Theory of Computation

Notes By: **J.JAGADEESAN.,**

Asst. Professor of Computer Science

Arignar Anna Govt. Arts & Sci. College, Karaikal

Theory of Computation – Unit-1

- Alphabets
- Strings
- Language
- Basic Operations on Language
    - Concatenation
    - Union
    - Kleene Star

---

1. Alphabet (Σ)

Definition:

An **alphabet** is a **finite, non-empty set of symbols**. These symbols are the basic building blocks used to construct strings and languages.

Examples:

- Σ = {0, 1} → Binary alphabet
- Σ = {a, b, c, ..., z} → Lowercase English letters
- Σ = {a, b}
- Σ = {x, y, z, 1, 2, 3}

Notes:

- The alphabet is denoted by **Σ** (capital sigma).
- The alphabet must not be empty.
- Symbols in the alphabet are **atomic** (they cannot be broken down further in formal language theory).

---

2. String (Word)

Definition:

A **string** is a **finite sequence of symbols** from an alphabet.

Examples (for Σ = {a, b}):

- "a", "b", "ab", "ba", "aab", "baba"
- "" (empty string) is also a valid string, denoted by **ε**

Notation:

- **ε** = empty string (string of length 0)
- **Length of a string w** is denoted as |w|

Notes:

- Strings can be **concatenated**.
- Every string is made **only** from symbols in the given Σ.
- If w = "abc", then |w| = 3.

---

3. Language

Definition:

A **language** is a **set of strings** formed using the symbols of an alphabet Σ.

Examples:

- Let Σ = {0,1}
  - L = {ε, 0, 1, 00, 01, 10, 11}
  - L = {w | w contains equal number of 0's and 1's}
  - L = {w | w ends in 0}
- Finite language: L = {a, ab, abc}
- Infinite language: L = $\{a^n \mid n \geq 0\}$ = {ε, a, aa, aaa, …}

Types:

- **Finite Language** – has a limited number of strings
- **Infinite Language** – contains infinitely many strings

---

4. Basic Operations on Languages

Languages can be manipulated using several operations. Here are the fundamental ones:

4.1 Concatenation (L_1L_2)

Definition:
Concatenation of two languages **L_1 and L_2** is the set of strings formed by taking any string from L_1 and appending any string from L_2.

Notation:
$L_1 \cdot L_2$ or simply L_1L_2

Formally:
$L_1L_2 = \{ xy \mid x \in L_1 \text{ and } y \in L_2 \}$

Example:
Let $L_1 = \{a, b\}$, $L_2 = \{c, d\}$
→ $L_1L_2 = \{ac, ad, bc, bd\}$

Properties:

- Concatenation is **not necessarily commutative**: $L_1L_2 \neq L_2L_1$ in general
- **ε** is the identity element for concatenation: $\varepsilon L = L\varepsilon = L$

4.2 Union ($L_1 \cup L_2$)

Definition:
The union of two languages **L_1 and L_2** is the set containing all strings that are in **either L_1, L_2, or both**.

Notation:
$L_1 \cup L_2$

Formally:
$L_1 \cup L_2 = \{ x \mid x \in L_1 \text{ or } x \in L_2 \}$

Example:
$L_1 = \{ab, a\}$, $L_2 = \{a, b, c\}$
→ $L_1 \cup L_2 = \{ab, a, b, c\}$

Properties:

- Union is **commutative**: $L_1 \cup L_2 = L_2 \cup L_1$
- Union is **associative**

---

*4.3 Kleene Star (L)**

Definition:
The Kleene Star operation represents the **set of all strings** that can be formed by concatenating **zero or more strings** from a language L.

Notation:
L*

Formally:
$L^* = \cup (L^n)$ for $n = 0$ to ∞
Where:

- $L^0 = \{\varepsilon\}$
- $L^1 = L$
- $L^2 = LL = \{xy \mid x, y \in L\}$
- ...

Example:
$L = \{a\}$
→ $L^* = \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$

$L = \{ab\}$
→ $L^* = \{\varepsilon, ab, abab, ababab, \ldots\}$

Properties:

- Always includes ε
- Infinite if $L \neq \{\varepsilon\}$

Additional Concepts:

Power of a Language: L^n

- $L^0 = \{\varepsilon\}$
- $L^1 = L$
- $L^2 = L \cdot L$
- $L^3 = L \cdot L \cdot L$
- L^n = n-times concatenation of L with itself

Kleene Plus (L^+)

- Similar to Kleene Star, but excludes ε
- $L^+ = L \cdot L^*$

Summary Table

Operation	Symbol	Meaning
Concatenation	L_1L_2	All strings formed by $x \in L_1$ followed by $y \in L_2$
Union	$L_1 \cup L_2$	All strings in either L_1, L_2, or both
Kleene Star	L^*	All possible concatenations (0 or more times) of L
Kleene Plus	L^+	All possible concatenations (1 or more times) of L

Unit II - Regular Expressions & Deterministic Finite Automata (DFA)

---

1) Regular Languages at a Glance

Regular languages are the simplest robust class of formal languages. They can be characterized equivalently by:

- Regular expressions (REs)
- Deterministic finite automata (DFAs)
- Nondeterministic finite automata (NFAs)
- ε-NFAs (NFAs with ε-moves)
- Right/left-linear grammars

All these models define **exactly** the same family: the **regular languages**.

