Video chapters

- **Chapter-1 (Basic Concepts and Automata Theory):** Introduction to Theory of Computation- Automata, Computability and Complexity, Alphabet, Symbol, String, Formal Languages, Deterministic Finite Automaton (DFA)- Definition, Representation, Acceptability of a String and Language, Non Deterministic Finite Automaton (NFA), Equivalence of DFA and NFA, NFA with ε-Transition, Equivalence of NFA's with and without ε-Transition, Finite Automata with output- Moore Machine, Mealy Machine, Equivalence of Moore and Mealy Machine, Minimization of Finite Automata.
- **Chapter-2 (Regular Expressions and Languages):** Regular Expressions, Transition Graph, Kleen's Theorem, Finite Automata and Regular Expression- Arden's theorem, Algebraic Method Using Arden's Theorem, Regular and Non-Regular Languages- Closure properties of Regular Languages, Pigeonhole Principle, Pumping Lemma, Application of Pumping Lemma, Decidability- Decision properties, Finite Automata and Regular Languages
- **Chapter-3 (Regular and Non-Regular Grammars):** Context Free Grammar(CFG)-Definition, Derivations, Languages, Derivation Trees and Ambiguity, Regular Grammars-Right Linear and Left Linear grammars, Conversion of FA into CFG and Regular grammar into FA, Simplification of CFG, Normal Forms- Chomsky Normal Form(CNF), Greibach Normal Form (GNF), Chomsky Hierarchy, Programming problems based on the properties of CFGs.
- **Chapter-4 (Push Down Automata and Properties of Context Free Languages):** Nondeterministic Pushdown Automata (NPDA)- Definition, Moves, A Language Accepted by NPDA, Deterministic Pushdown Automata(DPDA) and Deterministic Context free Languages(DCFL), Pushdown Automata for Context Free Languages, Context Free grammars for Pushdown Automata, Two stack Pushdown Automata, Pumping Lemma for CFL, Closure properties of CFL, Decision Problems of CFL, Programming problems based on the properties of CFLs.
- **Chapter-5 (Turing Machines and Recursive Function Theory):** Basic Turing Machine Model, Representation of Turing Machines, Language Acceptability of Turing Machines, Techniques for Turing Machine Construction, Modifications of Turing Machine, Turing Machine as Computer of Integer Functions, Universal Turing machine, Linear Bounded Automata, Church's Thesis, Recursive and Recursively Enumerable language, Halting Problem, Post's Correspondance Problem, Introduction to Recursive Function Theory.

- **<u>Chapter-1 (Basic Concepts and Automata Theory)</u>:** Introduction to Theory of Computation- Automata, Computability and Complexity, Alphabet, Symbol, String, Formal Languages, Deterministic Finite Automaton (DFA)- Definition, Representation, Acceptability of a String and Language, Non Deterministic Finite Automaton (NFA), Equivalence of DFA and NFA, NFA with ε-Transition, Equivalence of NFA's with and without ε-Transition, Finite Automata with output- Moore Machine, Mealy Machine, Equivalence of Moore and Mealy Machine, Minimization of Finite Automata.

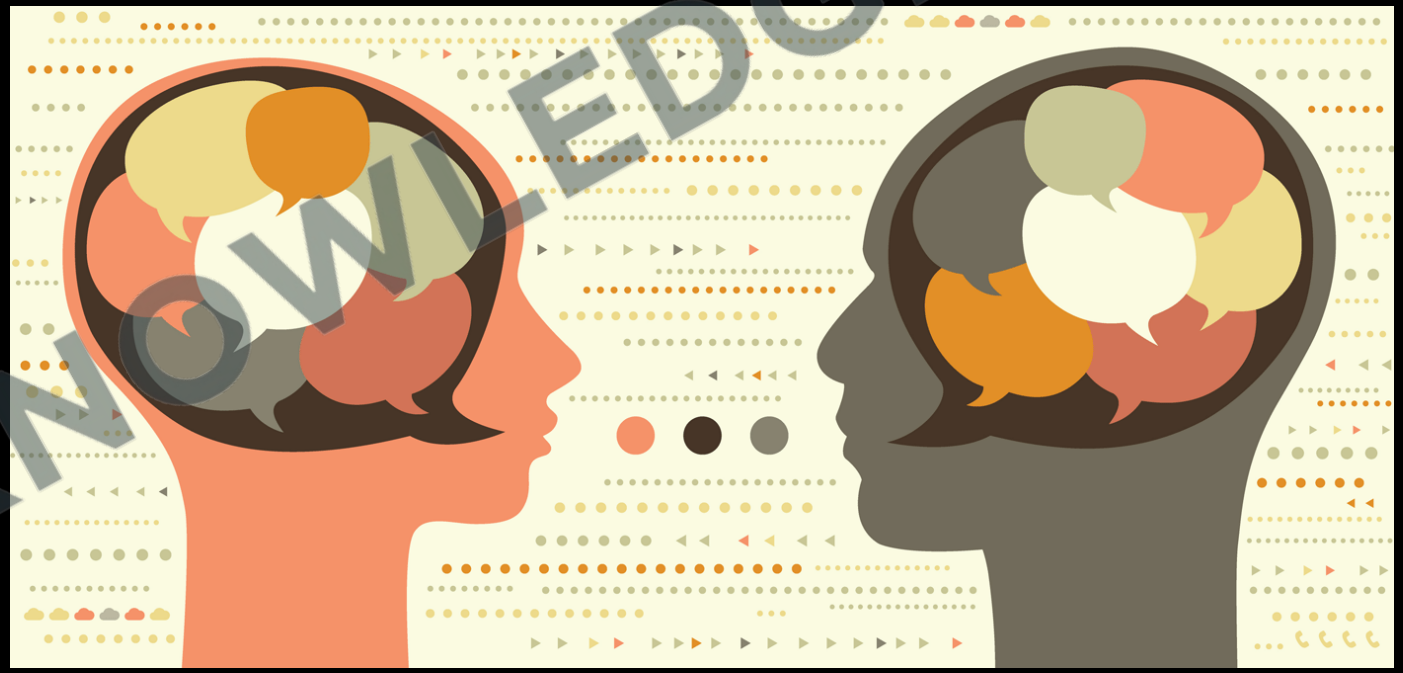
INTRODUCTION TO THEORY OF COMPUTATIONS

- The theory of computation is a branch of theoretical computer science that deals with the study of algorithms and computational complexity. It aims to answer fundamental questions about
  - what can be computed
  - how efficiently it can be done
  - and what limitations exist in terms of computational power.

- As word suggests ‘TOC’ is the study of 'mathematical' machines or systems called automata.
- Theory of computation can be considered as the study of all kinds of computational model in the field of computer science and it also considers how efficiently the problem can be solved (but not is depth).

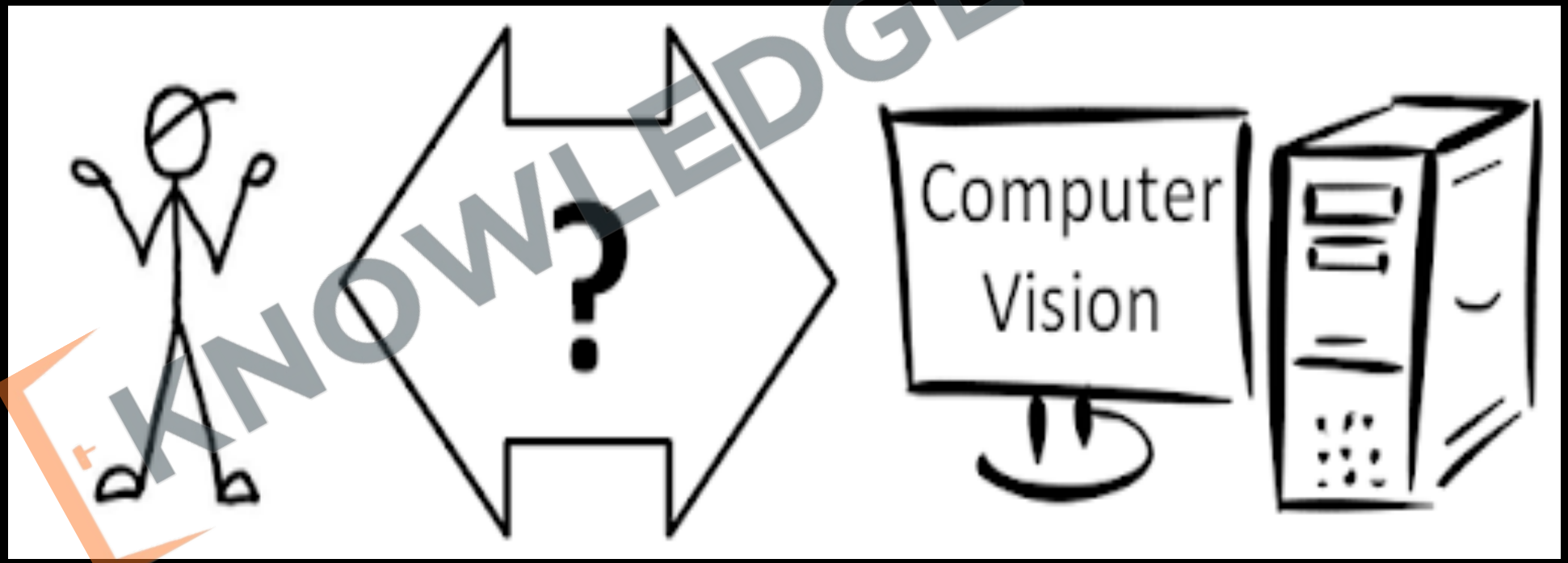
PROBLEM

- Now a day's machines (digital, analog, mechanical) play a very important role in the development of human, we need some mechanism (language) to communicate with the machines.



SOLUTION

- We need a language for communication with machines. But we do not require natural languages to communicate with the machines, as natural languages are very complex and machine interaction require very fewer complex languages compare to natural languages.

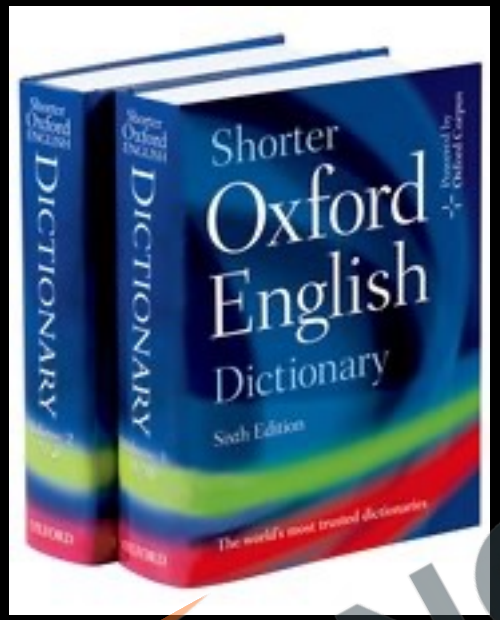


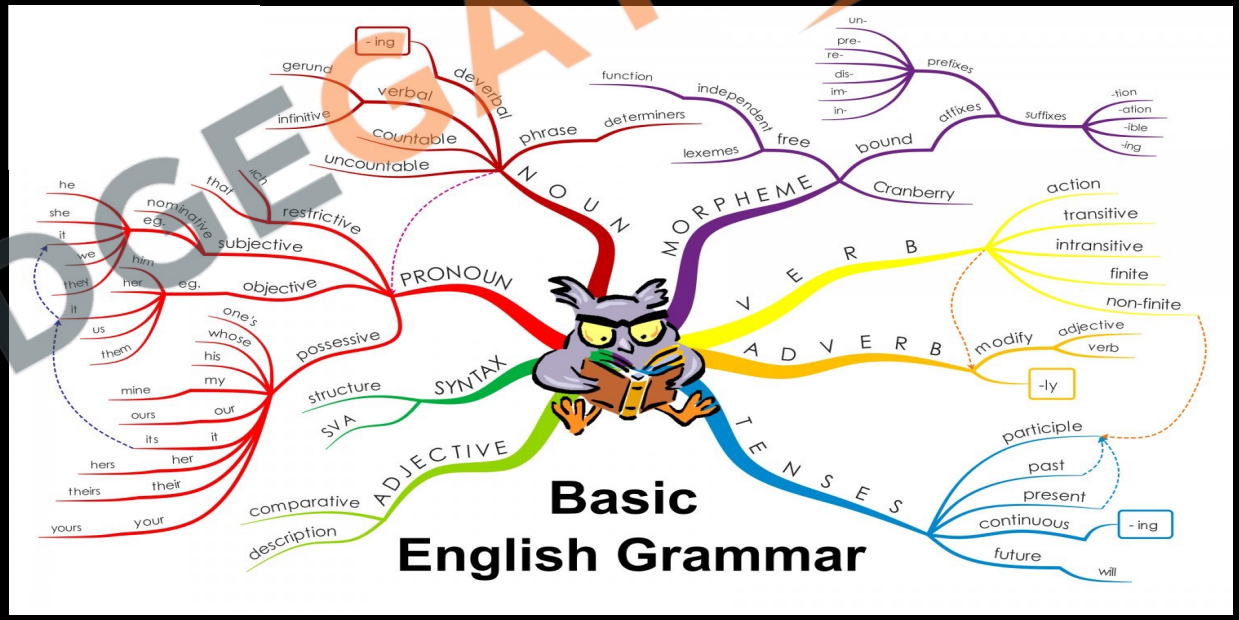
- Languages can be of two types formal languages and informal languages, here in this subject we will only discuss formal languages.
- Dictionary defines the term informally as ‘a system suitable for the expression of certain ideas, facts or concepts including a set of symbols for their manipulation’.

http://www.knowledgegate.in/gate

METHODS TO DEFINE LANGUAGE

- In natural language we define the list of words in a dictionary because they are finite and predefined, but we cannot list all the sentence which can be formed using these words as they are infinite.





- So, we have a mechanism called grammar/rules using which we can decide which sentence is valid and which is invalid.

MATHEMATICL DEFINATION OF LANGUAGE

- **SYMBOL**- Symbols are the basic building blocks, which can be any character/token. (cow, sheep,  , white flag,  ,  ,  etc.) (in English we called them as letters).

- **ALPHABET**- An alphabet is a finite non empty set of symbols, (every language has its own alphabet). here in toc, we use symbol Σ for depicting alphabet. e.g. Σ = {0,1}. for English Σ = {a, b, c, …., z} (in English also alphabet is a set of letters, thought in general we called them as alphabet).

- **STRING** - It is a finite sequence of symbols (which are the member of set alphabet). E.g. Σ = {a, b} String- aabb, aa, b, so on. (in English we called them as words).

- **LANGUAGE** - A language is defined as a set of strings. (in natural language (set of words(predefined) and grammar) we apply this model from words to sentence).
- In the next level we consider programs as a string and programming constructs/tokens like int, floats as letters/symbols.

- Similarly, in our system we have finite number of symbols/letters but using those letters we can generate infinite strings/words.
- So, we may have languages that have infinite number of words, so it is not possible for us to list them, we have to use some framework, which can somehow represent the same language. There are mainly two methods to represent a language
  - by a grammar that generates a language [RG generate RL]
  - by a machine that accepts a language [FA accept RL language]

Machine Grammar

- The Theory of Formal Languages is a branch of theoretical computer science that deals with the study of formal languages and their properties.
- The theory of formal languages includes the study of formal grammars, which are used to define the syntax of formal languages, and automata theory, which deals with the study of abstract machines used to recognize, generate, or process formal languages. Some of the key concepts studied in the theory of formal languages include regular languages, context-free languages, context-sensitive languages, recursively enumerable languages, and the Chomsky hierarchy.
- The theory of formal languages has important applications in areas such as compilers, parsers, and other software engineering tools, as well as in the design of programming languages and the study of natural language processing.

Q if $\sum = \{a, b\}$ then, find the following?

$\sum^0 =$

$\sum^1 =$

$\sum^2 =$

$\sum^3 =$

- $\sum^K$ is the set of all the strings from the alphabet $\sum$ of length exactly K.

- $\sum^k = \{W \mid |W| = K\}$ (using the symbols from the alphabet $\sum$)

Kleene closure- If ∑ is a set of symbols, then we use ∑* to denote the set of strings obtained by concatenating zero or more symbols from ∑ of any length, in general any string of any length which can have only symbols specified in ∑.

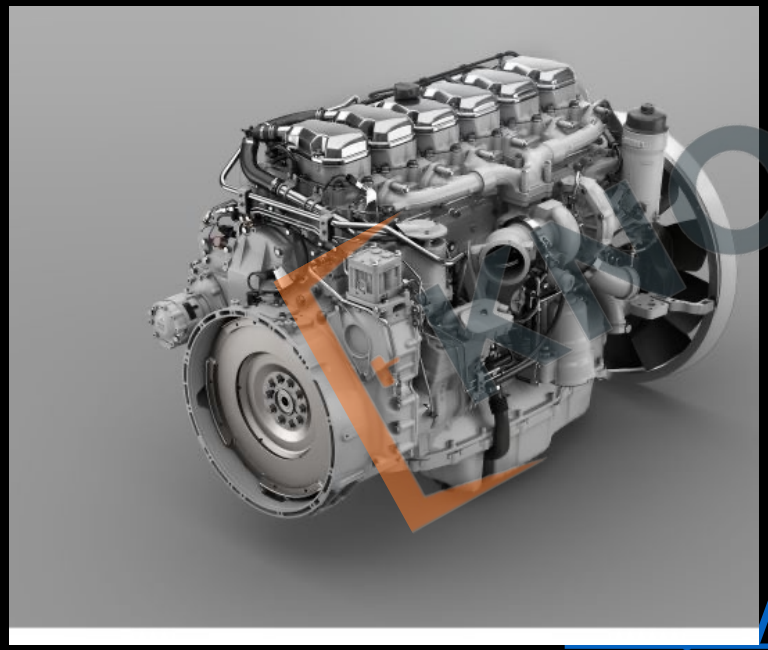
- $\sum^* = \bigcup_{i=0}^{i=\infty}\{w \mid |w| = i\}$ (using the symbols from the alphabet ∑)

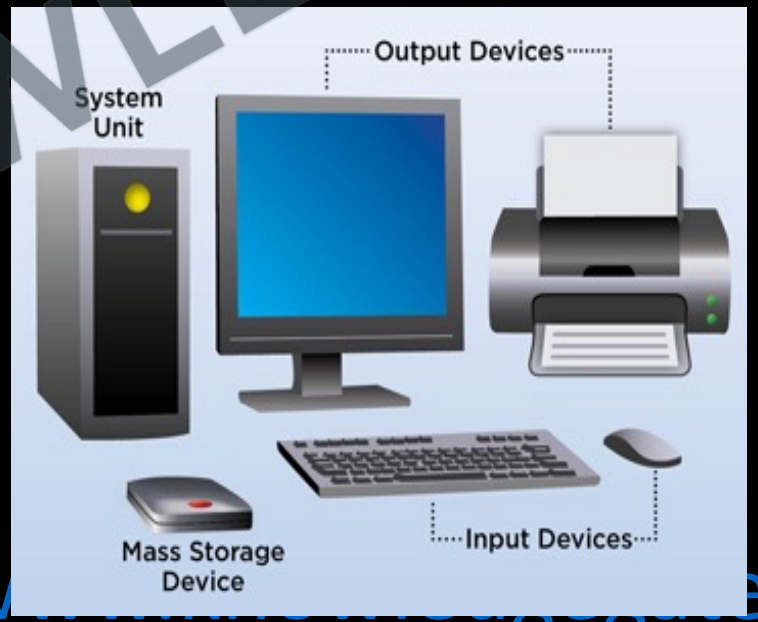
Positive closure – If ∑ is a set of symbols, then we use $\sum^+$ to denote the set of strings obtained by concatenating one or more symbols from ∑ of any length, in general any string of any length which can have only symbols specified in ∑ (except ∈).

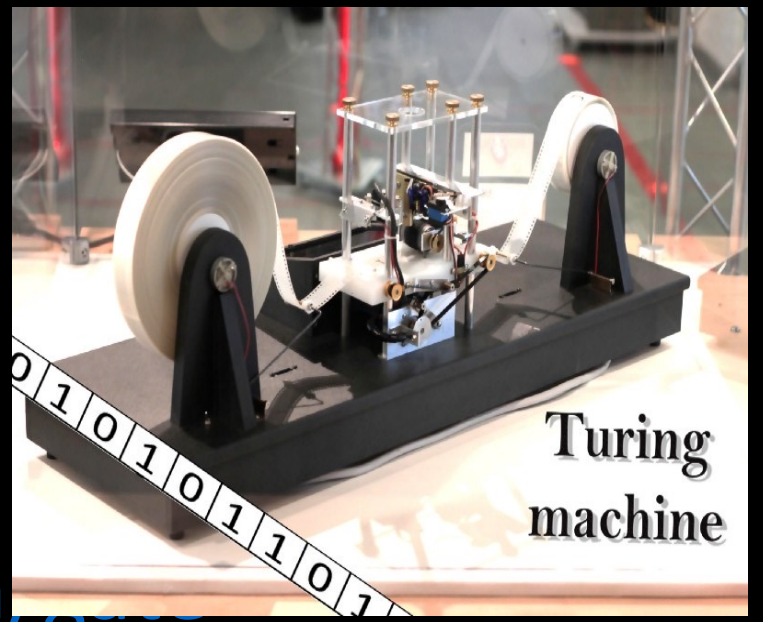
- $\sum^+ = \bigcup_{i=1}^{i=\infty}\{w \mid |w| = i\}$ (using the symbols from the alphabet ∑)

Automaton

- An automaton is defined as a self-operating system where energy, materials and information are transformed, transmitted and used for performing some functions without direct participation of man.
- The term is often used to describe a theoretical machine that operates according to a set of rules and is capable of carrying out complex operations without human intervention.
- Example: automatic machine tools, automatic packing machines, and automatic photo printing machines.



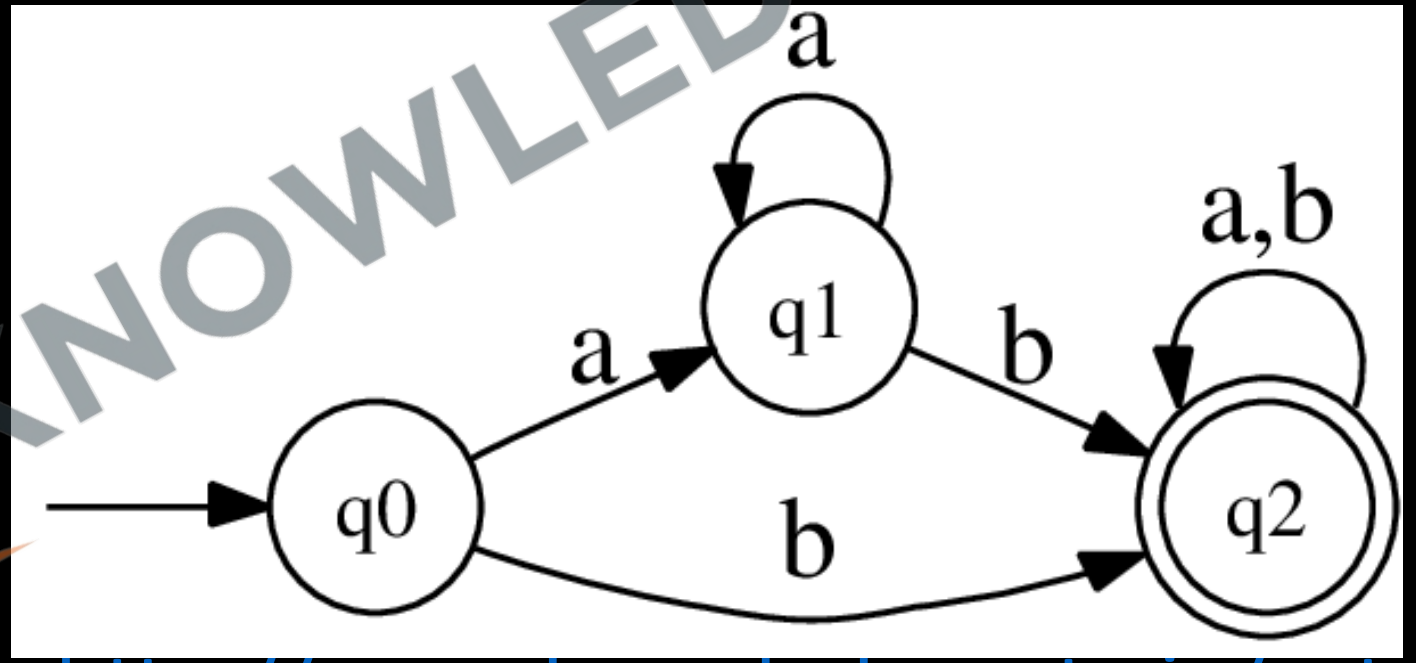




FINITE AUTOMATA

- A **Finite automaton** is a model that has a finite set of states (represented in the figure by circles) and its control moves from one state to another state in response to external inputs (represented by arrows).

- It is an abstract machine that is used to recognize patterns in strings of symbols. Finite automata are widely used in computer science for pattern matching, lexical analysis, and parsing, among other applications.

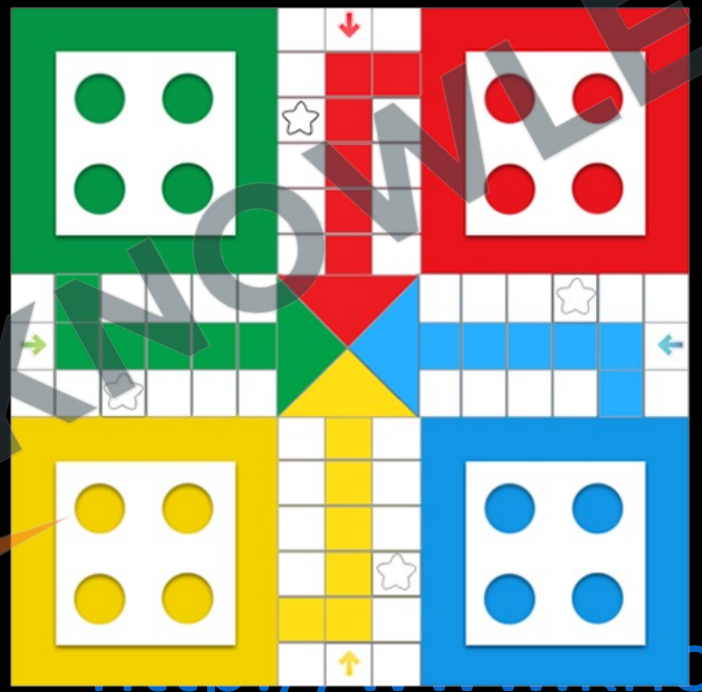


- Finite automata can be broadly classified into two types-

1. **Finite automata without output**
    1. Deterministic finite automata.
    2. Non deterministic finite automata.
    3. Non deterministic finite automata with $\in$

2. **Finite automata with output**
    1. Moore machine
    2. Mealy machine

- In general, this type of automata is characterized by machine having no temporary storage, as it is severely limited in its capacity to remember things during the computation.
- Only finite amount of information can be in the control unit by placing the unit into a specific state but since the number of states is finite, a finite automaton can only deal with situation in which the information to be stored at any time is strictly bounded.



