . | 15          |

### Examples Solved:

| # | Generating a parse tree for a given String w and Grammar G.   | Page number |
|---|---|-------------|
| 1 | Generate parse tree for the sting w=1010 given the grammar .<br>$S \rightarrow 0S 1S 0$   | 19          |
| 2 | Generate parse tree for the sting w=aaabbhhh given the grammar .<br>$S \rightarrow aaB$<br>$B \rightarrow aB bbbE$<br>$E \rightarrow bE  \lambda$ | 20          |

### Examples Solved:

| # | Constructing a left linear grammar from the finite automata.         | Page number |
|---|--|-------------|
| 1 | Constructing a left linear grammar for $L=\{aw, w \in \{a,b\}^*\}$ . | 22          |

### Examples Solved:

| # | Converting to left linear grammar to finite automata   | Page number |
|---|--|-------------|
| 1 | Convert a given left linear grammar to finite automata<br>Left Linear grammar<br>$B \rightarrow Ba Bb Aa$<br>$A \rightarrow \lambda$ | 23          |

## 1. Introduction to Grammar

Language is a set of strings constructed over an alphabet which satisfies certain properties or follows a certain set of rules. These rules can be encoded using the concept of Grammar. Grammars are used to compactly express and generate languages.

Grammars denote syntactic rules that means it is concerned only with the syntax of a string and not the meaning .

Grammar is another way to represent a set of strings. Rather than define the set through a notion of which strings are accepted or rejected, like we do with a machine model, we define the set by describing a collection of strings that the grammar can generate.

For example:She is a boy, is grammatically correct but semantically wrong.

There are various applications of regular grammars for example in compiler design the concept of grammar is used construct syntax of a programming language

Formal grammar or just grammar is a set of rules that describe how strings that are valid according to the language's syntax. can be generated from the language's alphabet.

## 2. Grammar definition

A grammar is a 4-tuple  $G = (V, T, P, S)$ , where

- $V$  is a set of variables/non-terminals. We can have
  - a. Lowercase names such as expr,var,op,stmt. or
  - b. Capital letter near the beginning of the alphabet. A,B,C,D or Uppercase letters near the end of alphabet X,Y,Z. as non-terminals
  - c. Usually Variable  $S$  is used as the-start symbol.
  - d. Lowercase letters near the end of the alphabet. U,v,w,x,y,z are used to represent the entire string.
- $T$  is the set terminal symbols
  - a. These are basically the symbols from the input alphabet for example could be a Keywords such as :
  - b. If ,else,then ,for ,while,do while etc
  - c. Or Digits 0 to 9
  - d. Or Lowercase alphabets such as a,b,c,d etc
  - e. Or Bold faced letters
  - f. Or Symbols such as + , - , \* , / , : , ; etc

We can use Greek letters- $\alpha, \beta, \gamma$  to denote a grammar symbol which could either be a either terminal or a non-terminal.

- $P$  is a finite set of production rules (also called simply rules or productions) of form  $\alpha \rightarrow \beta$  where  $\alpha$  is always a non terminal and cannot be  $\lambda$ . And  $\beta$  is a string over  $(V \cup T)^*$
- $S$  is the start symbol.

- The set of strings that can be generated or derived from Start symbol S of a grammar is called Language of the Grammar denoted by L(G).

### 3. Sentence and Sentential form

**Sentence** or the word string are used alternatively and are made up only of terminals.

Example :

For the Grammar  $S \rightarrow aS \mid a \mid \lambda$

Sentence (or string) is { a or  $\lambda$  or aa or aaaa)

A **sentential form** is made up over set of terminals and non-terminals (VUT)\*

Example:

For the Grammar  $S \rightarrow aS \mid a \mid \lambda$

Sentential form is { a or  $\lambda$  or  $aS$  or aa or or  $aS$  aaaa)

### 4. Chomsky Hierarchy

Noam Chomsky in 1956 described Chomsky hierarchy of grammars

**Chomsky Hierarchy** represents the class of languages that are accepted by the different machines. Chomsky classified grammars on the basis of the form of their productions. This is a hierarchy. Therefore every language of type 3 is also of type 2, 1 and 0. Similarly, every language of type 2 is also of type 1 and type 0, etc

Type 3 Grammar is known as Regular Grammar.

### 5. Linear and Non-Linear grammar

We can broadly categorize grammars into two categories as being linear and non-linear  
Linear grammar is the one which has at most one non terminal in the RHS of the production.

For Examples  $S \rightarrow aSa \mid aS \mid Sa \mid \epsilon$

here we have only one nonterminal/variable in the right hand side of each of its productions.

In non linear grammar there is no restriction on the number non terminals that can appear on the RHS of a production for example :  $S \rightarrow SS \mid aSS \mid a \mid \epsilon$

Here we have two non terminals in the RHS .

A regular grammar has more restriction and is a subset of linear grammar.

## 5.1 Right Linear grammar and Left Linear Grammar

A regular grammar can either be right linear or left linear.

A grammar is **left linear** iff the non-terminal on RHS of a production appears on the leftmost side. That means the non-terminal is the first grammar symbol on the RHS of a production

For example

$$S \rightarrow Sa \mid \epsilon \quad (\text{left linear})$$

A grammar is **right linear** iff the non-terminal on RHS of a production appears on the rightmost end. That means the non-terminal is the last grammar symbol on the RHS of a production

For example

$$S \rightarrow aS \mid \epsilon \quad (\text{right linear})$$

Since both the forms are linear, we can have only one non terminal on RHS of a production.

**The right- and left-linear grammars are equivalent.** That means for a given language we can have both right and left linear grammar constructed. We could also convert one form to another.

## **6. Constructing regular grammar for given language descriptions.**

### **Example 1:**

**Construct a regular grammar for the language  $L = \{a\}$ .**

| Language    | Regular Expression | Regular Grammar   |
|-------------|--------------------|-------------------|
| $L = \{a\}$ | a                  | $S \rightarrow a$ |

We start the regular grammar with a start symbol “S”.

We can convert regular expressions to regular grammar as the regular expressions are a form of representing regular languages.

Here ,since ‘a’ is the only symbol in the language ,the regular grammar from S generates only ‘a’,  $S \rightarrow a$  (production rule)

### **Example 2:**

**Construct a regular grammar for the language  $L = \{a, b\}$ .**

| Language       | Regular Expression | Regular Grammar          |
|----------------|--------------------|--------------------------|
| $L = \{a, b\}$ | $a+b$              | $S \rightarrow a \mid b$ |

Here ,since ‘a’ or ‘b’ is a symbol in the language ,the regular grammar from S generates either ‘a’ or ‘b’ ,  $S \rightarrow a$  or  $S \rightarrow b$ .

### **Example 3:**

**Construct a regular grammar for the language  $L = \{ab\}$ .**

| Language     | Regular Expression | Regular Grammar                         |
|--------------|--------------------|---|
| $L = \{ab\}$ | a.b                | $S \rightarrow aA$<br>$A \rightarrow b$ |

Here ,since 'a.b' is symbol in the language ,the regular grammar produces  $S \rightarrow ab$

#### Example 4:

Construct a regular grammar for the language  $L = \{ab, ba\}$ .

| Language     | Regular Expression | Regular Grammar         |
|--------------|--------------------|-------------------------|
| $L = \{ab\}$ | a.b                | $S \rightarrow ab   ba$ |

#### Example 5:

Construct a regular grammar for the language  $L = \{a^n | n \geq 0\}$

| Language                             | Regular Expression | Regular Grammar              |
|--------------------------------------|--------------------|------------------------------|
| $L = \{\lambda, a, aa, aaa, \dots\}$ | $a^*$              | $S \rightarrow aS   \lambda$ |

The language with any number of a's including empty string .

The regular grammar from S generates any number of a's with the production  $S \rightarrow aS$  or empty string with production  $S \rightarrow \lambda$

We combine the productions,

$S \rightarrow aS | \lambda$

#### Example 6:

Construct a regular grammar for the language  $L = \{a^n | n \geq 1\}$

| Language                    | Regular Expression | Regular Grammar        |
|-----------------------------|--------------------|------------------------|
| $L = \{a, aa, aaa, \dots\}$ | $a^+$              | $S \rightarrow aS   a$ |

The language with any number of a's but not an empty string ,minimum string is one 'a'.

The regular grammar from S generates any number of a's with the production  $S \rightarrow aS$  or single 'a' with production  $S \rightarrow a$

We combine the productions,

$$S \rightarrow aS | a$$

### Example 7:

**Construct a regular grammar for the regular expression  $(a+b)^*$**

| Language   | Regular Expression | Regular Grammar                   |
|--|--------------------|-----------------------------------|
| $L = \{\lambda, a, b, abb, baaaab, \dots\}$<br>$L = \{\text{any number of 'a's and b's}\}$ | $(a+b)^*$          | $S \rightarrow aS   bS   \lambda$ |

The language with any number of a's and any number of b's including an empty string.

The regular grammar from S generates any number of a's with the production  $S \rightarrow aS$  and any number of b's with the production  $S \rightarrow bS$  and empty string with production

$$S \rightarrow \lambda.$$

We combine the productions,

$$S \rightarrow aS | bS | \lambda$$

### Example 8 :

**Construct a regular grammar for the regular expression  $(a+b)^+$**

| Language   | Regular Expression | Regular Grammar                          |
|--|--------------------|--|
| $L = \{\lambda, a, b, ab, abb, baaaab, \dots\}$<br>$L = \{\text{any number of 'a's and b's}\}$ | $(a+b)^*$          | $S \rightarrow aS \mid bS \mid a \mid b$ |

The language with any number of a's and any number of b's but not an empty string.

The regular grammar from S generates any number of a's with the production  $S \rightarrow aS$  and any number of b's with the production  $S \rightarrow bS$ , single 'a' with production  $S \rightarrow a$  and single 'b' with production  $S \rightarrow b$ .

We combine the productions,

$$S \rightarrow aS \mid bS \mid a \mid b$$

### Example 9:

**Construct a regular grammar for the regular expression  $(ab)^*$**

| Language   | Regular Expression | Regular Grammar                  |
|--|--------------------|----------------------------------|
| $L = \{\lambda, ab, ababababab, \dots\}$<br>$L = \{\text{sequences of ab's}\}$ | $(ab)^*$           | $S \rightarrow abS \mid \lambda$ |

The language with sequences of ab's including empty string.

### Example 10:

**Construct a regular grammar for the regular expression  $(ab+ba)^*$**

| Language  | Regular Expression | Regular Grammar                           |
|---|--------------------|---|
| $L = \{\lambda, ab, abababab, \dots\}$<br>$L = \{\text{sequences of } ab's\}$ | $(ab)^*$           | $S \rightarrow abS \mid baS \mid \lambda$ |

The language with sequences of ab's or ba's including empty string.

### Example 11:

Construct a regular grammar for the language  $L = \{a^m b^n \mid n, m \geq 0\}$

| Language  | Regular Expression | Regular Grammar  |
|---|--------------------|--|
| $L = \{\lambda, a, b, bb, abb, \dots\}$<br>$L = \{\text{any number of } a's \text{ followed by any number of } b's\}$ | $a^*b^*$           | $S \rightarrow aS \mid A$<br>$A \rightarrow bA \mid \lambda$ |

$aS$  generates as many a's as we want and when we go ahead in the pattern we introduce a new non-terminal  $A$  which generates only b's.

### Example 12:

**Construct a regular grammar for the given language with an even number of a's.**  
 $L=\{a^{2n} \mid n \geq 0\}$

| Language  | Regular Expression | Regular Grammar                  |
|---|--------------------|----------------------------------|
| $L=\{\lambda,$<br>$aa,$<br>$aaaa,$<br>$aaaaaaaa ....$ | $(aa)^*$           | $S \rightarrow aaS \mid \lambda$ |

### Example 13:

**Construct a regular grammar for the given language with an odd number of a's.**  
 $L=\{a^{2n+1} \mid n \geq 0\}$ .

| Language   | Regular Expression | Regular Grammar            |
|--|--------------------|----------------------------|
| $L=\{a,$<br>$aaa,$<br>$aaaa,$<br>$aaaaaaaa ....$ | $(aa)^*a$          | $S \rightarrow aaS \mid a$ |

### Example 14:

**Construct a regular grammar for the given language with a number of a's as multiples of 4.  $L=\{a^{4n} | n \geq 0\}$**

| Language  | Regular Expression | Regular Grammar                    |
|---|--------------------|------------------------------------|
| $L=\{\lambda,$<br>$aaaa,$<br>$aaaaaaaa,$<br>$aaaaaaaaaaaa, \dots$ | $(aaaa)^*$         | $S \rightarrow aaaaS \mid \lambda$ |

### **Example 15:**

**Convert finite automata to regular grammar for the language to accept at least one 'a' over the alphabet  $\Sigma=\{a,b\}$**

| Language  | Regular Expression | Regular Grammar   |
|---|--------------------|---|
| $L=\{a,$<br>$bb\dots a,$<br>$ba \text{ any } \# \text{ of } a's \& b's$ | $b^* a (a+b)^*$    | $S \rightarrow bS \mid aA$<br>$A \rightarrow aA \mid bA \mid \lambda$ |

## **7. Aspects of a Grammar**

There are two Aspects of a Grammar.

First is the Generative(derivation) aspect that means given a Grammar G describing language we can generate all strings that belong to the language of the Grammar.

Second is the Analytical(parsing) aspect where given a string w we can make a check whether w belongs to the language of the Grammar or not.

## 8. Parse tree/ derivation tree

We all know what a tree looks like. It has leaves, branches and a root.



Technically we specify the tree upside down. That is we specify the root first and then leaves where branches will help connect root to each of the leaves.



Parse/Derivation tree is a graphical representation for the derivation of the given production rules .

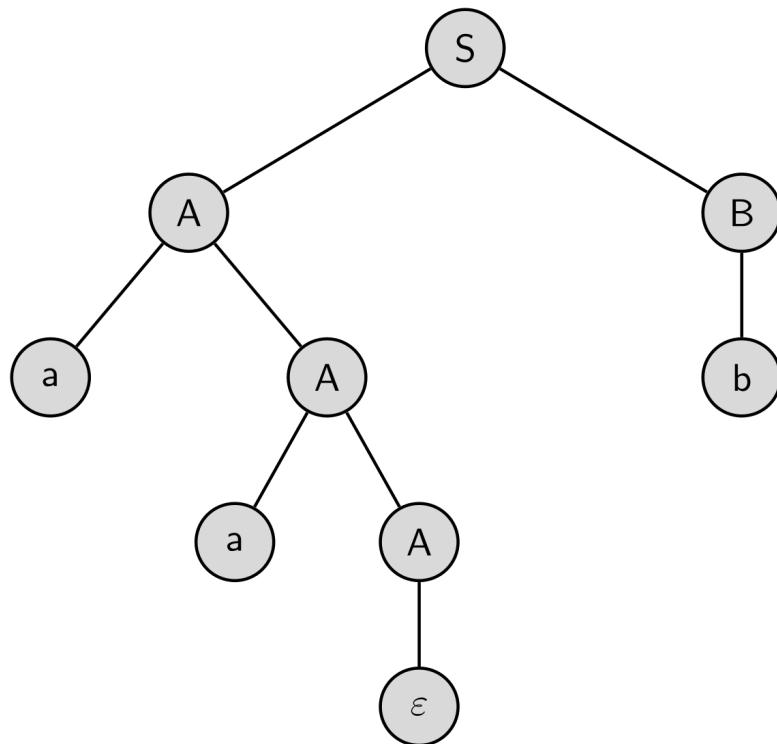
A parse tree/derivation tree contains the following properties:

- The root node is always a node indicating start symbols.
- The interior nodes are always non-terminal.
- The leaf node is always represented by a terminal.
- The derivation is read from left to right. Yield or output of the parse tree is the string derived.

### 9.1 Tree and its representation

- A tree is nothing but a graph.
- It is made up of a finite set of nodes.
- Nodes are usually denoted by circles or ovals .
- We use the word Edge instead of Branch.
- Edges are the connections between the nodes. It connects two nodes.
- Edges are usually represented by lines, or lines with arrows.
- Tree is a graph without cycles. That is When following the graph from node to node, we will never visit the same node twice.
- From the root node we can reach every other node. So, tree is completely connected.
- Hence we can say a tree is a connected acyclic graph.

Representation of a tree using data



S is the root note .

V is the set of non -terminals which are the interior nodes.

T is the set of terminal symbols ,which are the leaf nodes.

**Yield** of the parse tree is abb.

## 9.2 Parsing

Parsing is the process of determining whether a String  $w$  belongs to the language of the Grammar or not. Stated another way, we can say whether the String  $w$  can be derived from the Grammar or not.

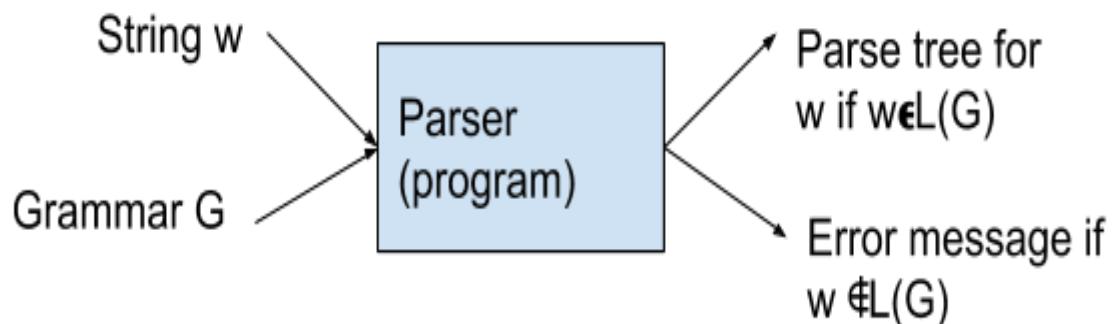
There are two ways in which parsing can be performed:

- a) Top Down Parsing
- b) Bottom up Parsing

### 9.2.1 Top-down Parsing and bottom up Parsing

Top-down Parsing is a parsing technique that first looks at the highest level of the parse tree which is the Start Symbol and works down the parse tree by using the rules of grammar while Bottom-up Parsing is a parsing technique that first looks at the lowest level of the parse tree that means the string  $w$  and works up the parse tree till the Start symbol by using the rules of grammar

**Parser** is a program which takes as input the string  $w$  and grammar  $g$  and produces as output parse tree for  $w$  if  $w$  belongs to lang of the grammar otherwise outputs an error message.



### 9.2.2 Generating a parse tree for a given String $w$ and

## Grammar G.

### Example 1:

Generate parse tree for the sting w=1010 given the grammar .

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

Derivation: w=1010

$S \Rightarrow 1S$  (using  $S \rightarrow 1S$ )

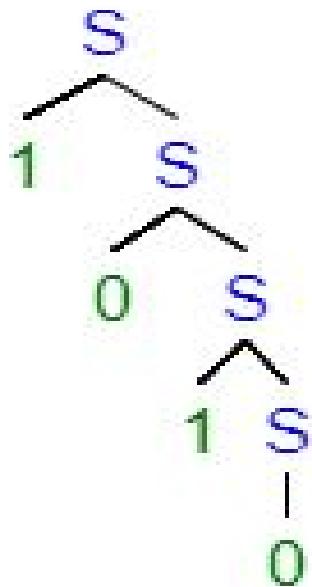
$\Rightarrow 10S$  (using  $S \rightarrow 0S$ )

$\Rightarrow 101S$  (using  $S \rightarrow 1S$ )

$\Rightarrow 1010$  (using  $S \rightarrow 0$ )

Sentence or string =1010

Parse Tree: w=1010



Yield of the tree=1010

### Example 2:

Generate parse tree for the sting  $w=aaabbbb$  given the grammar .

$S \rightarrow aaB$

$B \rightarrow aB | bbbE$

$E \rightarrow bE | \lambda$

Derivation:  $w=aaabbbb$

$S \Rightarrow aaB \quad (\text{using } S \rightarrow aaB)$

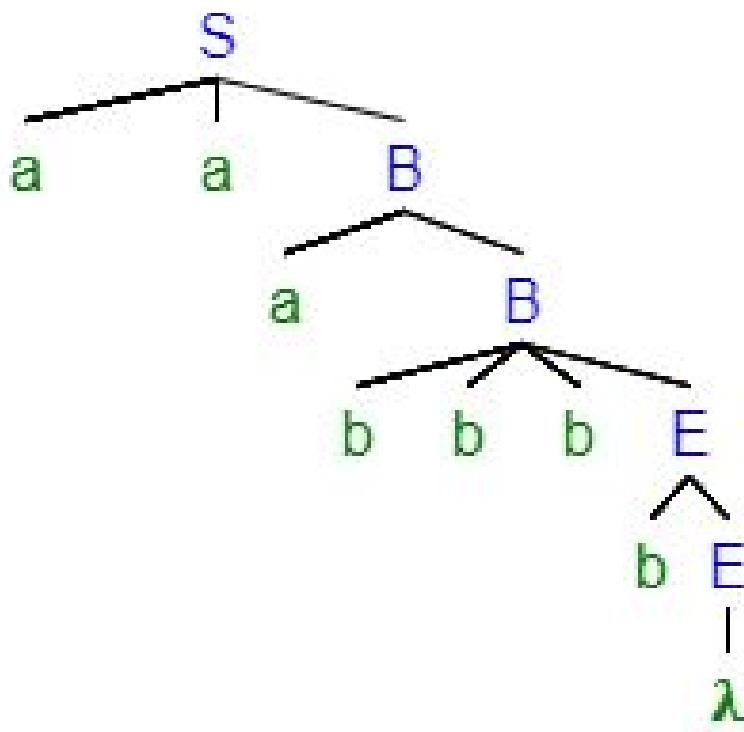
$\Rightarrow aaaB \quad (\text{using } S \rightarrow aB)$

$\Rightarrow aaabbbE \quad (\text{using } B \rightarrow bbbE)$

$\Rightarrow aaabbbbE \quad (\text{using } E \rightarrow bE)$

$\Rightarrow aaabbbb \quad (\text{using } E \rightarrow \lambda)$

Parse Tree:  $w=aaabbbb$



Yield of the tree:aaabbbb

### 9.3 Constructing a left linear Grammar

There is another form in which regular grammars could possibly be written and it is known as the Left linear form. That means the non-terminal on RHS of a production rule must be present at the leftmost side or should be the first symbol on RHS of the production rule.

Let us consider the right linear grammar for the lang where all the strings must start with an 'a'.

Right Linear Grammar

$A \rightarrow aB$

$B \rightarrow aB | bB | \lambda$

$L = \{\text{starting with 'a'}\}$

You might think that reversing all the symbols on RHS of every production rule will get us an equivalent Left linear grammar for the language L.

But surprisingly, the language represented by this grammar is the reverse of the language represented by the Right linear form.

When we reverse all the symbols in RHS of every production rule what we get is a language where every string ends with an a.

So, the conversion or construction of LLG is not as straightforward. We need to work our way via finite automata.

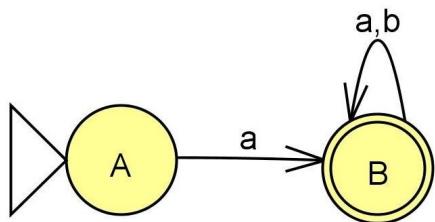
## 9.4 Constructing a left linear grammar from the finite automata.

- Step 1: Consider the Finite automata for language "L".
- Step 2: Rever the finite automata L .
  - Steps to reverse:
    - Change initial to final state.
    - Final state to initial state.
    - Reverse the directions of the transitions.
    - We get  $L^R$ .
- Step 3: Construct a right linear grammar for this language which is reverse of  $L^R$ .
- Step 4: Now Reverse the symbols in RHS of every production rule.
  - Now with this step we have got a left linear grammar for a language which is a reverse of the reversed language, thereby constructing a left linear grammar for L.

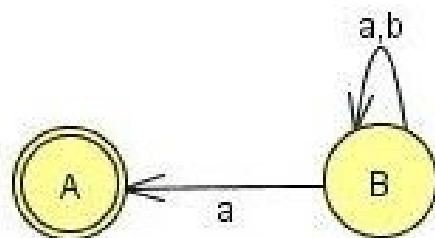
### Example 1:

Constructing a left linear grammar for  $L = \{aw, w \in \{a,b\}^*\}$ .

Step 1: Finite automata for the language where the strings must start with an  $a$ ,  $L = \{aw, w \in \{a,b\}^*\}$



Step 2: Reversing directions in this automata represents automata for the reverse of the language  $L$  which will be a set of strings ending with an  $a$ .



Here, B is the start state.