---

2) Regular Expressions (RE)

2.1 Alphabet and Strings

- **Alphabet (Σ)**: A finite, nonempty set of symbols (e.g., Σ = {0,1} or {a,b}).
- **String**: Finite sequence of symbols from Σ. The empty string is **ε**.
- **Language**: Any set of strings over Σ.

2.2 Syntax of Regular Expressions (Inductive Definition)

Base cases (atomic REs):

1. **Ø** is a regular expression denoting the empty language **Ø**.
2. **ε** is a regular expression denoting the language {ε}.
3. **a** is a regular expression for every symbol a ∈ Σ, denoting {a}.

Recursive formation rules: If **R** and **S** are regular expressions, then so are:

2.2.1 (R)|(S) (union/alternation), denoting **L(R) ∪ L(S)**.

Meaning:

- If you have two regular expressions R and S,
- R | S means “strings that are in **either** L(R) **or** L(S) (or both).”
- L(R) means “the language defined by R” (set of strings R matches).

Example:

- R = cat → L(R) = { "cat" }
- S = dog → L(S) = { "dog" }
- R | S → L(R) ∪ L(S) = { "cat", "dog" }

This is called **alternation** because it gives you a choice.

2.2.2 (R)(S) (concatenation), denoting **L(R) · L(S)** = { xy : x∈L(R), y∈L(S) }.

Meaning:
- Take one string from L(R) and **immediately** follow it with one string from L(S).
- That's why we write L(R) • L(S) = { xy : x ∈ L(R), y ∈ L(S) }.

Example:
- R = `hi` → L(R) = { "hi" }
- S = `there` → L(S) = { "there" }
- RS → { "hithere" }

If R = { "a", "b" } and S = { "x", "y" },

- RS = { "ax", "ay", "bx", "by" }.

2.2.3 (R) — **Kleene** Star*

Meaning:
- `(R)*` means "zero or more repetitions of strings from L(R)."
  where:
  - $L(R)^0$ = { ε } (zero repetitions → empty string)
  - $L(R)^1$ = L(R) (one repetition)
  - $L(R)^2$ = { xy : x ∈ L(R), y ∈ L(R) } (two repetitions), and so on.

Example:
- R = `a` → L(R) = { "a" }
- R* = { ε, "a", "aa", "aaa", "aaaa", ... }

If R = { "ab" },
- R* = { ε, "ab", "abab", "ababab", ... }

- **(R)*** (Kleene star), denoting **L(R)*** = ⋃_{k≥0} L(R)^k, with L^0 = {ε}.

2.3 Shorthand and Precedence

- Often write **R|S** instead of (R)|(S), and **RS** for (R)(S).
- **Kleene star** has highest precedence, then **concatenation**, then **union**.
- + (Kleene plus): R^+ = RR*.
- **Optional**: R? = (R|ε).
- **Character classes** (e.g., [abc]) are conveniences; formally they expand by union.

2.4 Semantics: Language Denotation L(R)

Defined by the inductive clauses above. Key points:

- $\varepsilon \in L(R)^*$ always, because k=0 case gives ε.
- $\emptyset = \{\varepsilon\}^*$ (non-intuitive but important!).
- **εR = Rε = R** (as languages, via concatenation with ε).

2.5 Algebraic Laws (Useful Identities)

For all REs R,S,T (interpreting equality as language equivalence):

- **Union**: R|S = S|R (commutative); (R|S)|T = R|(S|T) (associative); R|R = R (idempotent); R|∅ = R.
- **Concatenation**: (RS)T = R(ST) (associative); R∅ = ∅R = ∅; Rε = εR = R.
- **Distributive**: R(S|T) = RS | RT; (R|S)T = RT | ST.
- **Star**: R** = R*; ε ∈ R*; R* = ε | RR* = ε | R*R.
- **Arden's Lemma** (for linear language equations): If X = AX | B and ε ∉ A, then the least solution is **X = A*B**.

2.6 Typical Design Patterns

- Strings over {0,1} with **even number of 0s**: (1*01*01*)*1*.
- **All strings not containing 11**: (ε|0)(10)*1?.
- **Strings over {a,b} ending with ab**: (a|b)*ab.
- **Decimal integers without leading zeros**: 0 | ([1-9][0-9]*).