DETERMINISTIC FINITE AUTOMATA

A **deterministic finite automaton (DFA)** is defined by 5-tuple (Q,Σ,δ,S,F) where:

- Q is a finite and non-empty set of states
- Σ is a finite non-empty set of finite input alphabet
- δ is a transition function, ($\delta: Q \times \Sigma \rightarrow Q$)
- S is **initial state** (always one) ($S \in Q$)
- F is a set of final states ($F \subseteq Q$) ($0<=|F|<=N$, where n is the number of states)

DFA Designing

- A language a said to be regular language if it can be accepted by a DFA.
- The best knowledge about DFA can be only understood by designing a number of DFA, by doing so we will first understand the process of DFA designing and secondly, we will understand Regular Language.

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b}, where every accepted string ‘w’ starts with substring s = ‘abb’

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b}. where every accepted string ‘w’ ends with substring ‘s = bab’ ?

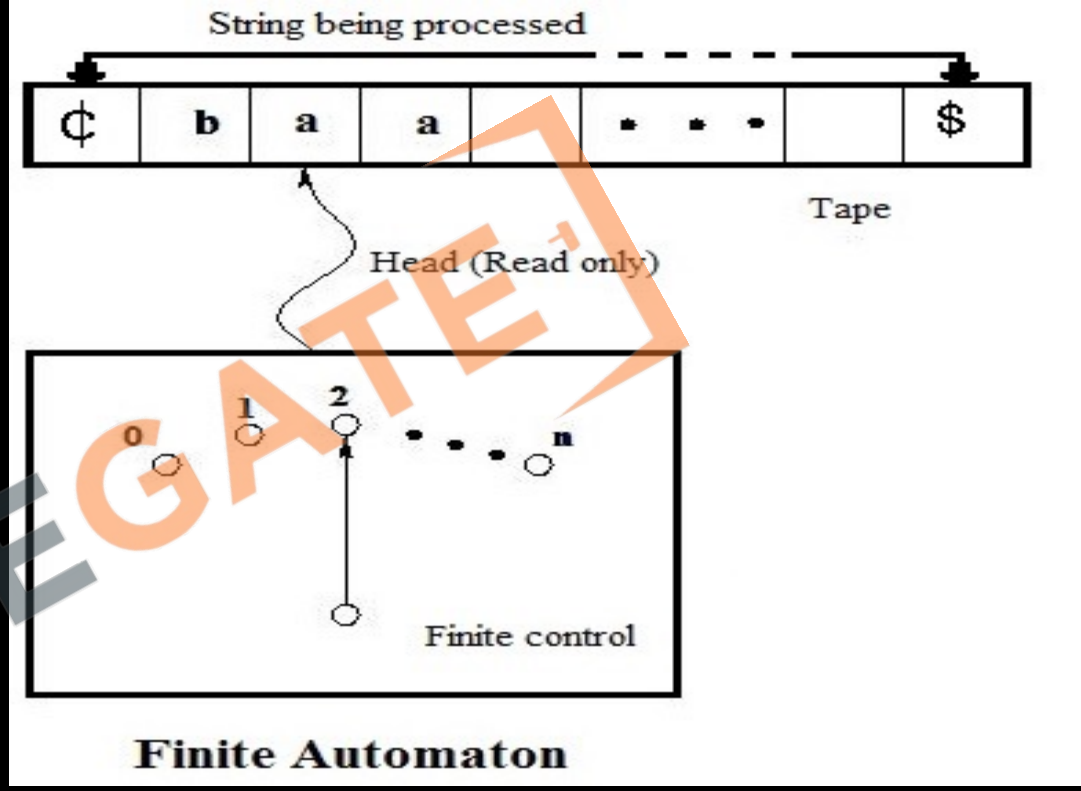
Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b}. where every accepted string ‘w’ contains sub string s= ‘aba’?

- The various components of the block diagram are explained as follows:

- **Input tape**: The input tape is divided into squares, each square containing a single symbol from the input alphabet ∑. The end squares of the tape contain the end marker ¢ at the left end and the end marker $ at the right end. The absence of end markers indicates that the tape is of infinite length. The left-to-right sequence of symbols between the two end markers is the input string to be processed.

- **Reading head (R-head)**: The head examines only one square at a time and will move one square to the right.

- **Finite control:** is the inference engine take care of transition



Finite Automaton

NOTE-

- Produces a unique computation (or run) of the automaton for each input string.
- Deterministic refers to the uniqueness of the computation.
- DFAs are useful for doing lexical analysis (Spell check) in compiler design.

Representation

TRANSITION STATE DIAGRAM- (Graphical, easy to understand, easy to design) A transition graph or a transition system is a finite directed labelled graph in which each circle represents a state and the directed edges indicate the transition of one state to another state. The initial state is represented by a circle with an arrow pointing towards it, the final state by two concentric circles.

δ	Σ	
	a	b
Q	q_0	
	q_1	

TRANSITION TABLE- it is a two-dimensional table where number of columns is equal to number of input alphabets and number of rows is equal to number of states.

TRANSITION ID- $\delta \{q_i, a) = q_j$, this means that on a state q_i take an input symbol a machine will make a transition q_j

ACCEPTANCE BY DFA

- Let ‘w’ be any string designed from the alphabet Σ, corresponding to w, if there exist a transition for which it starts at the initial state and ends in any One of the final states, then the string ‘w’ is said to be accepted by the finite automata. $\delta^*(q_0, w) = q_f$ for some $q_f \in F$.

- Mathematically, it can be represented as: - $L(M) = \{w \in \Sigma^* \mid \delta^*(S, w) \in F\}$

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b} such that every accepted string start and end with ‘a’?

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b} such that every accepted string start and end with same symbol?

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b} such that every accepted string start and end with different symbol?

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b} such that every accepted string w, is like w=SX. S=aaa/bbb?

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b} such that every accepted string w, is like w=XS. S=aaa/bbb?

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b} such that every accepted string w, is like w=XSX. S=aaa/bbb?

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b}, such that every string ‘w’ accepted must be like

i) |w| = 3

ii) |w|<=3

iii) |w|>=3

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b}, such that every string accepted must contain exactly two a's, $|w|_a=2$?

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b}, such that every string accepted must contain at least two a's, $|w|_a >= 2$?

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b}, such that every string accepted must contain at most two a's, $|w|_a <= 2$?

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b}, such that every string ‘w’ accepted must be like

i) |w| = 0(mod 3)

ii) |w|=1(mod 4)

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b}, such that every string accepted must contain

i) number of a is, $|w|_a = 0 \pmod 2$

iii) number of a is $|w|_b = 2 \pmod 3$

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b} such that for every accepted string 2^{th} from left end is always a ?

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b} such that for every accepted string 2nd from right end is always b ?

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b}, such that every string accepted must contain be like,
no of a = 0(mod 2) || no of b = 0(mod 2) ?

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {0,1}, such that every string ‘w’ which is accepted has a decimal equivalent 0(mod 3) ?

COMPLIMENT OF DFA

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b}, such that every string accepted must not contain a substring ’aaa’?

NON DETEMINISTIC FINITE AUTOMATA

- Non determinism means choice of move for an automaton. So rather than prescribing a unique move in each situation, we allow a set of possible moves.
- Non deterministic machine are only theoretical machine i.e. in the first place they are not implementable and neither we want to implement them, the only reason we study non-determinism is because they are easy to design and easily be converted into deterministic machine.

FORMAL DESCRIPTION OF NDFA

- A Non-Deterministic finite automaton (NDFA) is a 5-tuple (Q,Σ,δ,S,F) where:
  - Q is a finite and non-empty set of states
  - Σ is a finite non-empty set of finite input alphabet
  - δ is a transition function $\delta: Q \times \Sigma \rightarrow 2^Q$
  - q_0 is **initial state** (always one) ($q_0 \in Q$)
  - F is a set of final states ($F \subseteq Q$) ($0<=|F|<=N$), where n is the number of states

Some points to remember

- Every DFA is also an NFA.
- Every NFA can be translated to an equivalent DFA, so their language accepting capability is same.
- NFAs like DFA's only recognize ***regular languages***.
- It need not to be a complete system. There can be a state that doesn't have any transition on some input symbol.
- It is possible that a single state led to multiple transition on same input to different states.
- NOTE- A null transition is also possible for NFA, such special NFA are called Null-NFA. We will discuss it later.

PROPERTIES OF NFA

- i) Accepting power of NDFA= Accepting power of DFA.
- ii) NDFA is a theoretical engine and is not implementable, but it is very easy to design compare to DFA.
- iii) No concept of dead state, therefore complementation of DFA is also not possible.
- iv) NDFA will respond for only valid strings and no need to respond for invalid strings. (it is a Incomplete system)

ACCEPTANCE BY NDFA

- Let ‘w’ be any string defined over the alphabet Σ, corresponding to w, there can be multiple transitions for NFA starting from initial state, if there exist at least one transition for which we start at the initial state and ends in any One of the final state, then the string ‘w’ is said to be accepted by the non-deterministic finite automata, otherwise not.

- Mathematically, it can be represented as, $L(M) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$

Q Design a NDFA that accepts all strings over the alphabet ∑ = {a, b}, where every accepted string ‘w’ starts with substring s, Where s = ‘aba’ ?

Q Design a NDFA that accepts all strings over the alphabet ∑ = {a, b}. where every accepted string ‘w’ ends with substring ‘s’, Where s = ‘bab’?

Q Design a minimal DFA that accepts all strings over the alphabet ∑ = {a, b}. where every accepted string ‘w’ contains sub string s, Where s = ’aba’ ?

Q Design a NDFA that accepts all strings over the alphabet ∑ = {a, b} such that every accepted string start and end with same symbol ?

Q Design a NDFA that accepts all strings over the alphabet ∑ = {a, b} such that every accepted string start and end with different symbol ?

Q Design a minimal NDFA that accepts all strings over the alphabet ∑ = {a, b} such that every accepted string w, is like w=SX , Where s = aaa/bbb

Q Design a minimal NDFA that accepts all strings over the alphabet ∑ = {a, b} such that every accepted string w, is like w=XS , Where s = aaa/bbb

Q Design a minimal NDFA that accepts all strings over the alphabet ∑ = {a, b} such that every accepted string w, is like w=XSX , Where s = aaa/bbb ?

Q Design a NDFA that accepts all strings over the alphabet ∑ = {a, b} such that for every accepted string 3rd from right end is always a ?

NFA and DFA Equivalence

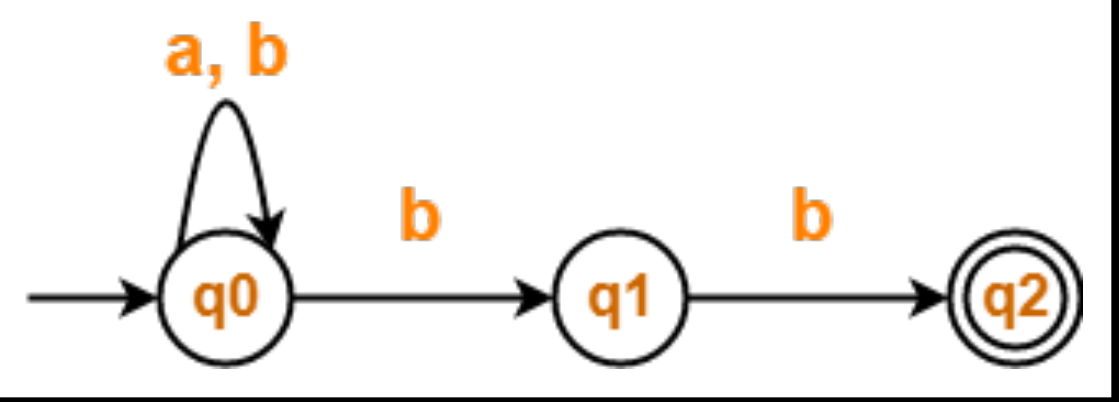
- In this topic we will be learning about the equivalence of NFA and DFA and how an NFA can be converted to equivalent DFA. Let us take an example and understand the conversion.
- Since every NFA and DFA has equal power that means, for every language if a NFA is possible, then DFA is also possible.
- So, every NFA can be converted to DFA.
- The process of conversion of an NFA into a DFA is called Subset Construction.

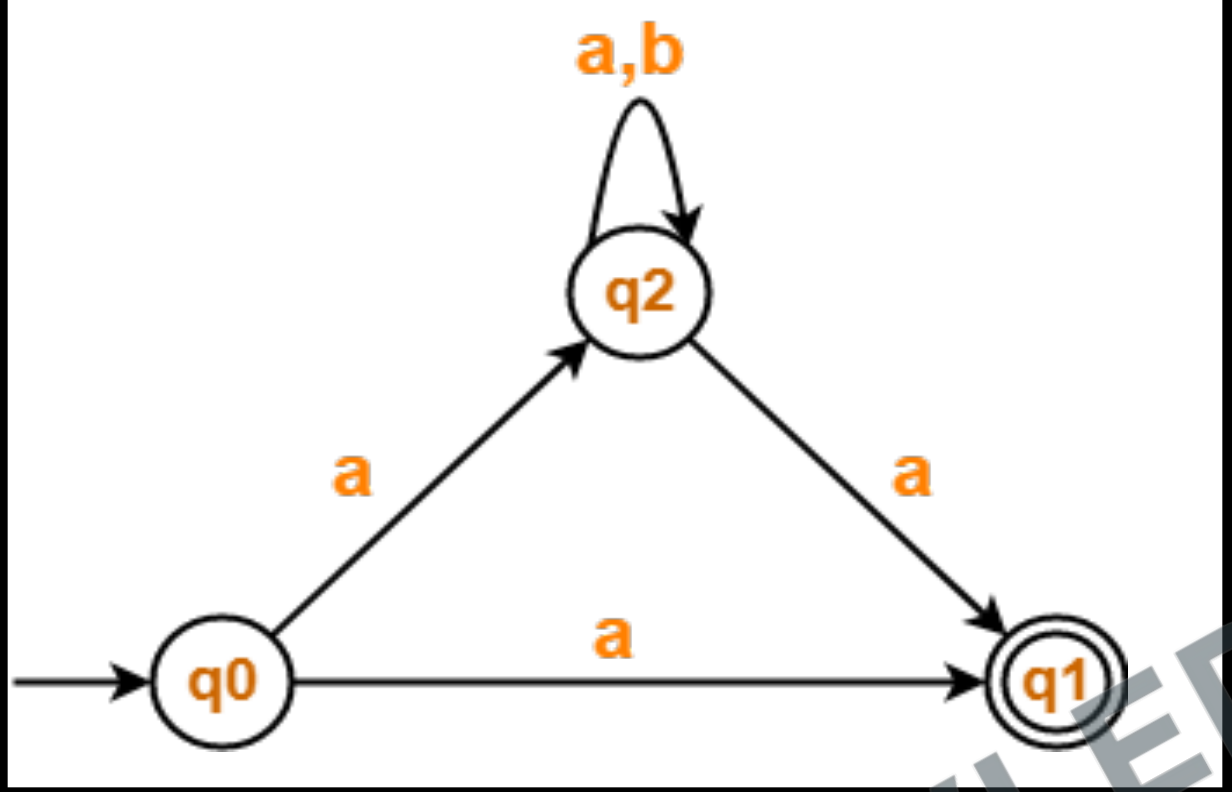
- If NFA have ‘n’ states which is converted into DFA which ‘m’ states than the relationship between n and m will be

- $1 <= m <= 2^n$

Procedure for Conversion

- There lies a fixed algorithm for the NFA and DFA conversion. Following things must be considered
  - Initial state will always remain same.
  - Start the construction of δ' with the initial state & continue for every new state that comes under the input column and terminate the process whenever no new state appears under the input column.
  - Every subset of states that contain the final state of the NFA is a final state in the resulting DFA.
  - $\delta'(q_0, q_1, q_2, q_{3\text{--------}}, q_{n-1}, a) = \bigcup_{i=0}^{i=n-1} \delta(q_i, a)$





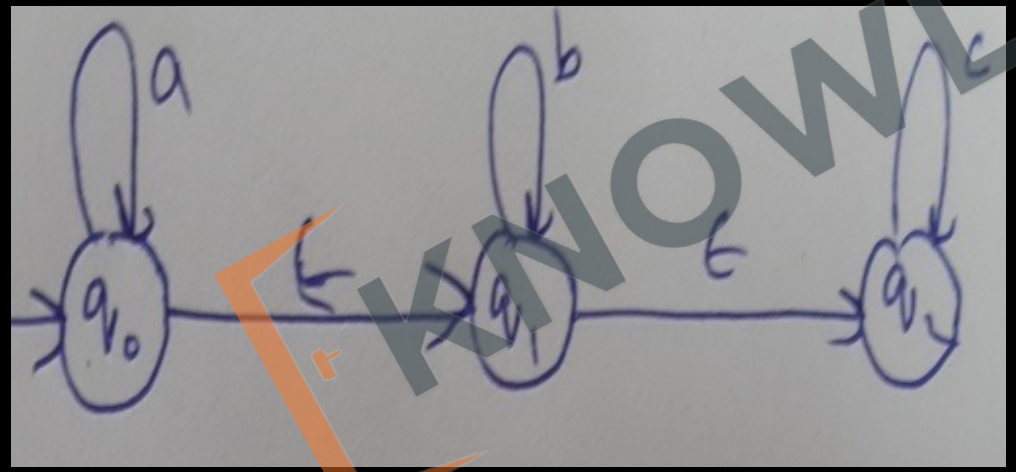
Aspect	DFA (Deterministic Finite Automata)	NDFA (Nondeterministic Finite Automata)
State Transition	On each input symbol, transitions to exactly one state.	Can transition to multiple states or none on the same input symbol.
Determinism and Uniqueness	Each state has a unique transition for each input symbol.	A state can have multiple transitions for the same input symbol.
Computation Path	Always has a single, unique computation path for any input string.	May have multiple computation paths for the same input string.
Ease of Construction	Generally simpler and more straightforward to construct.	Can be more complex to construct due to non-determinism.
Acceptance of Input	Accepts an input if it reaches a final state after processing all input symbols.	Accepts an input if at least one computation path reaches a final state.

NFA WITH EPSILON MOVES (€-NFA)

- An automaton that consist of null transitions is called a Null- NFA i.e. we allow a transition on null means empty string.
- €-NFA is a 5-tuple (Q,Σ,δ,S,F) where:
  - Q is a finite and non-empty set of states
  - Σ is a finite non-empty set of finite input alphabet
  - δ is a transition function $\delta: (Q \times \{\Sigma \cup \varepsilon\}) \rightarrow 2^Q$
  - S is **initial state** (always one) $(S \in Q)$
  - F is a set of final states $(F \subseteq Q)$ $(0<=|F|<=N$, where n is the number of states

NULL-CLOSURE

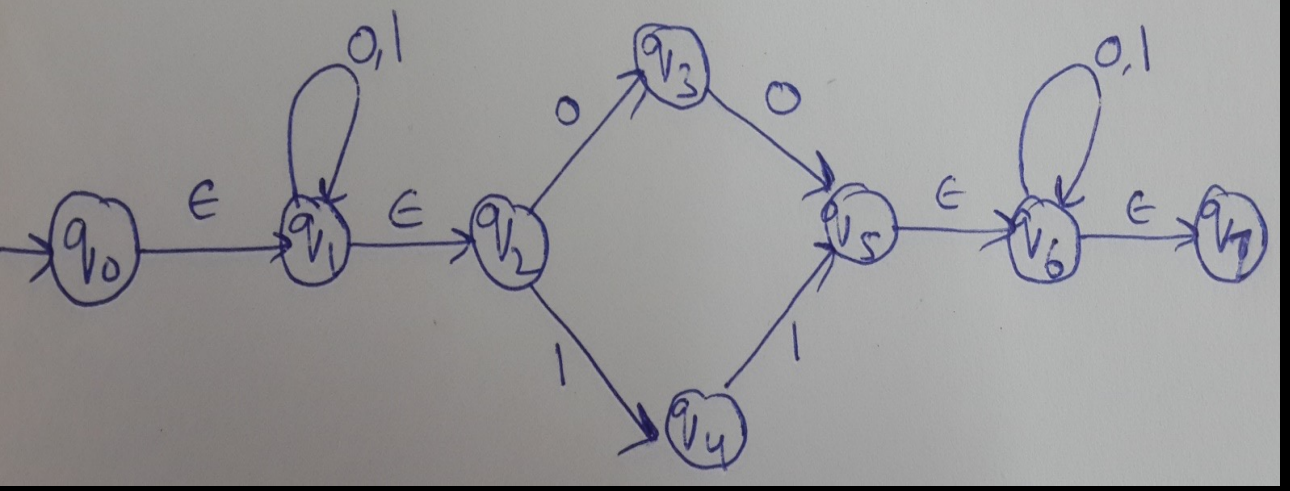
- Null closure of a set Q is defined as a set of all the states, which are at zero distance from the state Q. A set of all the states, that can be reached from the state and along a null- transition.
- **€-Closure(q_i)-** The set of all the states which are at zero distance from the state q_i is called €-closure(q_i). Or the set of all the states that can be reached from the state q_i along € labelled transition path, is known as €-closure(q_i).



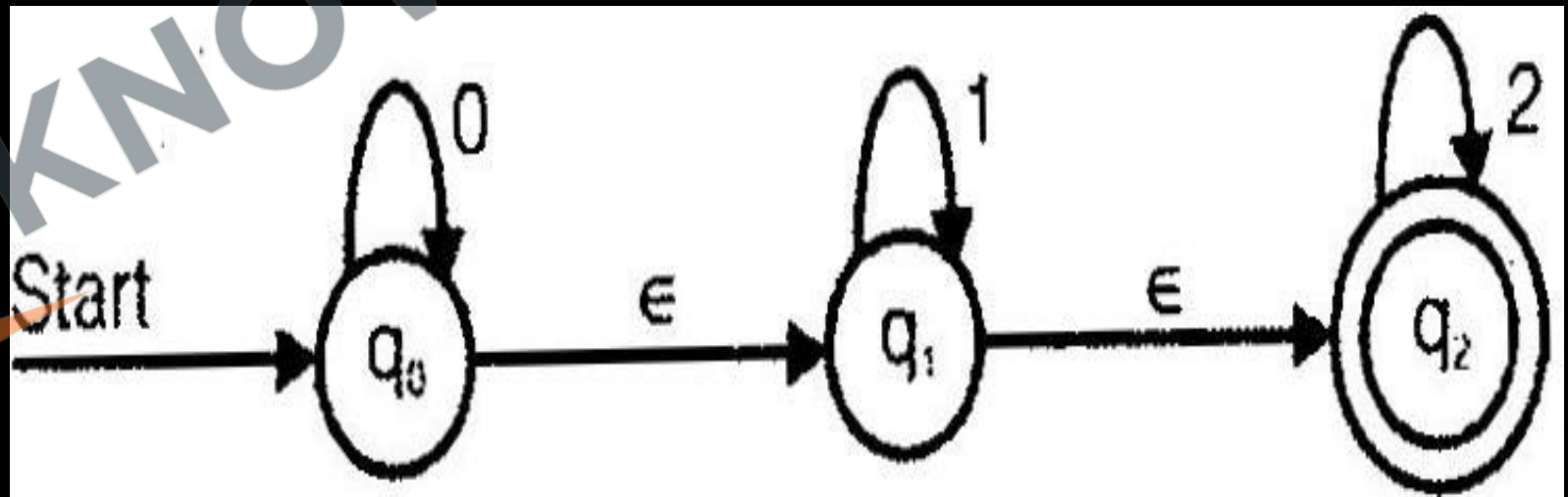
- Every state is at zero distance to itself.
- In NFA or DFA, distance between two states is always 1, because there could be no null transitions.
- The Null closure Q is always a non-empty and finite state, because every state's null closure is that state only.
- €-closure(Φ) = Φ
- €-closure (q_0 , q_1, q_2 , q_3--------, q_n) =$U_{i=0}^{i=n}$ δ(€-closure(q_i))

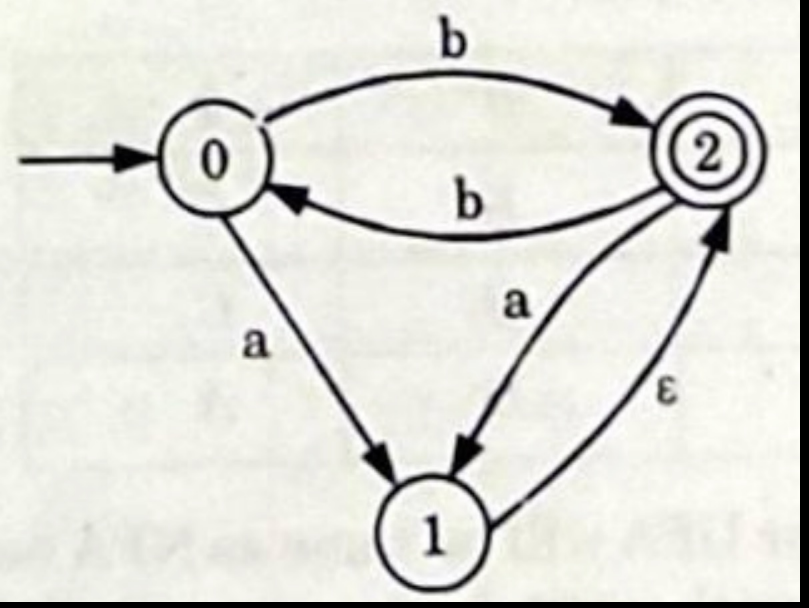
EQUIVALENCE BETWEEN NULL NFA TO NFA

- There will be no change in the initial state.
- No change in the total no. of states
- May be change in the number of final states.
- All the states will get the status of the final state in the resulting NFA, whose €-closure contains at least one final state in the initial €-NFA.