Step 3: We will now generate the right linear grammar for this reversed language. We very well know how to convert finite automata to regular grammar.

$$\begin{aligned} B &\rightarrow aB | bB | aA \\ A &\rightarrow \lambda \end{aligned}$$

Step 4: In order to get LLG for the language  $L$  is to reverse the symbols in RHS of every production rule of right linear grammar.

$$\begin{aligned} B &\rightarrow Ba | Bb | Aa \\ A &\rightarrow \lambda \end{aligned}$$

## 9.5 Converting to left linear grammar to finite automata

This is exactly a reverse process of conversion from FA to Left linear grammar.

- Step 1: Given the Left linear grammar for Language “L”,reverse the symbols on RHS of each production rule in order to get a right linear grammar for reversal of the language.
- Step 2: Construct finite automata for reverse of the language using right linear grammar.
- Step 3: Reverse this finite automata. This automata will now accept the language L.

### **Example 1:**

**Convert a given left linear grammar to finite automata**

**Left Linear grammar**

$$B \rightarrow Ba|Bb|Aa$$

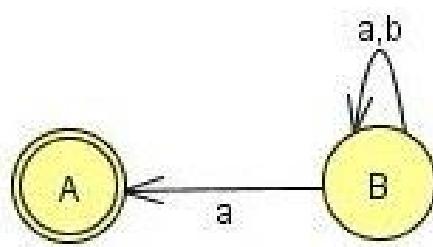
$$A \rightarrow \lambda$$

- Step 1: Reverse the symbols on RHS of each production rule in order to get a right linear grammar for reversal of the language.

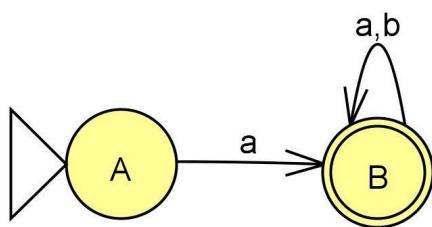
$$B \rightarrow aB|bB|aA$$

$$A \rightarrow \lambda$$

- Step 2: Construct finite automata for reverse of the language using right linear grammar.



- Step 3: Reverse this finite automata. This automata will now accept the language L.



Here ,B is the start state.

## **9.6 Which one is easier left linear or right linear?**

Given the right linear grammar,

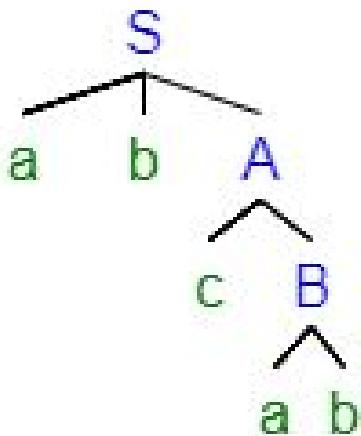
$$S \rightarrow abA$$

$$A \rightarrow cB|aC$$

$$B \rightarrow ab$$

$$C \rightarrow b$$

Parse tree for the string abcab



We start with the Start Symbol S. Since there is only one alternative and the first symbol in our string is also a, we will use the production rule  $S \rightarrow abA$  and expand our parse tree.

So we have successfully got a match for the first two symbols.

The next symbol to be matched is c and we have the non-terminal A.

Therefore we will expand the parse tree using the alternative  $A \rightarrow cB$ .

Next we must match a and the non-terminal to be expanded is B. We use  $B \rightarrow ab$  to expand the tree which will also match the last symbol in our string which is b.

We see the process of derivation of string is easy. We just need to see which input symbol is next and pick the alternative for non-terminal which will match that symbol. It is quite easy as we scan the string left to right and also derive the string left to right.

Consider this Left linear grammar

$$S \rightarrow Aabc$$

$$A \rightarrow Bb|C$$

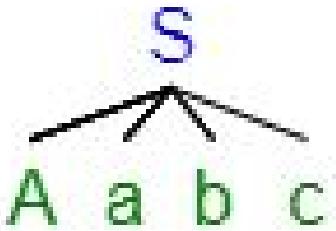
$$B \rightarrow a$$

$$C \rightarrow b$$

Parse tree for the string ababc

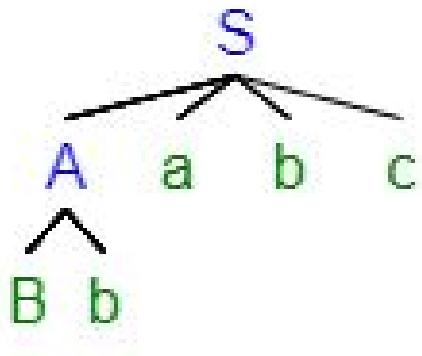
Can the rule  $S \rightarrow Aabc$  recognize the string ababc?

We see that the string is ending with abc and it matches with the last part of the rule.

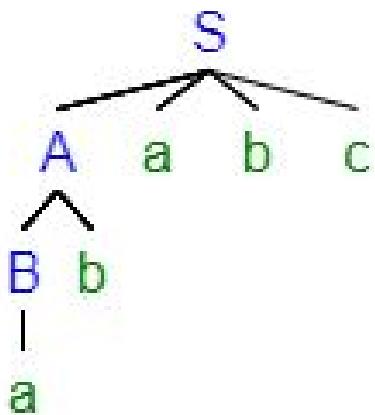


Now, which rule do we choose to  $A \rightarrow Bb$  or  $A \rightarrow C$

We choose  $A \rightarrow Bb$  as we see from the input string that the next symbol to be parsed is b



Then we choose  $B \rightarrow a$  which completes parsing the string.



This way of parsing although possible is not very intuitive as we need to match the symbols in the string backwards.

**Hence right linear form is always preferred over left linear form.**





**Department of Computer Science and Engineering**  
**PES University, Bangalore, India**

## UE23CS243A: Automata Formal Language and Logic

Prakash C O, Associate Professor, Department of CSE

### Regular Grammar (type-3 grammar):

#### Why do we care about regular grammars?

Programs are composed of tokens:

- Identifiers
- Literals (Integer literals, floating point literals, character literals, string literals, ... )
- Keywords
- Operators and
- Special symbols (i.e., Punctuation symbols, ... )

Each of these can be defined by regular grammars.

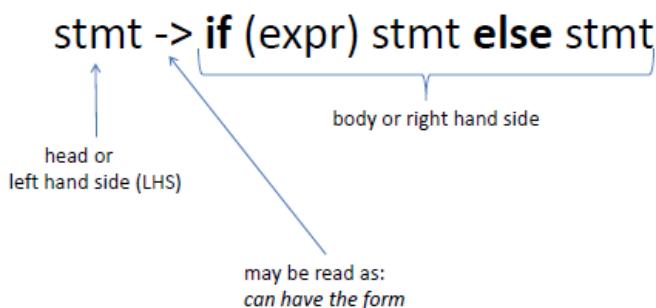
### Formal Definition of a Grammar

Formally, a grammar is a 4-tuple  $G = (V, T, P, S)$

where:

- **V - Set of nonterminals (or variables)**
- **T - Set of terminal symbols**
- **P - Set of productions(or rules)**
  - The head is nonterminal
  - The body is a sequence of teminals and/or nonterminals
- **S – Start Symbol (Designation of one nonterminal as starting symbol)**

#### Example:



➤ Production rules.

$\text{stmt} \rightarrow \text{if } (\text{expr}) \text{ stmt } \text{else } \text{stmt}$

Nonterminals

They need more rules to define them.

$\text{stmt} \rightarrow \text{if } (\text{expr}) \text{ stmt } \text{else } \text{stmt}$

Terminals

No more rules needed for them

**Note:** All the variables in your grammar must be reachable from the start symbol of the grammar and all variables must derive something (or end)

## Derivation:

A derivation in compiler design is the successive application of production rules to produce the desired input string.

- Given the grammar (i.e. productions)
- begin with the start symbol
- repeatedly replacing nonterminal by the body
- We obtain the language string/statement defined by the grammar (i.e. group of terminal strings)

## There are two types of derivation in compiler design:

- 1) Left-most Derivation
- 2) Right-most Derivation

## Left-most Derivation

The left-most derivation is a method of transforming an input string according to the grammar rules of a programming language. The leftmost non-terminal is selected at each stage of left-most derivation.

### Grammar:

$S \rightarrow ABC$   
 $A \rightarrow a$   
 $B \rightarrow b$   
 $C \rightarrow c$

**Input string: abc**

**Left-most derivation:**  $S \Rightarrow ABC \Rightarrow aBC \Rightarrow abC \Rightarrow abc$

## Right-most Derivation

The right-most derivation is a method for transforming an input text depending on the grammar rules of a programming language.

The rightmost non-terminal is selected for expansion at each step, which is regulated by the production rule associated with that non-terminal.

### Grammar:

**S → ABC**

**A → a**

**B → b**

**C → c**

**Input string: abc**

**Right-most derivation: S ⇒ ABC ⇒ ABc ⇒ Abc ⇒ abc**

**Parse Tree:** Parse Tree is the geometrical representation of a derivation.

- Parse tree is the hierarchical representation of terminals or non-terminals.
- These symbols (terminals or non-terminals) represent the derivation of the grammar to yield input strings.
- In parsing, the string springs using the beginning symbol.
- The starting symbol of the grammar must be used as the root of the Parse Tree.
- Leaves of parse tree represent terminals.
- Each interior node represents non-terminals (or productions) of a grammar.

**Example 1:**

**S → sAB**

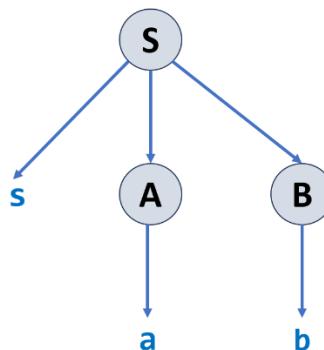
**A → a**

**B → b**

The input string is “**sab**”,

**Derivation: S ⇒ sAB ⇒ saB ⇒ sab**

then the Parse Tree is:



**Example-2:**

**S → AB**

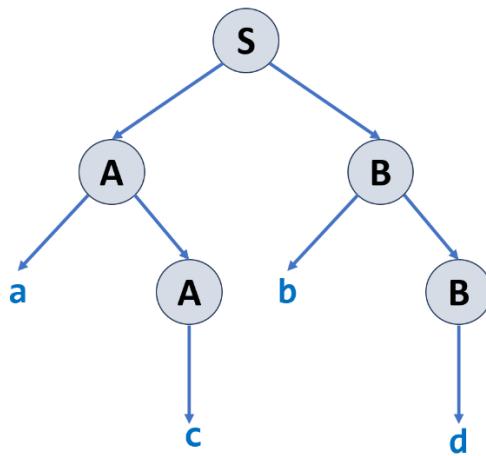
**A → c | aA**

**B → d | bB**

The input string is “**acbd**”,

**Derivation: S ⇒ AB ⇒ aAB ⇒ acB ⇒ acd ⇒ acbd**

then the Parse Tree is as follows:



## Sentential Form:

**A sentential form is any string consisting of non-terminals and/or terminals that is derived from a start symbol.** Therefore, every sentence is a sentential form, but only **sentential forms without non-terminals** are called sentences.

**Example:**      Grammar:  $S \rightarrow aSb \mid \lambda$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabb$

**Sentential Forms**      **sentence**

## We write:

$$S \Rightarrow aaabbb$$

## Instead of:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabb$$

## Linear Grammar

**Grammars with at most one variable (or Nonterminal) at the right hand side of a production.**

### **Example-1:**

$$\begin{array}{l} S \rightarrow aSb \\ S \rightarrow \lambda \end{array}$$

### **Example-2:**

$$\begin{array}{ll} S \rightarrow Ab \\ A \rightarrow aAb \\ A \rightarrow \lambda \end{array}$$

## **Non-Linear Grammar**

## Grammars with more than one variable (or Nonterminal) at the right hand side of a production.

## Example:

**S → SS**

$$\begin{aligned} S &\rightarrow \lambda \\ S &\rightarrow aSb \\ S &\rightarrow bSa \end{aligned}$$

$$L(G) = \{ w \mid n_a(w) = n_b(w) \}$$

## Right-Linear Grammar

- **Right linear grammar:** The non-terminal symbol should be at the right end of the production body.  
All productions have the form:

$$\begin{array}{ll} S \rightarrow wS & \text{or} \\ S \rightarrow w & A \rightarrow w \end{array}$$

where  $S$  and  $A$  are non-terminals in  $V$  and  $w$  is a string of terminals i.e.,  $w \in \Sigma^*$

- **Right linear grammar:** The grammars, in which all rules are of the form  $A \rightarrow w\alpha$  where  $w$  is a string of terminals and  $\alpha$  is either empty or a single nonterminal.

### Examples:

$$\begin{array}{l} 1) \quad S \rightarrow abS \\ \quad \quad S \rightarrow a \end{array}$$

$$\begin{array}{l} 2) \quad S \rightarrow 00B \mid 11S \\ \quad \quad B \rightarrow 0B \mid 1B \mid 0 \mid 1 \end{array}$$

$$\begin{array}{l} 3) \quad S \rightarrow cS \\ \quad \quad S \rightarrow \lambda \end{array}$$

## Left-Linear Grammar

- **Left linear grammar:** The non-terminal symbol should be at the left end of the production body.  
All productions have the form:

$$\begin{array}{ll} S \rightarrow Sw & \text{or} \\ S \rightarrow w & A \rightarrow w \end{array}$$

where  $S$  and  $A$  are non-terminals in  $V$  and  $w$  is a string of terminals i.e.,  $w \in \Sigma^*$

- **Left linear grammar:** The grammars, in which all rules are of the form  $A \rightarrow \alpha w$  where  $\alpha$  is either empty or a single nonterminal and  $w$  is a string of terminals.

### Example:

$$\begin{array}{l} S \rightarrow Aab \\ A \rightarrow Aab \mid B \\ B \rightarrow a \end{array}$$

## Regular Grammar

- A regular language can be described by a special kind of grammar called **regular grammar**.
- A **regular grammar** is a grammar that is **left-linear** or **right-linear**.
- Regular grammars generate **regular languages**.

**Example:** Construct a regular grammar for the language of the regular expression  $(a|b)^* a$

Regular Grammar is:

$$\left. \begin{array}{l} S \rightarrow aS \\ S \rightarrow bS \end{array} \right\} \text{ or } S \rightarrow aS \mid bS \mid a$$

$$S \rightarrow a$$

Does this sentence (or string) **baaba** conform to the written grammar. YES

**Derivation:**  $S \Rightarrow bS \Rightarrow baS \Rightarrow baaS \Rightarrow baabS \Rightarrow baaba$

## Construct a Regular Grammar for the given language description.

**Hint:** Easy way is to construct NFA(or  $\lambda$ -NFA with state names as uppercase letters) and then writing the regular grammar.

|     |   |
|-----|---|
| 1   | $L = \{a\}$   |
| 2   | $L = \{ab\}$  |
| 3   | $L = \{ w \mid w \in \{a, b\}^* \text{ and }  w  = 2 \}$  |
| 4   | $L = \{ w \mid w \in \{a, b\}^* \text{ and }  w  \leq 2 \}$   |
| 5   | $L = \{aaaa\}$  |
| 6   | $L = \{a, b\}$  |
| 7   | $L = \{ab, ba\}$  |
| 8   | $L = \{ \lambda, a, aa, aaa, \dots \}$  |
| 9   | $L = \{ a, aa, aaa, \dots \}$   |
| 10  | $L = \{ \lambda, ab, abab, \dots \}$  |
| 11  | $L = \{ \lambda, a, b, ab, ba, aaa, bbb, bba, bab, abb, aab, aba, baa, \dots \}$  |
| 12  | $L = \{ a, b, ab, ba, aaa, bbb, bba, bab, abb, aab, aba, baa, \dots \}$   |
| 13  | Strings with zero or more a's only and Strings with zero or more b's only.  |
| 14  | Strings with one or more a's only and Strings with one or more b's only.  |
| 15  | Strings begins with zero or more a's followed by single b.  |
| 16  | Strings begins with a and followed by zero or more b's.   |
| 17  | $L = \{ a^{2n} \mid n \geq 0 \}$  |
| 17a | $L = \{ a^{2n} \mid n \geq 1 \}$  |
| 18  | $L = \{ a^{2n+1} \mid n \geq 0 \}$  |
| 19  | $L = \{ a^{4n} \mid n \geq 0 \}$  |
| 20  | $L = \{ w_1aw_2 \mid w_1 \in \{b\}^* \text{ and } w_2 \in \{a, b\}^* \}$  |
| 21  | $L = \{ a^m b^n \mid m \geq 0 \text{ and } n > 0 \}$  |
| 22  | $L = \{ a^m b^n \mid m > 0 \text{ and } n \geq 0 \}$  |
| 23  | $L = \{ a^m b^n \mid m > 0 \text{ and } n > 0 \}$   |
| 24  | $L = \{ a^m b^n \mid m \geq 0 \text{ and } n \geq 0 \}$<br>$L = \{ \lambda, a, aa, aaa, \dots, b, bb, bbb, bbbb, \dots, ab, aab, abb, abbb, aabb, aaab, \dots \}$ |
| 25  | 0 or more a's, followed by 0 or more b's, followed by 0 or more c's. $L = \{\epsilon, a, b, c, aa, ab, ac, bb, bc, cc, aaa, \dots\}$                              |
| 26  | $L = \{ a^m b^n \mid m > 0 \text{ or } n > 0 \}$  |
| 27  | Strings ending with abb. $L = \{abb, aabb, babb, aaabb, ababb, baabb, bbabb, \dots\}$   |
| 28  | Strings starting with ab. $L = \{ab, aba, abb, abaa, abab, abba, abbb, \dots\}$   |

|    |  |
|----|--|
| 29 | Strings that contains aa. L = {aa, aaa, baa, aab, ...}   |
| 30 | 1 or more a's, followed by 1 or more b's, followed by 1 or more c's. L = {abc, aabc, abbc, abcc, aabbc, aabcc, abbcc, ...} |
| 31 | Strings that end with a or bb. L = {a, bb, aa, abb, ba, bbb, ...}  |
| 32 | Strings with even number of a's followed by odd number of b's. L = {b, aab, bbb, aabbb, ...}                               |
| 33 | Binary strings ending with 3 0's. L = {000, 0000, 1000, 00000, 01000, 10000, 11000, ...}                                   |
| 34 | Strings with even number of 1's. L = {ε, 11, 1111, 111111, ...}  |

## Solutions:

| Sl. No. | Language   | Regex                                 | Regular Grammar   |
|---------|--|---------------------------------------|---|
| 1       | L = {a}  | a                                     | $S \rightarrow a$   |
| 2       | L = {ab}   | ab                                    | $S \rightarrow ab$  |
| 3       | L = { w   w ∈ {a, b}* and  w  = 2 }  | (a b)(a b)<br>or<br>[ab][ab]          | $S \rightarrow aA \mid bA$<br>$A \rightarrow a \mid b$  |
| 4       | L = { w   w ∈ {a, b}* and  w  ≤ 2 }  | (a b)? (a b)?<br>or<br>(ε a b)(ε a b) | $S \rightarrow aA \mid bA \mid \lambda$<br>$A \rightarrow a \mid b \mid \lambda$  |
| 5       | L = {aaaa}   | aaaa                                  | $S \rightarrow aaaa$  |
| 6       | L = {a, b}   | a b                                   | $S \rightarrow a \mid b$  |
| 7       | L = {ab, ba}   | ab ba                                 | $S \rightarrow ab \mid ba$  |
| 8       | L = { λ, a, aa, aaa, ... }   | a*                                    | $S \rightarrow aS \mid \lambda$   |
| 9       | L = { a, aa, aaa, ... }  | a <sup>+</sup>                        | $S \rightarrow aS \mid a$   |
| 10      | L = { λ, ab, abab, ... }   | (ab)*                                 | $S \rightarrow abS \mid \lambda$  |
| 11      | L = { λ, a, b, ab, ba, aaa, bbb, bba, bab, abb, aab, aba, baa, ... }       | (a b)*                                | $S \rightarrow aS \mid bS \mid \lambda$   |
| 12      | L = { a, b, ab, ba, aaa, bbb, bba, bab, abb, aab, aba, baa, ... }          | (a b)+                                | $S \rightarrow aS \mid bS \mid a \mid b$  |
| 13      | Strings with zero or more a's only and Strings with zero or more b's only. | a* b*                                 | $S \rightarrow A \mid B$<br>$A \rightarrow aA \mid \lambda$<br>$B \rightarrow bB \mid \lambda$<br>or<br>$S \rightarrow aA \mid bB \mid \lambda$<br>$A \rightarrow aA \mid \lambda$<br>$B \rightarrow bB \mid \lambda$ |
| 14      | Strings with one or more a's only and Strings                              | a <sup>+</sup>  b <sup>+</sup>        | $S \rightarrow A \mid B$  |

|     |   |                                    |   |
|-----|---|------------------------------------|---|
|     | <b>with one or more b's only.</b>   | <b>or</b><br>$aa^* bb^*$           | $A \rightarrow aA \mid a$<br>$B \rightarrow bB \mid b$<br><b>or</b><br>$S \rightarrow aA \mid bB$<br>$A \rightarrow aA \mid \lambda$<br>$B \rightarrow bB \mid \lambda$ |
| 15  | <b>Strings begins with zero or more a's and ends with single b.</b>                             | $a^*b$                             | $S \rightarrow aS \mid b$<br><b>or</b><br>$S \rightarrow Ab$<br>$A \rightarrow Aa \mid \lambda$   |
| 16  | <b>Strings begins with a and followed by zero or more b's.</b>                                  | $ab^*$                             | $S \rightarrow aB$<br>$B \rightarrow bB \mid \lambda$<br><b>or</b><br>$S \rightarrow a \mid Sb$   |
| 17  | $L=\{ a^{2n} \mid n \geq 0 \}$  | $(aa)^*$                           | $S \rightarrow aA \mid \lambda$<br>$A \rightarrow aS$<br><b>or</b><br>$S \rightarrow aaS \mid \lambda$  |
| 17a | $L=\{ a^{2n} \mid n \geq 1 \}$  | $(aa)^+$                           | $S \rightarrow aA$<br>$A \rightarrow aS \mid a$<br><b>or</b><br>$S \rightarrow aaS \mid aa$   |
| 18  | $L=\{ a^{2n+1} \mid n \geq 0 \}$  | $(aa)^*a$                          | $S \rightarrow aaS \mid a$<br><b>or</b><br>$S \rightarrow aA \mid a$<br>$A \rightarrow aS$<br><b>or</b><br>$S \rightarrow aA$<br>$A \rightarrow aS \mid \lambda$        |
| 19  | $L=\{ a^{4n} \mid n \geq 0 \}$  | $(aaaa)^*$                         | $S \rightarrow aaaaS \mid \lambda$<br><b>or</b><br>$S \rightarrow aA \mid \lambda$<br>$A \rightarrow aB$<br>$B \rightarrow aA$<br>$A \rightarrow aS$                    |
| 20  | $L = \{ w_1aw_2 \mid w_1 \in \{b\}^* \text{ and } w_2 \in \{a, b\}^* \}$                        | $b^*a(a b)^*$                      | $S \rightarrow bS \mid aA \mid a$<br>$A \rightarrow aA \mid bA \mid \lambda$  |
| 21  | $L=\{ a^m b^n \mid m \geq 0 \text{ and } n > 0 \}$<br>$L = \{ b, bb, ab, aab, abb, bbb, ... \}$ | $a^*bb^*$<br><b>or</b><br>$a^*b^+$ | $S \rightarrow aS \mid b \mid bB$<br>$B \rightarrow bB \mid \lambda$  |
| 22  | $L=\{ a^m b^n \mid m > 0 \text{ and } n \geq 0 \}$<br>$L = \{ a, aa, ab, aab, abb, aaa, ... \}$ | $aa^*b^*$<br><b>or</b><br>$a^*b^*$ | $S \rightarrow aA$<br>$A \rightarrow aA \mid \lambda \mid bB$<br>$B \rightarrow bB \mid \lambda$  |

|    |   |   |  |
|----|---|---|--|
| 23 | $L = \{ a^m b^n \mid m > 0 \text{ and } n > 0 \}$   | $a a^* b b^*$<br>or<br>$a^+ b^+$                                  | $S \rightarrow aA$<br>$A \rightarrow aA \mid bB$<br>$B \rightarrow \lambda \mid bB$  |
| 24 | $L = \{ a^m b^n \mid m \geq 0 \text{ and } n \geq 0 \}$<br>$L = \{ \lambda, a, aa, aaa, \dots, b, bb, bbb, bbbb, \dots, ab, aab, abb, abbb, aabb, aaab, \dots \}$ | $a^* b^*$   | $S \rightarrow aS \mid bB \mid \lambda$<br>$B \rightarrow bB \mid \lambda$   |
| 25 | 0 or more a's, followed by 0 or more b's, followed by 0 or more c's. $L = \{\epsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, \dots\}$                  | $a^* b^* c^*$   | $S \rightarrow aS \mid bB \mid cC \mid \epsilon$<br>$B \rightarrow bB \mid cC \mid \epsilon$<br>$C \rightarrow cC \mid \epsilon$                                 |
| 26 | $L = \{ a^m b^n \mid m > 0 \text{ or } n > 0 \}$  | $aa^* b^* \mid a^* b^* b$<br>or<br>$a^* b^* \mid a^* b^+$         | $S \rightarrow aS \mid aA \mid bA$<br>$A \rightarrow bA \mid \lambda$  |
| 27 | Strings ending with abb. $L = \{abb, aabb, babb, aaabb, ababb, baabb, bbabb, \dots\}$   | $(a b)^* abb$   | $S \rightarrow aS \mid bS \mid aB$<br>$B \rightarrow bA$<br>$A \rightarrow b$  |
| 28 | Strings starting with ab. $L = \{ab, aba, abb, abaa, abab, abba, abbb, \dots\}$   | $ab(a b)^*$   | $S \rightarrow aB$<br>$B \rightarrow bA$<br>$A \rightarrow aA \mid bA \mid \epsilon$   |
| 29 | Strings that contains aa. $L = \{aa, aaa, baa, aab, \dots\}$  | $(a b)^* aa(a b)^*$   | $S \rightarrow aS \mid bS \mid aA$<br>$A \rightarrow aB$<br>$B \rightarrow aB \mid bB \mid \epsilon$   |
| 30 | 1 or more a's, followed by 1 or more b's, followed by 1 or more c's. $L = \{abc, aabc, abbc, abcc, aabbc, aabcc, abbcc, \dots\}$                                  | $a^+ b^+ c^+$<br>or<br>$aa^* bb^* cc^*$                           | $S \rightarrow aA$<br>$A \rightarrow aA \mid bB$<br>$B \rightarrow bB \mid cC$<br>$C \rightarrow cC \mid \epsilon$   |
| 31 | Strings that end with a or bb. $L = \{a, bb, aa, abb, ba, bbb, \dots\}$   | $(a b)^* (a bb)$  | $S \rightarrow aS \mid bS \mid a \mid bB$<br>$B \rightarrow b$   |
| 32 | Strings with even number of a's followed by odd number of b's. $L = \{b, aab, bbb, aabbb, \dots\}$  | $(aa)^* (bb)^* b$   | $S \rightarrow aA \mid bB \mid b$<br>$A \rightarrow aC$<br>$C \rightarrow aA \mid bB \mid \epsilon$<br>$B \rightarrow bD \mid \epsilon$<br>$D \rightarrow bB$    |
| 33 | Binary strings ending with 3 0's. $L = \{000, 0000, 1000, 00000, 01000, 10000, 11000, \dots\}$  | $(0+1)^* 000$   | $S \rightarrow 0S \mid 1S \mid 0A$<br>$A \rightarrow 0B$<br>$B \rightarrow 0$  |
| 34 | Strings with even number of 1's. $L = \{\epsilon, 11, 1111, 111111, \dots\}$  | $(11)^*$  | $S \rightarrow 1A \mid \epsilon$<br>$A \rightarrow 1S$   |
| 35 | Strings with atmost 3 a's and $\Sigma \in \{a, b\}$   | $b^* a? b^* \mid$<br>$b^* ab^* ab^* \mid$<br>$b^* ab^* ab^* ab^*$ | $S \rightarrow bS \mid aA \mid \lambda$<br>$A \rightarrow bA \mid aB \mid \lambda$<br>$B \rightarrow bB \mid aC \mid \lambda$<br>$C \rightarrow bC \mid \lambda$ |