- Let us check for $\delta(q_0, a)$ = **ε-Closure [δ[ε-Closure(q_0), *a*]]**
- first, we find **ε-Closure** of q_0: {q_0, q_1, q_2}
- Now check transition of all **three** states on input symbol *a* we get: q_0
- After that we again calculate the **ε-Closure** of the above result q_0, we get the result as: **{q_0, q_1, q_2}**
- Similarly, we will check it for every state.
- All those states will be considered as a final state in which we have q_2(final state in initial **ε-NFA**) in there ε-Closure, i.e. all states will be final in this case.





Moore and Mealy Machine

- Both moore and mealy machine are special case of DFA
- Both acts like o/p producers rather than language acceptors
- In moore and mealy machine no need to define the final states
- No concepts of dead states and no concepts of final states
- Mealy and Moore Machines are equivalent in power.

George H Mealy

Edward F Moore

Moore Machine

A Moore machine is a six-tuple ($Q, \sum, \Delta, \delta, \lambda, q_0$), where

- Q is a finite set of states:
- ∑ is the input alphabet:
- Δ is the output alphabet.
- δ is the transition function Q x ∑ into Q
- λ is the output function mapping Q into Δ and
- q_0 is the initial state.

Examples: The below table shows the transition table of a Moore Machine.

Present state	Next state δ		Output
	$a = 0$	$a = 1$	λ
$\rightarrow q_0$	q_3	q_1	0
q_1	q_1	q_2	1
q_2	q_2	q_3	0
q_3	q_3	q_0	0

- In moore machine for every state output is associated.
- If the length of i/p string is n, then length of o/p string will be n+1
- Moore machine response for empty string $\in$

Q construct a Moore machine take all the string of a's and b's as i/p and counts the no of a's in the i/p string in terms of 1, ∑ = {a, b}, Δ = {0, 1}?

Q construct a Moore machine take all the string of a's and b's as i/p and counts the no of occurrence of sub-string 'ab' in terms of 1, ∑ = {a, b}, Δ = {0, 1}?

Q construct a Moore machine where ∑ = {0, 1}, Δ = {a, b, c}, machine should give o/p a, if the i/p string ends with 10, b if i/p string ends with 11, c otherwise?

Mealy Machine

- Mealy machine is a six-tuple **$(Q, \sum, \Delta, \delta, \lambda, q_0)$**, where all the symbols except λ have the same meaning as in the Moore machine. λ is the output function mapping Q x ∑ into Δ.

- In case of mealy machine, the output symbol depends on the transition.

Example: The below table shows the transition table of a Mealy Machine.

Present state	Next state			
	$a = 0$		$a = 1$	
	state	output	state	output
$\rightarrow q_1$	q_3	0	q_2	0
q_2	q_4	1	q_4	0
q_3	q_2	1	q_1	1
q_4	q_4	1	q_3	0

- If the length of i/p string is n, then length of o/p string will be n
- Mealy machine do not response for empty string $\in$

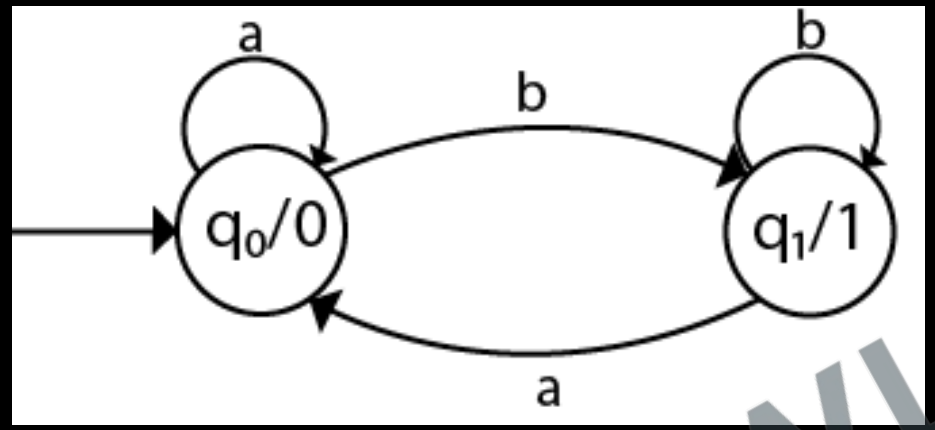
Q construct a Mealy machine take all the string of a's and b's as i/p and counts the no of a's in the i/p string in terms of 1, $\sum = \{a, b\}$, $\Delta = \{0, 1\}$?

Q construct a Mealy machine take all the string of a's and b's as i/p and counts the no of occurrence of sub-string 'ab' in terms of 1, $\sum = \{a, b\}$, $\Delta = \{0, 1\}$?

Q construct a Mealy machine where ∑ = {0, 1}, Δ = {a, b, c}, machine should give o/p a, if the i/p string ends with 10, b if i/p string ends with 11, c otherwise?

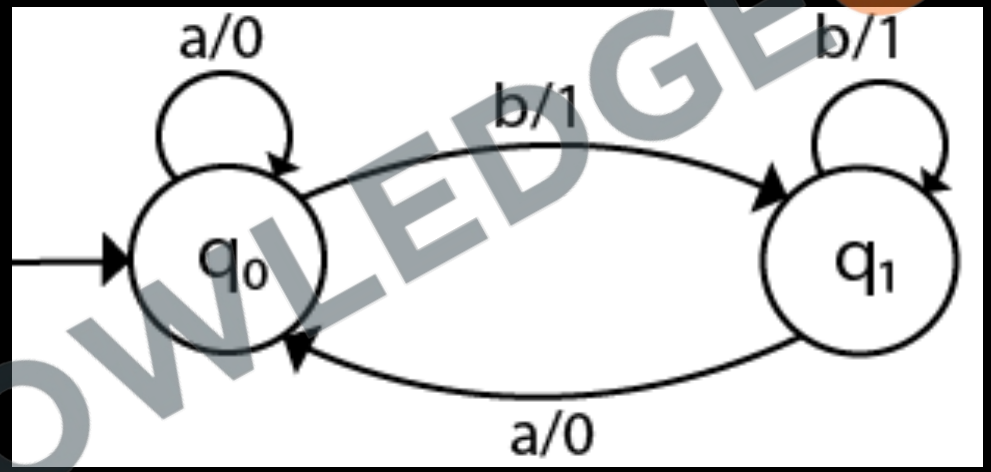
CONVERSION OF MOORE TO MEALY MACHINE

- Let us take an example to understand the conversion:
- Convert the following Moore machine into its equivalent Mealy machine.



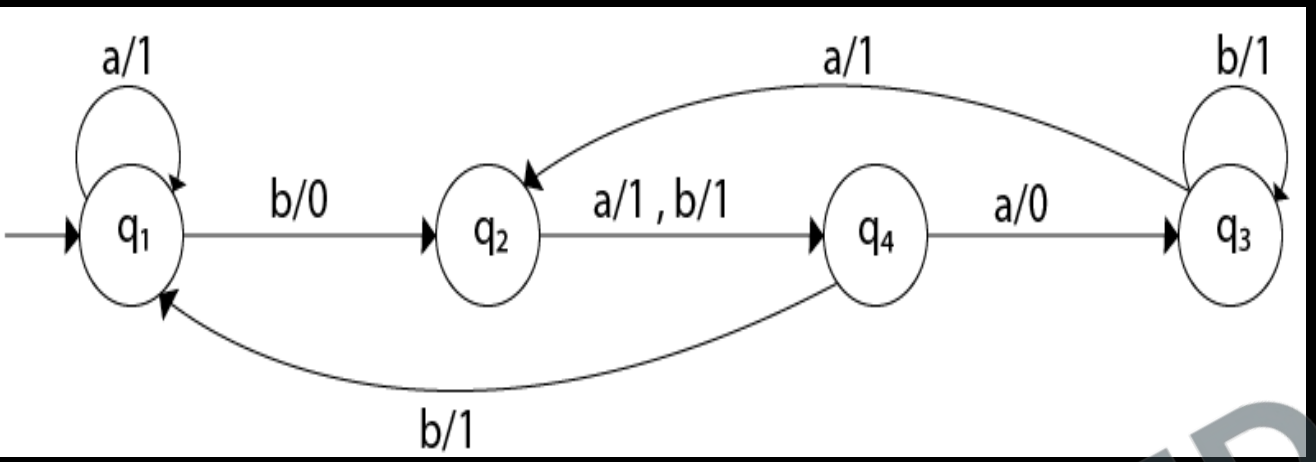
Q	a	b	Output(λ)
q0	q0	q1	0
q1	q0	q1	1

- To convert a mealy machine to moore machine all you need to do is just push out the outputs of states onto to the incoming transitions.
- While conversion from moore to mealy machine, the number of states will we same and there will be no extra states.
- The equivalent Moore Machine will be:



PROCEDURE FOR TRANSFORMING A MEALY MACHINE INTO A MOORE MACHINE

Consider the Mealy Machine:



Present State	Next State			
	a		b	
	State	O/P	State	O/P
q_1	q_1	1	q_2	0
q_2	q_4	1	q_4	1
q_3	q_2	1	q_3	1
q_4	q_3	0	q_1	1

Aspect	Moore Machine	Mealy Machine
Response to Null Input	Produces an output corresponding to the initial state.	Does not produce any output.
Output Length for Input of Length n	Generates an output string of length n+1.	Generates an output string of length n.
Initial Output	The initial output is determined by the initial state.	No initial output until an input is provided.
Output Synchronization with Input	Output is one step behind the input.	Output is synchronized with the input; changes immediately with input.
Typical Use Cases	Suitable for applications where stable output is necessary.	Ideal for applications requiring immediate response to input changes.

MINIMIZATION OF FINITE AUTOMATA

- The process of elimination of states whose presence or absence doesn't affect the language accepting capability of deterministic Finite Automata is called minimization of automata and the result is minimal deterministic finite automata or commonly called as minimal finite automata as MFA.

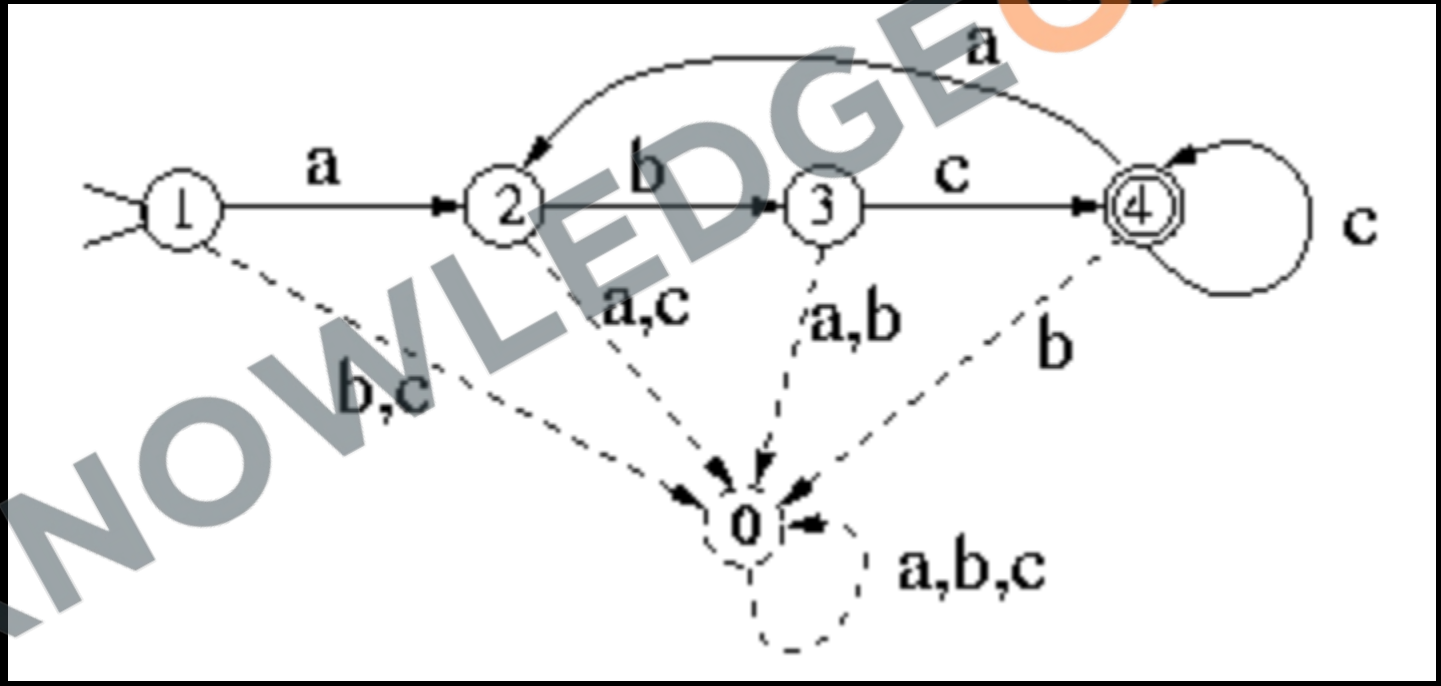
- **NOTE- MFA is always unique for a language.**

- It is sometimes difficult to design a minimal DFA directly so, a better approach is to first design the DFA and then minimize it.
- Based on **productivity**, the states of DFA can be mainly classified in two types-
- **PRODUCTIVE STATES**- State is said to be productive, if it adds any accepting power to the machine that is its presence and absence effect the language accepting capability of the machine.
- **NON- PRODUCTIVE STATES**- These states don't any add anything to the language accepting power to the machine.

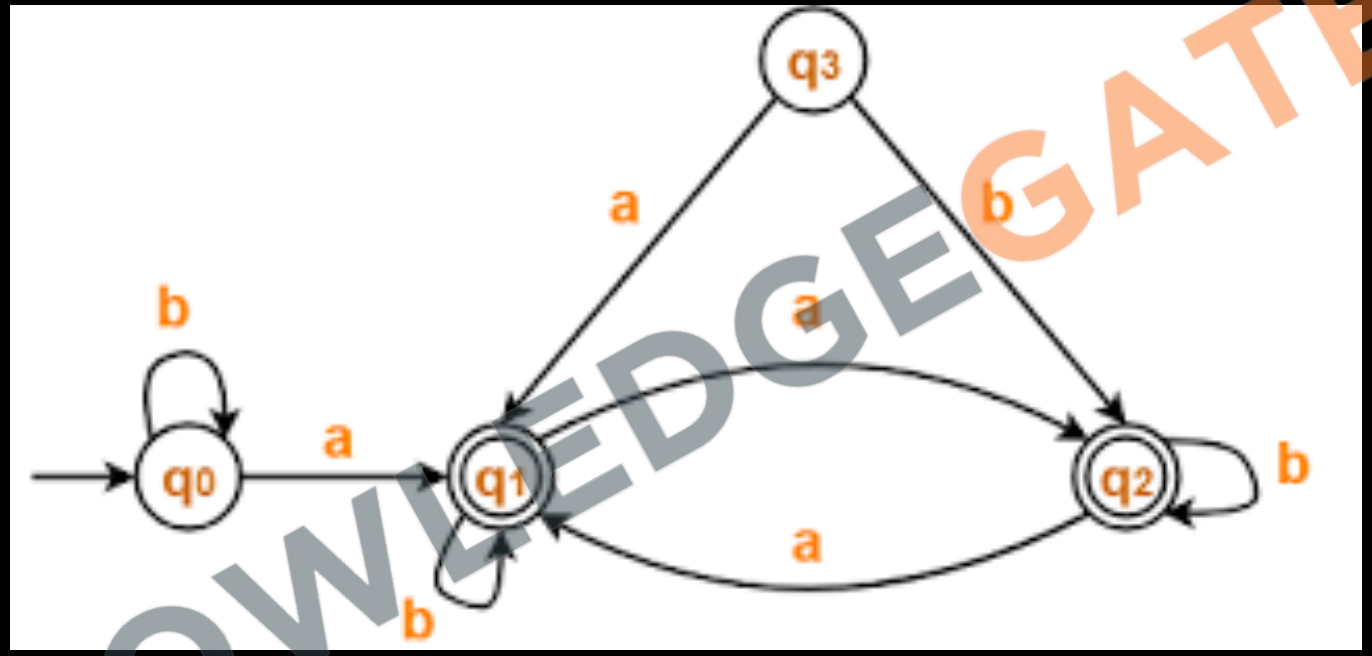
NON- PRODUCTIVE STATES- These states don't add anything to the language accepting power to the machine. They can further be divided into three types-

- Dead State
- Unreachable Sate
- Equal State

- **Dead State**- It is basically created to make the system complete, can be defined as a state from which there is no transition possible to the final state.
- In a DFA there can be more than one dead state but logically always one dead state is sufficient to complete the functionality.



- **Unreachable Sate**- It is that state which cannot be reached starting from initial state by parsing any input string.

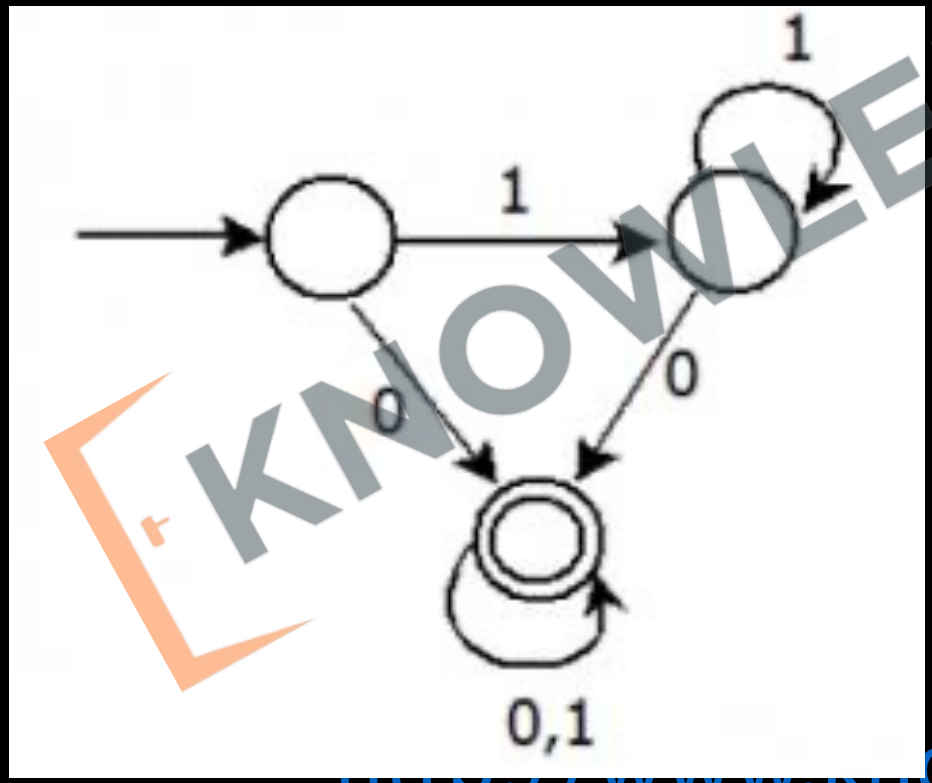


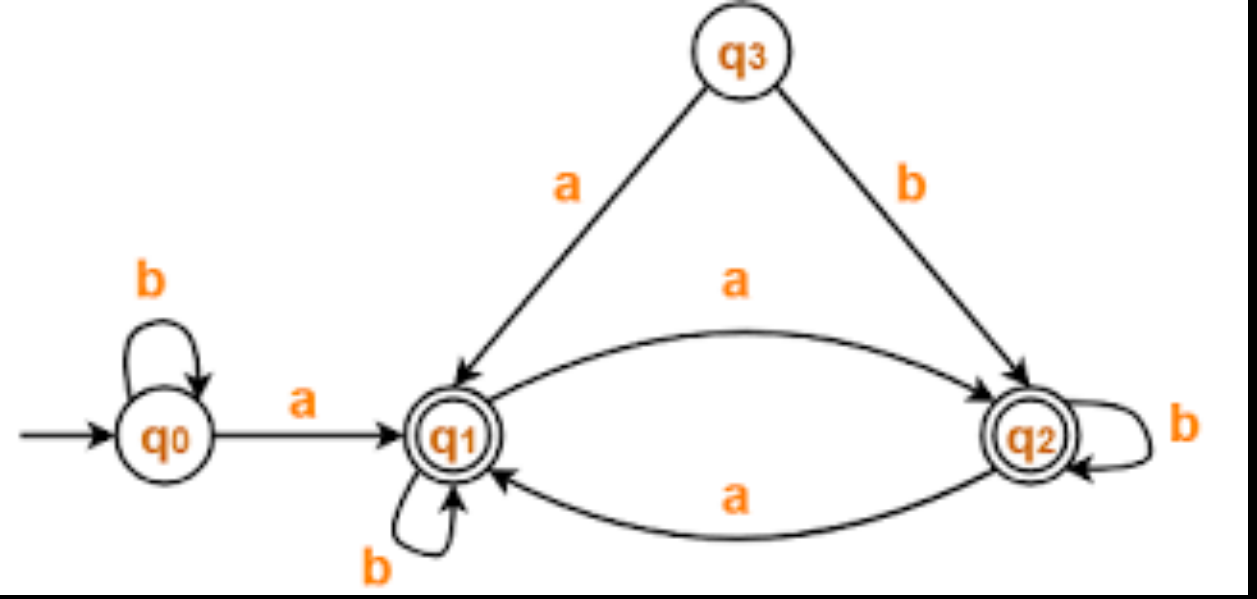
- **Equal State**-These are those states that behave in same manner on each and every input string. That is for any string w where w $\in \Sigma^*$ either both of the states will go to final state or both will go to non-final state. (remember the example of an equal state DFA).

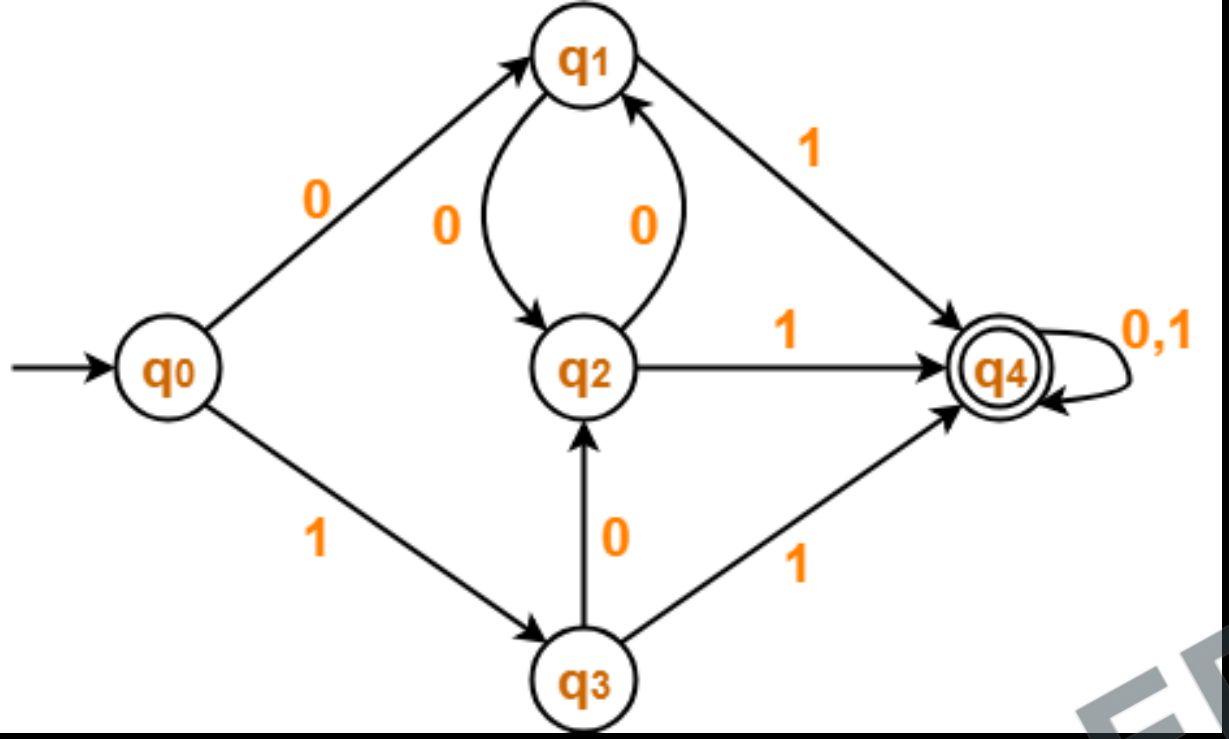
- More formally, two states q_l and q_2 are equivalent (denoted by $q_1 \cong q_2$) if both $\delta(q_1, x)$ and $\delta(q_2, x)$ are final states or both of them are non-final states for all $x \in \sum^*$. If q_1 and q_2 are k-equivalent for all $k \geq 0$, then they are *k*-equivalent.

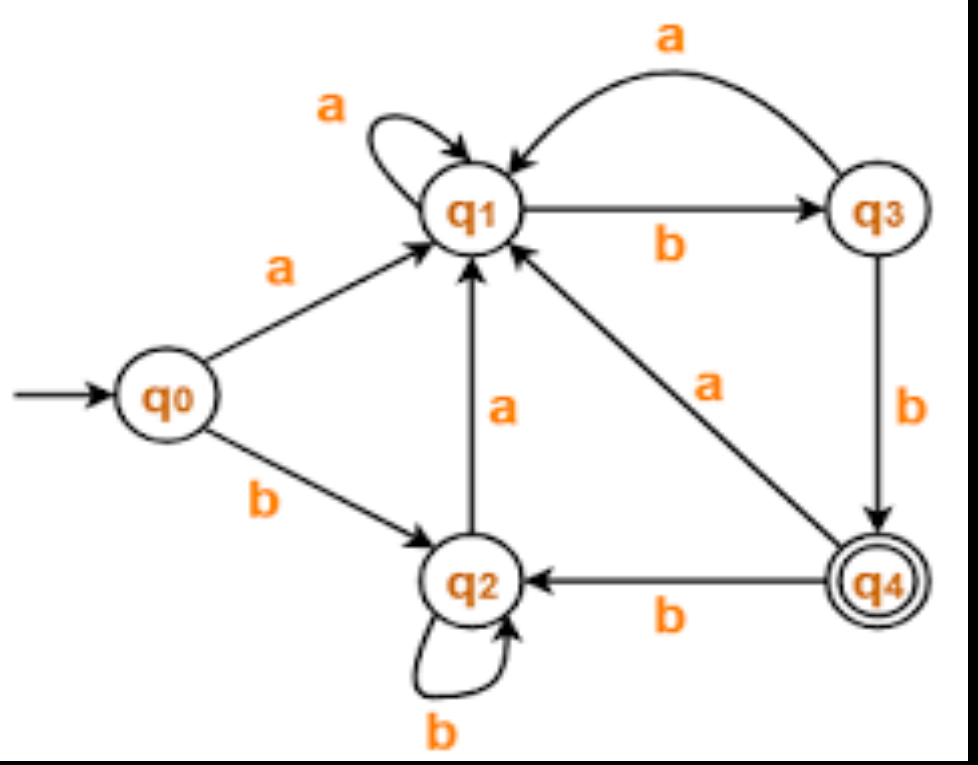
Procedure of Minimization

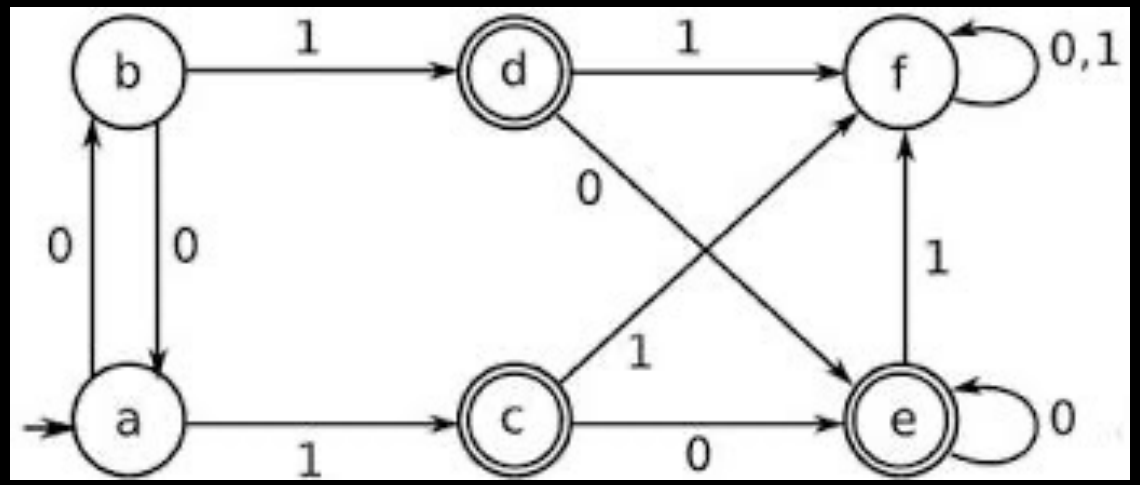
- For this first of all, group all the non-final states in one set and all final states in another set.
- Now, on both the sets, individually check, whether any of the underlying elements (states) of that particular set are behaving in the same way, that is are they having same transition(to same set) on each input alphabet.
- if the answer is yes, then these two states are equal, otherwise not.











- **Chapter-2 (Regular Expressions and Languages):** Regular Expressions, Transition Graph, Kleen's Theorem, Finite Automata and Regular Expression- Arden's theorem, Algebraic Method Using Arden's Theorem, Regular and Non-Regular Languages- Closure properties of Regular Languages, Pigeonhole Principle, Pumping Lemma, Application of Pumping Lemma, Decidability- Decision properties, Finite Automata and Regular Languages

Regular Expressions

- One way of describing regular language is via the notation of regular expression. An expression of strings which represents regular language is called regular expression. The regular expressions are useful for representing certain sets of strings(Language) in an algebraic fashion.
- We give a formal recursive definition of regular expressions over ∑ as follows:
  - Any terminal symbol (i.e. an element of ∑), ∈ and Φ are regular expressions **(Primitive regular expressions)**.
  - A regular expression is valid iff it can be derived from a primitive regular expression by a finite number of applications of operators.

Regular Language:- Any set(language) represented by a regular expression is called a **Regular language**. If for example, a, b ∈ ∑, then

- **R =** a denotes the L = {a}
- **R =** a.b denotes L = {ab} concatenation
- **R =** a + b denotes L = {a, b} Union
- **R =** a* denotes the set {∈, a, aa, aaa, ...} known as Kleene closure.
- **R =** a^+ Positive closure {a, aa, aaa…}
- **R =**(a + b)* denotes {a, b}*

Operators

- When we view *a* in ∑ as a regular expression, we denote it by **a**.
  - If R is a regular expression, then **(R)** is also a regular expression.
  - The iteration (or closure) of a regular expression **R** written as **R***, is also a regular expression.
  - The iteration (or closure) of a regular expression **R** written as $\mathbf{R^+}$, is also a regular expression.
  - The concatenation of two regular expressions $\mathbf{R_1}$ and $\mathbf{R_2}$, written as $\mathbf{R_1\ R_2}$, is also a regular expression.
  - The union of two regular expressions $\mathbf{R_1}$ and $\mathbf{R_2}$, written as $\mathbf{R_1 + R_2}$, is also a regular expression.

Operator Precedence

- The precedence order to solve is
  - ()Bracket
  - * (Kleene Closure)
  - $^{+}$ Positive Closure
  - Concatenation
  - Union

IDENTITIES FOR Regular Expression

- Two regular expressions P and Q are equivalent (we write P = Q)
  - if P and Q represent the same set of strings.

- Every regular expression can generate only one regular language but, a regular language can be generated by more than one regular expression I.e. means two different regular expression can generate same language.

- Two regular expression are said to be equal if they generate same language.

- $r_1 = a^*$

- $r_2 = a^* + (aa)^*$

हमारी language बताओ

1. R={a}
2. R={a + b}
3. R={a + b + c}

1. R={a.b}

2. R={a.b + a}b

Q Design a regular expression that represent a language ‘L’, where L={a} over the alphabet ∑={a}.

Q design a regular expression that represent all strings over the alphabet ∑ = {a, b}, where every accepted string ‘w’ starts with substring s = ‘abb’

Q design a regular expression that represent all strings over the alphabet ∑ = {a, b}. where every accepted string ‘w’ ends with substring s = ‘bab’?

Q Design a regular expression that represent all strings over the alphabet ∑ = {a, b}. where every accepted string ‘w’ contains sub string s = ‘aba’ ?

Q Design a regular expression that represent all strings over the alphabet ∑ = {a, b} such that every accepted string start and end with a.

Q Design a regular expression that represent all strings over the alphabet ∑ = {a, b} such that every accepted string start and end with same symbol?

Q Design a regular expression that represent all strings over the alphabet ∑ = {a, b} such that every accepted string start and end with different symbol ?

Q Design a regular expression that represent all strings over the alphabet ∑ = {a, b} such that every accepted string w, is like w=SX S = ‘aaa/bbb’ ?

Q Design a regular expression that represent all strings over the alphabet ∑ = {a, b} such that every accepted string w, is like w=XS. S = ‘aaa/bbb’ ?

Q Design a regular expression that represent all strings over the alphabet ∑ = {a, b} such that every accepted string w, is like w=XSX. S = aaa/bbb ?

Q Design a regular expression that represent all strings over the alphabet ∑ = {a, b}, such that every string ‘w’ accepted must be like

i) |w| = 3 **ii)** |w|<=3 **iii)** |w|>=3

Q Design a regular expression that represent all strings over the alphabet ∑ = {a, b} such that for every accepted string 2nd from left end is always b.

Q Design a regular expression that represent all strings over the alphabet ∑ = {a, b} such that for every accepted string 4^{th} from right end is always a.

Q Design a regular expression that represent all strings over the alphabet ∑= {a, b}, such that every string ‘w’ where |W| = 0(mod 3)?

Q Design a regular expression that represent all strings over the alphabet ∑= {a, b}, such that every string ‘w’ where |W| = 3(mod 4)?

Q Design a regular expression that represent all strings over the alphabet ∑= {a, b}, such that every string ‘w’ where $|W|_a = 0(\text{mod } 3)$?

Q Design a regular expression that represent all strings over the alphabet ∑= {a, b}, such that every string ‘w’ where $|W|_b = 2(\text{mod } 3)$?

Algebraic Properties of regular expression

- **Closure Property** - Regular expressions satisfy closure property with respect to Union, Concatenation and kleene closure. If R_1 and R_2 are regular expression then the following will also be regular expression.

$r_1 + r_2$

$r_1.r_2$

r_1^*

r_1^+

- **Associative Property**- Regular expression satisfy associative property with respect to union and intersection

$$(r_1 + r_2) + r_3 \square r_1 + (r_2 + r_3)$$

$$(r_1 . r_2) . r_3 \square r_1 . (r_2 . r_3)$$

- **Identity Property**- The identity property is satisfied as follows-

$r.\square = r$

$r+\square = r$

- **Commutative Property**- Regular expressions are commutative with respect to union but not with respect to concatenation.

$r_1 + r_2 \square r_2 + r_1$

$r_1 . r_2 \square r_2 . r_1$

- **Distributive Property**-Regular expression satisfy this property as follows-

$r_1(r_2+r_3) = r_1r_2+r_1r_3$

$(r_1+r_2)\, r_3 = r_1r_3+r_2r_3$

$r_1+(r_2.\, r_3) = (r_1+r_2)(r_1+r_3)$

$(r_1.r_2) + r_3 = (r_1 + r_3).\, (r_2 + r_3)$

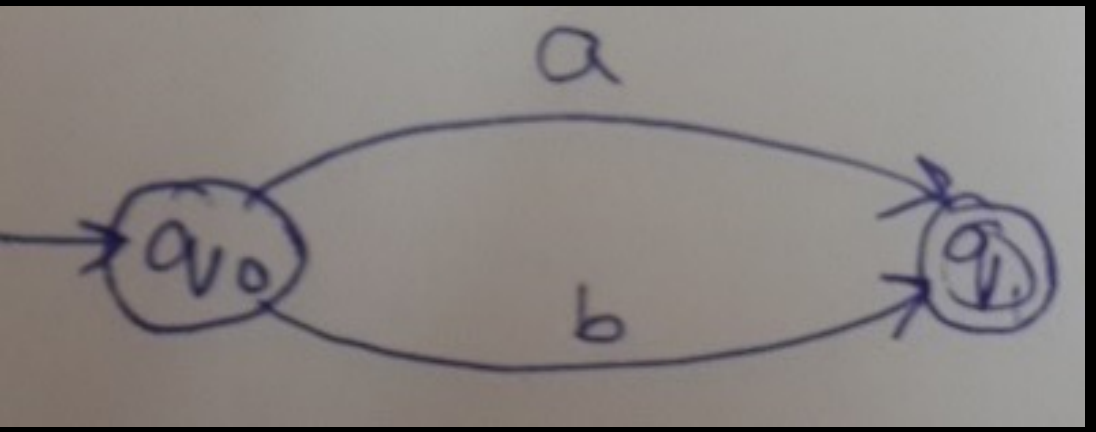
- **Idempotent Property**-regular expressions satisfies idempotent property with respect to union but not with respect to concatenation.

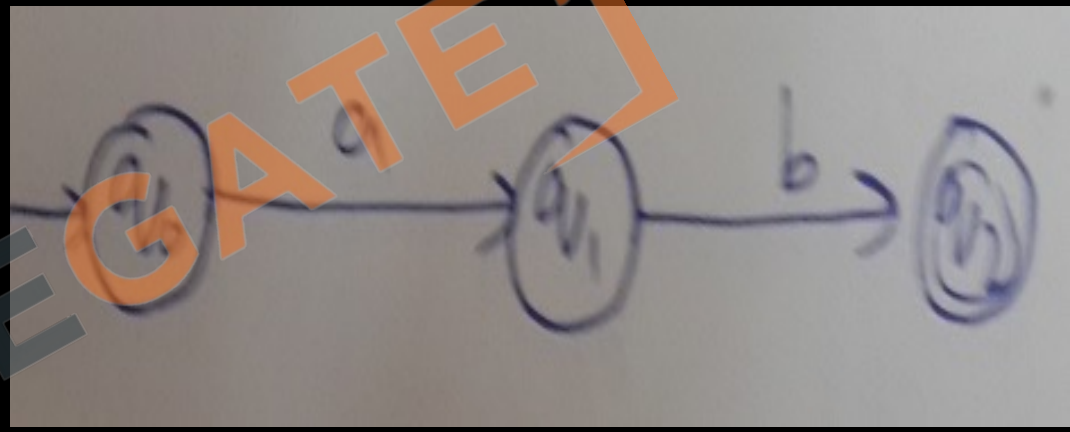
$r_1 + r_1 \square r_1$

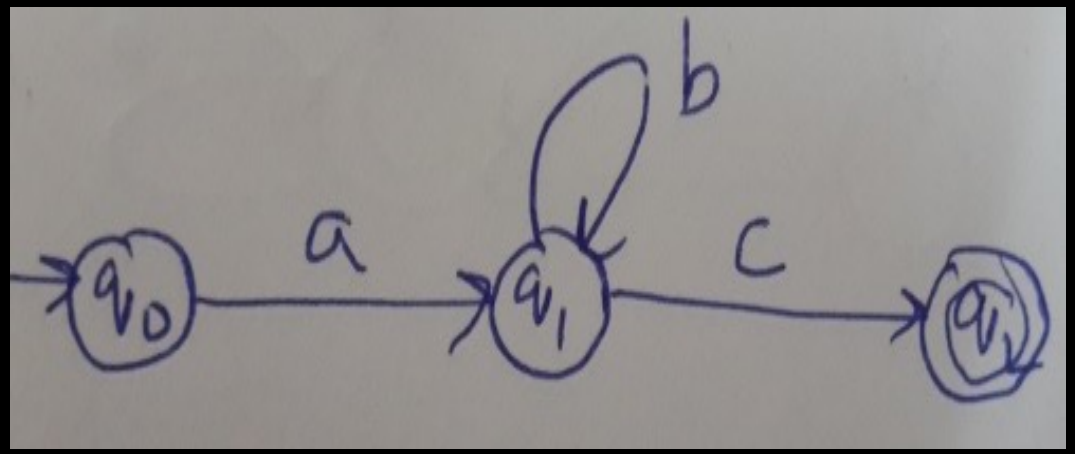
$r_1 . r_1 \square r_1$

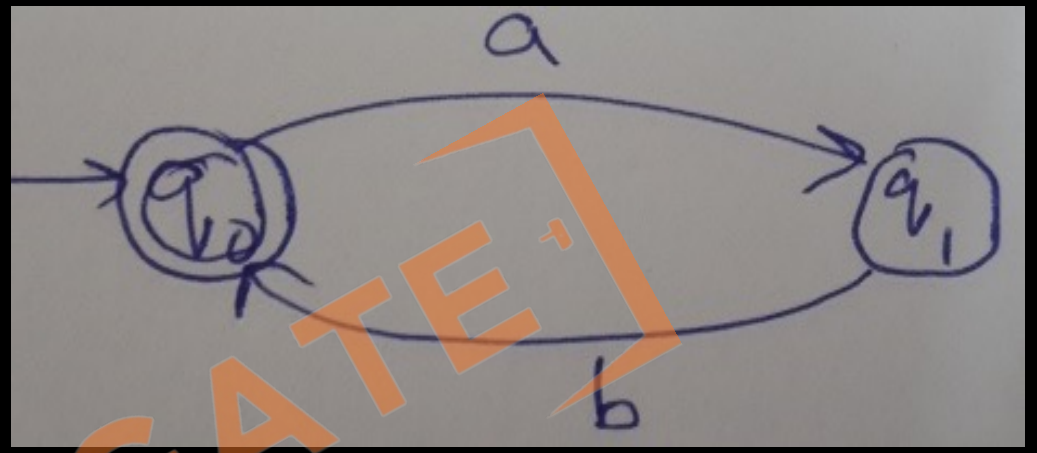
Conversion from Finite Automata to regular expression

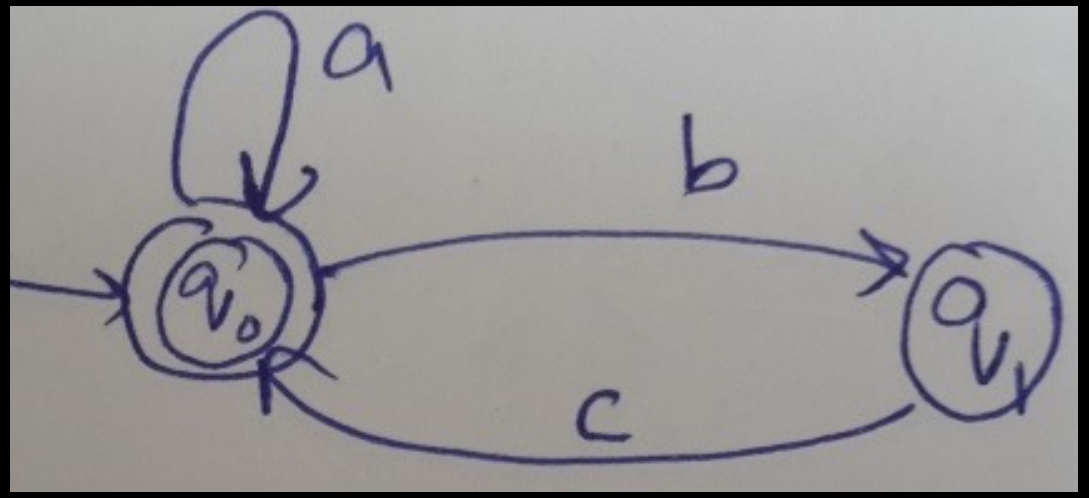
Q Write regular expressions for the following machines