**Note:**

Every regular language can be specified by a finite state automaton, a regular expression, or a regular grammar. Furthermore, all of these specifications are equivalent in the sense that they all capture the class of regular languages.

|                               |                   |   |
|-------------------------------|-------------------|---|
| <b>Finite State Automaton</b> | <b>Recognizer</b> | Determines whether a particular input string in the regular language is recognized by the FSA or not. |
| <b>Regular Expression</b>     | <b>Expresser</b>  | Expresses a regular language as a pattern to which every string in the regular language conforms.     |
| <b>Regular Grammar</b>        | <b>Generator</b>  | Provides grammar rules for generating strings in the regular language                                 |

**Note:** Regular grammar is a formal specification of a regular language by way of grammar rules.

## Exercises:

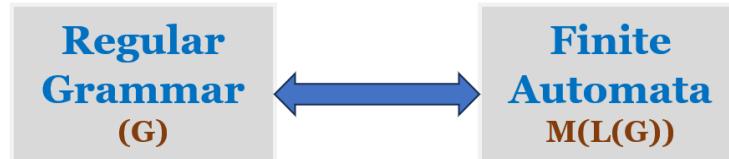
**Construct a Regular Grammar for the given language description and Regular expression.**

1. Strings of a's and b's of length 2.      **Regex = aa | ab | ba | bb    OR    (a|b)(a|b)**
2. Strings of a's and b's of length  $\leq 2$ .      **Regex =  $\epsilon$ |a|b|aa|ab|ba|bb    OR    ( $\epsilon$  | a | b)( $\epsilon$  | a | b)  
OR    (a|b)? (a|b)?**
3. Strings of a's and b's of length  $\leq 5$ .      **Regex = ( $\epsilon$  | a | b)<sup>5</sup>**
4. Even-lengthed strings of a's and b's.      **Regex = (aa | ab | ba | bb)\*    OR    ((a|b)(a|b))\***
5. Odd-lengthed strings of a's and b's.      **Regex = (a|b) ((a|b)(a|b))\***
6.  $L(R) = \{ w : w \in \{0,1\}^* \text{ with at least three consecutive } 0\text{'s } \}$       **Regex = (0|1)\* 000 (0|1)\***
7. Strings of 0's and 1's with no two consecutive 0's.      **Regex = (1 | 01)\* (0 |  $\epsilon$ )**
8. Strings of a's and b's starting with a and ending with b.      **Regex = a (a|b)\* b**
9. Strings of a's and b's whose second last symbol is a.      **Regex = (a|b)\* a (a|b)**
10. Strings of a's and b's whose third last symbol is a and fourth last symbol is b.  
  
**Regex = (a|b)\* b a (a|b) (a|b)**
11. Strings of a's and b's whose first and last symbols are the same. **Regex = (a (a|b)\* a) | (b (a|b)\* a)**
12. Strings of a's and b's whose first and last symbols are different. **Regex = (a (a|b)\* b) | (b (a|b)\* a)**
13. Strings of a's and b's whose last and second last symbols are same. **Regex = (a|b)\* (aa | bb)**
14. Strings of a's and b's whose length is even or a multiple of 3 or both.  
  
**Regex = R1 | R2      where R1 = ((a|b)(a|b))\*    and    R2 = ((a|b)(a|b)(a|b))\***
15. Strings of a's and b's such that every block of 4 consecutive symbols has at least 2 a's.  
  
**Regex = (aaxx | axax | axxa | xaax | xaxa | xxaa)\* where x = (a|b)**
16.  $L = \{a^n b^m : n \geq 0, m \geq 0\}$       **Regex = a\* b\***

|   |   |
|---|---|
| 17. $L = \{a^n b^m : n > 0, m > 0\}$                              | Regex = $aa^* bb^*$ OR $a^+ b^+$  |
| 18. $L = \{a^n b^m : n \mid m \text{ is even}\}$                  | Regex = $aa^* bb^* \mid a(aa)^* b(bb)^*$  |
| 19. $L = \{a^{2n} b^{2m} : n \geq 0, m \geq 0\}$                  | Regex = $(aa)^* (bb)^*$   |
| 20. Strings of a's and b's containing not more than three a's.    | Regex = $b^* (\epsilon \mid a) b^* (\epsilon \mid a) b^* (\epsilon \mid a) b^*$ |
| 21. $L = \{a^n b^m : n \geq 3, m \leq 3\}$                        | Regex = $aaa a^* (\epsilon \mid b) (\epsilon \mid b) (\epsilon \mid b)$         |
| 22. $L = \{w :  w  \bmod 3 = 0 \text{ and } w \in \{a,b\}^*\}$    | Regex = $((a b)(a b)(a b))^*$   |
| 23. $L = \{w : n_a(w) \bmod 3 = 0 \text{ and } w \in \{a,b\}^*\}$ | Regex = $b^* a b^* a b^* a b^*$   |
| 24. Strings of 0's and 1's that do not end with 01.               | Regex = $(0 1)^* (00 \mid 10 \mid 11)$  |
| 25. $L = \{vuv : u, v \in \{a,b\}^* \text{ and }  v  = 2\}$       | Regex = $(aa \mid ab \mid ba \mid bb) (a b)^* (aa \mid ab \mid ba \mid bb)$     |
| 26. Strings of a's and b's that end with ab or ba.                | Regex = $(a b)^* (ab \mid ba)$  |
| 27. $L = \{a^n b^m : m, n \geq 1 \text{ and } mn \geq 3\}$        | Regex = $a bbb b^* \mid aaa a^* b \mid aa a^* bb b^*$                           |

## Equivalence of Regular Grammar and Finite Automata

The relationship of regular grammar and finite automata is shown below:



If G is a regular grammar, then  $L(G)$  is a regular language accepted by Finite automata.

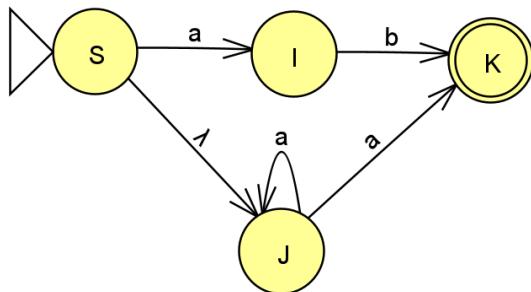
## Converting Finite Automata to Regular Grammars

### Algorithm: Finite Automata to Regular Grammar

Perform the following steps to construct a regular grammar that generates the language of a given Finite automata:

1. Rename the states to a set of uppercase letters (if the state names are named with  $q_0, q_1, q_2 \dots$  or  $1, 2, 3, \dots$ )
2. The start symbol of the grammar is the Finite Machines start state.
3. For each state transition from I to J labelled with a, create the production  $I \rightarrow aJ$
4. For each state transition from I to J labelled with  $\lambda$ , create the production  $I \rightarrow J$
5. For each final state K, create a null production  $K \rightarrow \lambda$

**Example:** Convert the following finite automata to regular grammar.

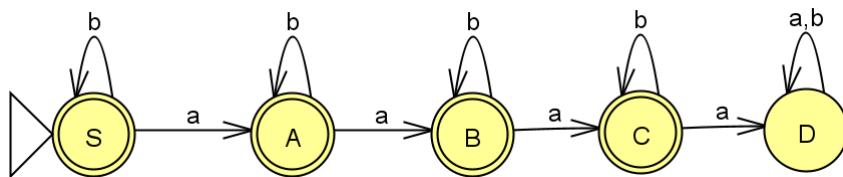


## Regular Grammar:

- o  $S \rightarrow aI$
- o  $S \rightarrow J$
- o  $I \rightarrow bK$
- o  $J \rightarrow aJ$
- o  $J \rightarrow aK$
- o  $K \rightarrow \lambda$

## Exercise:

1. Convert the following finite automata to regular grammar for the language to accept at most 3 'a's over the alphabet  $\Sigma = \{a, b\}$ .



## Solution:

$$S \rightarrow bS \mid aA \mid \lambda$$

$$A \rightarrow bA \mid aB \mid \lambda$$

$$B \rightarrow bB \mid aC \mid \lambda$$

$$C \rightarrow bC \mid aD \mid \lambda$$

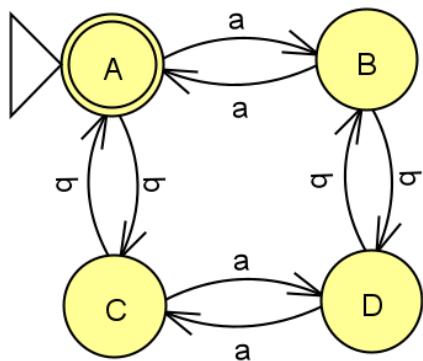
State D is a trap (or dead) state and its rules are

$$D \rightarrow aD \mid bD \quad (\text{Not used for string derivation, it's an useless production})$$

Note: The start symbol of the grammar is S, because the start state is S.

2. Convert the following finite automata to regular grammar.

Where  $L = \{ n_a(w) \bmod 2 = 0 \text{ and } n_b(w) \bmod 2 = 0 \}$



## Solution:

$$A \rightarrow aB \mid bC \mid \lambda$$

$$B \rightarrow aA \mid bD$$

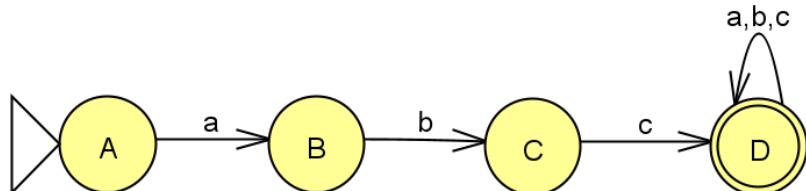
$$C \rightarrow aD \mid bA$$

**D → aC | bB**

**Note:** The start symbol of the grammar is A, because the start state is A.

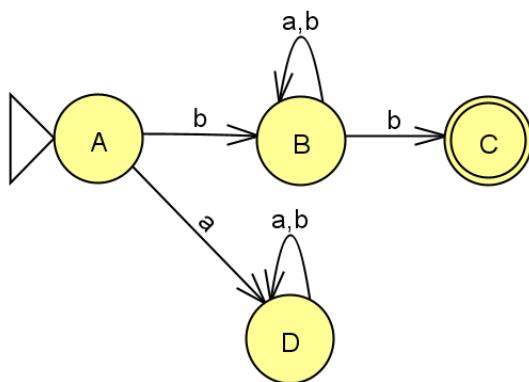
**3. Convert the following finite automata to regular grammar.**

Where L = { abcw | where w $\in\{a,b,c\}^*$  }



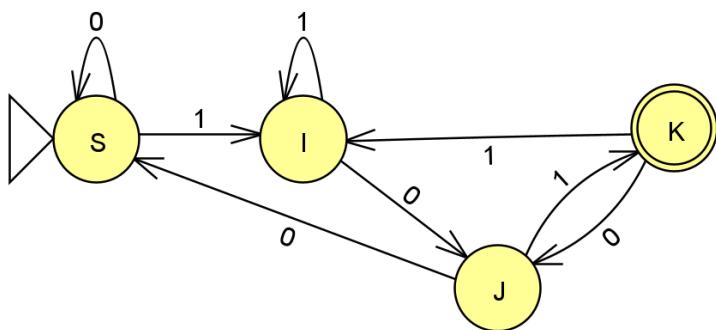
**4. Convert the following finite automata to regular grammar.**

Where L = { bw<sub>b</sub> | where w $\in\{a,b\}^*$  }



**5. Convert the following finite automata to regular grammar.**

Where language contains binary strings ends with 101.



## Converting Regular Grammars to Finite Automata

### Algorithm-01: Regular Grammar to Finite Automata

Perform the following steps to construct an Automata that accepts the language of a given regular grammar:

1. If necessary, transform the grammar so that all productions have the form  $A \rightarrow x$  or  $A \rightarrow xB$ , where  $x$  is either a single letter (i.e., terminal symbol) or  $\lambda$ .
2. The start state of the Automata is the grammar's start symbol.
3. For each production  $I \rightarrow aJ$ , construct a state transition from  $I$  to  $J$  labelled with the letter  $a$ .
4. For each production  $I \rightarrow J$ , construct a state transition from  $I$  to  $J$  labelled with  $\lambda$ .
5. If there are productions of the form  $I \rightarrow a$  for some letter  $a$ , then create a single new state symbol  $F$ .  
For each production  $I \rightarrow a$ , construct a state transition from  $I$  to  $F$  labelled with  $a$ .
6. If there is a production of the form  $I \rightarrow \lambda$  or  $I \rightarrow \epsilon$  (i.e., Null production), then state  $I$  is a final state.

### Examples:

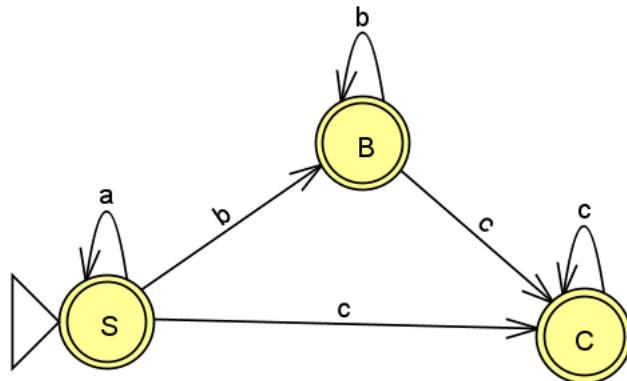
1) Convert the following regular grammar to finite automata.

$$S \rightarrow aS \mid bB \mid cC \mid \epsilon$$

$$B \rightarrow bB \mid cC \mid \epsilon$$

$$C \rightarrow cC \mid \epsilon$$

### Solution:



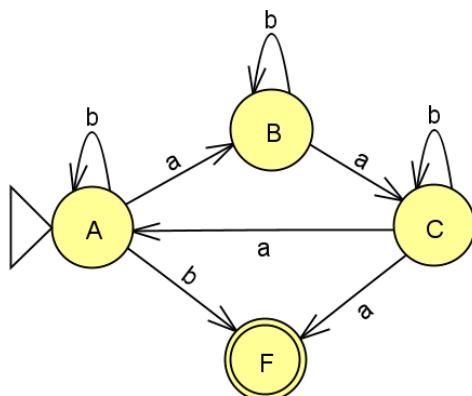
2) Convert the following regular grammar to finite automata.

$$A \rightarrow aB \mid bA \mid b$$

$$B \rightarrow aC \mid bB$$

$$C \rightarrow aA \mid bC \mid a$$

### Solution:



3) Convert the following regular grammar to finite automata.

$$S \rightarrow bS \mid aA$$

$$A \rightarrow aA \mid aB \mid bA$$

$$B \rightarrow bbB$$

$$B \rightarrow \lambda$$

**Solution:**

Transform the given grammar so that all productions have the form  $A \rightarrow x$  or  $A \rightarrow xB$ , where  $x$  is either a single letter (i.e., terminal symbol) or  $\lambda$

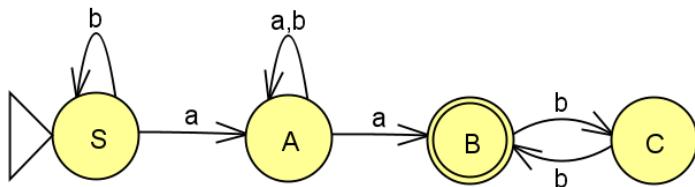
$$S \rightarrow bS \mid aA$$

$$A \rightarrow aA \mid aB \mid bA$$

$$B \rightarrow bC$$

$$C \rightarrow bB$$

$$B \rightarrow \lambda$$



## Algorithm-02: Regular Grammar to Automata

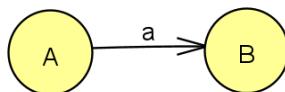
Assume that a regular grammar is given in its right-linear form, this grammar may be easily converted to a Automata. A right-linear grammar, defined by  $G = (V, T, P, S)$ , may be converted to a automata, defined by  $M = (Q, \Sigma, \delta, q_0, F)$  by:

1. Create a state for each variable.

2. Convert each production rule into a transition.

a) If the production rule is of the form  $V_i \rightarrow aV_j$ , where  $a \in T$ , add the transition  $\delta(V_i, a) = V_j$  to automata M.

i) For example,  $A \rightarrow aB$  becomes:

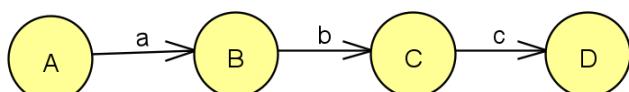


ii) For example,  $A \rightarrow aA$  becomes:



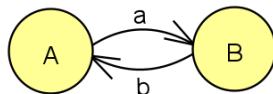
b) If the production rule is of the form  $V_i \rightarrow wV_j$ , where  $w \in T^*$ , create a series of states which derive w and end in  $V_j$ . Add the states in between to set Q.

i) For example,  $A \rightarrow abcD$  becomes:



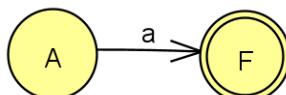
**Note: For intermediate states, give new names (i.e., names that are not there in the Non-terminals list V of the given grammar).**

ii) For example,  $A \rightarrow abA$  becomes:

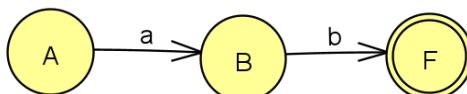


c) If the production rule is of the form  $V_i \rightarrow w$ , where  $w \in T^*$ , create a series of states which derive  $w$  and end in a final state.

i) For example,  $A \rightarrow a$  becomes:



ii) For example,  $A \rightarrow ab$  becomes:



d) If the production rule is of the form  $V_i \rightarrow \lambda$  or  $V_i \rightarrow \epsilon$  (i.e., Null production), then **state  $V_i$  is a final state**.

### Examples:

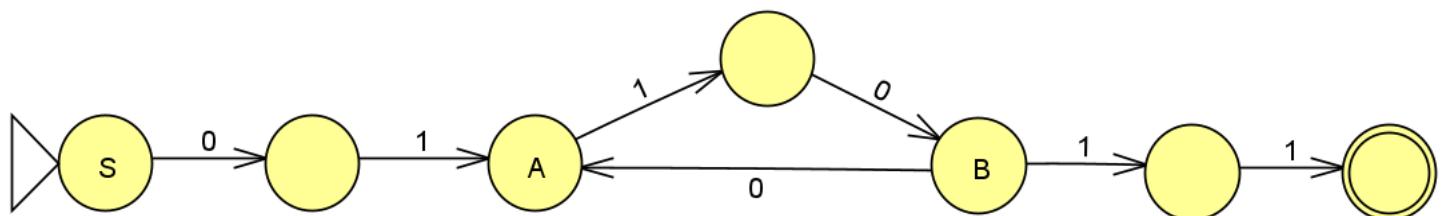
1) Convert the following regular grammar to Automata.

$$S \rightarrow 01A$$

$$A \rightarrow 10B$$

$$B \rightarrow 0A \mid 11$$

### Solution:



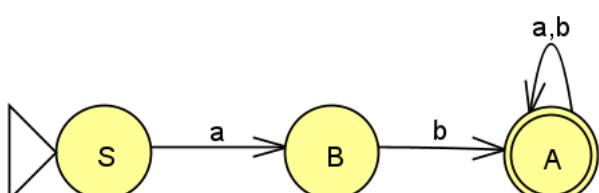
2) Convert the following regular grammar to Automata.

$$S \rightarrow aB$$

$$B \rightarrow bA$$

$$A \rightarrow aA \mid bA \mid \epsilon$$

### Solution:



# Reverse of a Regular Language

If  $L$  is a regular language, then  $L^R$  is also regular

- Let  $L$  be a regular language, then there exists an automata  $M$  such that  $L = L(M)$ .  
 $M = (Q, \Sigma, \delta, q_0, F)$  (where machine  $M$  has a single final state)
- Construct a new machine  $M^R$  (i.e.,  $L^R=L(M^R)$ ) by toggling initial and final state and by reversing the arrows (i.e., swapping initial and final state and changing the directions of the edges).
- The reverse of a regular language i.e.,  $L^R$  is the language accepted by automata  $M^R$ .