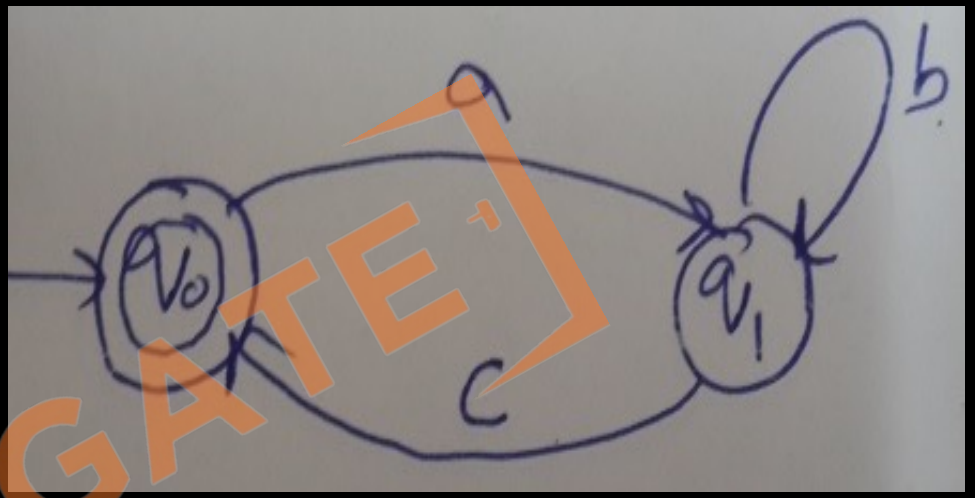


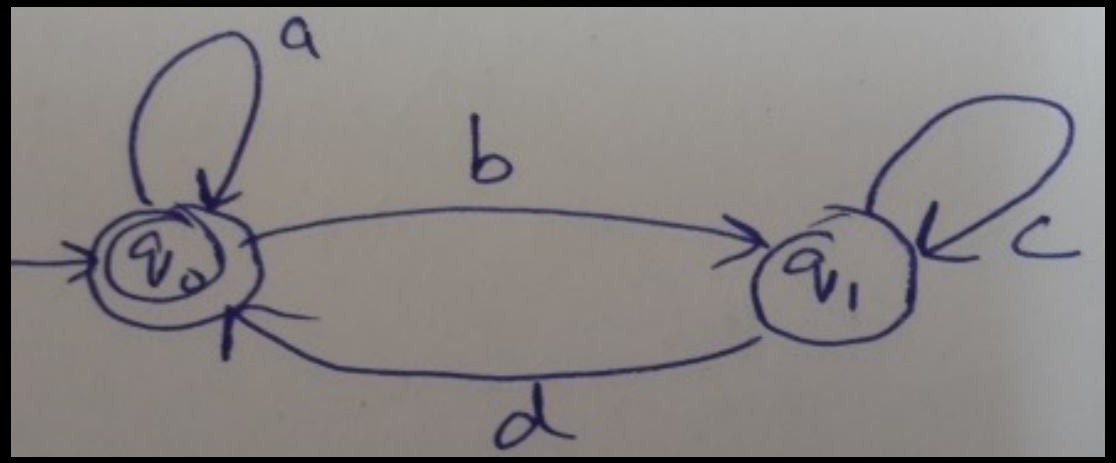


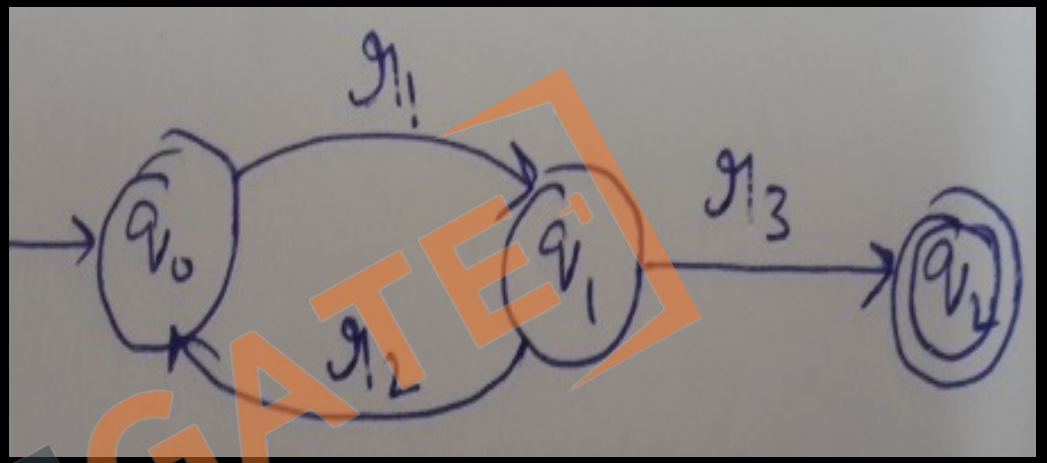


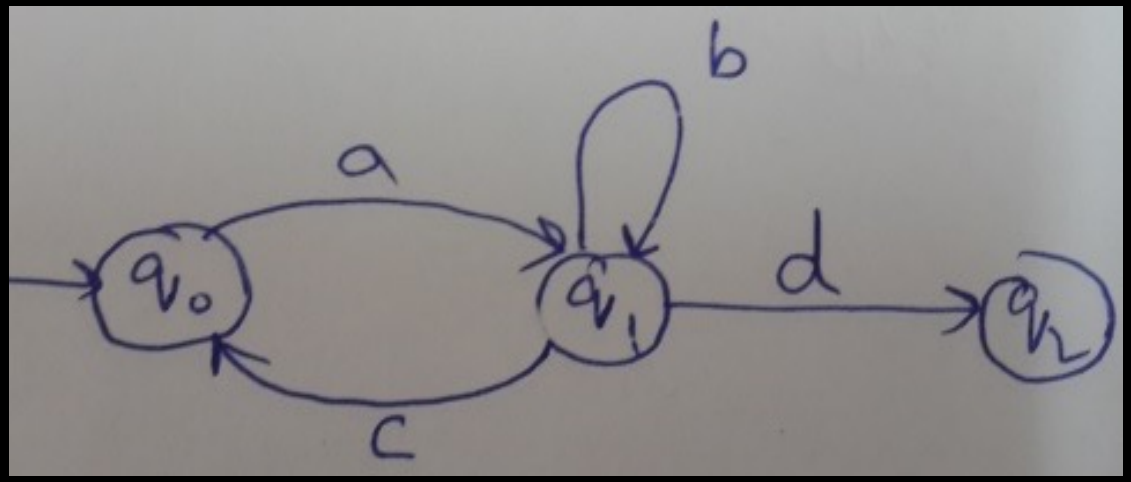


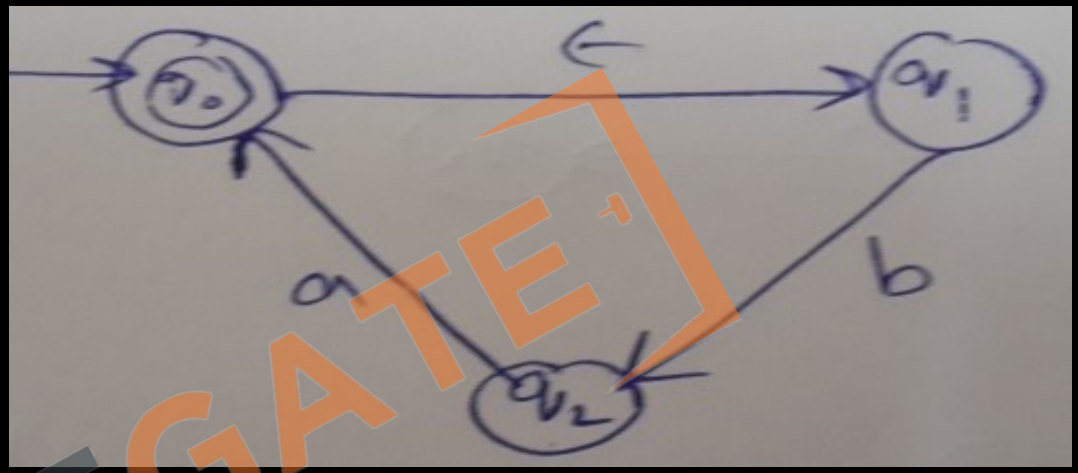


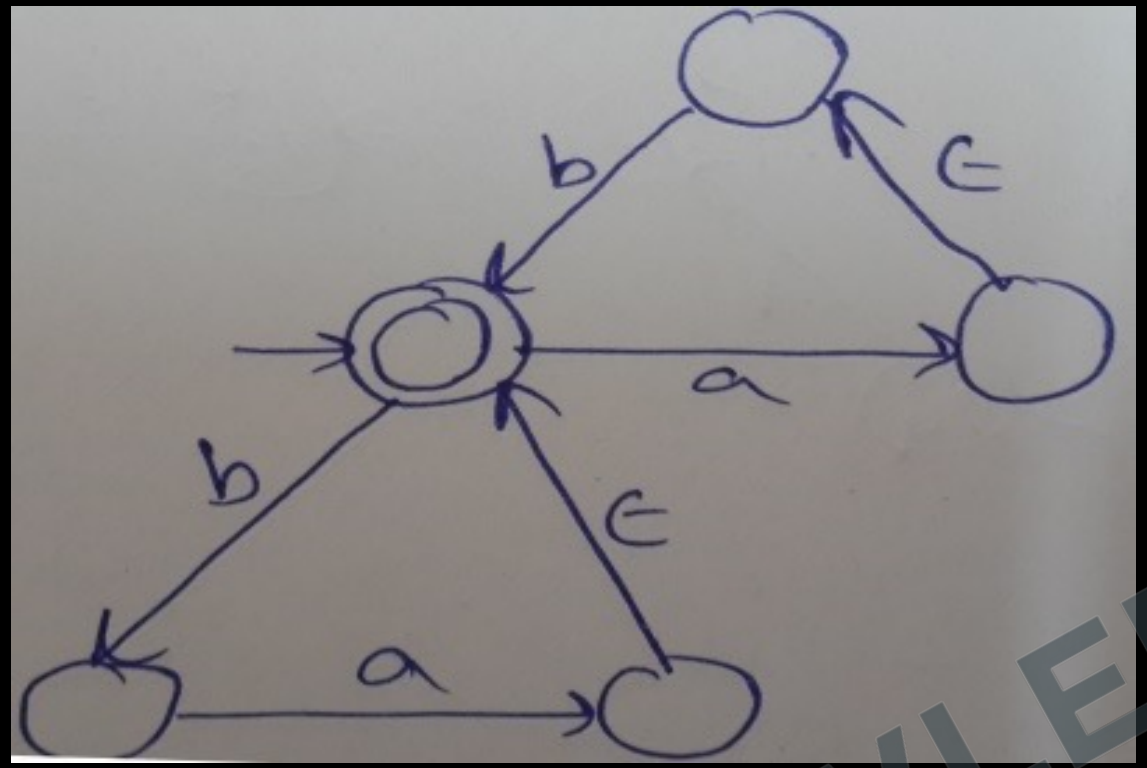


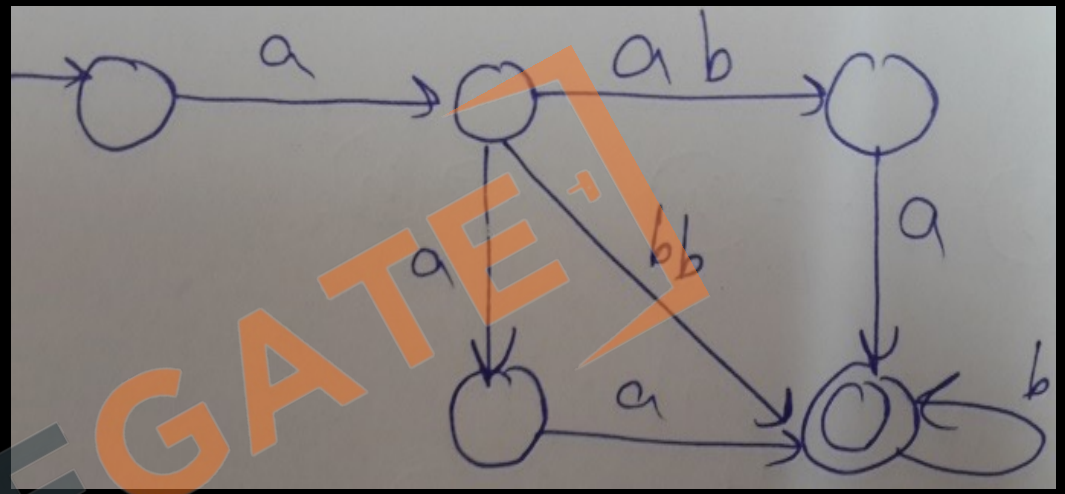


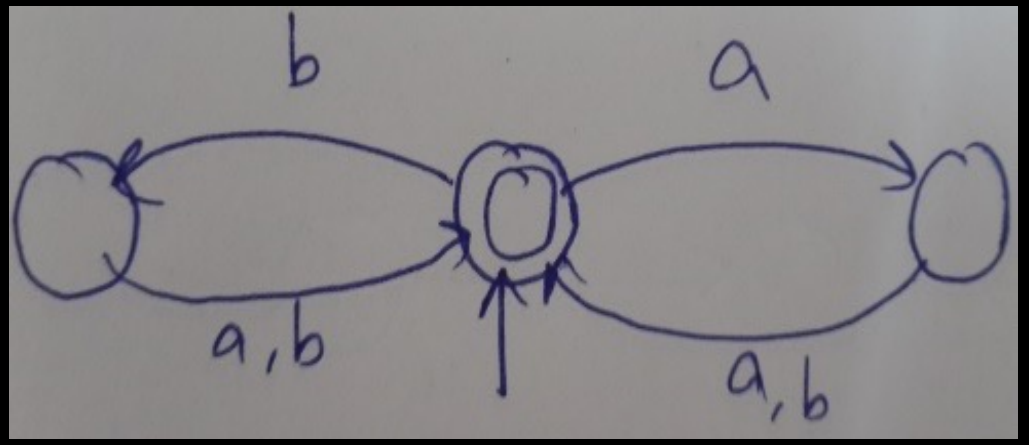


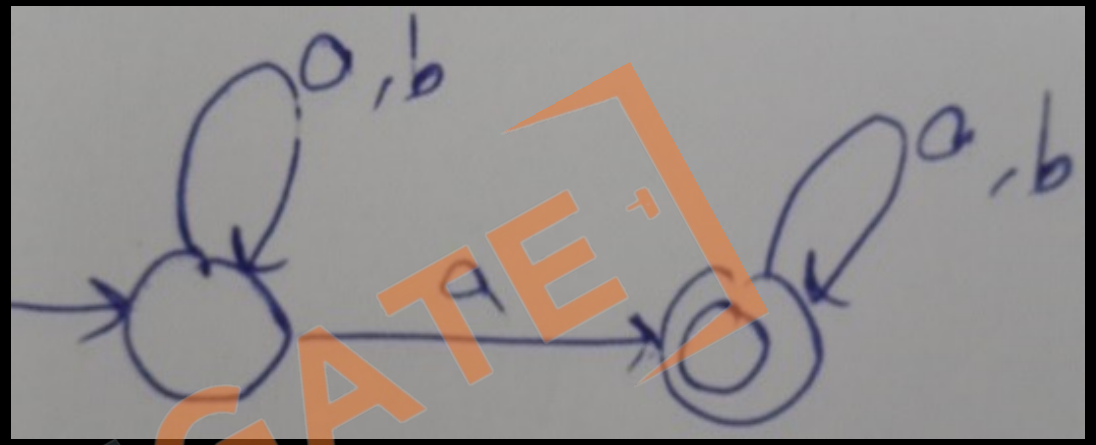


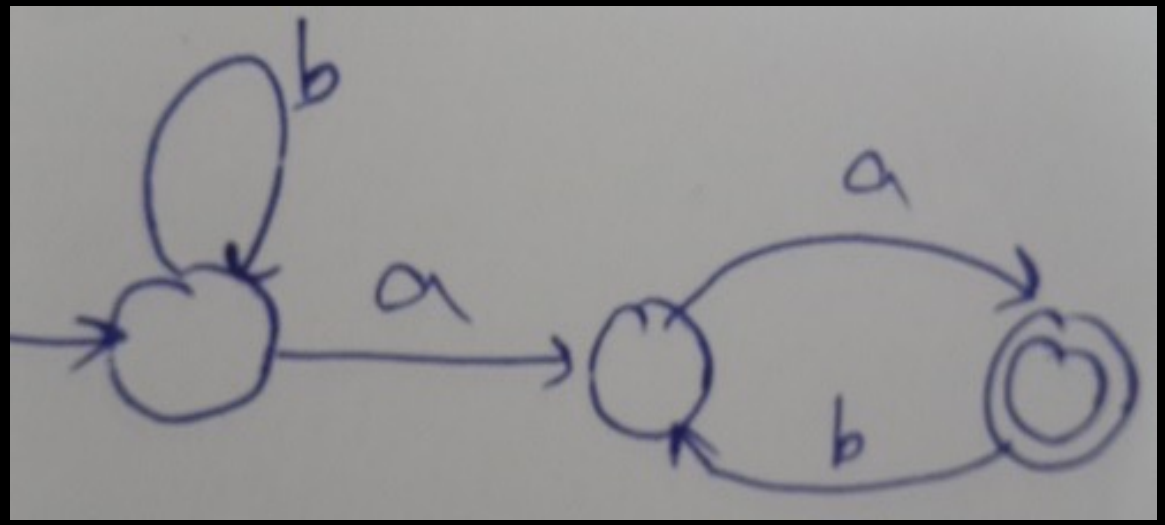


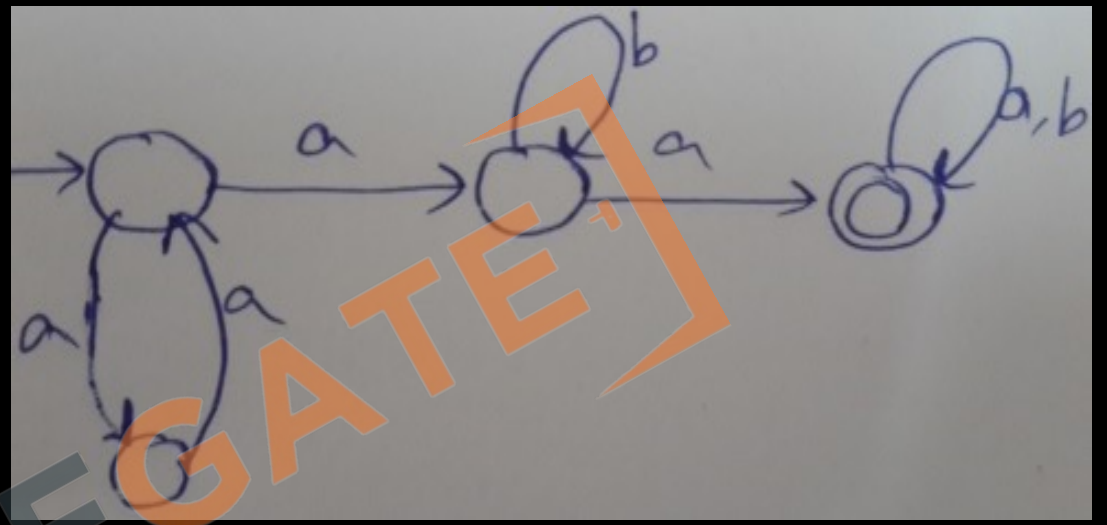


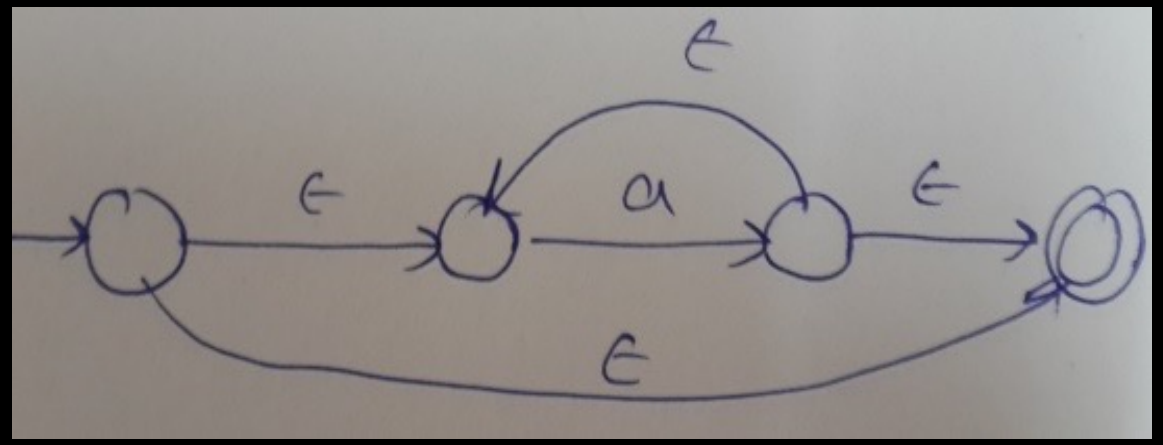


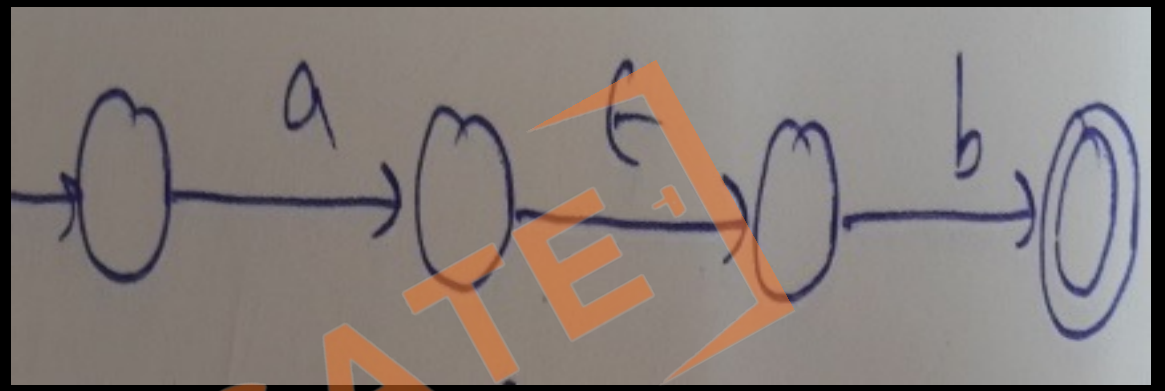


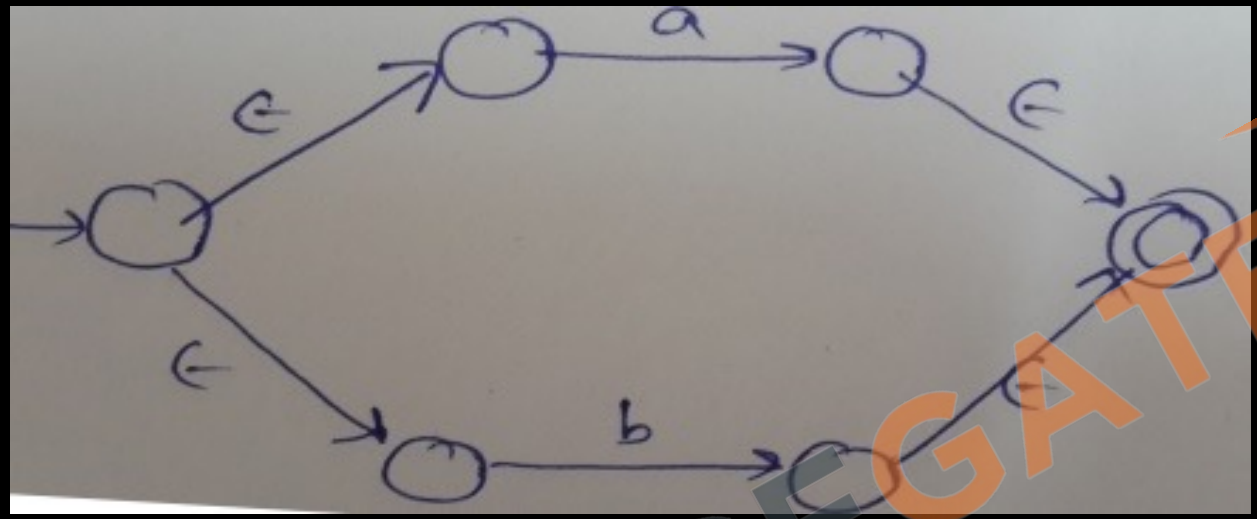












Conversion from regular expression in Finite Automata

i) R^*

ii) $(R_1.R_2)^*$

iii) $(R_1+R_2)^*$

iv) $(R_1 * R_2 . R_3 *)$

v) R = a*b(ab)*

vi) R = (a + ba)*ab*

vii) R = (aa + aaa)*

viii) R = (a + aaaaa)*

1. 0 * 10 * 10 *

2. ((ab)* + c*)*

3. (∈ + a + aa + aaa)b* + (a + b)* ba(a + b)*

4. 0*(10 * 1*)*0*

1. a*b(b * + aa*b)*

2. ((11*0+0)(0 + 1)*0*1*)

3. (ab+ bc+ acc).(a+ bc)*.(∈+ bc*a)*

EQUIVALENCE BETWEEN Regular Expression AND FINITE AUTOMATA

- **ARDEN'S THEOREM** is the mechanism for the construction of a regular expression from a finite automaton.

Q Consider a DFA and convert it into regular expression using Arden's theorem?

$\delta(A, a) = A$

$\delta(A, b) = B$

$\delta(B, a) = B$

$\delta(B, b) = B$

A is the initial state and B Is the final state

- Steps used-
  - For, every individual state of the DFA, write an expression for every incoming and outgoing input alphabet.
  - Apply Arden's theorem as follows-
    - If P is free from NULL, then equation R=Q+RP has unique solution, R=QP*
    - If P contains NULL, then equation R=Q+RP has infinitely many solutions.

Q Consider a DFA and convert it into regular expression using Arden's theorem?

$\delta(A, b) = A$

$\delta(A, b) = B$

$\delta(B, a) = C$

$\delta(C, b) = C$

A is the initial state and B,C Is the final state

Transition Graph

- Same as $\in$-nfa Can have more than one initial state .
- Non-Deterministic in nature (so no dead state is required, can take multiple moves).
- Can have string as a input to transition from one state to another.

Kleene's Theorem

- The theorem has two main parts:
  - **For any given regular expression, there is an equivalent finite automaton (either deterministic or non-deterministic) that recognizes the same language.**
  - **For any finite automaton, there is an equivalent regular expression that generates the same language.**

Closure Properties of Regular Languages

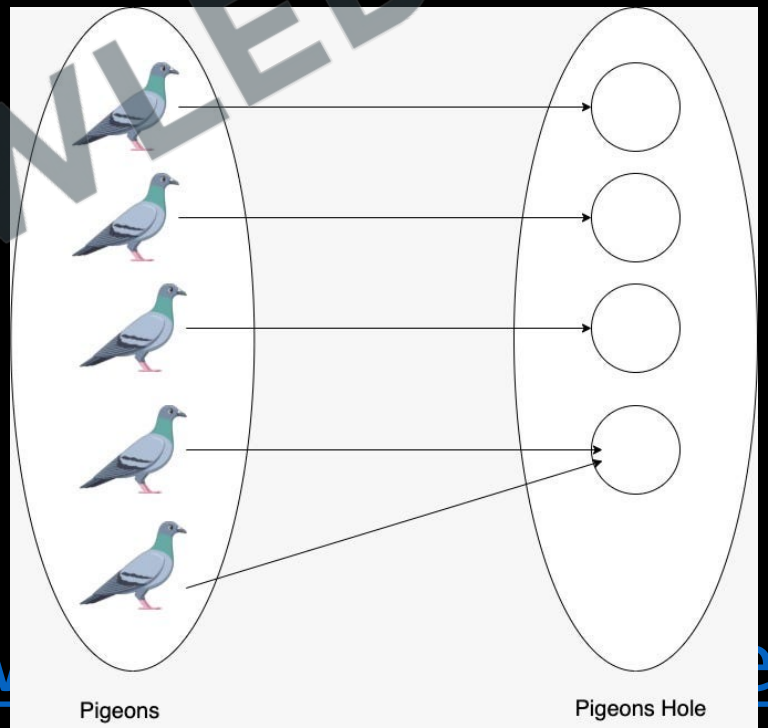
- Regular languages are closed under following operations
  - Kleen Closure
  - Positive closure
  - Complement
  - Reverse Operator
  - Prefix Operator
  - Suffix operator
  - Concatenation
  - Union
  - Intersection
  - Set Difference operator
  - Symmetric Difference

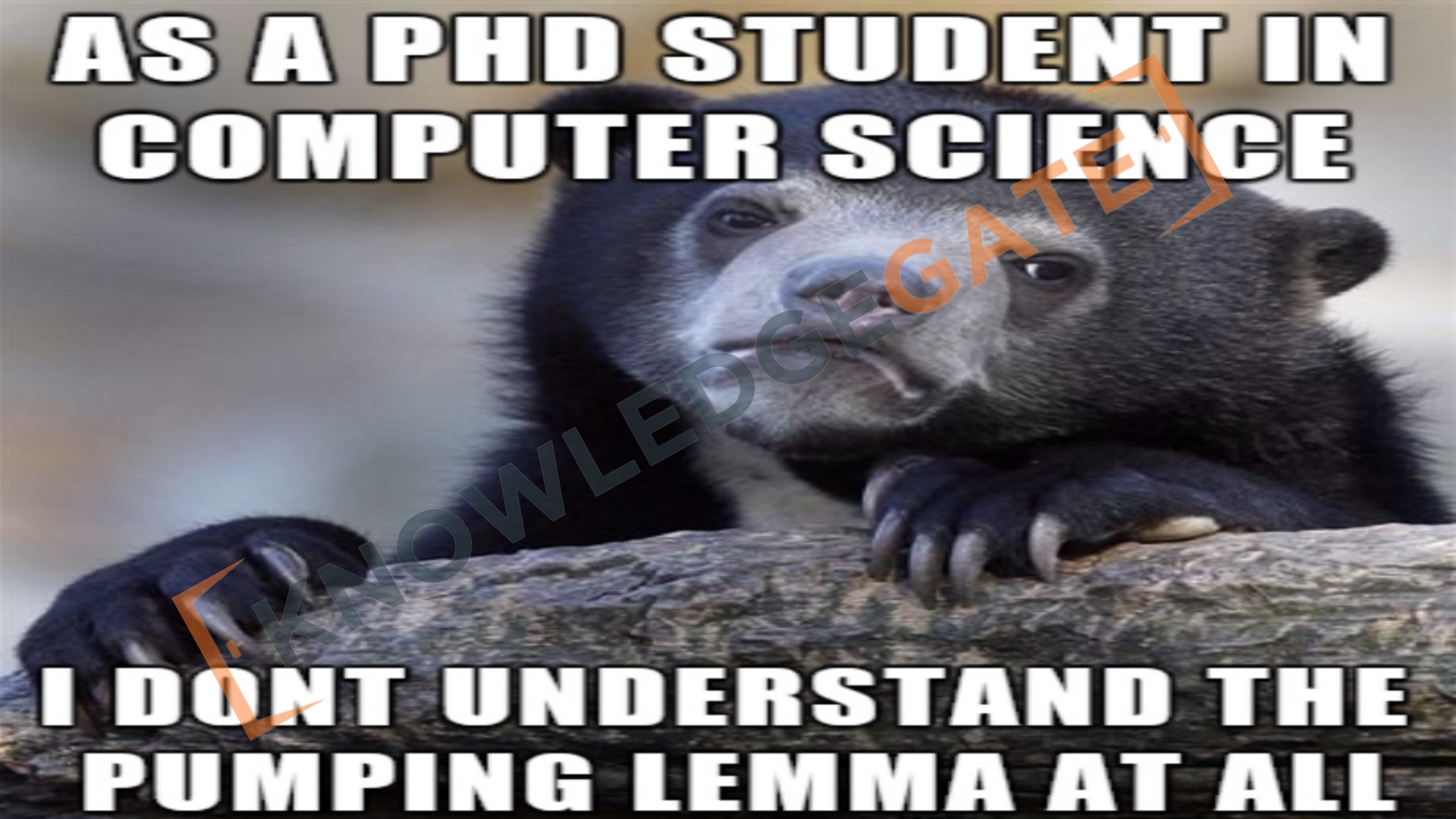
	RL	DCFL	CFL	CSL	RS	RES
Union	Y	N	Y	Y	Y	Y
Intersection	Y	N	N	Y	Y	Y
Complement	Y	Y	N	Y	Y	N
Set Difference	Y	N	N	Y	Y	N
Kleene Closure	Y	N	Y	Y	Y	Y
Positive Closure	Y	N	Y	Y	Y	Y
Concatenation	Y	N	Y	Y	Y	Y
Intersection with regular set	Y	Y	Y	Y	Y	Y
Reverse	Y	Y	Y	Y	Y	Y
Subset	N	N	N	N	N	N

	RL	DCFL	CFL	CSL	RS	RES
Homomorphism	Y	N	Y	N	N	Y
∈ Free Homomorphism	Y	N	Y	Y	Y	Y
Inverse Homomorphism	Y	Y	Y	Y	Y	Y
Substitution	Y	N	Y	N	N	Y
∈ Free Substitution	Y	N	Y	Y	Y	Y
Quotient with regular set	Y	Y	Y	N	Y	Y

Pigeonhole Principle

- The Pigeonhole Principle is a fundamental principle of combinatorial mathematics that states:
- If *n* items are put into *m* containers, with *n*>*m*, then at least one container must contain more than one item.
- This principle seems intuitive, but it has powerful implications and can be used to prove a variety of seemingly unrelated results across mathematics and computer science.



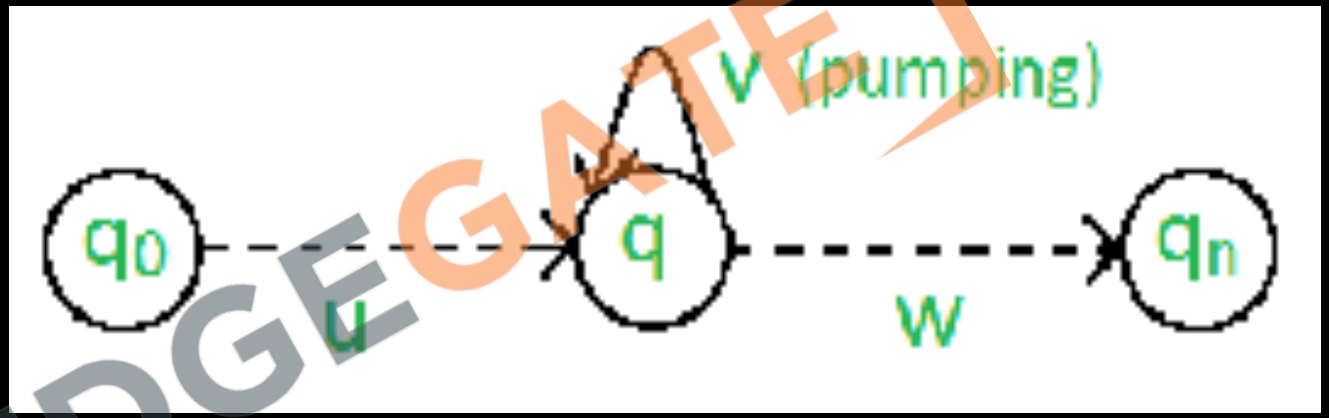

AS A PHD STUDENT IN COMPUTER SCIENCE
KNOWLEDGEGATE
I DONT UNDERSTAND THE PUMPING LEMMA AT ALL

Pumping Lemma For Regular Languages

- Pumping Lemma is used as a proof for irregularity of a language. Thus, if a language is regular, it always satisfies pumping lemma. If there exists at least one string made from pumping which is not in L, then L is surely not regular.
- The opposite of this may not always be true. That is, if Pumping Lemma holds, it does not mean that the language is regular.
- Pumping Lemma is used to prove that some of the language is non-regular.
- For the pumping lemma, i/p is NRL & o/p is also NRL.