## Converting Right Linear Grammar to Left Linear Grammar

If  $G$  is a Right Linear Grammar, then there is a Left Linear Grammar  $G''$  such that  $L(G) = L(G'')$

**Conversion outline:**

- From  $G$ , construct  $M$
- From  $M$  construct  $M^R$  for  $L^R$
- Generate  $G'$  a Right Linear Grammar for  $L^R$
- Generate  $G''$  a Left Linear Grammar for  $(L^R)^R=L$  (i.e., getting  $G''$  from  $G'$  by reversing all symbols on RHS of productions)

**Example:**

1) Convert the following Right Linear Grammar to Left Linear Grammar

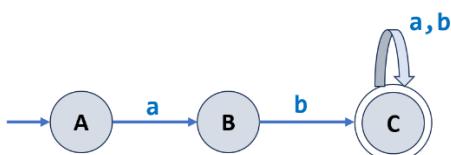
The Right Linear Grammar  $G$  is:

$$\begin{aligned}A &\rightarrow aB \\B &\rightarrow bC \\C &\rightarrow aC \mid bC \mid \epsilon\end{aligned}$$

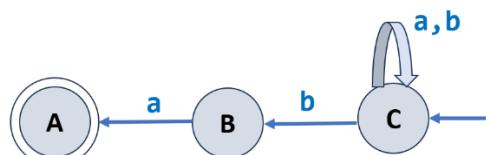
Where  $L(G) = \{ abw \mid w \in \{a,b\}^*\}$

**Solution:**

a) From  $G$ , construct automata  $M$



b) From  $M$  construct  $M^R$  for  $L^R$



c) Generate  $G'$  a Right Linear Grammar from  $M^R$  for  $L^R$

$$C \rightarrow aC \mid bC \mid bB$$

$$B \rightarrow aA$$

$$A \rightarrow \epsilon$$

d) Generate  $G''$  a Left Linear Grammar for  $(L^R)^R = L$  (i.e., getting  $G''$  from  $G'$  by reversing all symbols on RHS of productions)

$$\begin{aligned} C &\rightarrow Ca \mid Cb \mid Bb \\ B &\rightarrow Aa \\ A &\rightarrow \epsilon \end{aligned}$$

**Note:**  $L(G) = L(G'')$

## Converting Left Linear Grammar to Right Linear Grammar

If  $G$  is a Left Linear Grammar, then there is a Right Linear Grammar  $G''$  such that  $L(G) = L(G'')$

### Conversion outline:

- From  $G$ , Generate  $G'$  a Right Linear Grammar for  $L^R$  (i.e., getting  $G'$  from  $G$  by reversing all symbols on RHS of productions)
- From  $G'$  construct  $M$  for  $L^R$
- Construct  $M^R$  from  $M$  for  $(L^R)^R$
- Generate  $G''$  a right Linear Grammar for  $(L^R)^R = L$

### Example:

1) Convert the following Left Linear Grammar to Right Linear Grammar

**Left Linear Grammar  $G$  is:**

$$\begin{aligned} C &\rightarrow Ca \mid Cb \mid Bb \\ B &\rightarrow Aa \\ A &\rightarrow \epsilon \end{aligned}$$

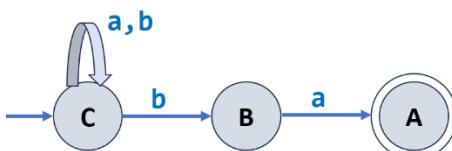
Where  $L(G) = \{ abw \mid w \in \{a,b\}^*\}$

### Solution:

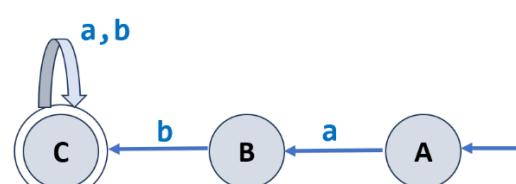
a) From  $G$ , Generate  $G'$  a Right Linear Grammar for  $L^R$  (i.e., getting  $G'$  from  $G$  by reversing all symbols on RHS of productions)

$$\begin{aligned} C &\rightarrow aC \mid bC \mid bB \\ B &\rightarrow aA \\ A &\rightarrow \epsilon \end{aligned}$$

b) From  $G'$  construct  $M$  for  $L^R$



c) Construct  $M^R$  from  $M$  for  $(L^R)^R$



d) Generate  $G'$  a right Linear Grammar for  $(L^R)^R = L$

$A \rightarrow aB$

$B \rightarrow bC$

$C \rightarrow aC \mid bC \mid \epsilon$

-----\*\*\*-----



# Automata Formal Languages & Logic

---

**Preet Kanwal**

Department of Computer Science & Engineering

# Automata Formal Languages & Logic

---

## Unit 2

**Preet Kanwal**

Department of Computer Science & Engineering

# Regular Grammar

- **Formal Definition of a Grammar**
- **Terminology : Sentence , Sentential Form**
- **Examples on how to construct a Regular Grammar for a given L**

### Grammar

V - Variables

expr, var, S, A, B..., w,x,y

T - Terminals

a, b, c ..., 0, 1, ... id, num

$\alpha$  or  $\beta = (V \cup T)^*$

$\alpha \rightarrow \beta$

P - Production Rules

$L(G) =$

$\{w \mid S \Rightarrow^* w, \text{ where } w \in \Sigma^*\}$

S - Start Symbol

### Sentence (or a String)

A sentence is made up only of terminals.

Example:

For the Grammar  $S \rightarrow aS \mid a \mid \lambda$

Sentence (or string) is { a or  $\lambda$  or aa or aaaa}

### Sentential Form

A sentential form is made up over set of terminals and non-terminals (VUT)\*

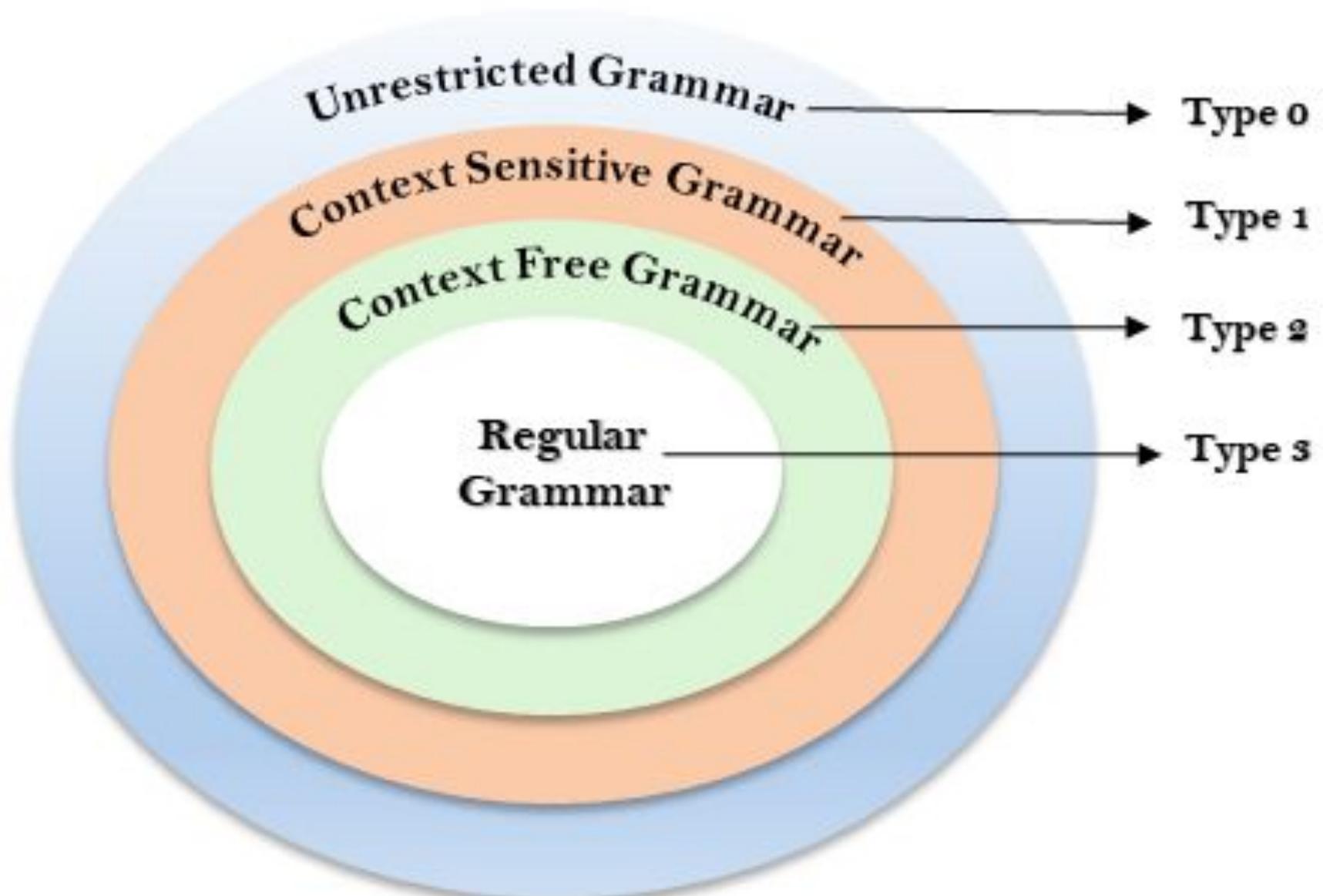
Example:

For the Grammar  $S \rightarrow aS \mid a \mid \lambda$

Sentential form is { a or  $\lambda$  or  $aS$  or aa or or  $aS$  aaaa)

# Automata Formal Languages and Logic

## Unit 2 - Regular Grammar



**Noam Chomsky**

## Chomsky Hierarchy

# Automata Formal Languages and Logic

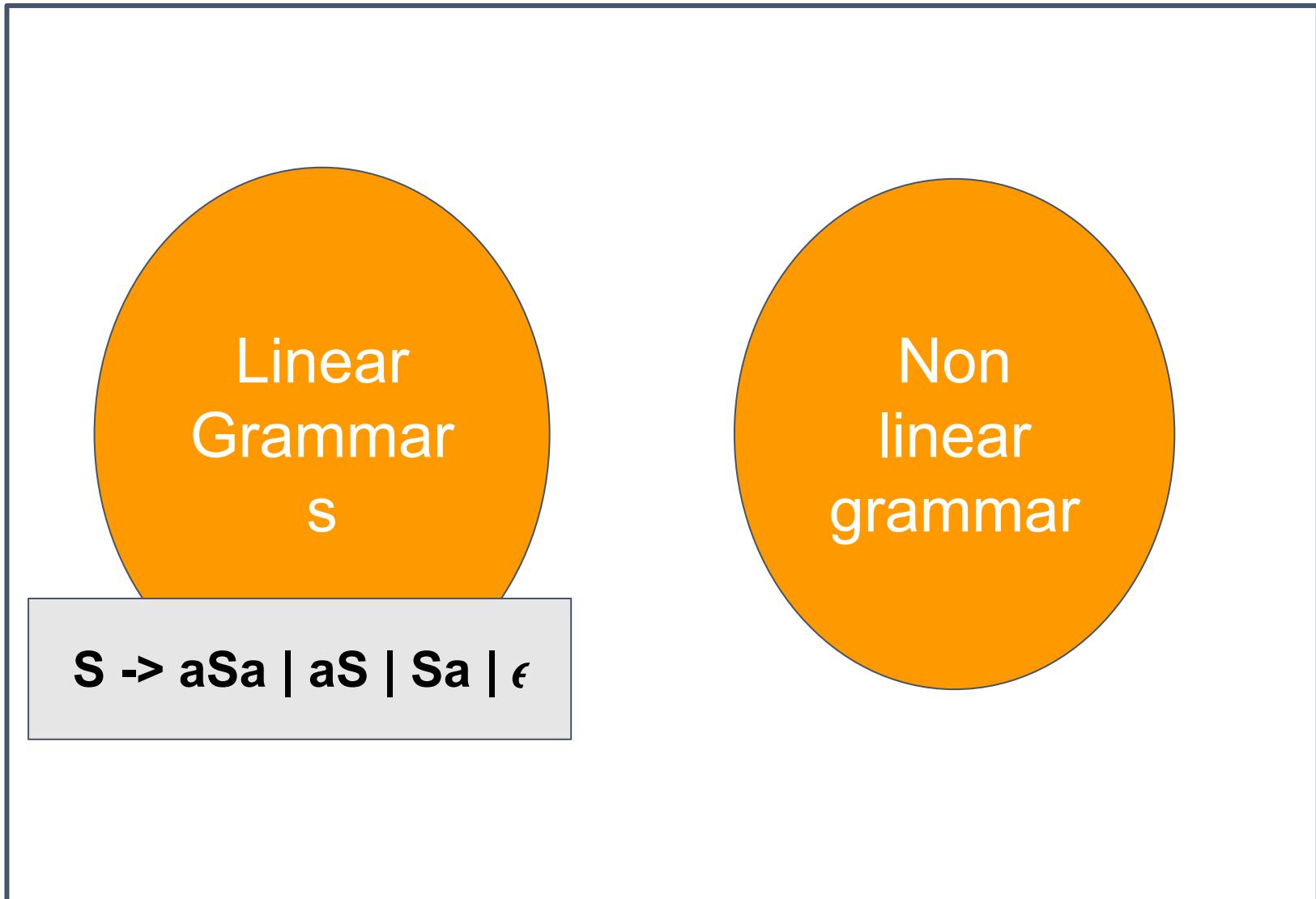
## Unit 2 - Regular Grammar

Linear  
Grammars

Non linear  
grammar

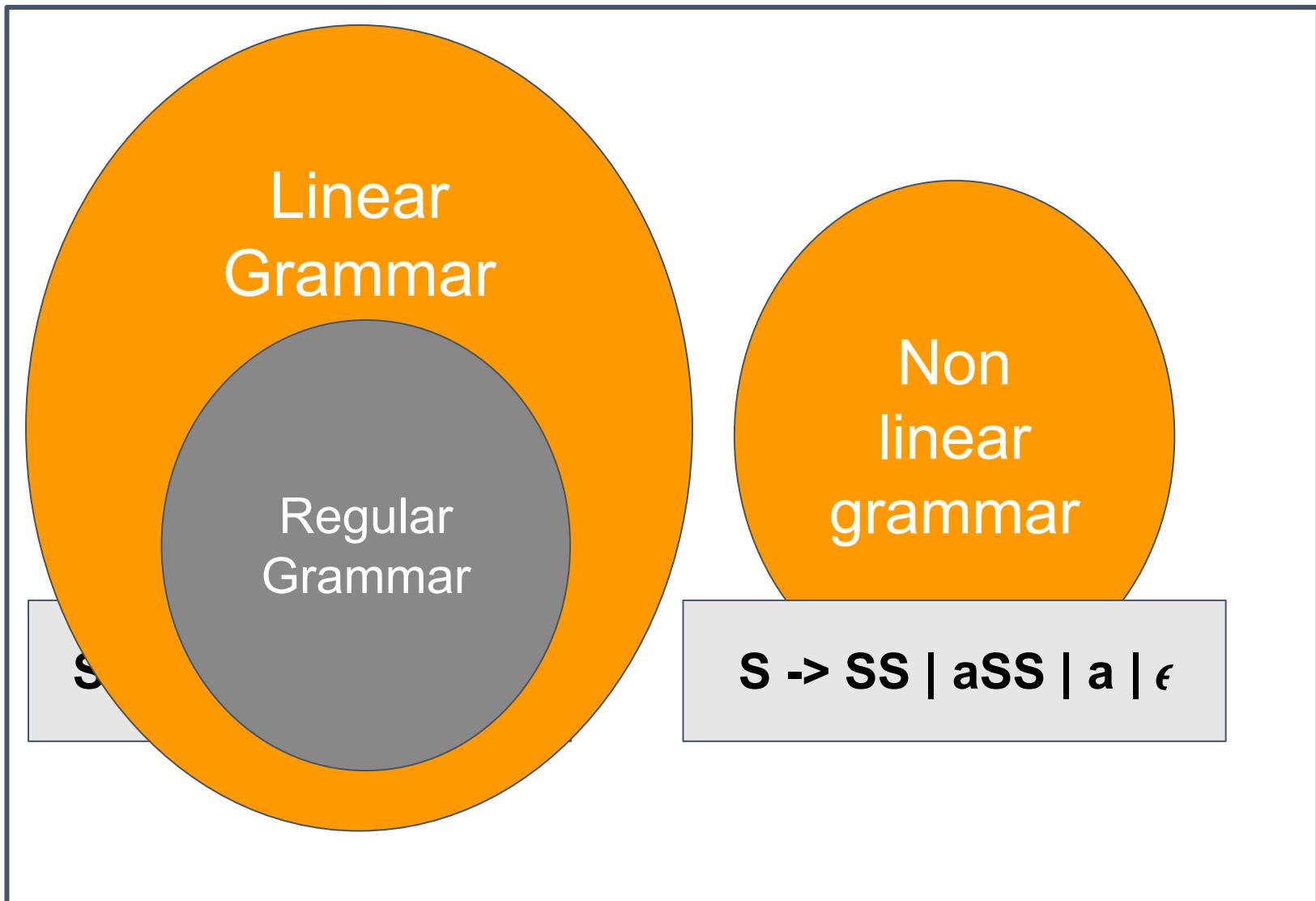
# Automata Formal Languages and Logic

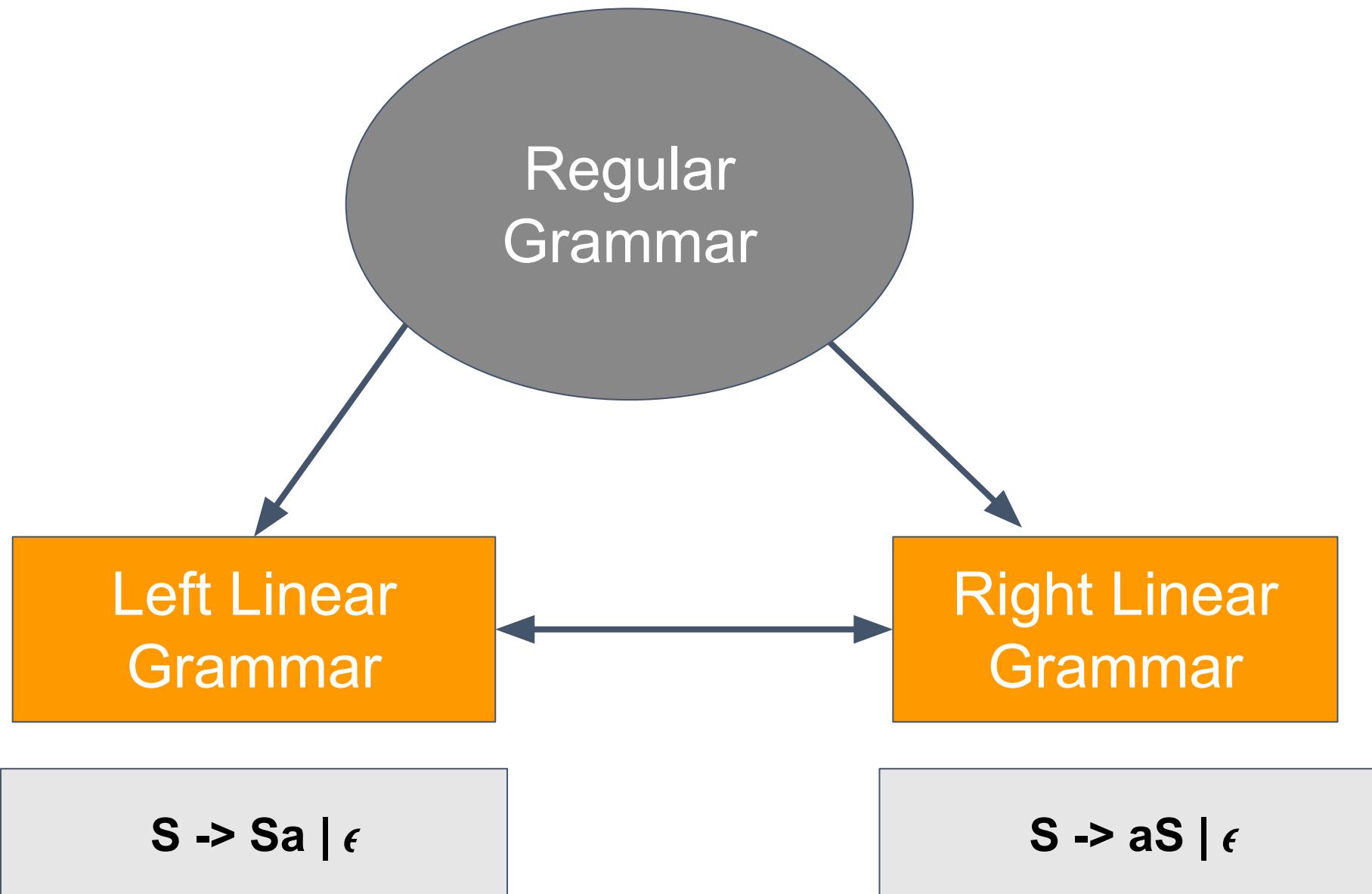
## Unit 2 - Regular Grammar



# Automata Formal Languages and Logic

## Unit 2 - Regular Grammar





# Automata Formal Languages and Logic

## Unit 2 - Regular Grammar

---



**Construct a regular grammar for a given language description**

Construct a regular grammar for the given language

| Language    | RE | RG                |
|-------------|----|-------------------|
| $L = \{a\}$ | a  | $S \rightarrow a$ |

Construct a regular grammar for the given language

| Language       | RE    | RG   |
|----------------|-------|--|
| $L = \{a, b\}$ | $a+b$ | $S \rightarrow a \mid b$<br>or<br>$S \rightarrow a$<br>$S \rightarrow b$ |

# Automata Formal Languages and Logic

## Unit 2 - Construct a regular grammar for the given language



Construct a regular grammar for the given language

| Language     | RE | RG                 |
|--------------|----|--------------------|
| $L = \{ab\}$ | ab | $S \rightarrow ab$ |

# Automata Formal Languages and Logic

## Unit 2 - Construct a regular grammar for the given language



Construct a regular grammar for the language  $L=\{a^n | n \geq 0\}$

| Language                             | RE    | RG                           |
|--------------------------------------|-------|------------------------------|
| $L = \{\lambda, a, aa, aaa, \dots\}$ | $a^*$ | $S \rightarrow aS   \lambda$ |

Construct a regular grammar for the language  $L=\{a^n | n \geq 1\}$

| Language                    | RE    | RG                        |
|-----------------------------|-------|---------------------------|
| $L = \{a, aa, aaa, \dots\}$ | $a^+$ | $S \rightarrow aS \mid a$ |

Construct a regular grammar for the regular expression

$(a+b)^*$

| Language  | RE        | RG                                      |
|---|-----------|---|
| $L =$<br>$\{\lambda, a, b, ab, abb, baaaab, \dots\}$<br>$L=\{ \text{any number of 'a's and b's} \}$ | $(a+b)^*$ | $S \rightarrow aS \mid bS \mid \lambda$ |

Construct a regular grammar for the regular expression

$(a+b)^+$

| Language  | RE        | RG                                       |
|---|-----------|--|
| $L =$<br>$\{\lambda, a, b, ab, abb, baaaab, \dots\}$<br>$L=\{ \text{any number of 'a's and b's} \}$ | $(a+b)^+$ | $S \rightarrow aS \mid bS \mid a \mid b$ |

Construct a regular grammar for the regular expression  $(ab)^*$

| Language  | RE       | RG                               |
|---|----------|----------------------------------|
| $L =$<br>$\{\lambda, a, b, ab, abab, \dots\}$<br>$L = \{ \text{any number of } ab's \}$ | $(ab)^*$ | $S \rightarrow abS \mid \lambda$ |

Construct a regular grammar for the language  $L=\{a^m b^n \mid n,m \geq 0\}$

| Language  | RE       | RG  |
|---|----------|---|
| $L=\{\lambda, a, b, bb, abb, \dots\}$<br>$L=\{\text{any number of } a's \text{ followed by any number of } b's\}$ | $a^*b^*$ | $S \rightarrow A B$<br>A: any number of a's<br>B: any number of b's<br>$A \rightarrow aA \lambda$<br>$B \rightarrow bB \lambda$ |

Construct a regular grammar for the given language with even number of a's.  $L=\{a^{2n} \mid n \geq 0\}$

| Language   | RE       | RG                               |
|--|----------|----------------------------------|
| $L=\{\lambda,$<br>$aa,$<br>$aaaa,$<br>$aaaaaaaa \dots$ | $(aa)^*$ | $S \rightarrow aaS \mid \lambda$ |

Construct a regular grammar for the given language with odd number of a's.  $L=\{a^{2n+1} \mid n \geq 0\}$

| Language  | RE        | RG                         |
|---|-----------|----------------------------|
| $L=\{a,$<br>$a\bar{a},$<br>$a\bar{a}\bar{a},$<br>$a\bar{a}\bar{a}\bar{a} \dots\}$ | $(aa)^*a$ | $S \rightarrow aaS \mid a$ |

Construct a regular grammar for the given language with number of a's as multiples of 4.  $L=\{a^{4n} \mid n \geq 0\}$