Pumping Lemma For Regular Languages

- For any regular language L, there exists an integer n, such that for all z ∈ L with |z| ≥ n, there exists u, v, w ∈ Σ^*, such that z = uvw, and
  - |uv| ≤ n
  - |v| ≥ 1
  - for all i ≥ 0: $uv^iw \in L$



- In simple terms, this means that if a string v is 'pumped', i.e., if v is inserted any number of times, the resultant string still remains in L.

Process of pumping Lemma

- Suppose that we need to prove that language L is a non-regular.
- Assume that L is a regular language, then L must satisfy pumping lemma property.
- Choose z ∈ L such that |z| <= n, split z into 3 parts.
  - |uv| ≤ n
  - |v| ≥ 1

- If there exist at least one variable for i such that uv^iw L, then L does not satisfy pumping lemma property so it is a contradiction, therefore language L is non-regular.

Q Proof that language L = $\{a^m b^n \mid m=n\}$ is non-regular?

L = {ab, aabb, aaabbb, aaaabbbb, ---}

$z \in L$

$Z = a^k b^k$

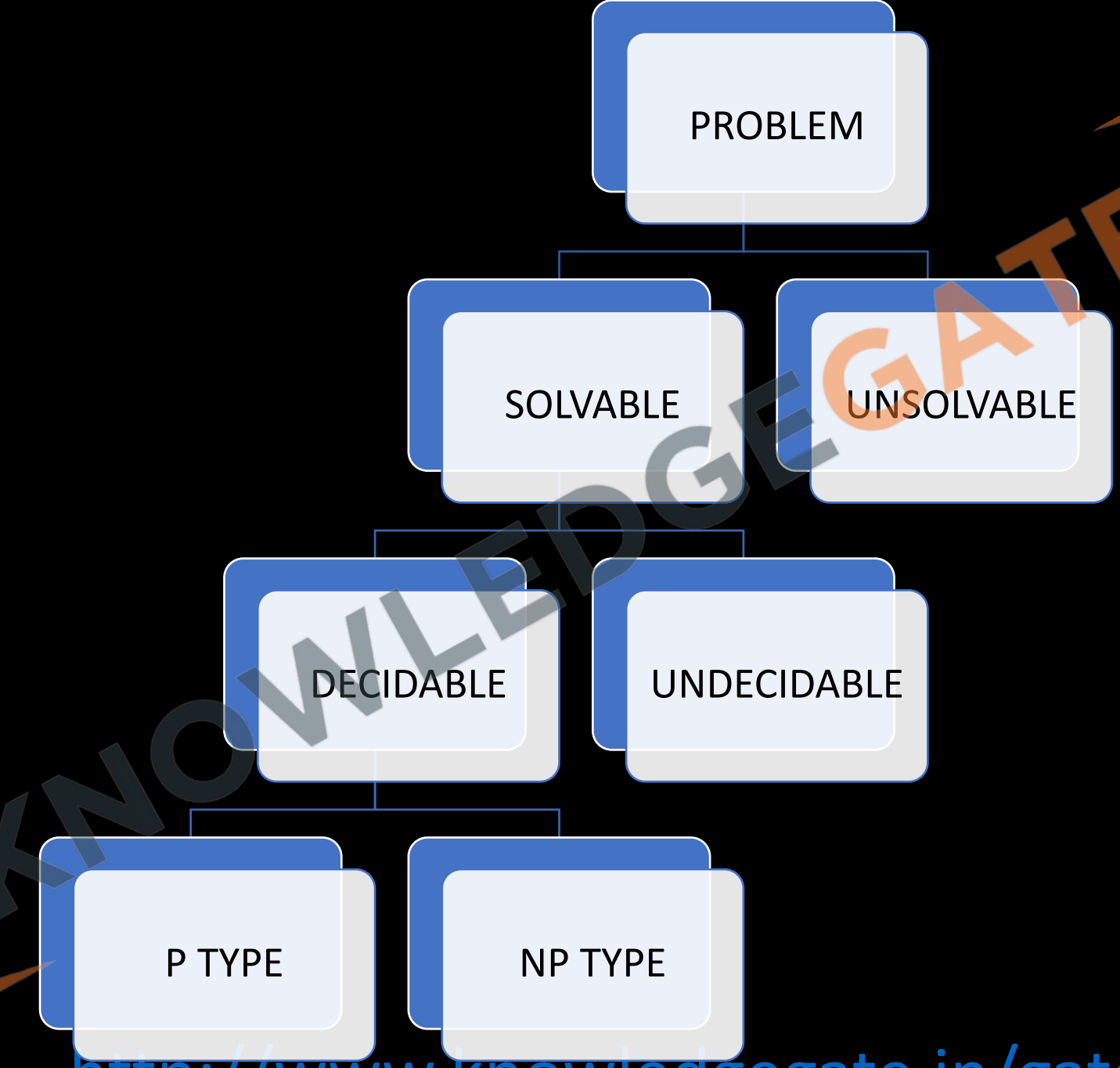
For i=1 → uv^iw → $a^k b^k$

For i=2 → uv^iw → $a^{k+1} b^k$

Q Proof that language L = $\{a^m b^n \mid m<n\}$ is non-regular?

Q Proof that language L = {$a^n b^n$ | n is a prime number} is non-regular?

Q Proof that language L = {$a^{n\hat{}2}$ | n>= 0} is non-regular?



- **SOLVABLE** - A problem is said to be solvable if either we can solve it or if we can prove that the problem cannot be solved

- **UNSOLVABLE** - A problem is said to be unsolvable if neither we can solve it, nor we can proof that the problem can not be solved

- P- if there exist a polynomial time algorithm to solve a problem then problem is said to be decidable.

- NP- if there exist a non- polynomial time algo to solve a problem.

Decision properties

- Approximately all the properties are decidable in case a finite automaton. Here we will use machine model to proof decision properties.

i) Emptiness

ii) Non-emptiness

iii) Finiteness

iv) Infiniteness

v) Membership

vi) Equality

Emptiness & Non-emptiness

- Step 1: - select the state that cannot be reached from the initial states & delete them (remove unreachable states)
- Step 2: - if the resulting machine contains at least one final states, so then the finite automata accepts the non-empty language.
- Step 3: - if the resulting machine is free from final state, then finite automata accepts empty language.

Finiteness & Infiniteness

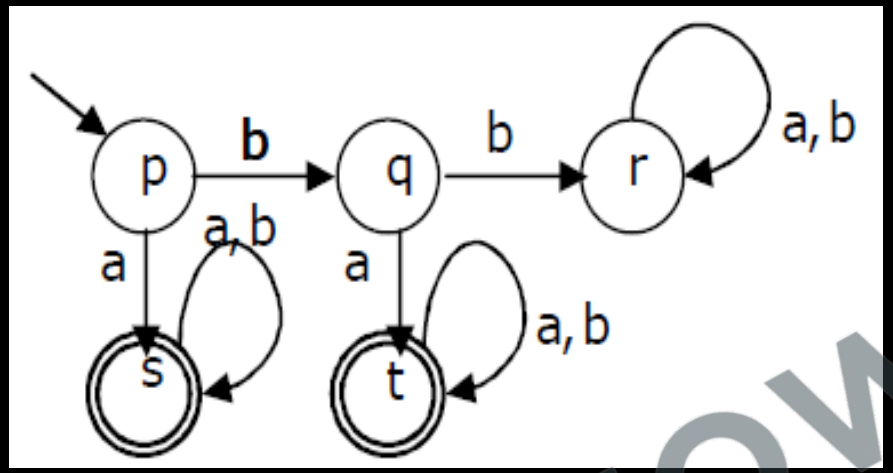
- Step 1: - Select the state that cannot be reached from the initial state & delete them (remove unreachable states)
- Step 2: - Select the state from which we cannot reach the final state & delete them (remove dead states)
- Step 3: - If the resulting machine contains loops or cycles then the finite automata accepts infinite language
- Step 4: - If the resulting machine do not contain loops or cycles then the finite automata accepts finite language.

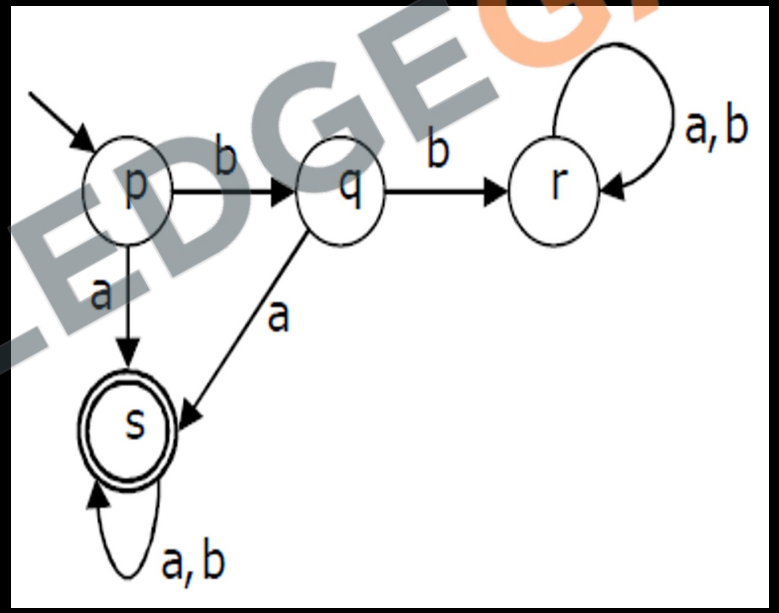
Membership

- Membership is a property to verify an arbitrary string is accepted by a finite automaton or not i.e. it is a member of the language or not.
- Let M is a finite automata that accepts some strings over an alphabet, and let ‘w’ be any string defined over the alphabet, if there exist a transition path in M, which starts at initial state & ends in anyone of the final state, then string ‘w’ is a member of M, otherwise ‘w’ is not a member of M.

Equality

- Two finite state automata M_1 & M_2 is said to be equal if and only if, they accept the same language.
- Minimise the finite state automata and and the minimal DFA will be unique.





	RL	DCFL	CFL	CSL	RS	RES
Emptiness	Y	Y	Y	X	N	N
Non-Emptiness	Y	Y	Y	X	N	N
Finiteness	Y	Y	Y	X	N	N
Infiniteness	Y	Y	Y	X	N	N
Membership	Y	Y	Y	X	Y	N
Equality	Y	N	N	X	N	N
Ambiguity	Y	N	N	X	N	N
Σ^*	Y	N	N	X	N	N
Halting	Y	Y	Y	X	Y	N

- **Chapter-3 (Regular and Non-Regular Grammars):** Context Free Grammar(CFG)-Definition, Derivations, Languages, Derivation Trees and Ambiguity, Regular Grammars-Right Linear and Left Linear grammars, Conversion of FA into CFG and Regular grammar into FA, Simplification of CFG, Normal Forms- Chomsky Normal Form(CNF), Greibach Normal Form (GNF), Chomsky Hierarchy, Programming problems based on the properties of CFGs.

Introduction

- Language usually contains infinite number of strings.
- We cannot tabulate each and every string to represent the language, therefore like automata, grammar is also a mathematical model of representing a language, using which we can generate the entire language.
- Therefore, a grammar is usually thought of as a language generator.

Formal Grammar

A phrase-structure grammar (or simply a grammar) is a 4-tuple (V_N, ∑, P, S), where

- V_N is a finite nonempty set whose elements are called variables,
- ∑ is a finite nonempty set whose elements are called terminals, $V_N \cap \sum = \Phi$.
- S is a special variable (i.e., an element of V_N ($S \in V_n$)) called the start symbol. Like every automaton has exactly one initial state, similarly every grammar has exactly one start symbol.
- P is a finite set whose elements are $\alpha \rightarrow \beta$. where α and β are strings on $V_N \cup \sum$. α has at least one symbol from V_N, the element of P are called productions or production rules or rewriting rules. $\{\Sigma \cup V_n\}^*$ some writer refers it as total alphabet

For a formal valid production

$\alpha \rightarrow \beta$

$\alpha \in \{\Sigma \cup V_n\}^* V_n \{\Sigma \cup V_n\}^*$

$\beta \in \{\Sigma \cup V_n\}^*$

If $G = (\{S\}, \{0, 1\}, \{S \rightarrow 0S1, S \rightarrow \Lambda\}, S)$, find $L(G)$.

Some points to note about productions

- Reverse substitution is not permitted. For example, if S → AB is a production, then we can replace S by AB but we cannot replace AB by S.
- No inversion operation is permitted. For example, if S → AB is a production, it is not necessary that AB →S is a production.

Defining a language by grammar

- The concept of defining a language using grammar is, starting from a start symbol using the production rules of the grammar any time, deriving the string. Here every time during derivation a production is used as its LHS is replaced by its RHS, all the intermediate stages(strings) are called sentential forms. The language formed by the grammar consists of all distinct strings that can be generated in this manner.

$$L(G) = \{w \mid w \in \sum^*, S \rightarrow^* W\}$$

- $\rightarrow^*$(reflexive, transitive closure) means from s we can derive w in zero or more steps

- Using same idea, we do processing of natural languages in computers, Actively used in compliers
- L (G) is the set of all terminal strings derived from the start symbol S.
- G_1 and G_2 are equivalent if L (G_1) = L (G_2).

Chomsky Classification of Languages

- Chomsky classified the grammars into four types in terms of productions (types 0-3).
- This hierarchy of grammars was described by Noam Chomsky in 1956. From type 0 to type 3, we will be putting more and more restrictions.
- We will see that more restrictive is grammar easy will be the language and more liberal is the grammar difficult will be the language. Based on the production rules of the grammar, we can classify the formal grammar into four types, based on which we generate different languages.

- **Avram Noam Chomsky** (born December 7, 1928) is an American linguist, philosopher, cognitive scientist, historian, social critic, and political activist. Sometimes called "the father of modern linguistics",

- Chomsky is also a major figure in analytic philosophy and one of the founders of the field of cognitive science. He is Laureate Professor of Linguistics at the University of Arizona and Institute Professor Emeritus at the Massachusetts Institute of Technology (MIT), and is the author of more than 150 books on topics such as linguistics, war, politics, and mass media. Ideologically, he aligns with anarcho-syndicalism and libertarian socialism.

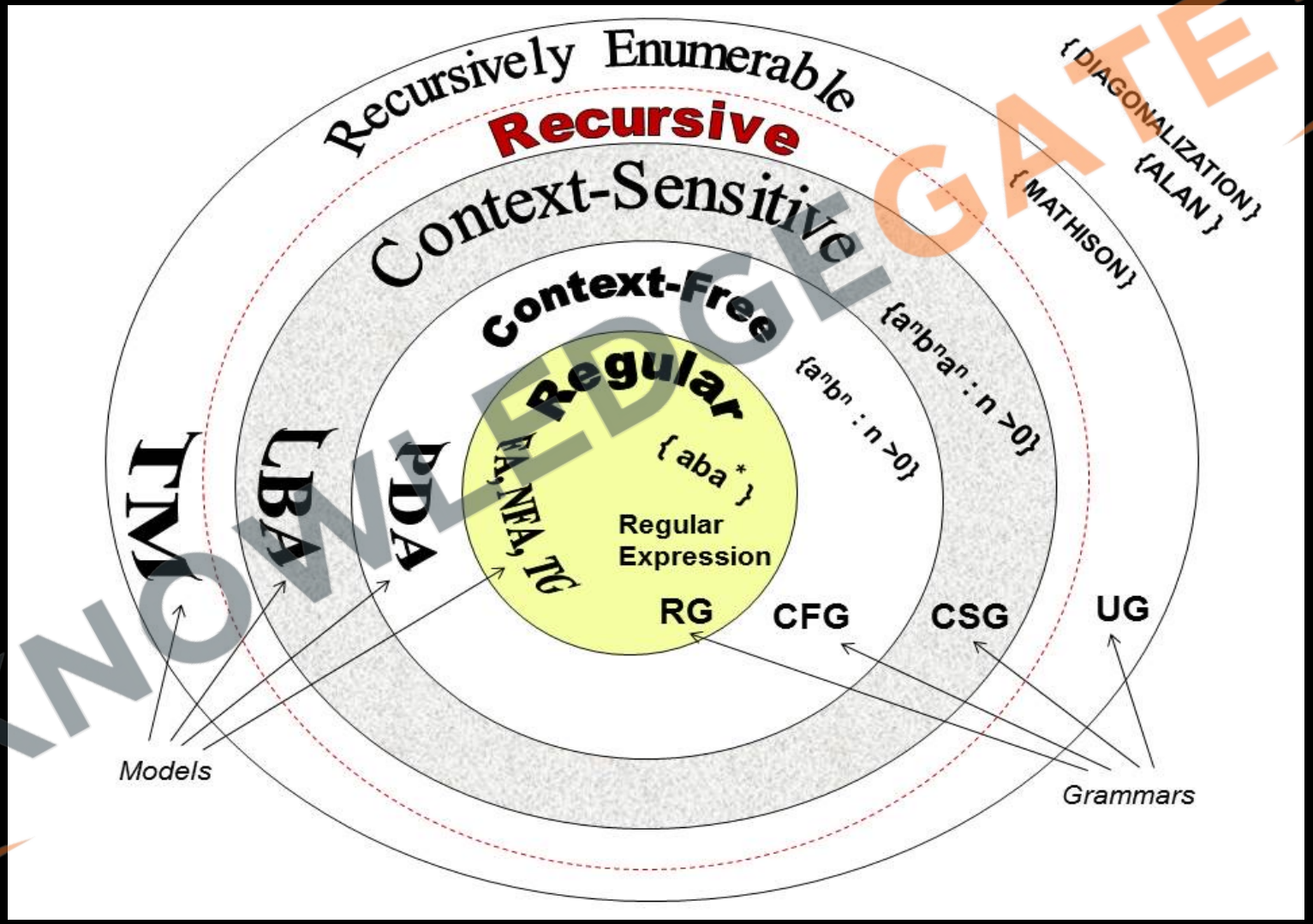
- Born to Jewish immigrants in Philadelphia, Chomsky developed an early interest in anarchism from alternative bookstores in New York City. He studied at the University of Pennsylvania. During his postgraduate work in the Harvard Society of Fellows, Chomsky developed the theory of transformational grammar for which he earned his doctorate in 1955. That year he began teaching at MIT, and in 1957 emerged as a significant figure in linguistics with his landmark work *Syntactic Structures*, which played a major role in remodeling the study of language.

- From 1958 to 1959 Chomsky was a National Science Foundation fellow at the Institute for Advanced Study. He created or co-created the universal grammar theory, the generative grammar theory, the Chomsky hierarchy, and the minimalist program. Chomsky also played a pivotal role in the decline of linguistic behaviorism, and was particularly critical of the work of B. F. Skinner.

- An outspoken opponent of U.S. involvement in the Vietnam War, which he saw as an act of American imperialism, in 1967 Chomsky rose to national attention for his anti-war essay "The Responsibility of Intellectuals".
- Associated with the New Left, he was arrested multiple times for his activism and placed on President Richard Nixon's Enemies List. While expanding his work in linguistics over subsequent decades, he also became involved in the linguistics wars. In collaboration with Edward S. Herman, Chomsky later articulated the propaganda model of media criticism in *Manufacturing Consent* and worked to expose the Indonesian occupation of East Timor.
- His defense of freedom of speech, including Holocaust denial, generated significant controversy in the Faurisson affair of the 1980s. Since retiring from MIT, he has continued his vocal political activism, including opposing the 2003 invasion of Iraq and supporting the Occupy movement. Chomsky began teaching at the University of Arizona in 2017.
- One of the most cited scholars alive, Chomsky has influenced a broad array of academic fields. He is widely recognized as having helped to spark the cognitive revolution in the human sciences, contributing to the development of a new cognitivistic framework for the study of language and the mind.
- In addition to his continued scholarship, he remains a leading critic of U.S. foreign policy, neoliberalism and contemporary state capitalism, the Israeli–Palestinian conflict, and mainstream news media. Chomsky and his ideas are highly influential in the anti-capitalist and anti-imperialist movements.

LANGUAGES AND AUTOMATA

- Following are the machines that accepts the following grammars.



Type 0 Grammar

- Also known as Unrestricted Grammar, phase structured grammar, recursively enumerable grammar used to generate recursive enumerable language which is accepted by a Turing machine.
- A type 0 grammar is without any restrictions.
- No restriction on the production rule, that is if there is a production from

$$\alpha \rightarrow \beta$$

$$\alpha \in \{\Sigma \cup V_n\}^* \, Vn \, \{\Sigma \cup V_n\}^*$$

$$\beta \in \{\Sigma \cup V_n\}^*$$

Type 1 Grammar

- Also known as case sensitive Grammar, length increasing grammar, non-contracting grammar, used to generate context sensitive language which is accepted by a linear bounded automaton.

$\alpha A \beta \rightarrow \alpha \delta \beta$

$\alpha , \beta \in \{\Sigma \cup V_n\}^*$ $A \in V_n$ $\delta \in \{\Sigma \cup V_n\}^+$

or

$\alpha \rightarrow \beta$

$\alpha \in \{\Sigma \cup V_n\}^* Vn \{\Sigma \cup V_n\}^*$

$\beta \in \{\Sigma \cup V_n\}^+$

$|\alpha| <= |\beta|$

- As from the rule we can understand that we cannot have null production, in order to solve that problem, Production S→∈, is allowed if S do not appear on the right-hand side of the production.

- A grammar is called type 1 or context-sensitive or context dependent if all its productions are type 1 productions.

- Very difficult to have a parse tree

- FORTRAN, PL1 with CGF we use STD

Type 2 Grammar

- Also known as Context Free Grammar, which will generate context free language that will be accepted by push down automata. (NPDA default case)

- if there is a production, from

$$\alpha \rightarrow \beta$$

$$\alpha \in V_n \qquad |\alpha| = 1$$

$$\beta \in \{\Sigma \cup V_n\}^*$$

- In other words, the L.H.S. has no left context or right context.

- A grammar is called a type 2 grammar if it contains only type 2 productions.

- Eg ALGOL 60, PASCAL

Type 3 Grammar

- Used to generate Regular Grammar which will generate regular language which will be accepted by finite machine.
- Regular grammar can be of two types either left linear or right linear.
- Left regular grammar, support two types of production

$A \rightarrow a / Ba$

$A, B \in V_n \quad |A| = |B| = 1$

$a \in \sum^*$

- Right regular grammar

$A \rightarrow a / aB$

$A, B \in V_n \quad |A| = |B| = 1$

$a \in \sum^*$

- however, if left-linear rules and right-linear rules are combined, the language need no longer be regular.

Q Consider the following grammar and identify it's language?

S → aAb

A → aB / b

B → c

Q Consider the following grammar and identify it's language?

S → AB / Bb

A → b / c

B → d

Q Consider the following grammar and identify it's language?

$S \rightarrow aSb / \in$

Q Consider the following grammar and identify it's language?

S → aA / abS

A → bS / b

Q Consider the following grammar and identify it's language?

$S \rightarrow aAB$

$A \rightarrow aA \mid \in$

$B \rightarrow b$

Q Consider the following grammar and identify it's language?

$S \rightarrow AB$

$A \rightarrow aA \mid \in$

$B \rightarrow bB \mid b$

Q Consider the following grammar and identify it's language?

$S \rightarrow aSa / bSb / \in$

Q Design a grammar that generates all strings over the alphabet ∑ = {a, b}, where every accepted string ‘w’ starts with substring s = abb

Q Design a grammar that generates all strings over the alphabet ∑ = {a, b}. where every accepted string ‘w’ ends with substring s = bab

Q Design a grammar that generates all strings over the alphabet ∑ = {a, b}. where every accepted string ‘w’ contains sub string s = aba

Q Design a grammar that generates all strings over the alphabet ∑ = {a, b} such that every accepted string start and end with a.

Q Design a grammar that generates all strings over the alphabet ∑ = {a, b}, such that every string ‘w’ accepted must be like

i) |w| = 3 **ii)** |w|<=3 **iii)** |w|>=3

Q Design a grammar that generates all strings over the alphabet ∑= {a, b}, such that every string ‘w’ where |W| = 0(mod 3)?

Q Design a grammar that generates all strings over the alphabet ∑= {a, b}, such that every string ‘w’ where |W| = 3(mod 4)?