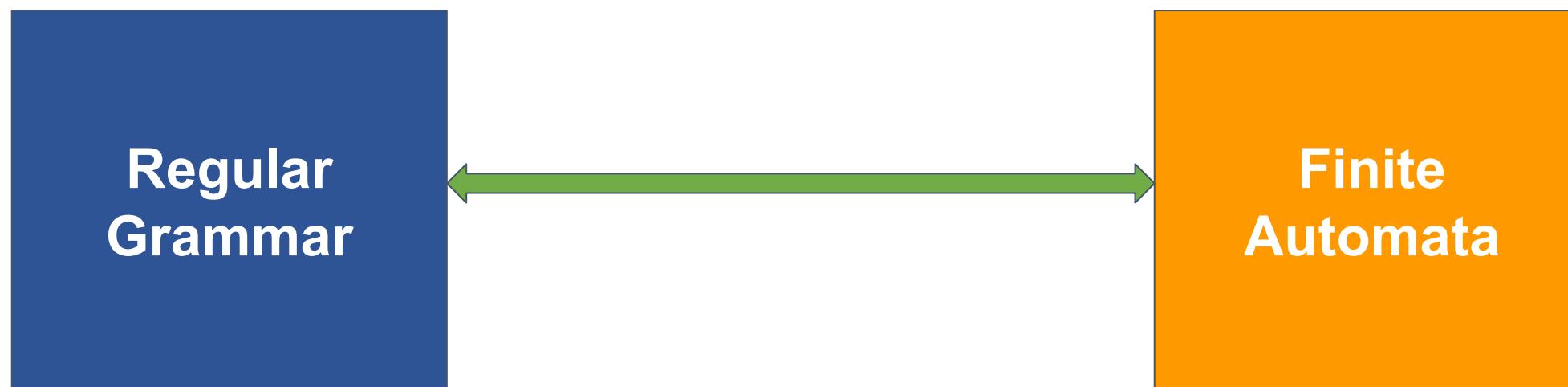
| Language  | RE         | RG                                 |
|---|------------|------------------------------------|
| $L=\{\lambda,$<br>$aaaa,$<br>$aaaaaaaa,$<br>$aaaaaaaaaaaa, \dots$ | $(aaaa)^*$ | $S \rightarrow aaaaS \mid \lambda$ |

Convert finite automata to regular grammar for the language to accept at least one 'a' over the alphabet  $\Sigma=\{a,b\}$

| Language   | RE              | RG  |
|--|-----------------|---|
| $L=\{a,$<br>$bb\dots a,$<br>$ba \text{ any } \# \text{ of } a's \&$<br>$b's$ | $b^* a (a+b)^*$ | $S \rightarrow bS \mid aA$<br>$A \rightarrow aA \mid bA \mid \lambda$ |

# Automata Formal Languages and Logic

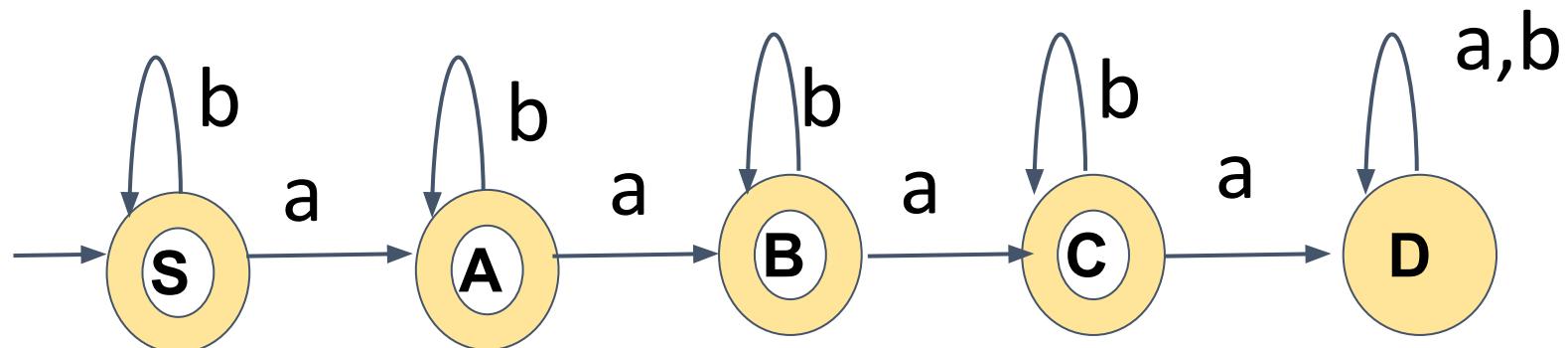
## Unit 2 - Equivalence of regular grammar and Finite automata



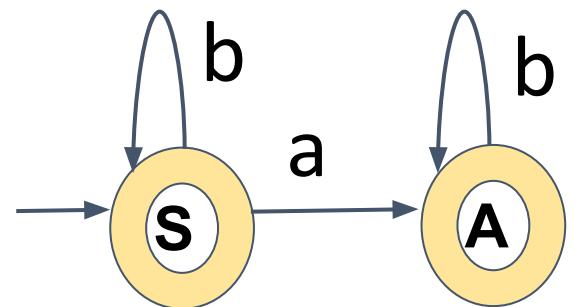
**Let's look at examples to convert :**

- 1. Finite automata to regular grammar**
- 2. Regular Grammar to Finite Automata**

1. Convert finite automata to regular grammar for the language to accept at most 3 'a's over the alphabet  $\Sigma=\{a,b\}$ .



Let's look at each transition and perform the conversion

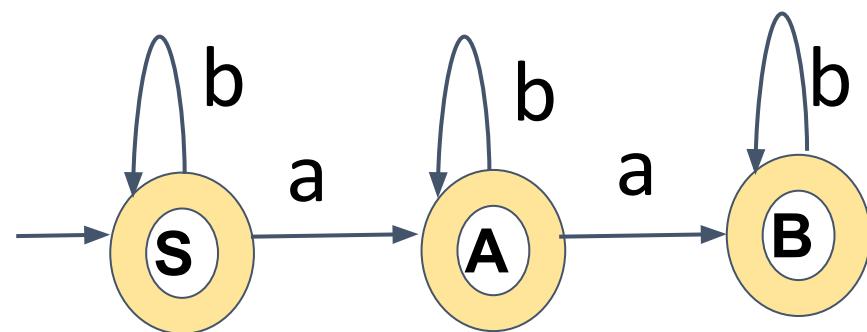


$$S \rightarrow bS \mid aA \mid \lambda$$

$$A \rightarrow bA \mid \lambda$$

$\lambda$  on RHS indicates S  
and A are final states

Let's look at each transition and perform the conversion



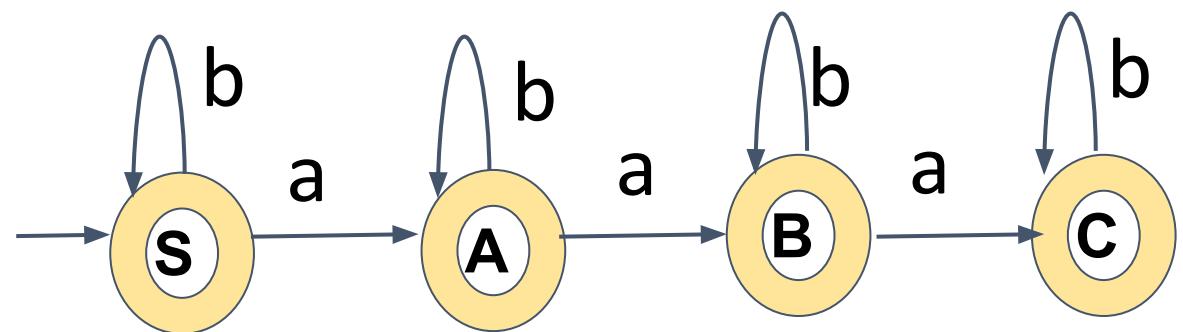
$$S \rightarrow bS | aA | \lambda$$

$$A \rightarrow bA | aB | \lambda$$

$$B \rightarrow bB | aC | \lambda$$

$\lambda$  on RHS indicates S,A  
and B are final states

Let's look at each transition and perform the conversion



$$S \rightarrow bS \mid aA \mid \lambda$$

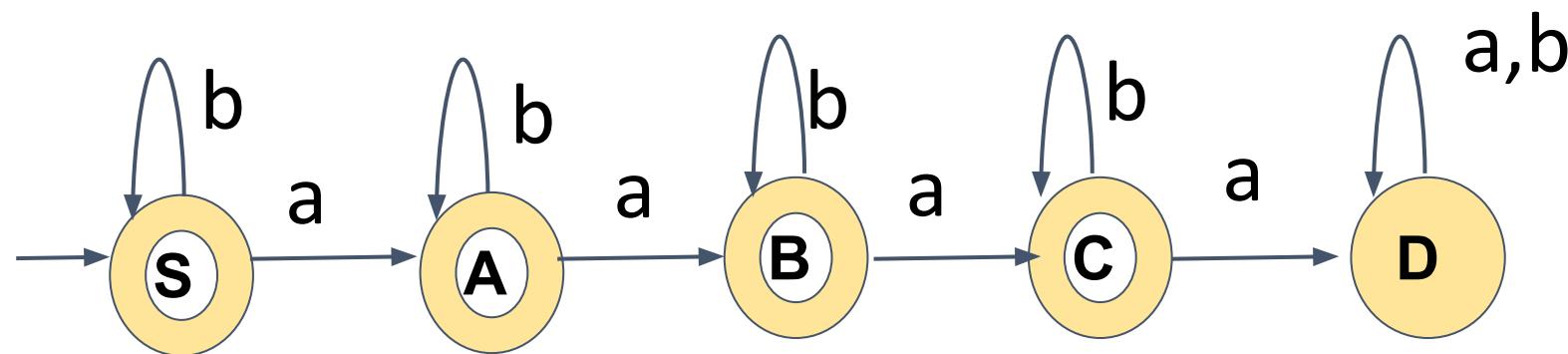
$$c \rightarrow bC \mid aD \mid \lambda$$

$$A \rightarrow bA \mid aB \mid \lambda$$

$$B \rightarrow bB \mid aC \mid \lambda$$

Let's look at each transition and perform the conversion

Since D is a dead state, we will not encode C-> aD and D -> aD | bD as D is a useless Non-terminal(that means it never terminates)



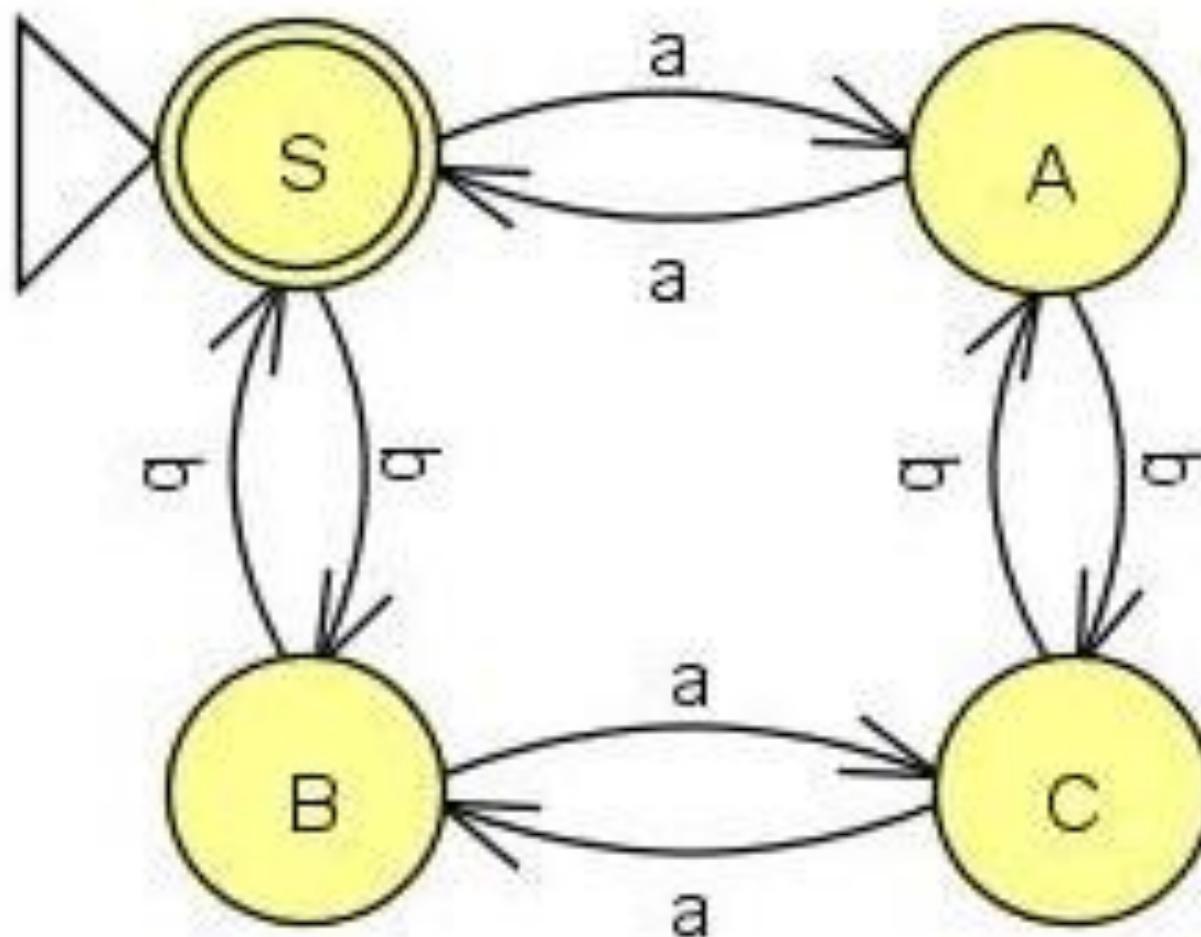
$$S \rightarrow bS \mid aA \mid \lambda$$

$$c \rightarrow bC \mid \lambda$$

$$A \rightarrow bA \mid aB \mid \lambda$$

$$B \rightarrow bB \mid aC \mid \lambda$$

2. Convert a given finite automata accepting  $L=\{n_a(w) \bmod 2=0 \text{ and } n_b(w) \bmod 2=0\}$  to regular grammar.



2. Convert a given finite automata accepting  $L = \{n_a(w) \bmod 2 = 0 \text{ and } n_b(w) \bmod 2 = 0\}$  to regular grammar.

Start state of automata will be the start symbol of the grammar.

We start with  $S, S$  on seeing terminal  $a$  it moves to state  $A (S \rightarrow aA)$  and on seeing terminal  $b$  it moves to state  $B (S \rightarrow bB)$ .

Since  $S$  is also the final state, we introduce the production  $S \rightarrow \lambda$ .

Grammar is

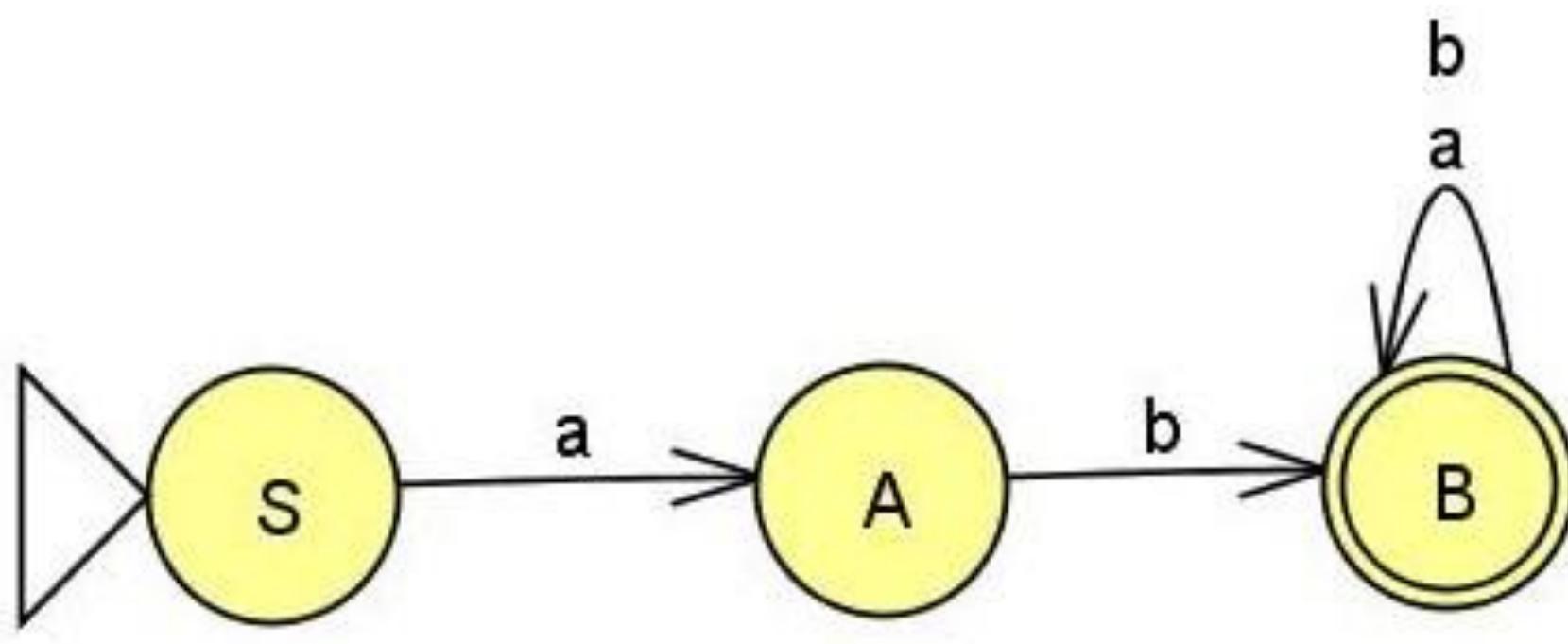
$$S \rightarrow aA \mid bB \mid \lambda$$

$$A \rightarrow aS \mid bC$$

$$B \rightarrow aC \mid bS$$

$$C \rightarrow aB \mid bA$$

3. Converting a given finite automata accepting  $L=\{abw, w \in \{a,b\}^*\}$  to regular grammar.



3. Converting a given finite automata accepting  $L=\{abw, w \in \{a,b\}^*\}$  to regular grammar.

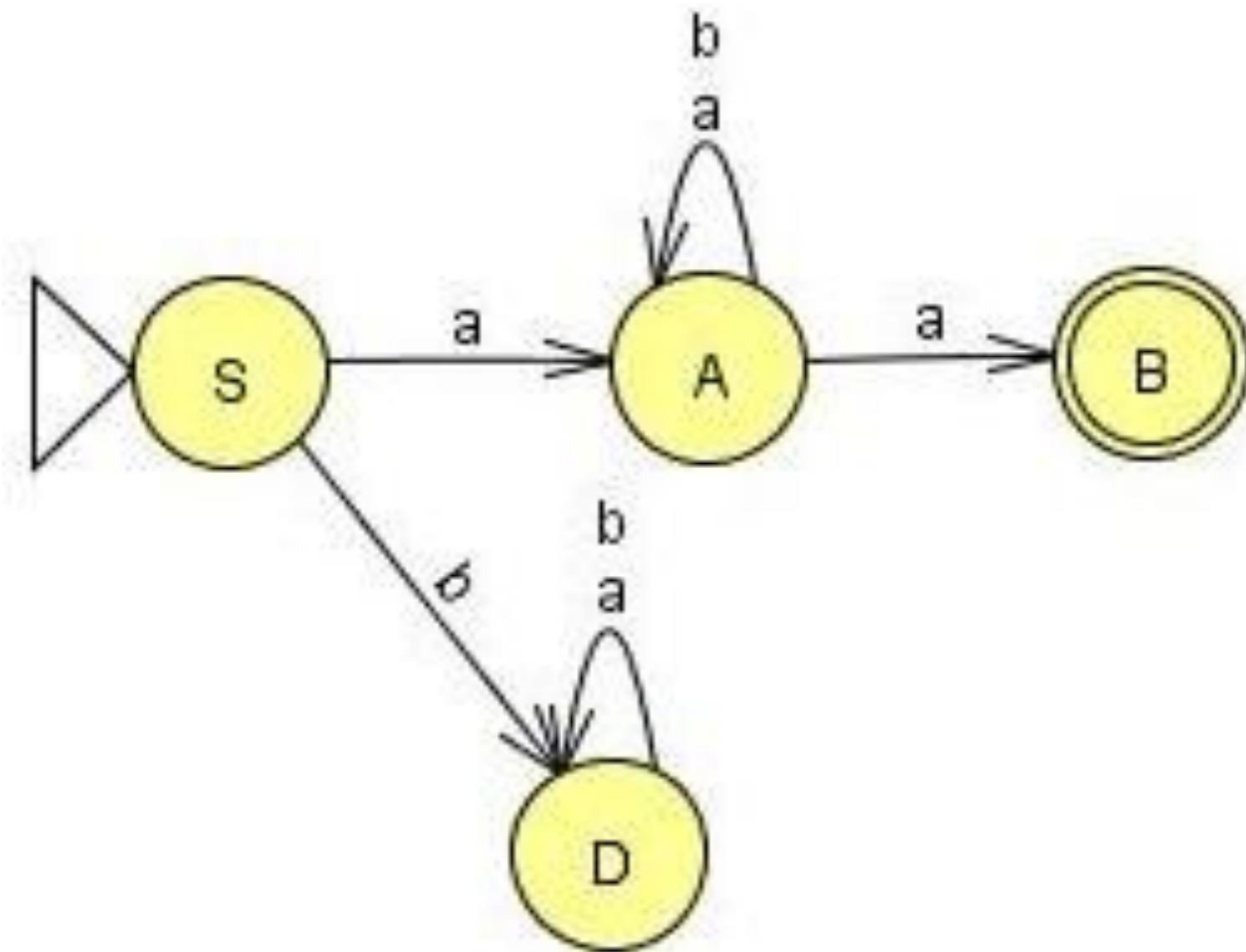
Grammar is,

$$S \rightarrow aA$$

$$A \rightarrow bB$$

$$B \rightarrow aB \mid bB \mid \lambda$$

4. Converting given finite automata accepting  $L=\{awa, w \in \{a,b\}^*\}$  to regular grammar.



4. Converting given finite automata accepting  $L=\{awa, w \in \{a,b\}^*\}$  to regular grammar.

So the grammar is,

$$S \rightarrow aA$$

$$A \rightarrow aA \mid bA \mid aB$$

$$B \rightarrow \lambda$$

### Converting Regular grammar to finite automata

**Example 1 :**

$A \rightarrow aB | bA | b$

$B \rightarrow aC | bB$

$C \rightarrow aA | bC | a$

**Variables->state**

**terminal symbols ->symbols of finite automata**

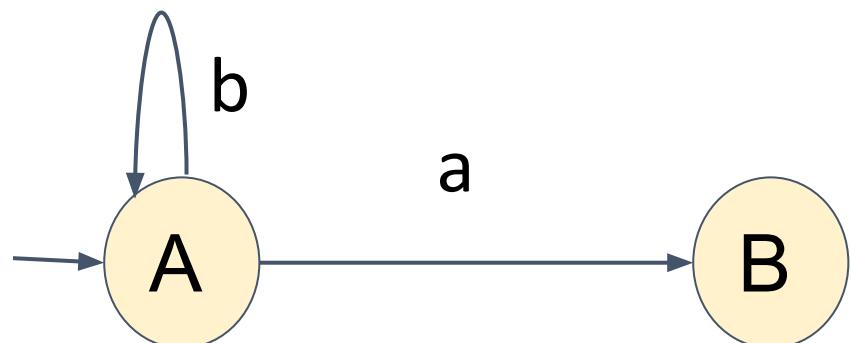
### Example 1 :

$A \rightarrow aB \mid bA \mid b$

$B \rightarrow aC \mid bB$

$C \rightarrow aA \mid bC \mid a$

- Initial start symbol will be the start state



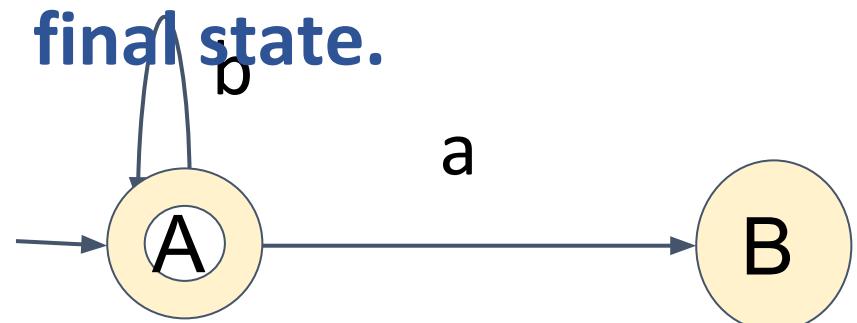
### Example 1 :

$A \rightarrow aB \mid bA \mid b$

$B \rightarrow aC \mid bB$

$C \rightarrow aA \mid bC \mid a$

- Transition  $A \rightarrow bA$  transforms to  $A \rightarrow b$ , when  $A \rightarrow \lambda$   $A \rightarrow \lambda$  indicates A is a final state.

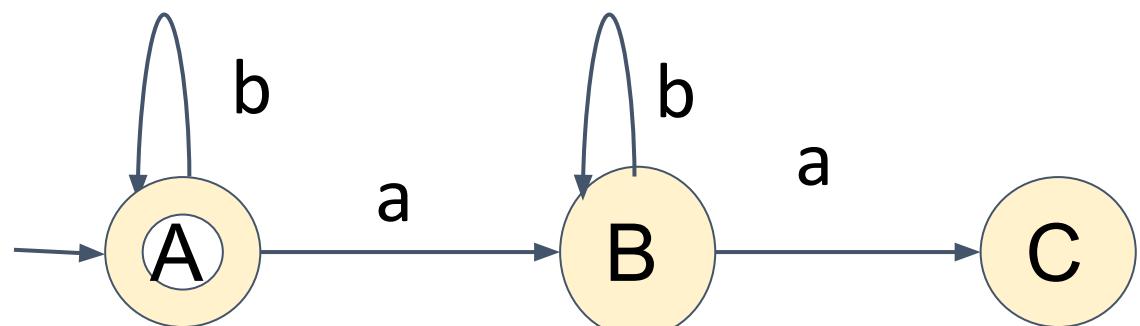


**Example 1 :**

$$A \rightarrow aB \mid bA \mid b$$

$$B \rightarrow aC \mid bB$$

$$C \rightarrow aA \mid bC \mid a$$



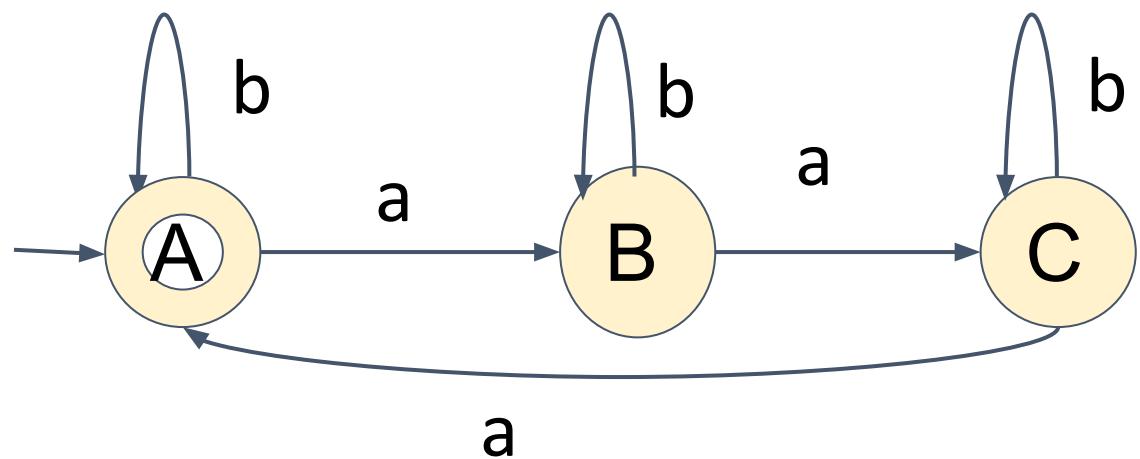
### Example 1 :

$A \rightarrow aB \mid bA \mid b$

$B \rightarrow aC \mid bB$

$C \rightarrow aA \mid bC \mid a$

- Transition  $C \rightarrow aA$  to  $C \rightarrow a$  when when  $A \rightarrow \lambda$  indicates A is a final state.



# Automata Formal Languages and Logic

## Unit 2 - Regular Grammar to Finite Automata



**Example 2 :**

$S \rightarrow 01A$

$A \rightarrow 10B$

$B \rightarrow 0A \mid 11$

# Automata Formal Languages and Logic

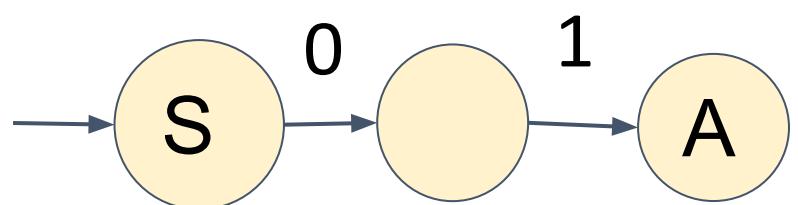
## Unit 2 - Regular Grammar to Finite Automata

**Example 2 :**

$S \rightarrow 01A$

$A \rightarrow 10B$

$B \rightarrow 0A \mid 11$

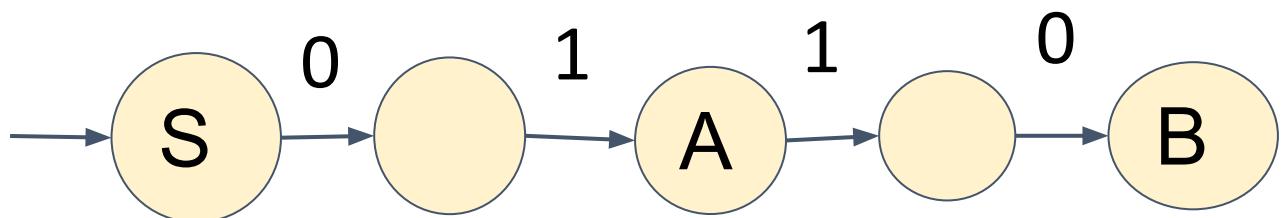


**Example 2 :**

$S \rightarrow 01A$

$A \rightarrow 10B$

$B \rightarrow 0A \mid 11$



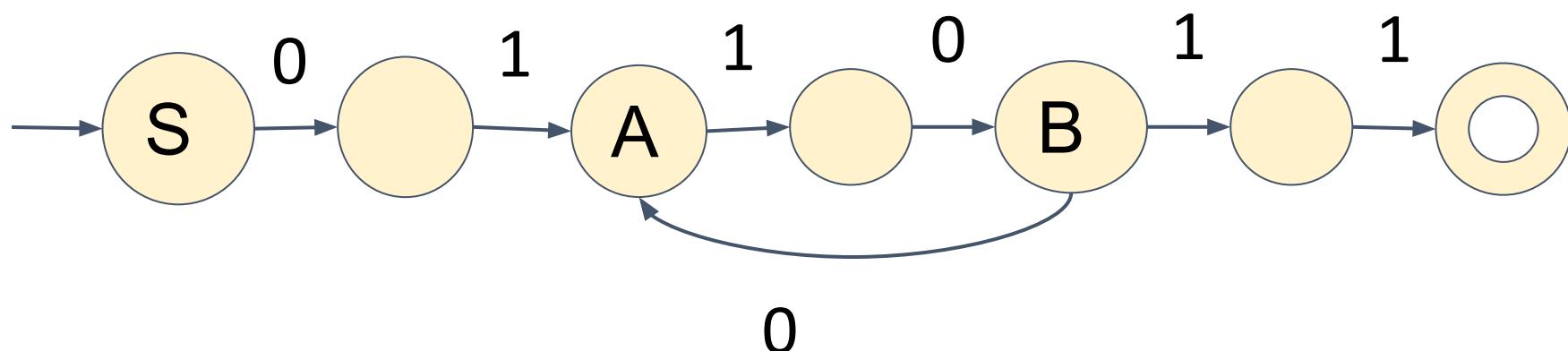
### Example 2 :

$S \rightarrow 01A$

$A \rightarrow 10B$

$B \rightarrow 0A \mid 11$

$B \rightarrow 11$  indicates on state B on consuming the input 11 we reach final state



### Example 3 :

$S \rightarrow bS | aA$

$A \rightarrow aA | bA$

$A \rightarrow aB$

$B \rightarrow bbB$

$B \rightarrow \lambda$

### Example 3 :

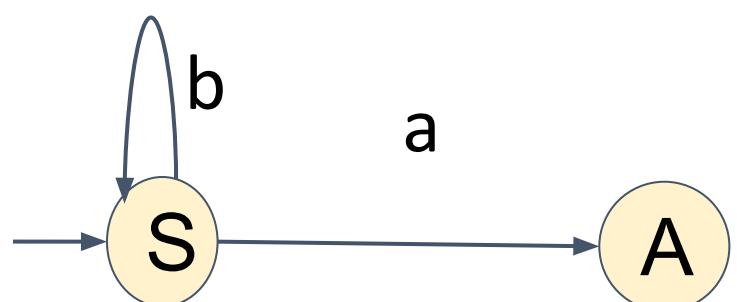
$S \rightarrow bS \mid aA$

$A \rightarrow aA \mid bA$

$A \rightarrow aB$

$B \rightarrow bbB$

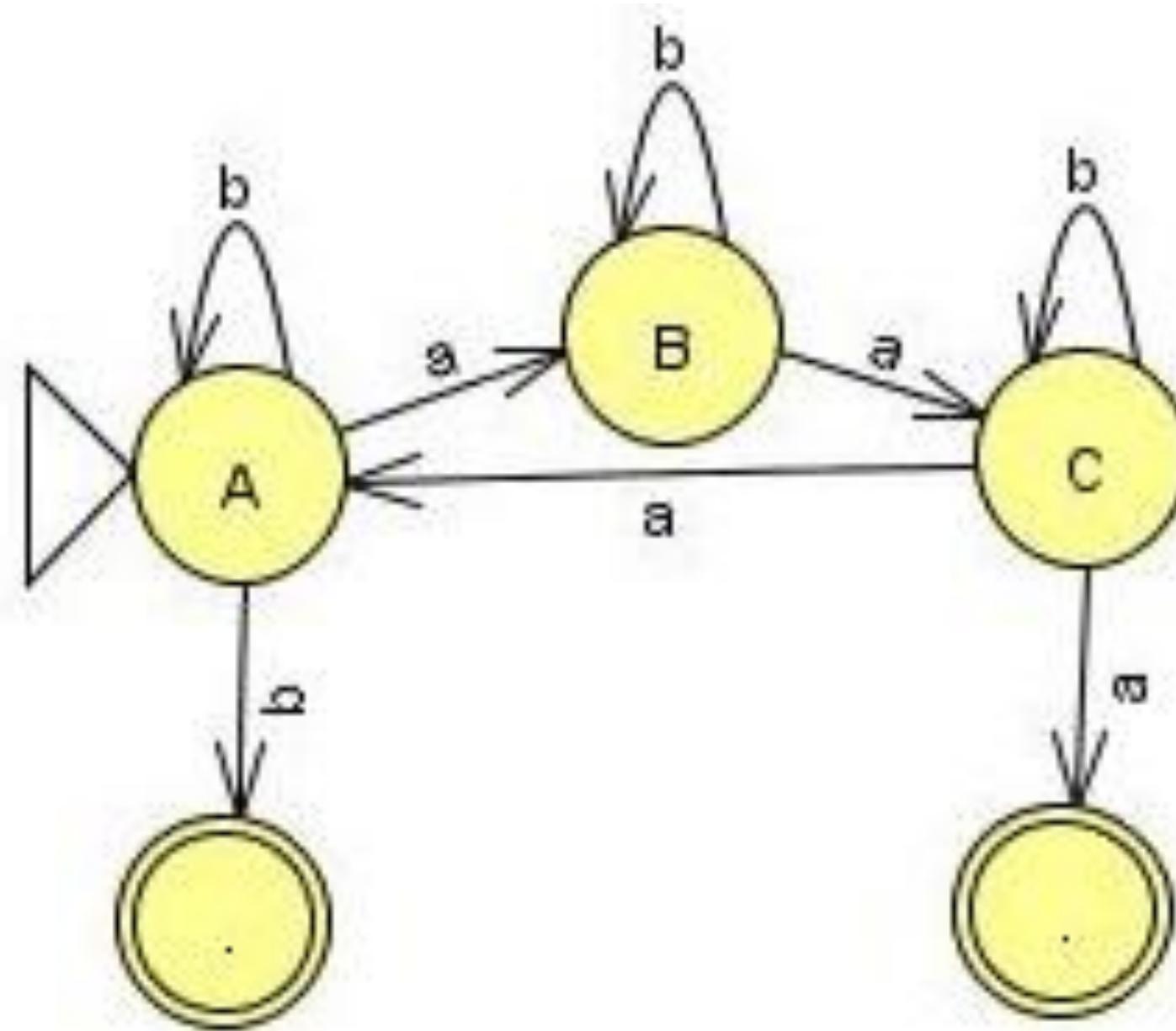
$B \rightarrow \lambda$

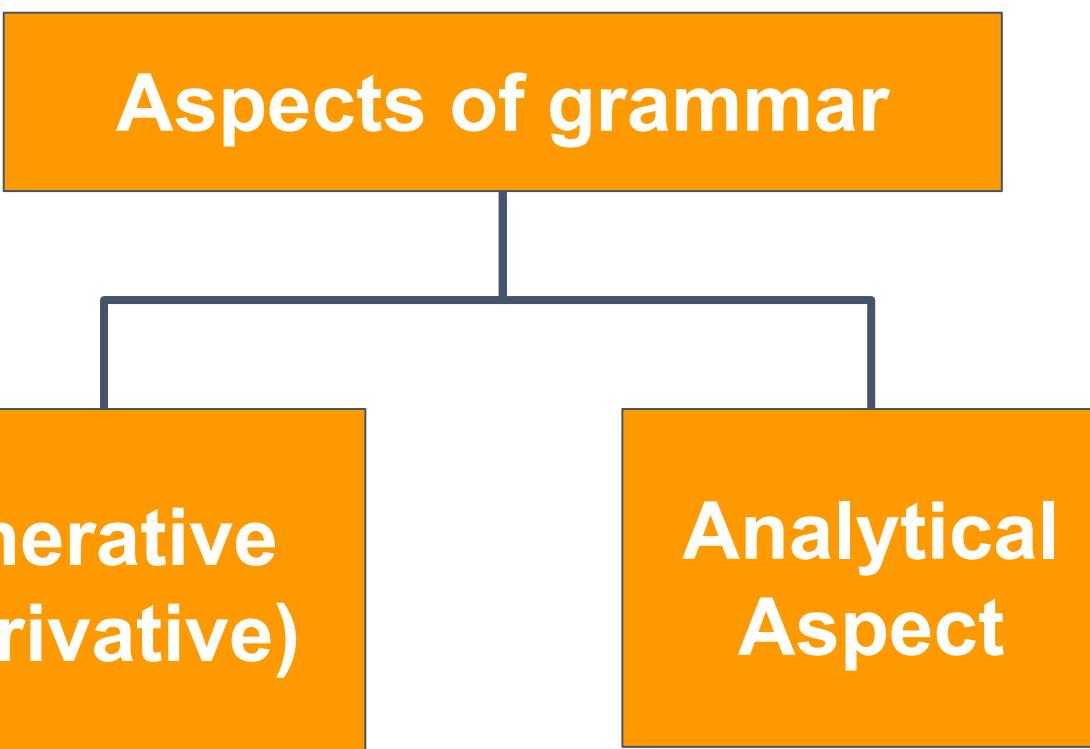


### Grammar

$$S \rightarrow bS | aA$$
$$A \rightarrow aA | bA | aB$$
$$B \rightarrow bbB | \lambda$$

### Automata





**Generate All  
Strings  $w \in L(G)$**

**Check whether  
 $w \in L(G) ??$**

# Parse Tree

### What is a Tree?

Leaves



Branches

Root

Technically we represent a Tree upside down

**Root-**



**Branches**

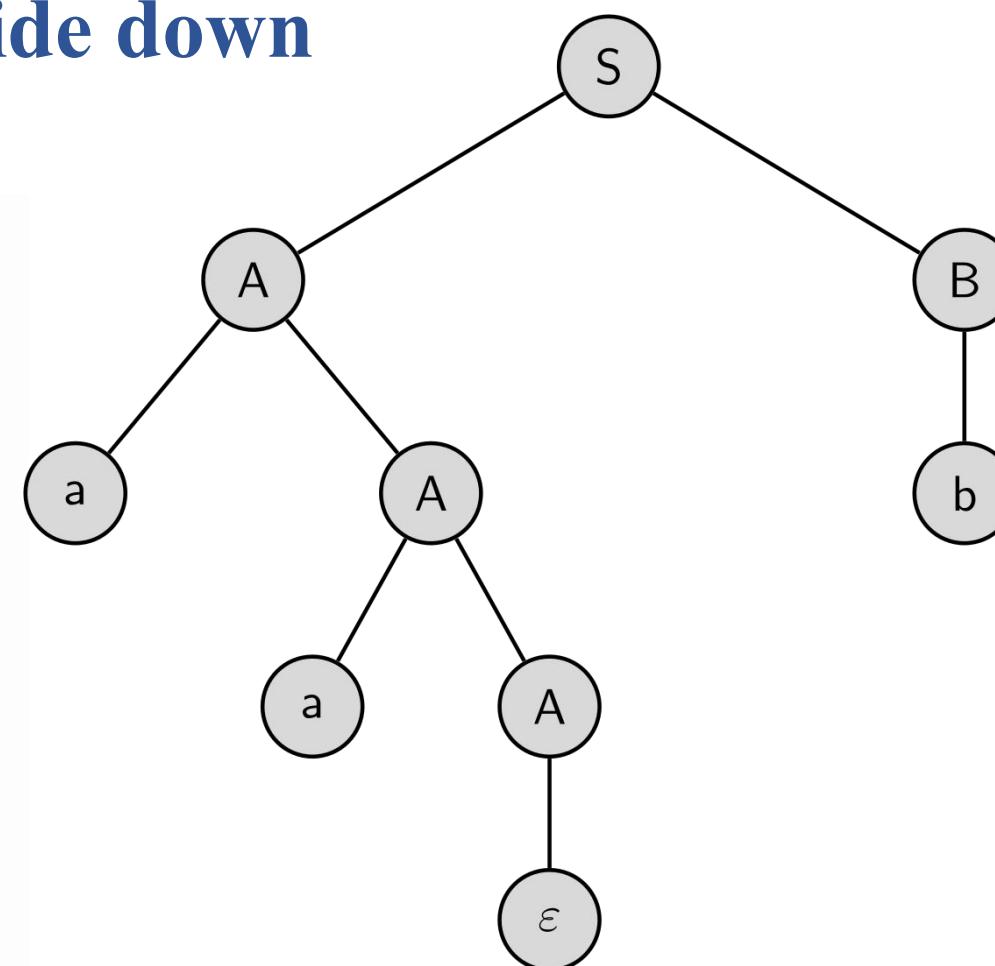
**Leaves**

Technically we represent a Tree upside down

**Root**

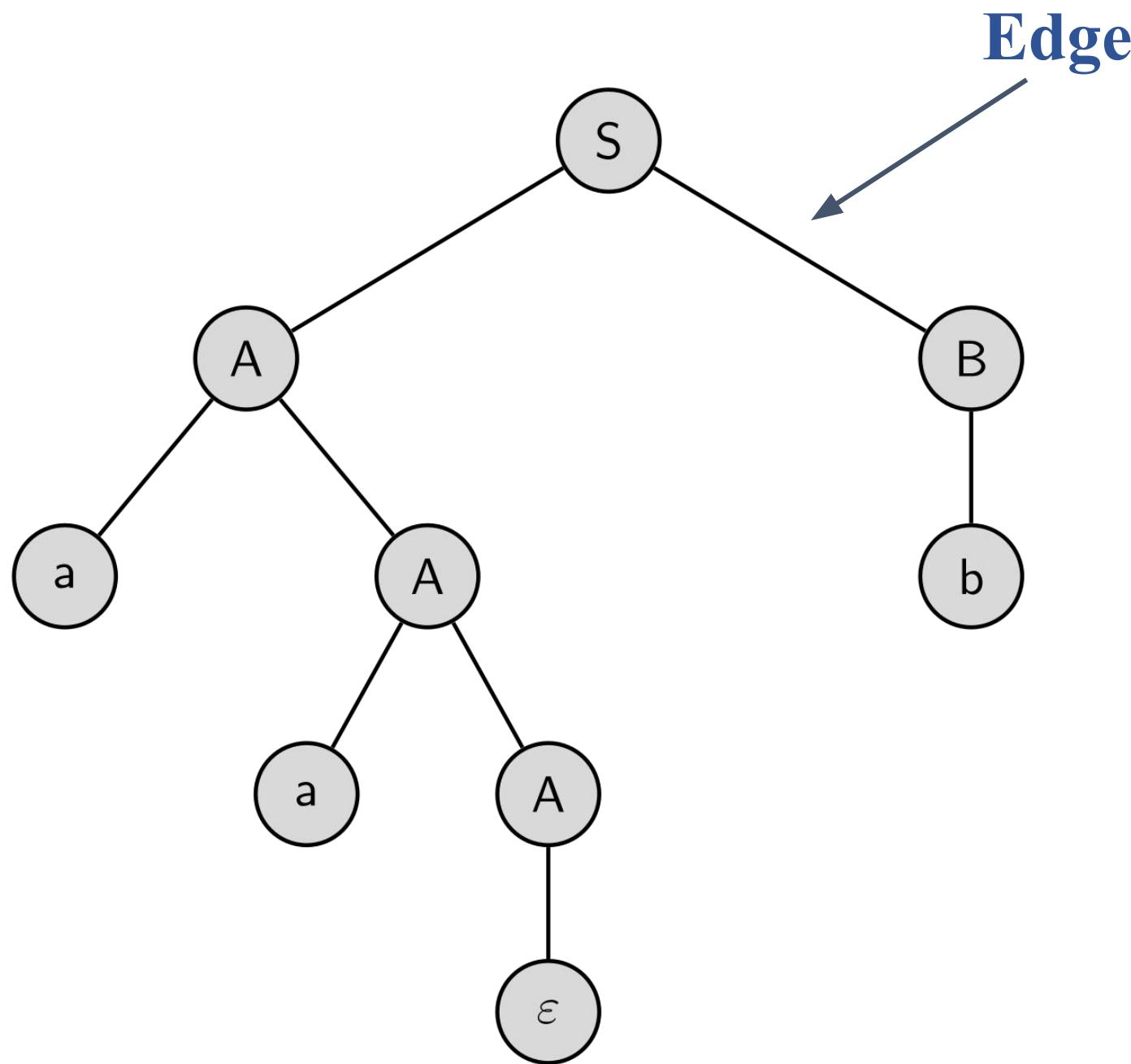


**Branches**

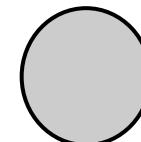


**Leaves**

### Trees



Edge

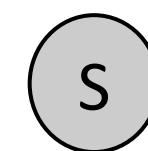
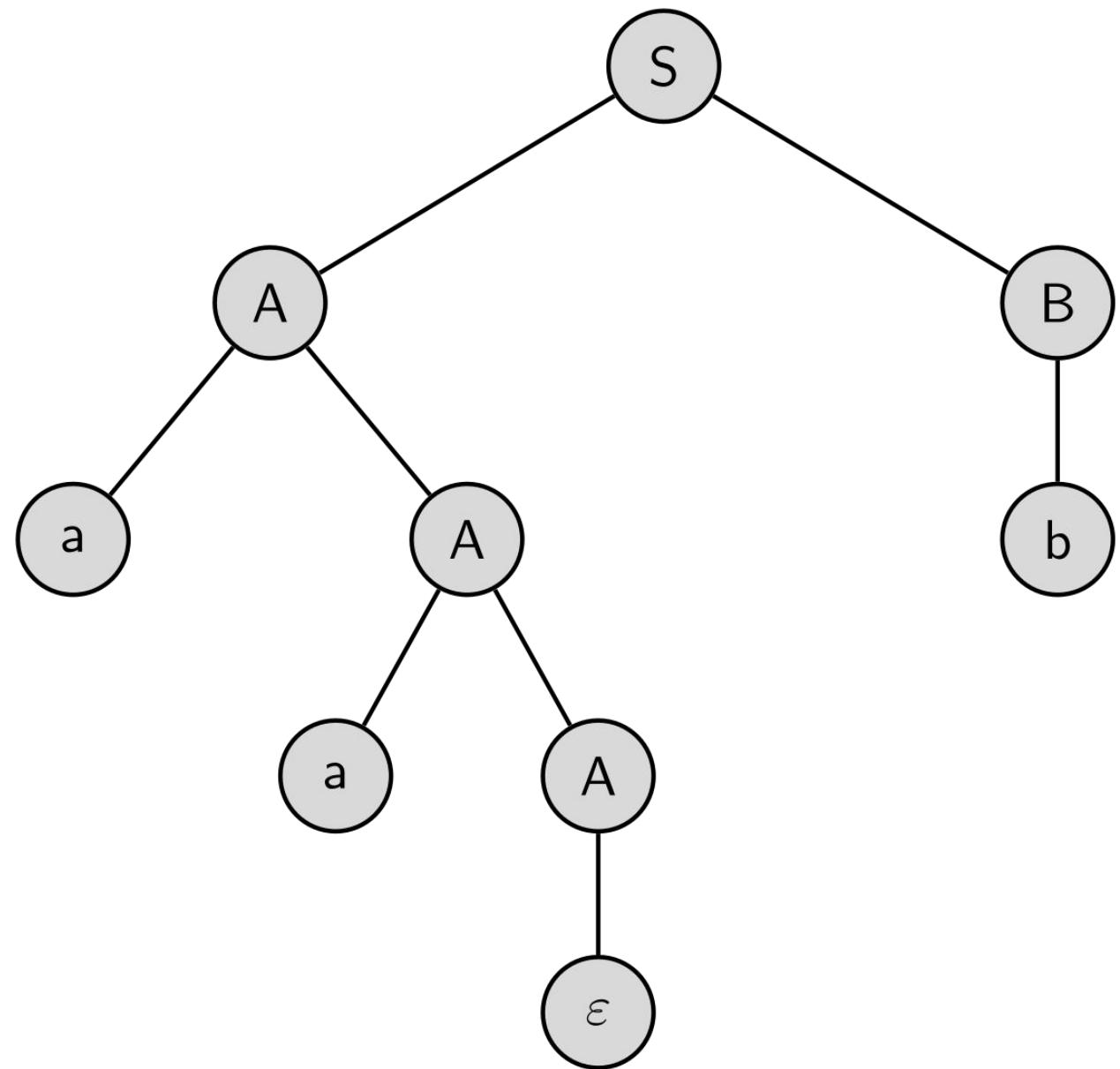