Regular Grammar to Regular Expression

http://www.knowledgegate.in/gate

S → 01S / 01

S → 0011S / 01 / 10

S → 01A / B11

A →011A / 01

B → 101B / 11

S → 011A / 101B

A →110A / 00

B → 11B / S

Regular Grammar to (Finite Automata)

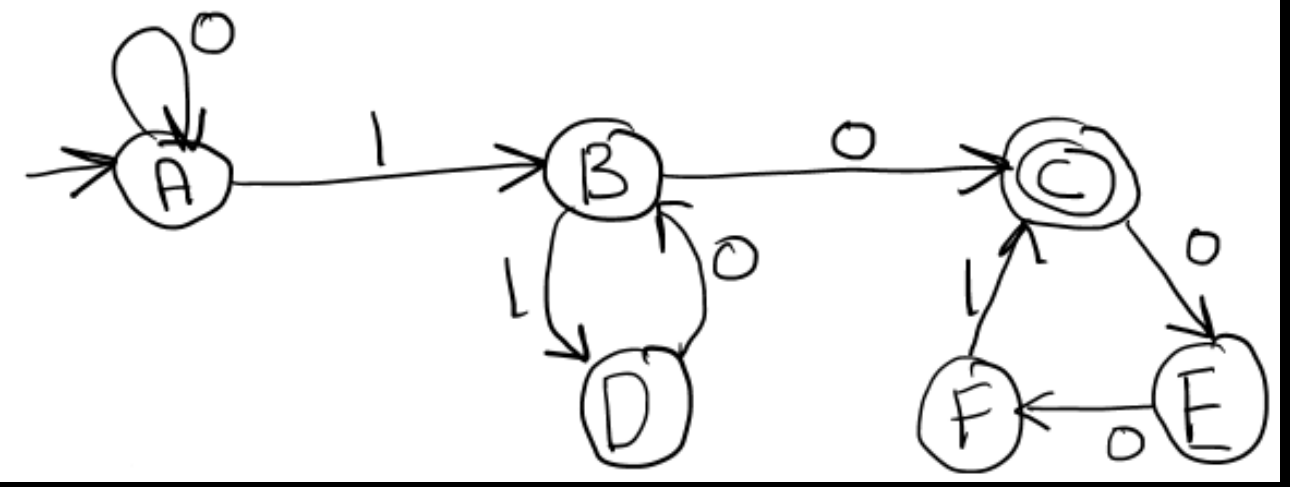
http://www.knowledgegate.in/gate

S → 01S / 1

S → 011S / 01

S → 001S / 10A

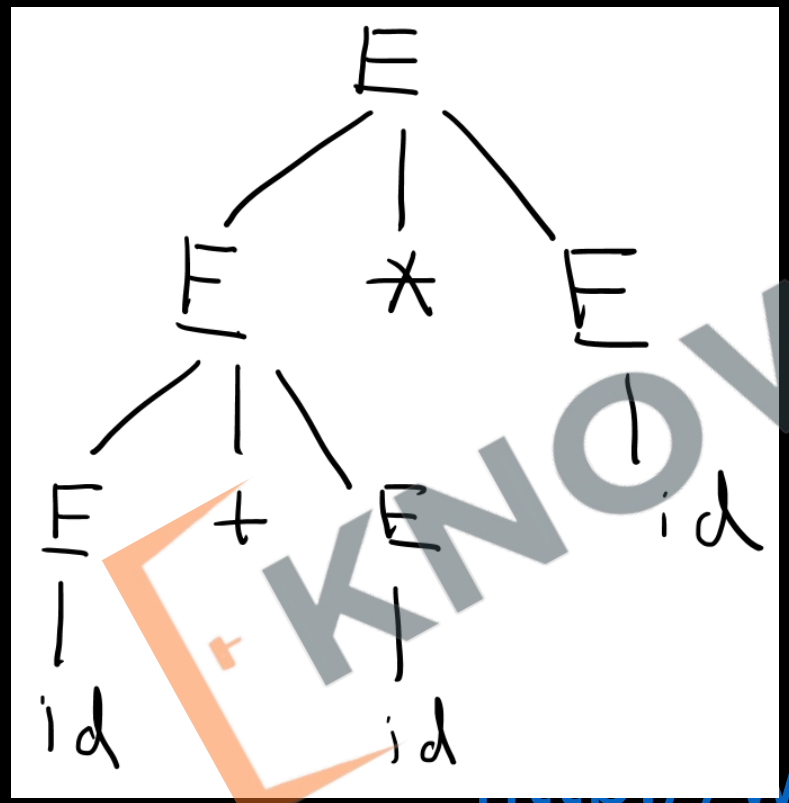
A → 101A / 0 / 1



- **Derivation:** - The process of deriving a string is known as derivation.

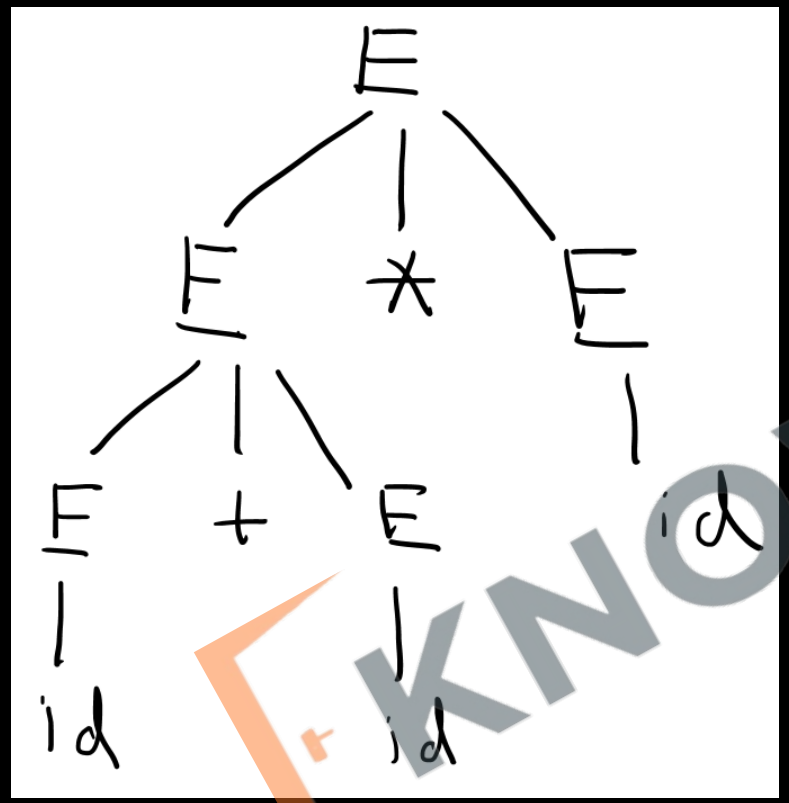
- **Derivation/ Syntax/ Parse Tree**: - The graphical representation of derivation is known as derivation tree.

E → E + E / E * E / E = E / id



- **Sentential form:** - Intermediate step involve in the derivation is known as sentential form.

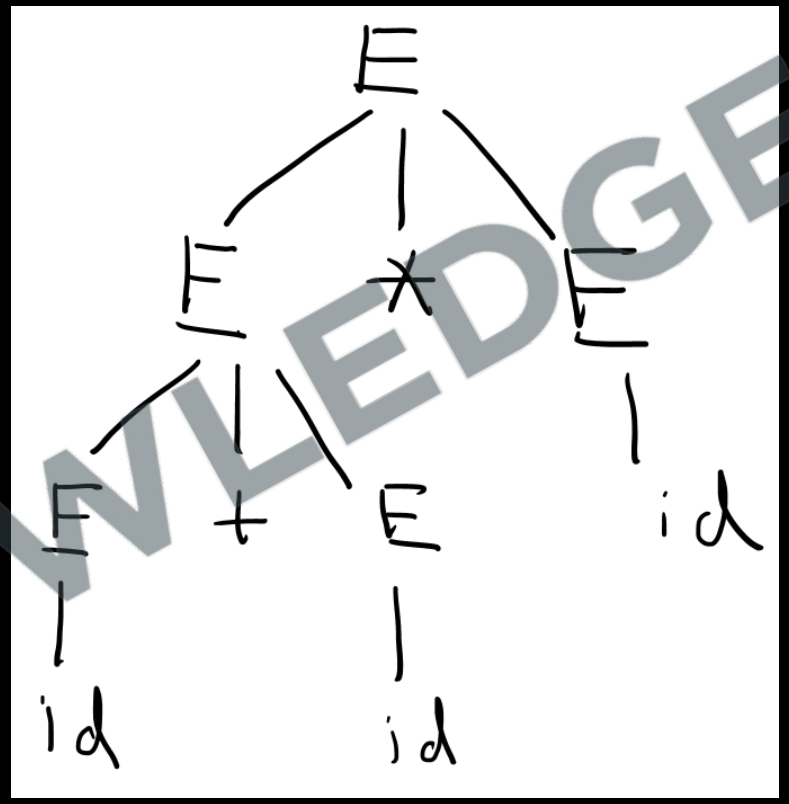
E → E + E / E * E / E = E / id



Sentential Form
E
E*E
E+E*E
ID+E*E
ID+ID*E
ID+ID*ID

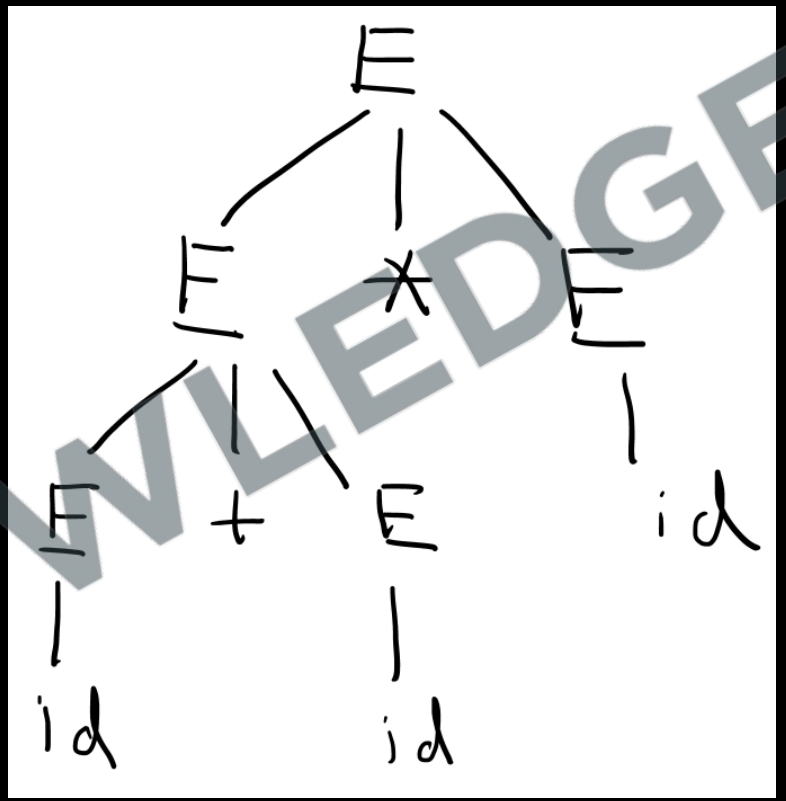
- **Left most derivation**: - the process of construction of parse tree by expanding the left most non terminal is known as LMD and the graphical representation of LMD is known as LMDT (left most derivation tree)

	LMD
E → E + E	E
E→ E * E	E*E
E→ E = E	E+E*E
E→ id	ID+E*E
E→ id	ID+ID*E
E→ id	ID+ID*ID



- **Right most derivation**: - the process of construction of parse tree by expanding the right most non terminal is known as RMD and the graphical representation of RMD is known as RMDT (right most derivation tree)

	RMD
E → E + E	E
E→ E * E	E+E
E→ E = E	E+E*E
E→ id	E+E*ID
E→ id	E+ID*ID
E→ id	ID+ID*ID



Ambiguous grammar: - The grammar CFG is said to be ambiguous if there are more than one derivation tree for any string i.e. if there exist more than one derivation tree (LMDT or RMDT), the grammar is said to be ambiguous.

S → aS/Sa/a

http://www.knowledgegate.in/gate

Unambiguous grammar: - The CFG is said to be unambiguous if there exist only one parse tree for every string i.e. if there exist only one LMDT or RMDT, then the grammar is unambiguous e.g.

S → aSb/ab

- Some CFL are called inherently ambiguous means there exist no unambiguous CFG to generate the corresponding CFL, proved by Rohit Parikh 1961,

- $\{a^n b^m c^m d^n\} \cup \{a^n b^n c^m d^m\}$

- Grammar which is both left and right recursive is always ambiguous, but the ambiguous grammar need not be both left and right recursive.

Simplification or Minimization of CFG

- The reason we simplify CFG to make it more efficient and compiler friendly.
- The process of deleting and eliminating of useless symbols, unit production and null production is known as simplification of CFG.

Removal of Null or Empty productions

- The production of the form A → ∈ is known as null production of empty production, here we try to remove them by replacing equivalent derivation.

S → AbB

A → a / ∈

B → b / ∈

S → AB

A → a / ∈

B → b / ∈

$S \rightarrow aSb / \in$

Removal of unit productions

- The production of the form A → B where A, B ∈ Vn, |A| = |B| = 1, is known as unit production.

S → Aa

A → a / B

B → d

S → aAb

A → B / a

B → C / b

C → D / c

D → d

$S \rightarrow aSb / \in$

Removal of useless symbols

- The variables which are not involved in the derivation of any string is known as useless symbol.
- Select the Variable that cannot be reached from the start symbol of the grammar and remove them along with their all production.

S → aAB
A → a
B → b
C → d

- Select variable that are reachable from the start symbol but which does not derive any terminal, remove them along with their productions

S → aA / aB
A → b

S → aAB / bA / aC

A → aB / b

B → aC / d

Chomsky Normal Form

- The Grammar G is said to be in Chomsky Normal Form, if every production is in the form

$A \rightarrow BC / a$

$B, C \in V_n$

$a \in \Sigma$

- **Avram Noam Chomsky** (born December 7, 1928) is an American linguist, philosopher, cognitive scientist, historian, social critic, and political activist. Sometimes called "the father of modern linguistics",

http://www.knowledgegate.in/gate

S → aSb / ab

S → aAb / bB

A → a / b

B → b

- if CFG is in CNF, then for a derivation of string w, with length we need exactly 2n -1 production. |w| = n, number of sentential forms will be 2n -1

Greiback Normal Form

- The Grammar G is said to be in Greiback Normal Form, if every production is in the form

$$A \rightarrow a\alpha$$

$A \in V_n$

$a \in \Sigma$

$\alpha \in V_n^*$

- **Sheila Adele Greibach** (born 6 October 1939 in New York City) is a researcher in formal languages in computing, automata, compiler theory and computer science.

S → aSb / ab

S → aAb / bB

A → a / b

B → b

- if CFG is in GNF, then for a derivation of string w, with length we need exactly n production. |w| = n, number of sentential forms will be n

- **Chapter-4 (Push Down Automata and Properties of Context Free Languages):** Nondeterministic Pushdown Automata (NPDA)- Definition, Moves, A Language Accepted by NPDA, Deterministic Pushdown Automata(DPDA) and Deterministic Context free Languages(DCFL), Pushdown Automata for Context Free Languages, Context Free grammars for Pushdown Automata, Two stack Pushdown Automata, Pumping Lemma for CFL, Closure properties of CFL, Decision Problems of CFL, Programming problems based on the properties of CFLs.

CONTEXT-FREE LANGUAGES AND PUSH DOWN AUTOMATA

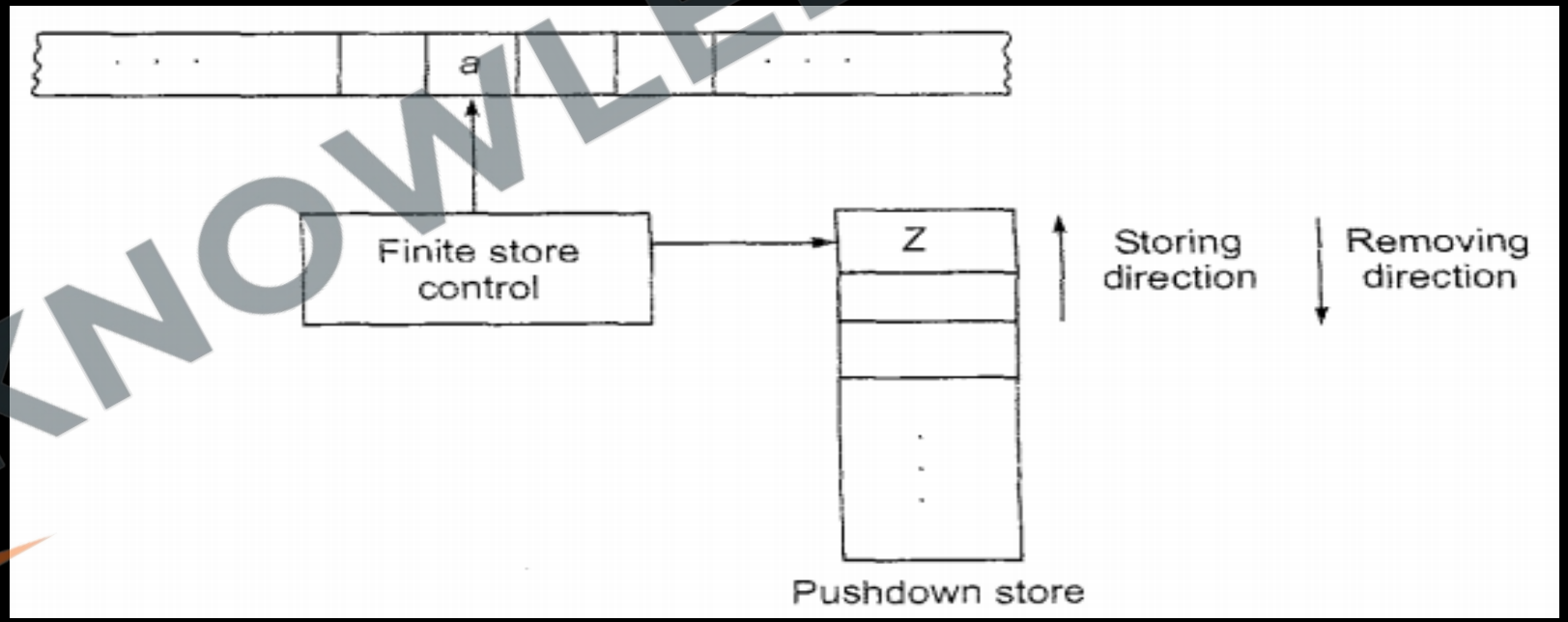
- Context-free languages are applied in parser design.
- They are also useful for describing block structures in programming languages.

Push Down Automata

- We Already understand the limitation of finite automata, that it cannot do the infinite comparison between the symbols.
- Let us consider L = {a^n b^n | n >= 1}. This is not regular, as it has to remember the number of a's in a string and so it will require an infinite number of states, which is logically not possible.
- This difficulty can be avoided by adding an auxiliary memory in the form of a 'stack'. The reason we choose stack because it is the simplest memory possible.
- This type of arrangement where ***a finite automaton has a stack leads to the generation of a pushdown automaton.***

BLOCK DIAGRAM OF PDA

- Finite control unit is also called as memory unit it is static and limited. So to process the i/p string if the static memory is not sufficient then we can use the stack.
- i/p tape is divided into cells where is cell is capable of holding one symbol at a time. At stack of infinite size, which support three operations push, pop and skip.
- The accepting power of a pda is more than that of finite automata and less than that of linear bounder automata
- The power of non-deterministic pda is more than the power of deterministic pda.



Formal Definition of DPDA

A DPDA is a 7-tuple, namely $(Q, \sum, \Gamma, \delta, q_0, Z_0, F)$, where

(i) Q – is a finite nonempty set of states,

(ii) $\sum$ – is a finite nonempty set of input symbols,

(iii) Γ – is a finite nonempty set of pushdown symbols,

(iv) q_0 – is a special state called the initial state,

(v) Z_0 – is a special pushdown symbol called the initial symbol on the pushdown store.

(vi) F – is a set of final states, a subset of Q and

(vii) δ – is a transition function from $Q \times (\sum \cup \{\in\}) \times \Gamma$ to the set of finite subsets of $Q \times \Gamma^*$.

Representation of States

(1) PUSH – one symbol can be inserted into the stack at one time.

$\delta(q_i, a, z_0) = (q_j, az_0)$

Representation of States

(2) POP – one symbol can be deleted from the stack at one time.

$\delta(q_i, a, z_0) = (q_j, \varepsilon)$

Representation of States

(3) SKIP – IT means no stack operation, status of the stack will remain same, before a after the operation

$$\delta(q_i, a, z_0) = (q_j, z_0)$$

note- if pda perform a push or a pop operation at least one's during processing of string than we say that pda is using the stack.

Instantaneous Description (ID)

- An **instantaneous description (ID)** is (q, *x*, α), where q ∈ Q, *x* ∈ ∑* and α ∈ Γ*.
- An initial ID is **(q_0, *x*, Z_0)**, this means that initially the pda is in the initial state q_0, the input string to be processed is *x* and the PDS has only one symbol, namely Z_0.

ACCEPTANCE BY PDA

- There is no change in the language acceptance capability of the pda either we accept by final state of empty stack.

Q Design a Deterministic Push Down Automata for $\{a^n b^n \mid n >= 1\}$?

Q Design a Deterministic Push Down Automata for $\{a^n b^{2n} \mid n >= 1\}$?

Q Design a PDA for $\{w\ c\ w^r \mid w \in (a, b)^*\}$?

Q Construct PDA that accepts $L = \{|w|_{a=b} \mid w \in (a, b)^*\}$?

Non- Deterministic PDA

- Non- Deterministic PDA can also be defined using 7 tuples.
- $\delta: Q \times \{\Sigma \cup \in\} \times \Gamma \rightarrow 2^{(Q \times \Gamma^*)}$
- i.e. on a given input symbol and stack symbol a NPDA can move to more than one state.
- Rest all other tuples are same as DPDA.

Q Construct pda that accepts a language L = {w w^r | w € $(a, b)^*$}?

Decision properties

Following properties are decidable in case a CFL. Here we will use Grammar model to proof decision properties.

i) Emptiness

ii) Non-emptiness

iii) Finiteness

iv) Infiniteness

v) Membership

Q consider the following CFG and identify which of the following CFG generate Empty language?

S → aAB / Aa
A → a

S → aAB
A → a / b

S → aAB / aB
A → aBb
B → aA

Q consider the following CFG and identify which of the following CFG generate Finite language?

S → SS / AB
A → BC / a
B → CC / b

S → AB
A → B / a

S → AB
A → BC / a
B → CC / b
C → AB

Q consider the following CFG and check out the membership properties?

S → AB / BB

A → BA / AS / b

B → AA / SB / a

w_1 = aba

w_2 = abaab

w_3 = abababba

CYK algorithm

- In computer science, the **Cocke–Younger–Kasami algorithm** (alternatively called **CYK**, or **CKY**) is a parsing algorithm for context-free grammars, named after its inventors, John Cocke, Daniel Younger and Tadao Kasami. It employs bottom-up parsing and dynamic programming.

- The standard version of CYK operates only on context-free grammars given in Chomsky normal form (CNF). However any context-free grammar may be transformed to a CNF grammar expressing the same language (Sipser 1997).

- The importance of the CYK algorithm stems from its high efficiency in certain situations. Using Big O notation, the worst case running time of CYK is O (n^3 .|G|), Where n is the length of the parsed string and |G| is the size of the CNF grammar G.

- This makes it one of the most efficient parsing algorithms in terms of worst-case asymptotic complexity, although other algorithms exist with better average running time in many practical scenarios.

Following properties are Undecidable in case a CFL

i) Equality

ii) Ambiguity

	RL	DCFL	CFL	CSL	RS	RES
Emptiness	Y	Y	Y	X	N	N
Non-Emptiness	Y	Y	Y	X	N	N
Finiteness	Y	Y	Y	X	N	N
Infiniteness	Y	Y	Y	X	N	N
Membership	Y	Y	Y	X	Y	N
Equality	Y	N	N	X	N	N
Ambiguity	Y	N	N	X	N	N
Σ^*	Y	N	N	X	N	N
Halting	Y	Y	Y	X	Y	N

Closure Properties of Deterministic Context Free Languages

- **Deterministic Context Free Languages** are closed under following operations
  - Complement
  - Intersection with regular set
  - Inverse Homeomorphism

- **Deterministic Context Free Languages** are not closed under following operations
  - Union
  - Concatenation
  - Kleen closure
  - homomorphism
  - Substitution
  - Reverse operator
  - Intersection

Closure Properties of Context Free Languages

- **Context Free Languages** are closed under following operations
  - Union
  - Concatenation
  - Kleen Closure
  - Substitution
  - Homomorphism
  - Inverse Homomorphism
  - Reverse Operator
  - Intersection with regular set

- **Context Free Languages** are not closed under following operations
  - Intersection
  - Complement
  - Symmetric Difference

	RL	DCFL	CFL	CSL	RS	RES
Union	Y	N	Y	Y	Y	Y
Intersection	Y	N	N	Y	Y	Y
Complement	Y	Y	N	Y	Y	N
Set Difference	Y	N	N	Y	Y	N
Kleene Closure	Y	N	Y	Y	Y	Y
Positive Closure	Y	N	Y	Y	Y	Y
Concatenation	Y	N	Y	Y	Y	Y
Intersection with regular set	Y	Y	Y	Y	Y	Y
Reverse	Y	Y	Y	Y	Y	Y
Subset	N	N	N	N	N	N

	RL	DCFL	CFL	CSL	RS	RES
Homomorphism	Y	N	Y	N	N	Y
∈ Free Homomorphism	Y	N	Y	Y	Y	Y
Inverse Homomorphism	Y	Y	Y	Y	Y	Y
Substitution	Y	N	Y	N	N	Y
∈ Free Substitution	Y	N	Y	Y	Y	Y
Quotient with regular set	Y	Y	Y	N	Y	Y

- **Chapter-5 (Turing Machines and Recursive Function Theory):** Basic Turing Machine Model, Representation of Turing Machines, Language Acceptability of Turing Machines, Techniques for Turing Machine Construction, Modifications of Turing Machine, Turing Machine as Computer of Integer Functions, Universal Turing machine, Linear Bounded Automata, Church's Thesis, Recursive and Recursively Enumerable language, Halting Problem, Post's Correspondance Problem, Introduction to Recursive Function Theory.