Node



Root Node

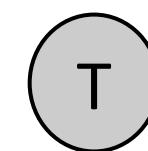
### Parse Tree / Derivation Tree



**Root Node (Start Symbol)**



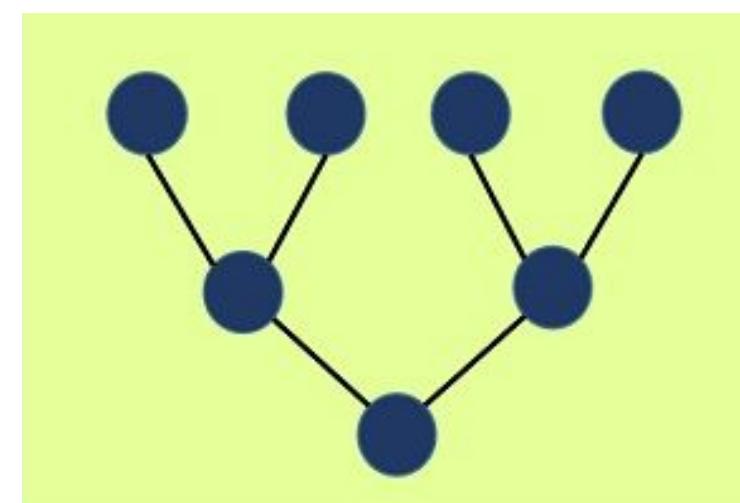
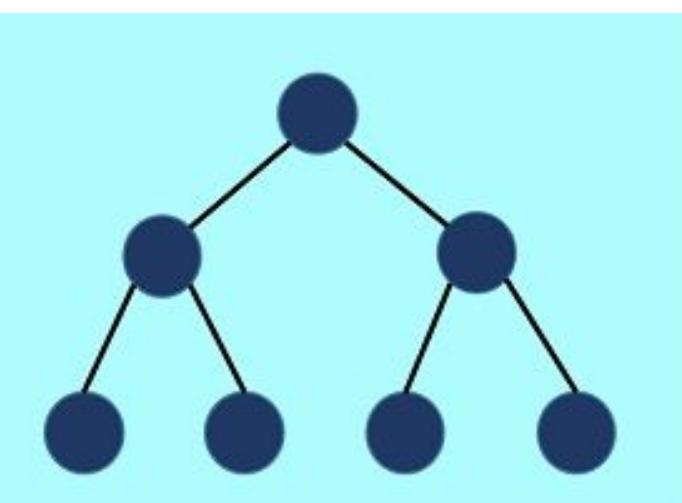
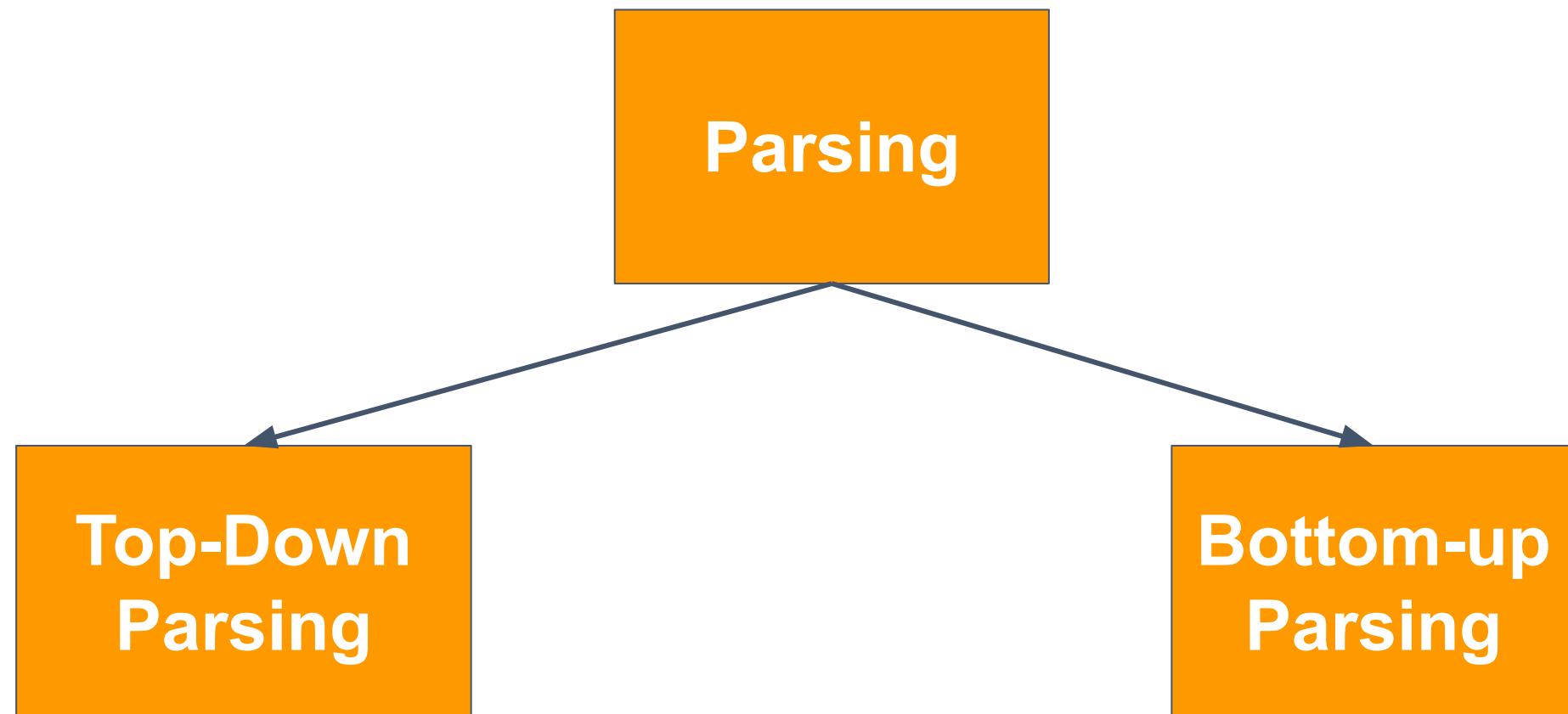
**Non-Terminals - Interior Nodes**

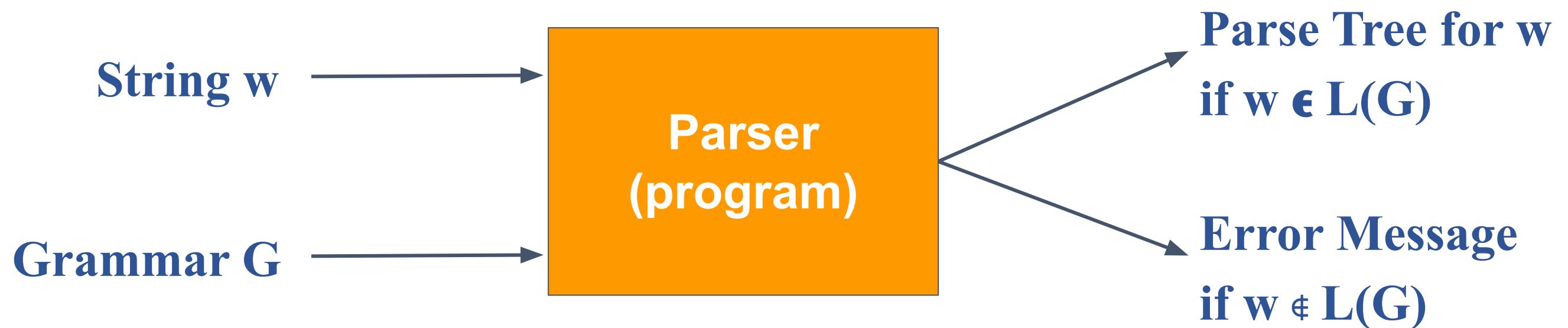


**Terminal Symbols - Leaf nodes**

**Yield of Parse Tree - aab**

**Parsing is the process of determining whether a String  
 $w \in L(G)??$**





Let us look at few examples and generate parse tree for a given String w and Grammar G.

### Parse Tree

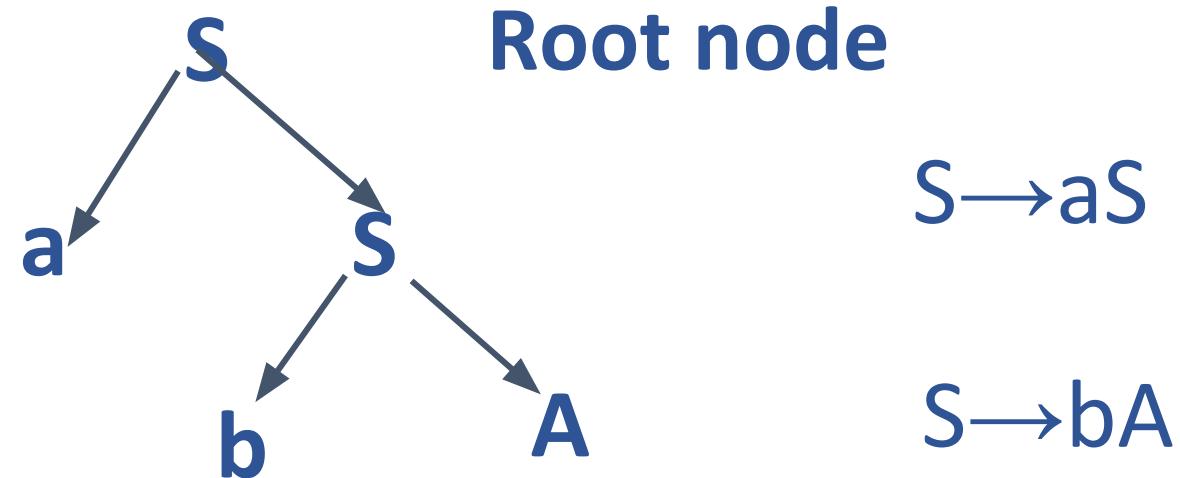
S

Root node

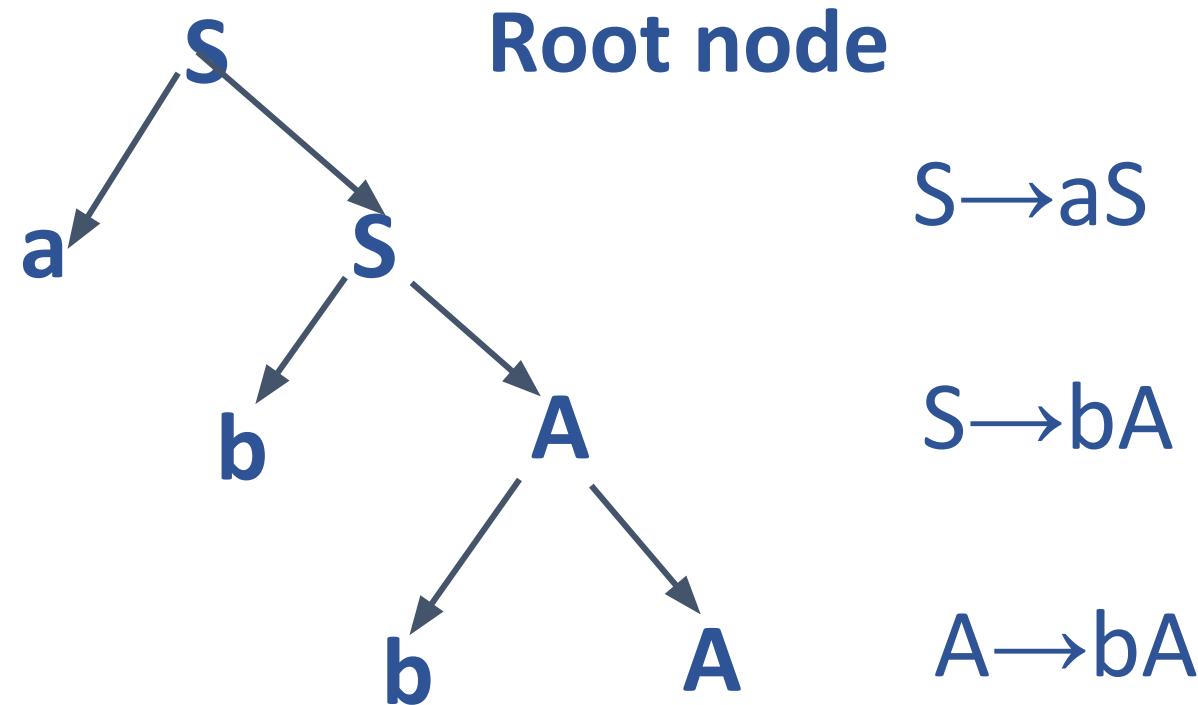
### Parse Tree



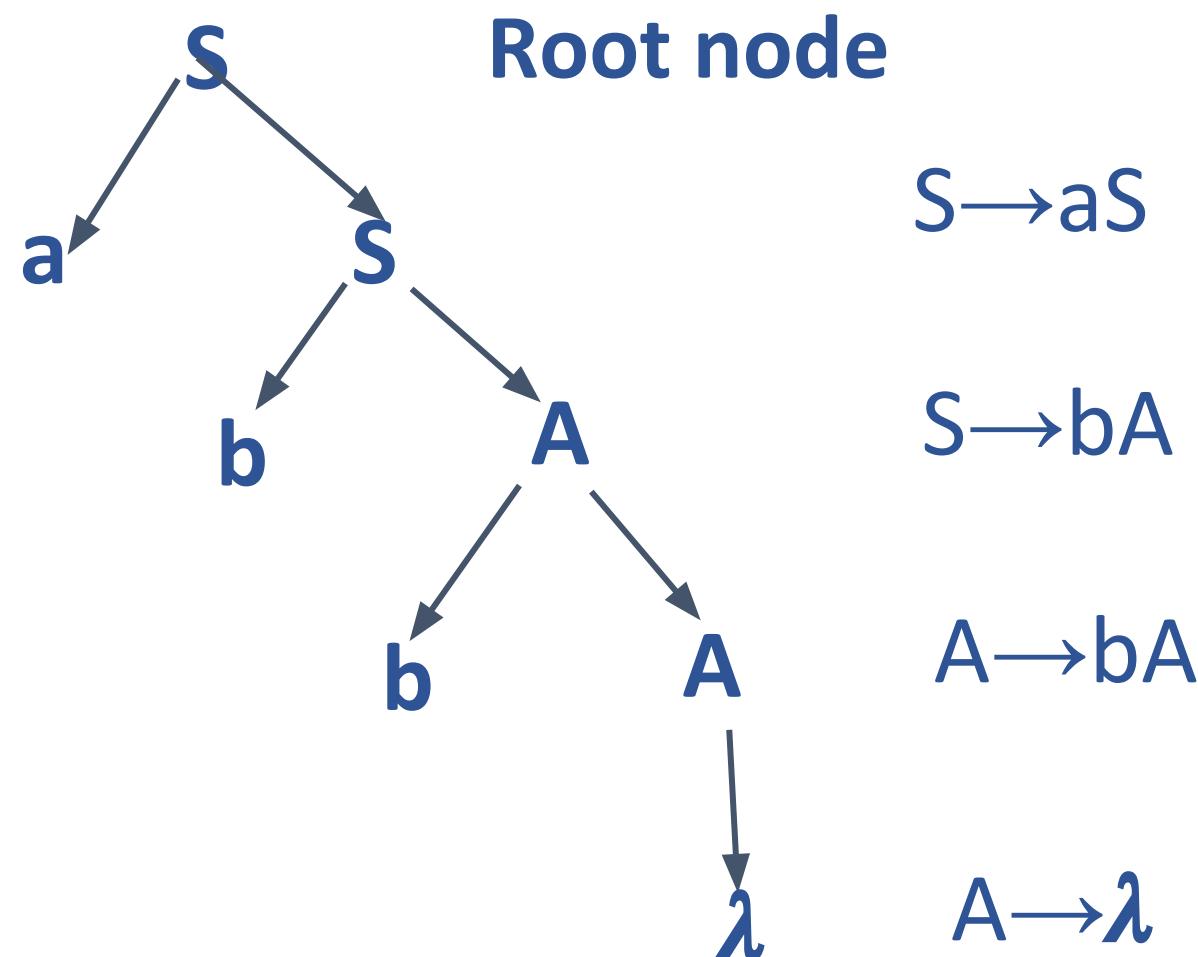
### Parse Tree



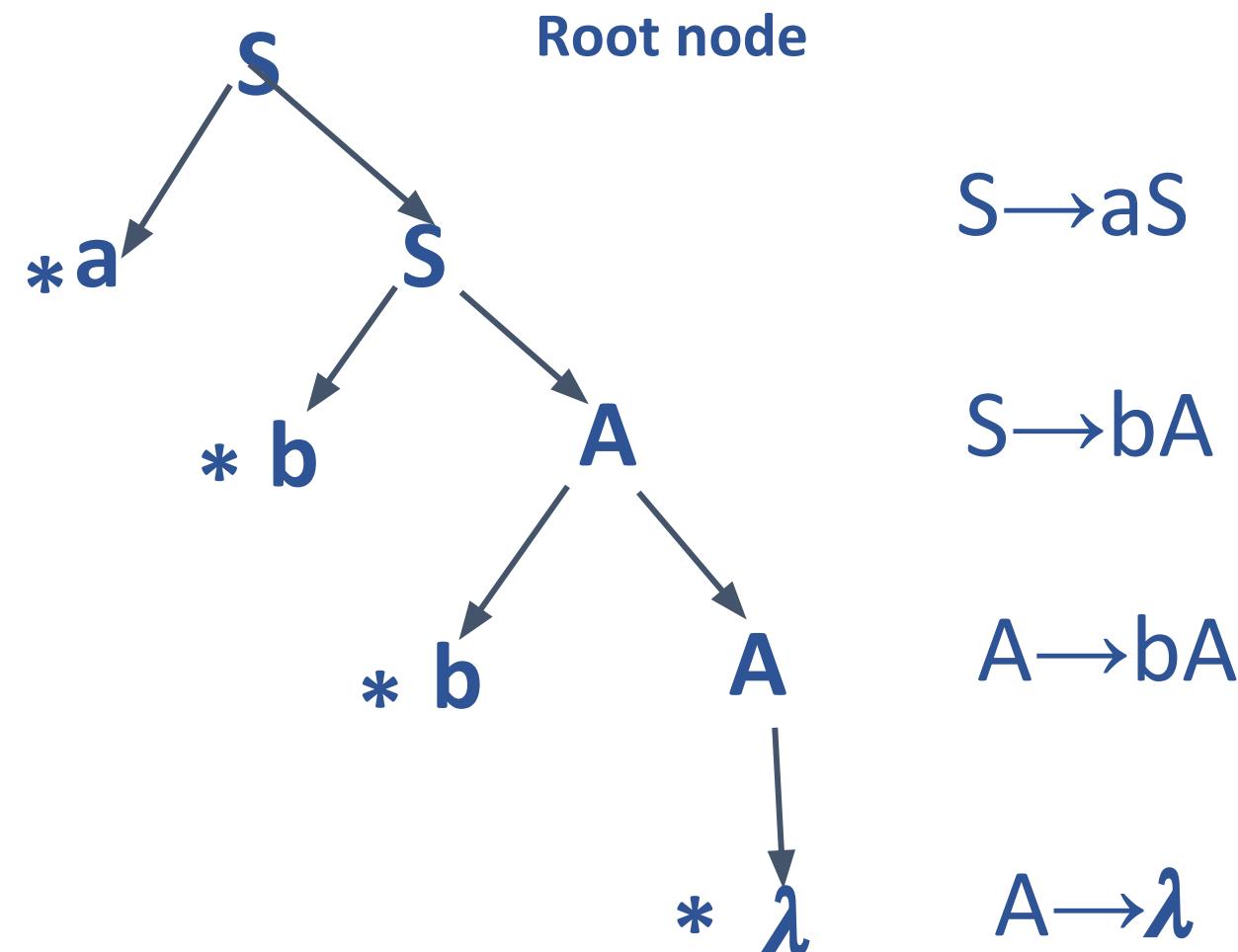
### Parse Tree



### Parse Tree



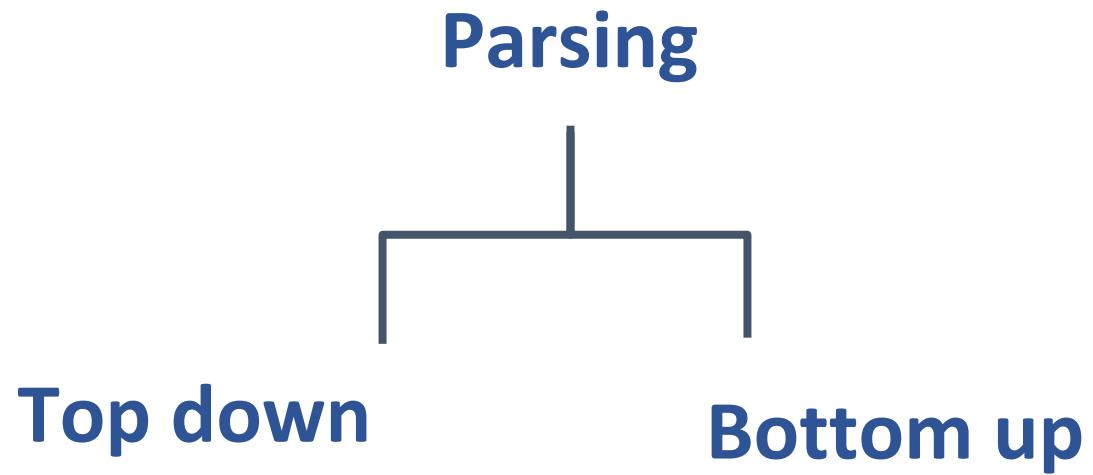
### Parse Tree



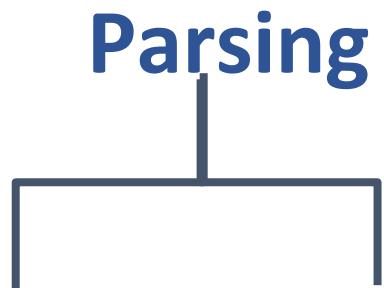
Leaf Node:\*

Yield of the tree:abb

### 2. Analytical aspect



### 2. Analytical aspect



$S \rightarrow aS$

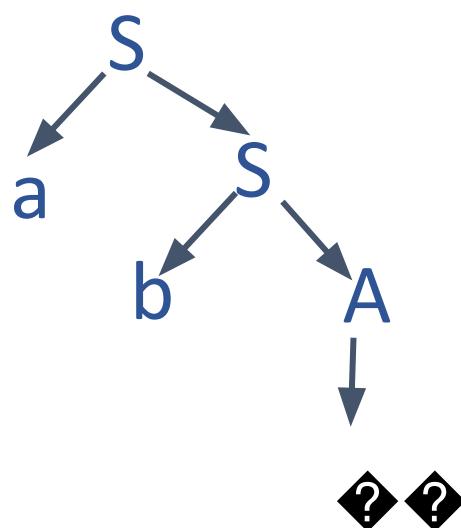
$S \rightarrow bA$

$A \rightarrow bA$

$A \rightarrow \lambda$

**Top down**

**Bottom up**

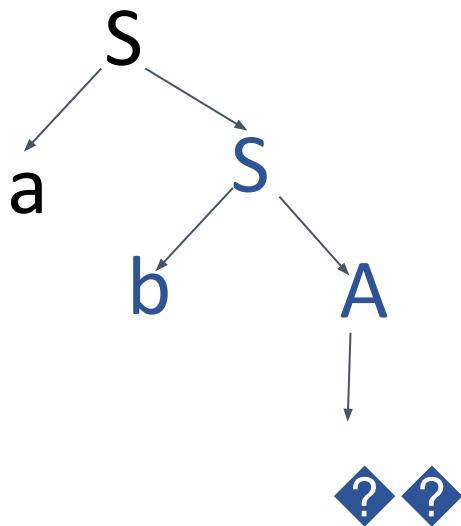


### 2. Analytical aspect

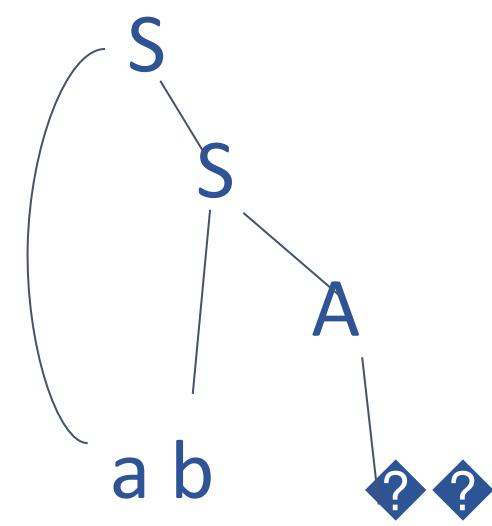
#### Parsing

$S \rightarrow aS$   
 $S \rightarrow bA$   
 $A \rightarrow bA$   
 $A \rightarrow \lambda$

#### Top down



#### Bottom up



### All about Left Linear Grammar

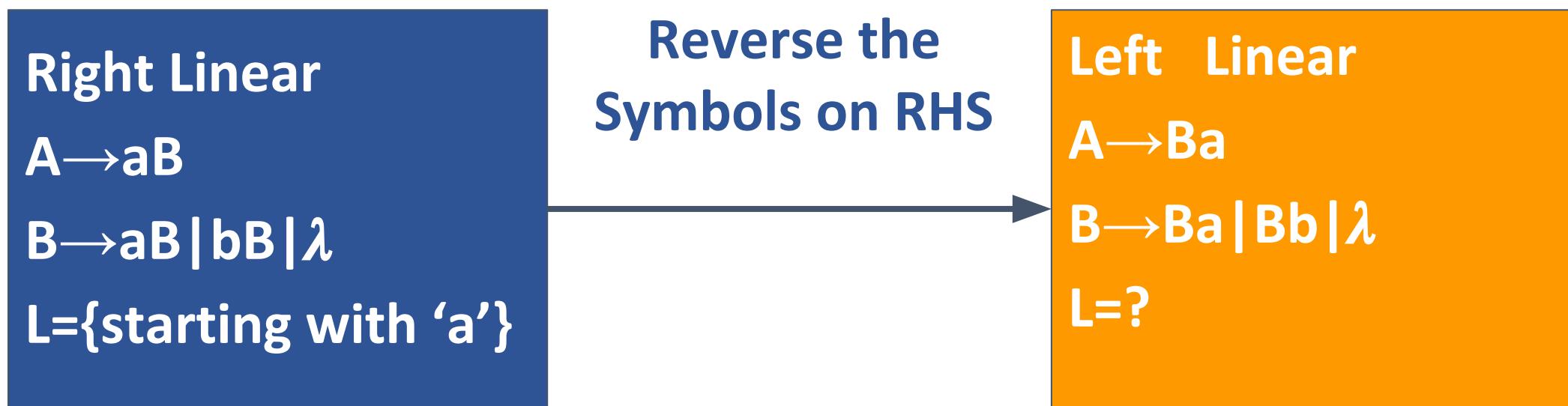
# Automata Formal Languages and Logic

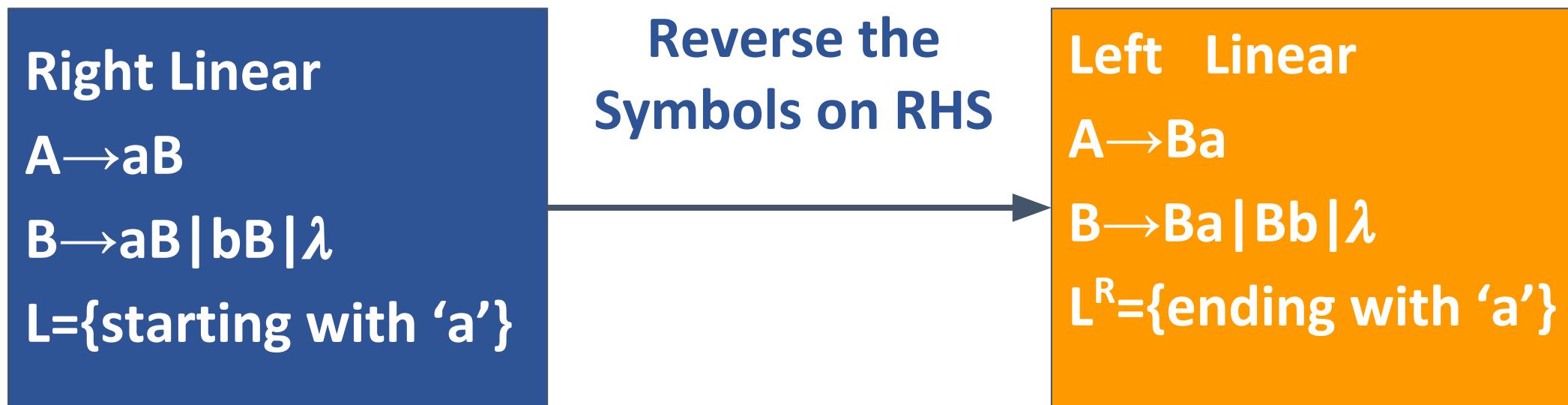
## Unit 2 - Constructing a Left Linear Grammar



# Automata Formal Languages and Logic

## Unit 2 - Constructing a Left Linear Grammar





When we reverse the RHS in every production in the right linear grammar for "L" we get a left linear grammar which will represent  $L^R$ .

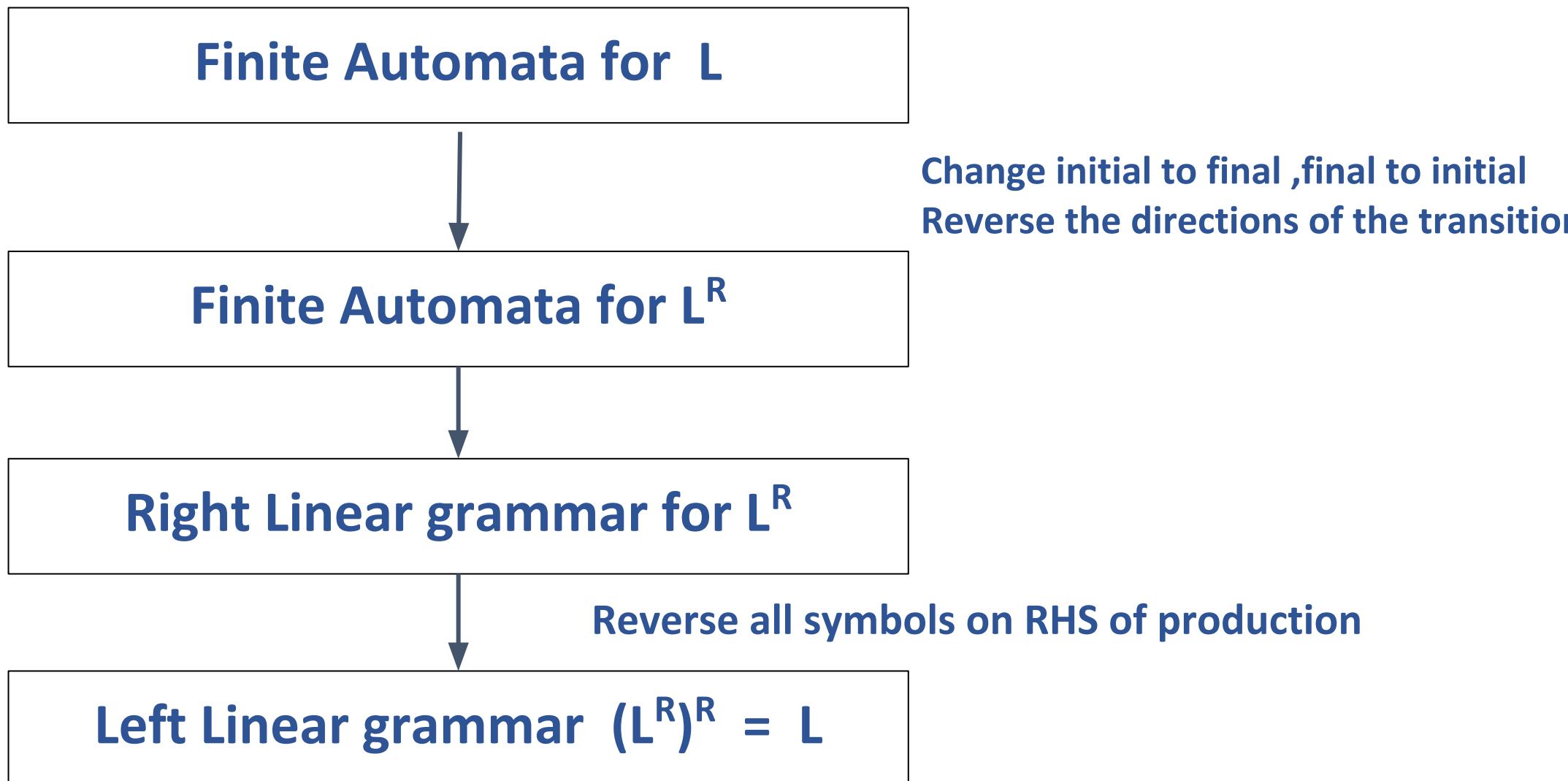
# Automata Formal Languages and Logic

## Unit 2 - Converting Finite automata to Left Linear Grammar

---



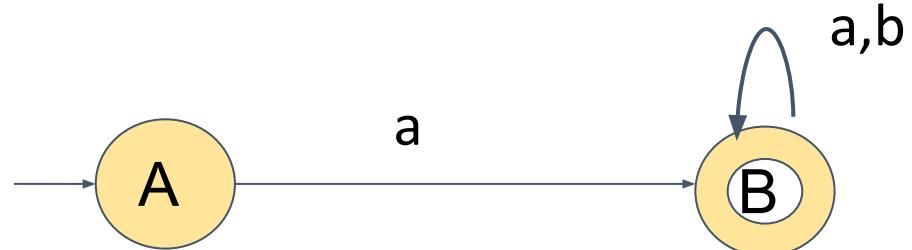
### Converting Finite Automata to Left linear grammar



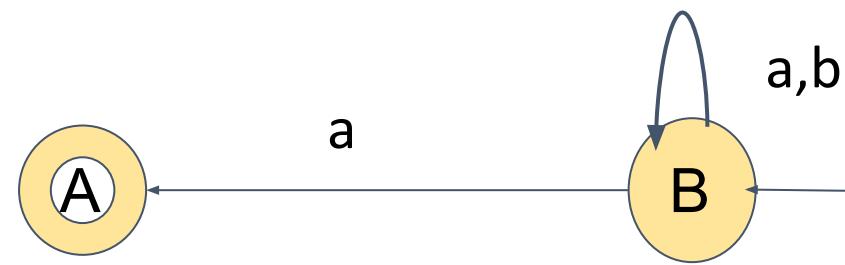
# Automata Formal Languages and Logic

## Unit 2 - Constructing a Left Linear Grammar

FA for  $L = \{aw, w \in \{a,b\}^*\}$



FA for  $L^R = \{wa, w \in \{a,b\}^*\}$



RLG for  $L^R$

Right Linear Grammar

$B \rightarrow aB \mid bB \mid aA$

$A \rightarrow \lambda$

LLG for  $(L^R)^R = L$

Left Linear Grammar

$B \rightarrow Ba \mid Bb \mid Aa$

$A \rightarrow \lambda$

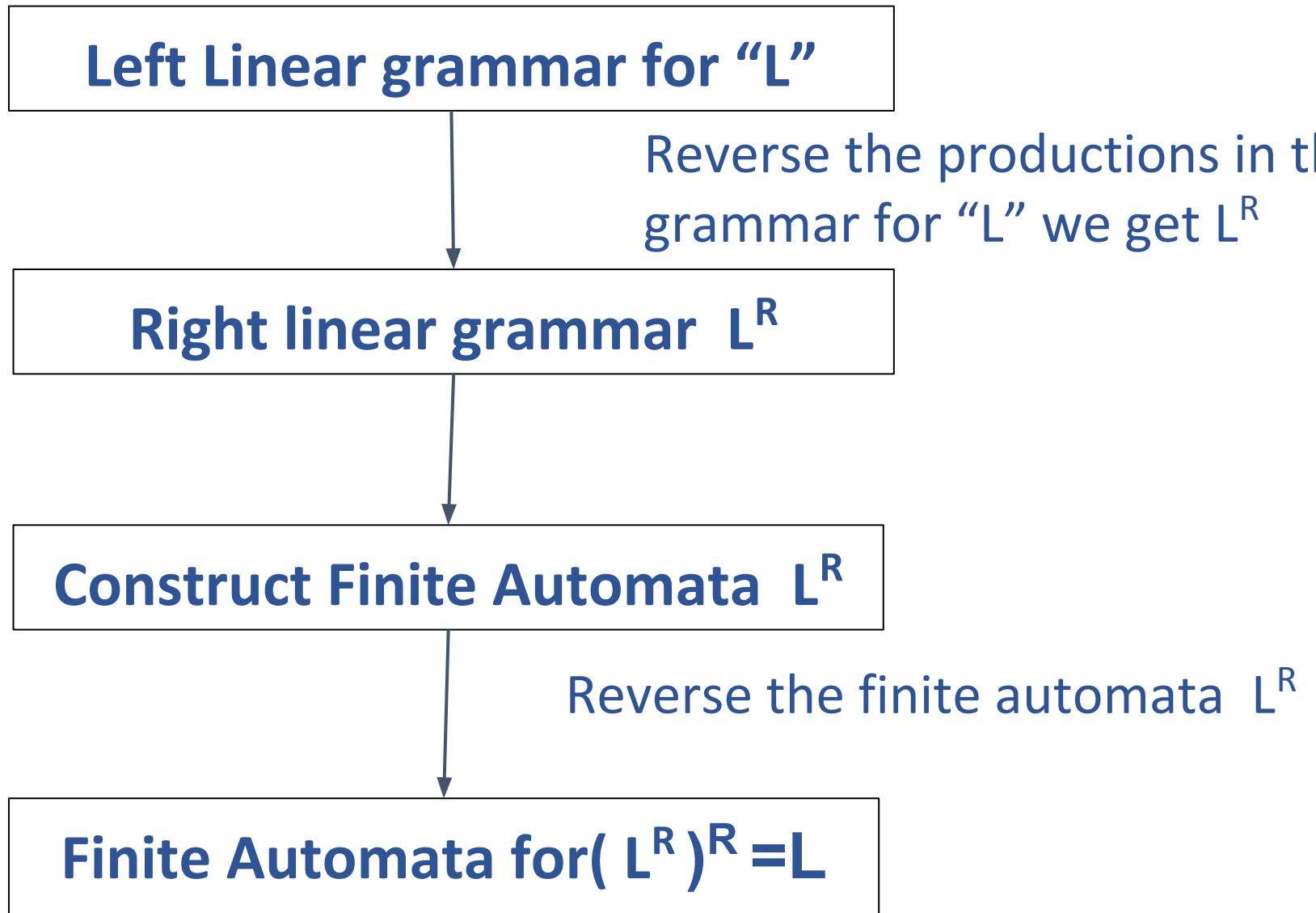
# Automata Formal Languages and Logic

## Unit 2 - Converting Left Linear Grammar to Finite automata

---



### Converting Left linear grammar to Finite automata



LLG for L

Left Linear Grammar

$$B \rightarrow Ba \mid Bb \mid Aa$$

$$A \rightarrow \lambda$$

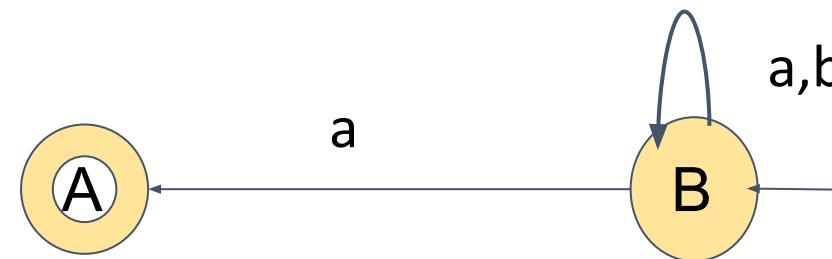
RLG for  $L^R$

Right Linear Grammar

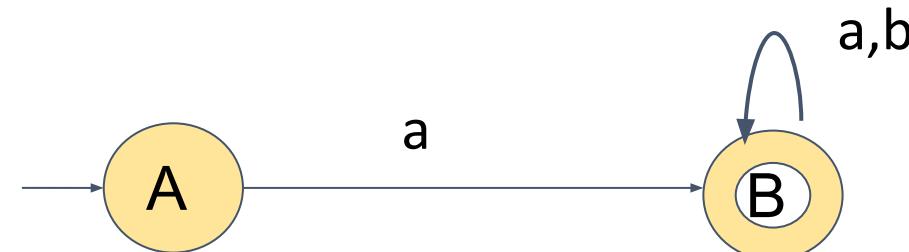
$$B \rightarrow aB \mid bB \mid aA$$

$$A \rightarrow \lambda$$

FA for  $L^R = \{wa, w \in \{a,b\}^*\}$



FA for  $(L^R)^R = L = \{aw, w \in \{a,b\}^*\}$



# Automata Formal Languages and Logic

## Unit 2 - LLG or RLG?

---



**Which one is easier LLG or RLG??**

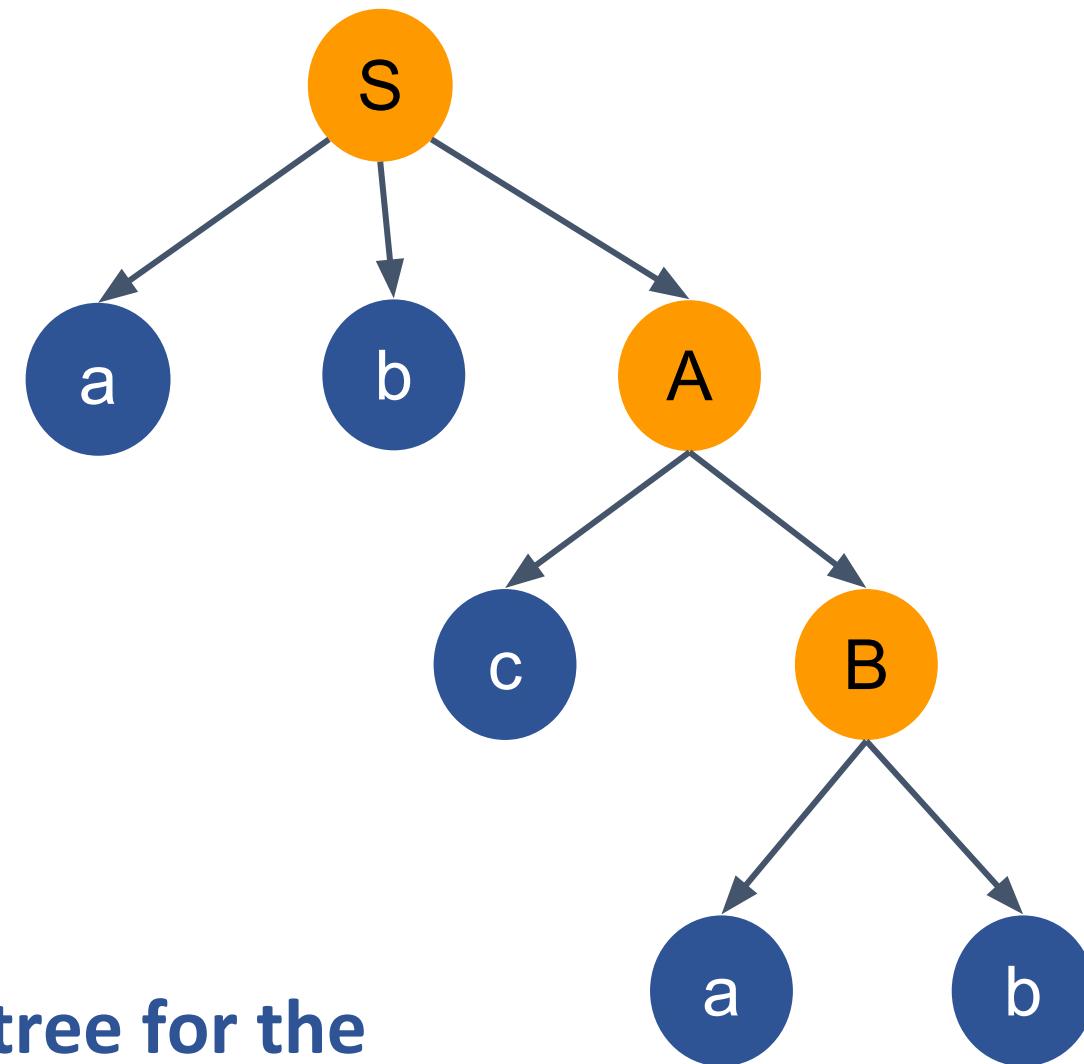
### Right Linear

$S \rightarrow abA$

$A \rightarrow cB|aC$

$B \rightarrow ab$

$C \rightarrow b$



Construct Parse tree for the String “abcab”

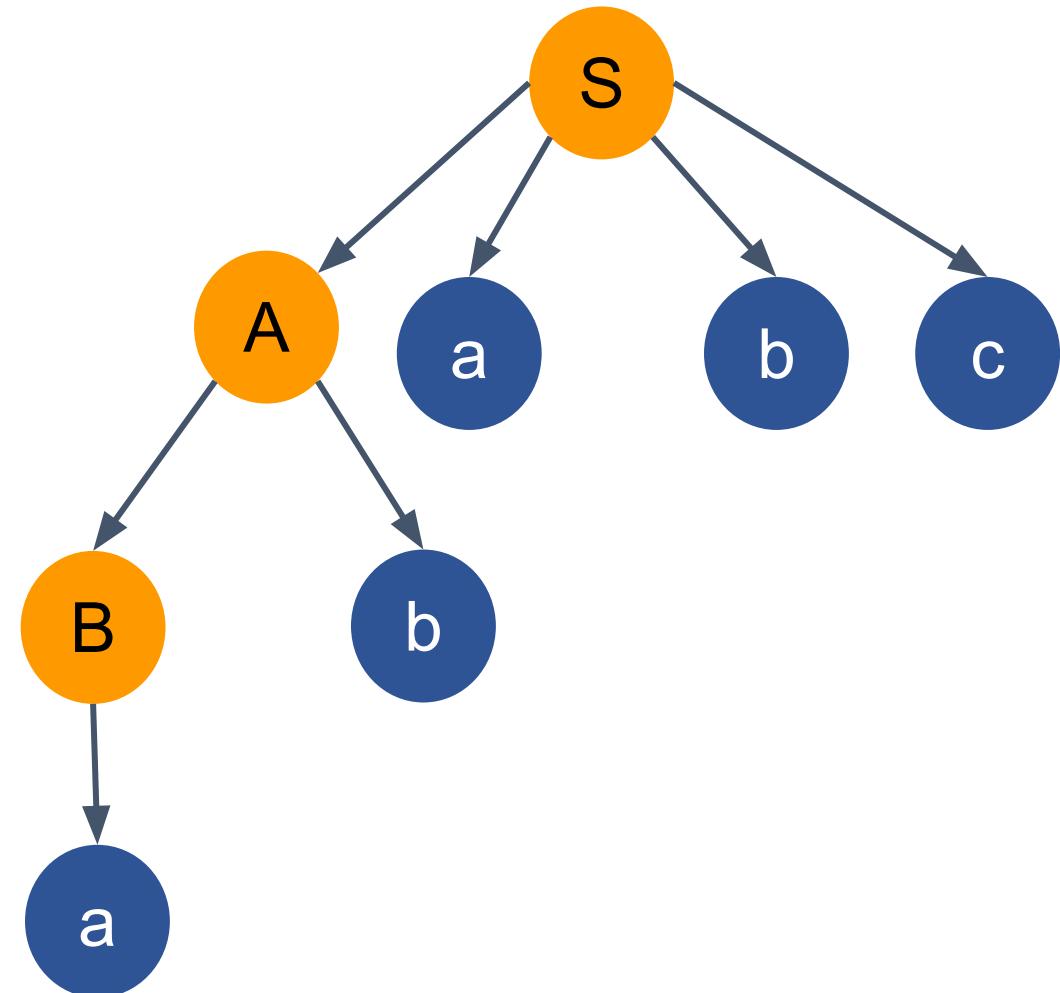
### Left Linear

$$S \rightarrow Aabc$$

$$A \rightarrow Bb|C$$

$$B \rightarrow a$$

$$C \rightarrow b$$



Construct Parse tree for the String “ababc”



# THANK YOU

---

**Preet Kanwal**

Department of Computer Science & Engineering

**[preetkanwal@pes.edu](mailto:preetkanwal@pes.edu)**

**+91 80 6666 3333 Extn 724**