Turing Machine

- *The Church-Turing thesis states that any algorithmic procedure that can be carried out by human beings/computer can be carried out by a Turing machine.*

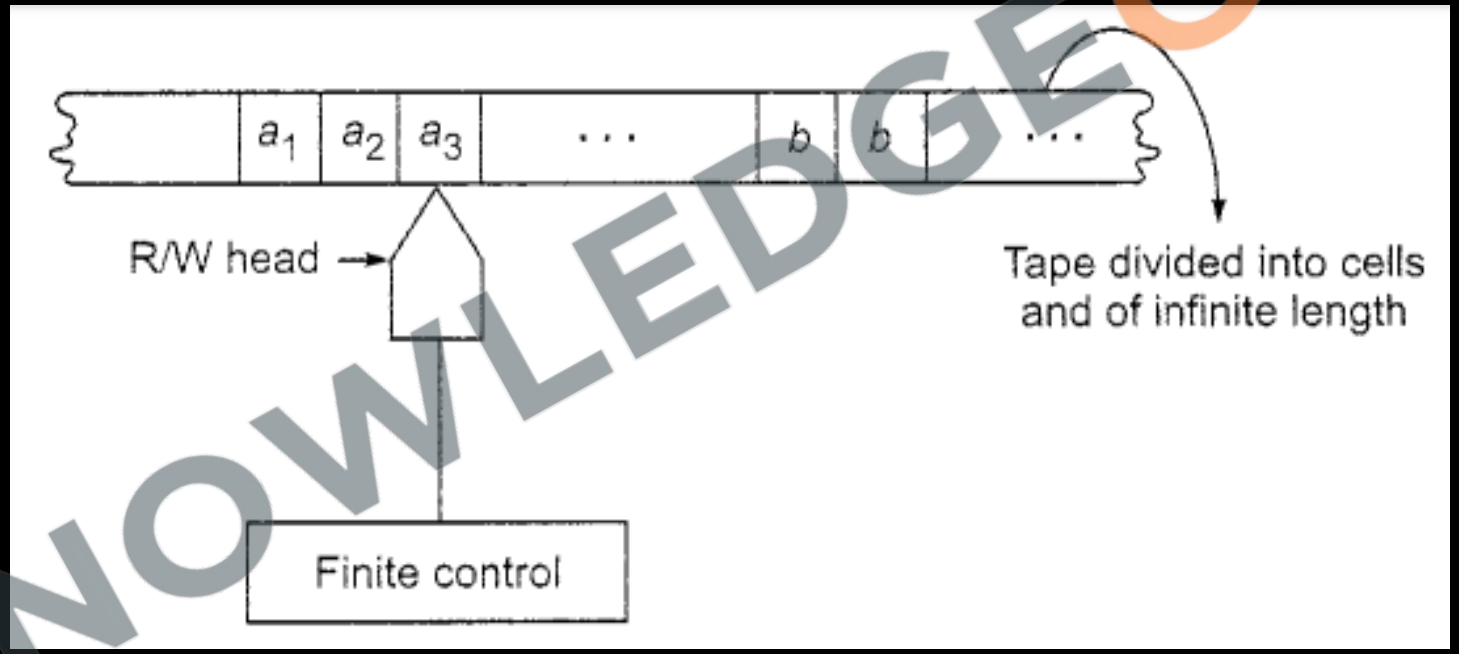
- It has been universally accepted by computer scientists that the Turing machine provides an ideal theoretical model of a computer.

- Turing machines are useful in several ways:
- As an automaton, the Turing machine is the most general model.
- It accepts type-0 languages.
- It can also be used for computing functions.
- Turing machines are also used for determining the undecidability of certain languages and measuring the space and time complexity of problems.

Components of Turing Machine

Infinite Tape

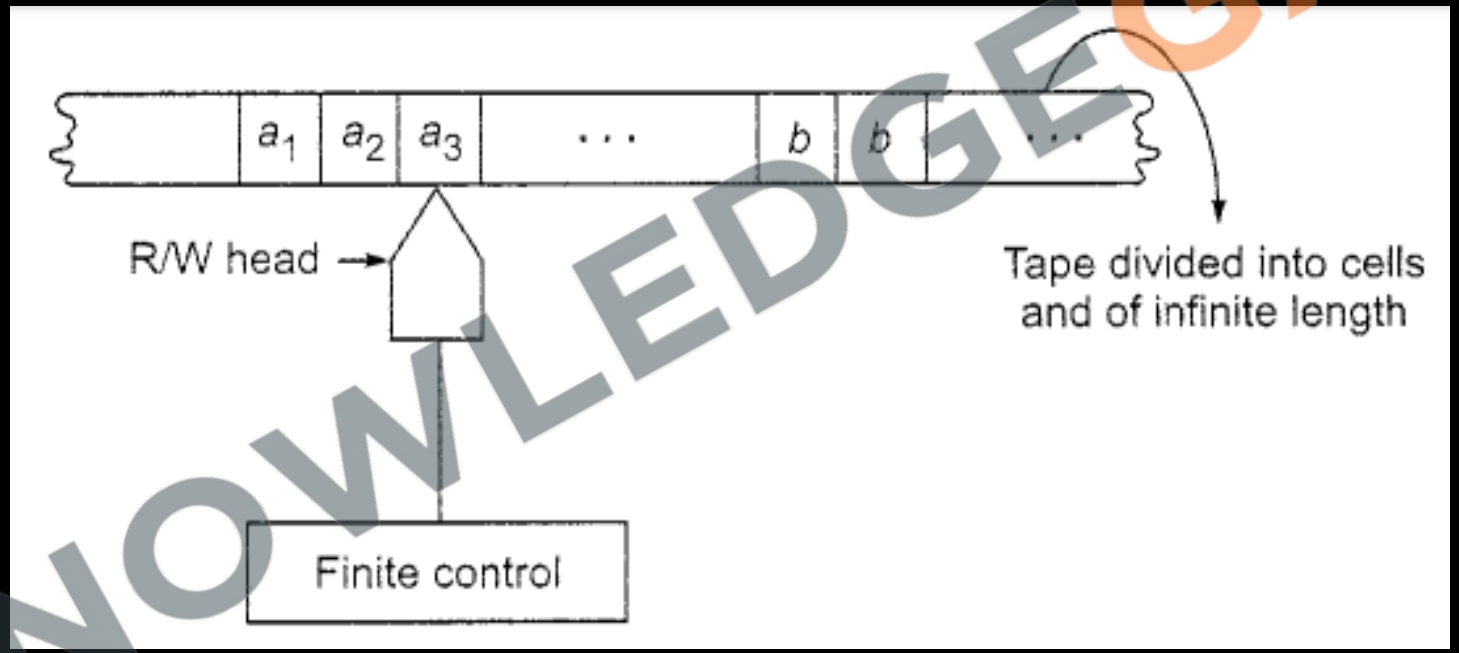
- An input infinite tape can be, used as an input buffer and also as a member element.
- It is random accessible. It is of infinite length divided into cells and each cell is capable of holding only one tape symbol.
- It is also called 2-way infinite tape.



Components of Turing Machine

Read-Write Buffer

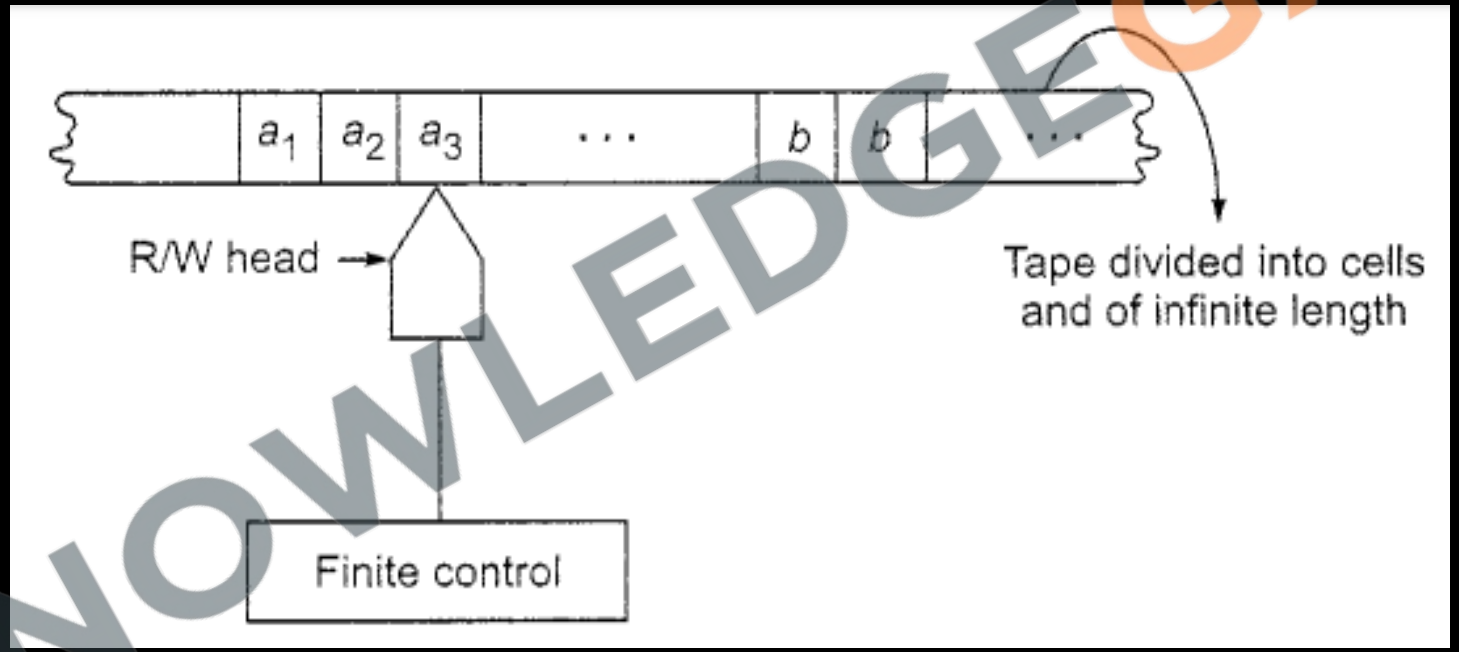
- Read the data from the tape and can also write the data over the tape. After reading the symbol from the tape, the read/write header moves to exactly one cell either in left or right direction.



Components of Turing Machine

Finite Control Unit

- As per the control unit, the transitions will take place, and will finally lead to some output.



FORMAL DEFINITION

A Turing machine *M* is a 7-tuple, namely (Q, ∑, Γ, δ, q_0, b, F), where

- Q is a finite nonempty set of states.
- Γ is a finite nonempty set of tape symbols,
- b ∈ Γ is the blank.
- ∑ is a nonempty set of input symbols and is a subset of Γ and b ∉∑.
- δ is the transition function mapping δ: Q ×𝕃 -> Q× 𝕃 × (L/R). It says that, on providing a tape symbol, from a particular state there will be a transition to another state, with a different or same tape symbol with defining whether next the machine needs to move in left/ right.
- q_0 ∈ Q is the initial state, and
- F ⊆ Q is the set of final states.

Q Design a Turing machine for L = {$a^n b^n$ | n >= 1}?

REPRESENTATION OF TURING MACHINES

Three representations:

- Instantaneous descriptions using move-relations.
- Transition table, and
- Transition diagram (transition graph).

LANGUAGE ACCEPTABILITY BY TURING MACHINES

- A string *w* in $\sum^*$ is said to be accepted by a TM(*M*) if after parsing the string w Turing machine must halts on final state.
- $q_0 w \vdash^* \alpha_1 p \alpha_2$ for some $p \in F$ and $\alpha_1, \alpha_2 \in \Gamma^*$.
- M does not accept *w* if the machine *M* either halts in a non-accepting state or does not halt and goes into a loop.

Q Design a Turing machine for $L = \{a^n b^n c^n \mid n >= 0\}$?

Q Design a Turing machine for L = {w c w | w ∈ {0, 1}* }?

Q Design a Turing machine for addition of two number in unary?

Q Design a Turing machine for converting unary number into binary number?

Versions of Turing Machine

- On the basis of determinism of the next transition to a state on a particular input, Turing Machine are divided in two types-
  - Determinism Turing Machine.
  - Non-Deterministic Turing Machine.

- In non-deterministic Turing machine, there can be more than one possible move for a given state and tape symbol, but non-deterministic TM does not add any power.
- ***Every non-deterministic TM can be converted into deterministic TM.***

Versions of Turing Machine

- There are multiple versions of Turing Machine which are as follows-
- Multi-tape Turing Machine.
  - In multi-tape Turing machine, there can be more than one tape and corresponding head pointers, but it does not add any power to Turing machine.
  - Every multi-tape TM can be converted into single tape TM.
  - Every language accepted by a multi-tape TM is acceptable by some single-tape TM (that is, the standard TM).
- Multi Read/Write head points.
- Multi-dimensional Turing Machine.
- TM with stay option δ: $Q \times \Gamma \rightarrow Q \times \Gamma \times (L/R/S)$.
- TM with one way infinite tape (Semi infinite tape)

- Offline TM, this TM have a restriction that input can not be changed
- Jumping TM, where it is allowed to take more that one moves in one transaction δ: Q $\times \mathbb{\Gamma}$ -> Q$\times \mathbb{\Gamma} \times$ (L/R) $\times\{n\}$.
- Non erasing TM, (where input can not be converted to blank)
- Always writing TM, (after reading a symbol from tape it must be replaced with other symbol)
- Multidimensional TM δ: Q $\times \mathbb{\Gamma}$ -> Q$\times \mathbb{\Gamma} \times$ (L/R/U/D)
- Multi-head TM
- FA with a Queue
- TM with only 3 states
- Multi-tape Turing Machine with stay option and at most 2 states.
- Non-Deterministic TM δ: Q $\times \mathbb{\Gamma}$ -> $2^{Q \times \mathbb{\Gamma} \times (L/R)}$
- A NPDA with two independent stacks δ: Q $\times(\Sigma \cup \in) \times \mathbb{\Gamma} \times \mathbb{\Gamma}$ -> $2^{Q \times \mathbb{\Gamma}^* \times \mathbb{\Gamma}^*}$

Q Consider the Turing machine M defined below
Choose the false statement

a) The machine loops on 01

b) The machine does not accept 0000

c) The machine accepts strings ending with a 1

d) The machine accepts strings ending with 10

	0	1	B
$\rightarrow Q_0$	$(Q_0, 0, R)$	$(Q_2, 1, L)$	$(Q_f, -, -)$
Q_1	$(Q_2, 1, L)$	$(Q_1, 1, R)$	$(Q_f, -, -)$
Q_2	$(Q_2, 1, L)$	$(Q_2, 0, L)$	$(Q_f, -, -)$
$*Q_f$	---	---	---

Halt

- The state of Turing machine, where no transition is defined or required is known as Halt. There are two types of Halt-
  - Final Halt- Machine has been halted on final state, then it is known as Final halt and hence it depicts that machine is accepted.
  - Non-Final Halt-Machine has been halted on non- final state, then it is known as non- final halt and hence the string is rejected.

- After taking an input string, there are three possibilities for Turing Machine
  - It may go to Final halt.
  - It may go to Non- Final halt.
  - It may go into Infinite loop.

- Final Halt and Non- Final Halt have been described above. After taking an input string, if the machine goes to infinite loop, then we can't say whether the string is Accepted/Rejected.

- So, broadly Recursively Enumerable Languages are categorized as-
  - **Recursive Set**- The Language L, which is accepted by Turing Machine, is said to be recursive set, where for all, 'w' that belongs to L, machine will go to final halt, and for all 'w' that does not belongs to L, machine will go to non-final halt. Hence, membership property is defined here.
  - **Recursively Enumerable Set**- The language L, which is accepted by Turing Machine is said to be REL, where for all, 'w' that belongs to L, machine will go to final halt, and for all 'w' that does not belongs to L, machine will either go to non-final halt or infinite loop. Hence, membership property is not defined here.

- If a set and its complement both are recognizable, then the set is decidable.
- There are some sets which are not recognizable (even members can't be identified) because there are more languages than programs.

Universal Turing Machine

Universal Turing Machine

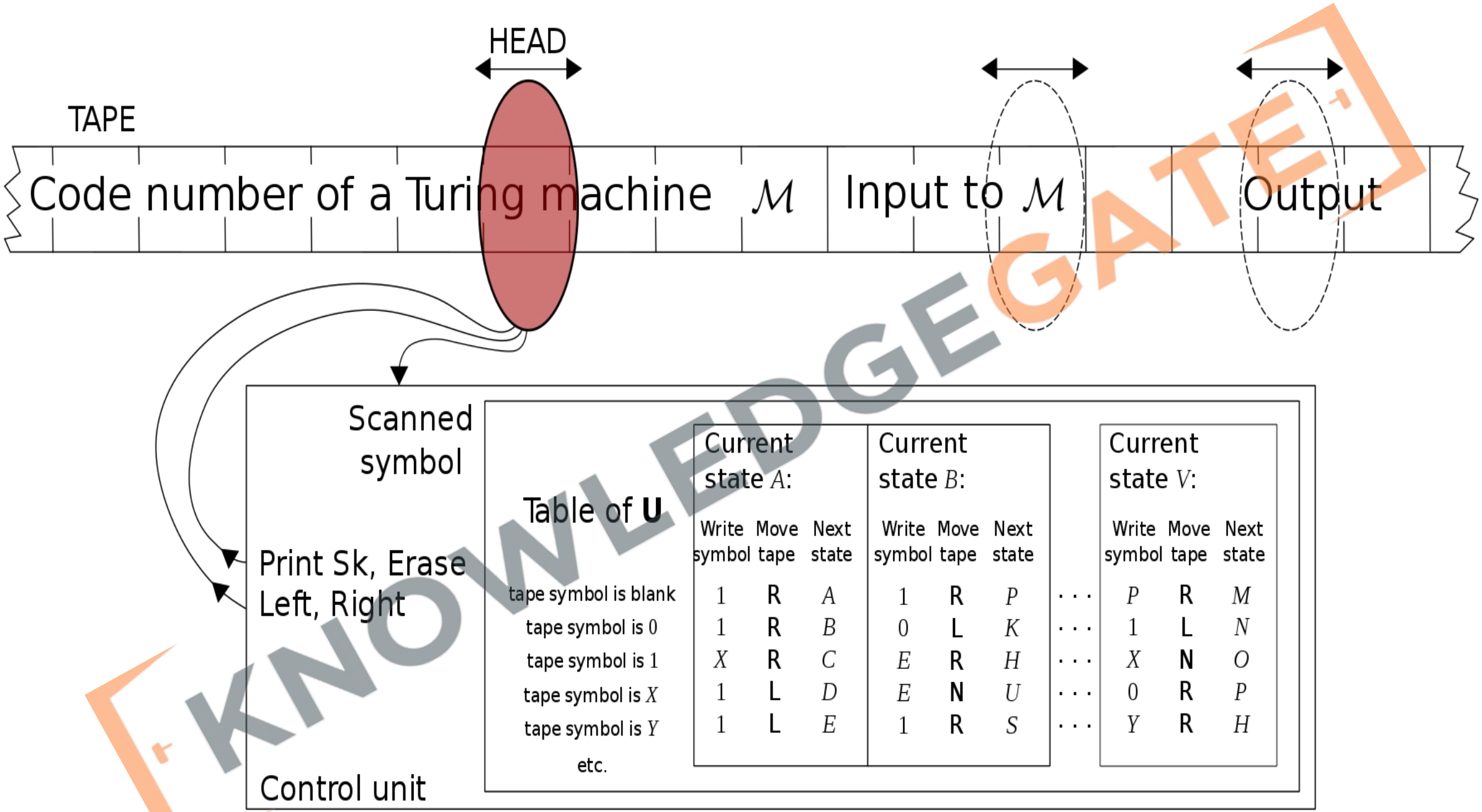
- In computer science, a **universal Turing machine** (**UTM**) is a Turing machine that simulates an arbitrary Turing machine on arbitrary input.

- The universal machine essentially achieves this by reading both the description of the machine to be simulated as well as the input to that machine from its own tape.

- Every Turing machine computes a certain fixed partial computable function from the input strings over its alphabet. In that sense it behaves like a computer with a fixed program.

- However, we can encode the action table of any Turing machine in a string. Thus we can construct a Turing machine that expects on its tape a string describing an action table followed by a string describing the input tape, and computes the tape that the encoded Turing machine would have computed. Turing described such a construction in complete detail in his 1936 paper:

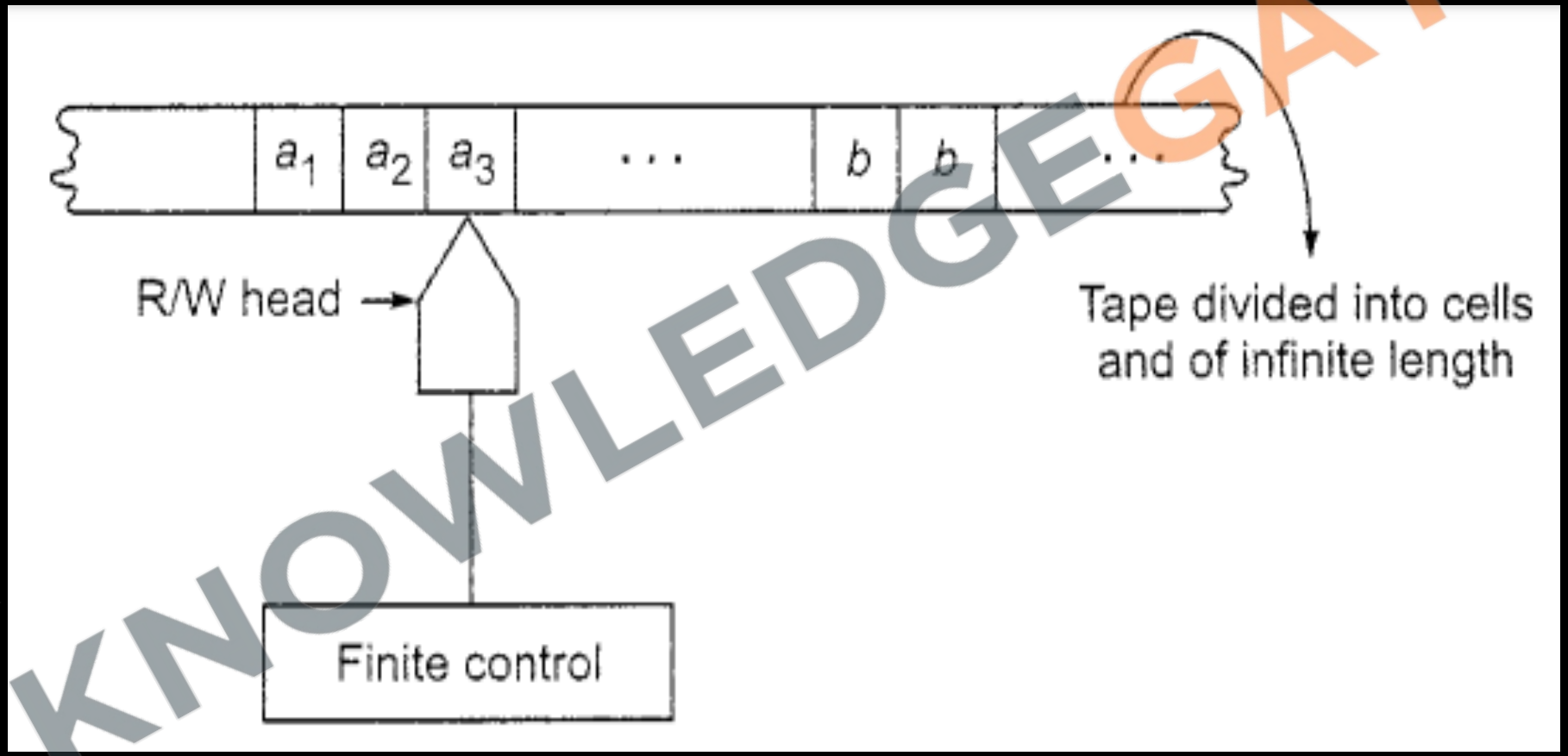
- "It is possible to invent a single machine which can be used to compute any computable sequence. If this machine **U** is supplied with a tape on the beginning of which is written the S.D ["standard description" of an action table] of some computing machine **M**, then **U** will compute the same sequence as **M**.



Linear Bounded Automata

A Linear Bounded Automaton (LBA) is similar to Turing Machine with property stated below:

- Turing Machine with a bounded finite length of the tape.



Turing-Church Thesis

- **Concept Origin**: Independently developed by Alan Turing and Alonzo Church in the 1930s, establishing a fundamental principle in computer science about computable functions.

1.Turing Machines: Turing proposed the concept of a 'Turing machine', a theoretical computing machine that can simulate any algorithm's logic.

2.Church's Lambda Calculus: Church introduced lambda calculus, a formal system for expressing computation based on function abstraction and application.

3.Equivalent Models: The thesis states that what can be computed on a Turing machine can also be computed in Church's lambda calculus, implying both models are equivalent in their computational power.

The Post Correspondence Problem (PCP)

- is a significant problem in the field of theoretical computer science. It was introduced by Emil Post in 1946 and is known for its undecidability. Here are the main points:
  - **Basic Concept**: The problem involves two lists of words (strings of symbols) over a common alphabet. The challenge is to find a sequence of indices where the concatenation of the words in the first list, in that sequence, is equal to the concatenation of the words in the second list in the same sequence.

- A = {1, 110, 0111}
- B = {111, 001, 11}

- **Undecidability**: The Post Correspondence Problem is known to be undecidable, meaning there is no algorithm that can determine for every instance of the problem whether or not a solution exists. This undecidability is a crucial aspect in the theory of computation, particularly in understanding the limits of algorithmic solvability.

- A = {b, babbb, ba}
- B = {bbb, ba, a}

Decision properties

- Following properties are decidable in case a RS.
  - Membership

- All properties are undecidable in case of a REL.

	RL	DCFL	CFL	CSL	RS	RES
Emptiness	Y	Y	Y	X	N	N
Non-Emptiness	Y	Y	Y	X	N	N
Finiteness	Y	Y	Y	X	N	N
Infiniteness	Y	Y	Y	X	N	N
Membership	Y	Y	Y	X	Y	N
Equality	Y	N	N	X	N	N
Ambiguity	Y	N	N	X	N	N
Σ^*	Y	N	N	X	N	N
Halting	Y	Y	Y	X	Y	N

Closure Properties of Recursive Set

- Recursive languages are closed under following operations
  - Union
  - Concatenation
  - Intersection
  - Reverse
  - Complement
  - Inverse homomorphism
  - Intersection with regular set
  - Set Difference

- Recursive languages are not closed under following operations
  - Kleen closure
  - Homomorphism
  - Substitution

Closure Properties of Recursive Enumerable Set

- Recursive Enumerable are closed under following operations
  - Union
  - Concatenation
  - Kleen Closure
  - Intersection
  - Substitution
  - Homomorphism
  - Inverse Homomorphism
  - Intersection with regular set
  - Reverse operation

- Recursive Enumerable are not closed under following operations
  - Compliment
  - Set Difference

	RL	DCFL	CFL	CSL	RS	RES
Union	Y	N	Y	Y	Y	Y
Intersection	Y	N	N	Y	Y	Y
Complement	Y	Y	N	Y	Y	N
Set Difference	Y	N	N	Y	Y	N
Kleene Closure	Y	N	Y	Y	Y	Y
Positive Closure	Y	N	Y	Y	Y	Y
Concatenation	Y	N	Y	Y	Y	Y
Intersection with regular set	Y	Y	Y	Y	Y	Y
Reverse	Y	Y	Y	Y	Y	Y
Subset	N	N	N	N	N	N

	RL	DCFL	CFL	CSL	RS	RES
Homomorphism	Y	N	Y	N	N	Y
∈ Free Homomorphism	Y	N	Y	Y	Y	Y
Inverse Homomorphism	Y	Y	Y	Y	Y	Y
Substitution	Y	N	Y	N	N	Y
∈ Free Substitution	Y	N	Y	Y	Y	Y
Quotient with regular set	Y	Y	Y	N	Y	Y