2.7 Worked Examples

1. Σ={a,b}. Language: strings with at least one **a** and at least one **b**.
    - One RE: (a|b)*a(a|b)*b(a|b)* | (a|b)*b(a|b)*a(a|b)*.
    - Alternative (shorter but more advanced): (a|b)*a(a|ab)*b(a|b)* | (a|b)*b(b|ba)*a(a|b)*.
2. Σ={0,1}. Language: **binary numbers divisible by 3**.

- Direct RE is tedious; better to build DFA for residues mod 3 and (optionally) convert to RE (state-elimination or GNFA). This illustrates practical limits of RE design vs. DFA construction.

2.8 Decision Properties (Regular Languages)

- **Emptiness**: decidable (e.g., DFA reachability of a final state).
- **Finiteness**: decidable.
- **Membership**: linear-time in |input| via DFA.
- **Equivalence/Containment**: decidable (e.g., via DFA minimization or product construction + emptiness test).

---

3) From RE to Automata (High-Level)

Every RE can be converted to an ε-NFA (e.g., Thompson construction). Then:

RE → ε-NFA → NFA → DFA (subset construction) → minimized DFA.

This pipeline proves: **REs and DFAs are equivalent in expressive power**.

---

4) Deterministic Finite Automata (DFA)

4.1 Formal Definition

A DFA is a 5-tuple $\mathbf{M = (Q, \Sigma, \delta, q_0, F)}$ where:

- **Q**: finite, nonempty set of states.
- **Σ**: finite input alphabet.
- $\boldsymbol{\delta: Q \times \Sigma \to Q}$: total transition function (exactly one next state for each state/symbol pair).
- $\mathbf{q_0 \in Q}$: start state.
- $\mathbf{F \subseteq Q}$: set of accept (final) states.

4.2 Computation and Acceptance

- On input $x = a_1a_2\ldots a_n$, the DFA starts at q_0 and iteratively applies δ.
- Let $\boldsymbol{\hat{\delta}(q, x)}$ (extended δ) be defined inductively: $\hat{\delta}(q, \varepsilon)=q$; $\hat{\delta}(q, xa)=\delta(\hat{\delta}(q, x), a)$.
- **Acceptance**: x is accepted iff $\hat{\delta}(q_0, x) \in F$. The **language of M**, L(M), is the set of all accepted strings.

4.3 Design Techniques

- **Direct counting/parity** (e.g., even number of 0s ⇒ 2 states toggling on 0).
- **Modular arithmetic** (e.g., residues mod k for divisibility properties).
- **Remembering last k symbols** (k+1 states often suffice for fixed-length suffixes/prefixes).
- **Product construction** to combine constraints: states are pairs (p,q) from machines M_1 and M_2.

4.4 Closure Properties via Automata

Regular languages are closed under:

- **Union, Intersection, Difference** (product DFA + appropriate accepting sets).
- **Complement** (swap accepting/non-accepting states in a complete DFA).
- **Concatenation, Kleene star** (easiest by NFA/ε-NFA constructions).
- **Reversal** (via NFA on reversed edges; or GNFA/RE methods).
- **Homomorphism & inverse homomorphism**.

4.5 Equivalence of DFA and NFA

- For any NFA **N**, there is a DFA **D** such that L(D)=L(N) (subset construction). DFA states correspond to subsets of N's states.
- Nondeterminism does not increase expressive power for finite automata, only potential succinctness.

4.6 DFA Minimization

Goal: find a DFA with the **fewest states** recognizing the same language.
Two mainstream views:

1. **Table-filling (distinguishability) algorithm**:
    - Mark pairs (p,q) where one is accepting and the other not.
    - Propagate markings: (p,q) is distinguishable if some $a \in \Sigma$ leads to a marked pair $(\delta(p,a), \delta(q,a))$.
    - Unmarked pairs are equivalent and can be merged.
2. **Partition-refinement (Hopcroft)**:
    - Start with $P = \{F, Q \setminus F\}$; split blocks using transitions until stable.
    - Hopcroft's algorithm runs in $O(|\Sigma|\,|Q| \log |Q|)$.

Myhill–Nerode theorem (conceptual foundation):

- A language L is regular iff it has **finitely many** Myhill–Nerode equivalence classes.

- Each class corresponds to a unique state in the minimal DFA; two strings x,y are equivalent iff for all z, $xz \in L \Leftrightarrow yz \in L$.

4.7 Proving Nonregularity (Contrast)

Although beyond DFA per se, it's vital to know limits:

- **Pumping lemma** for regular languages: If L is regular, there exists p (pumping length) such that any string $s \in L$ with $|s| \geq p$ can be decomposed $s=xyz$ with $|xy| \leq p$, $|y| \geq 1$, and for all $i \geq 0$, $xy^iz \in L$.
- Typical use: to show certain languages (e.g., $\{a^n b^n : n \geq 0\}$) are **not** regular.

---

5) Constructions & Proof Sketches

5.1 Thompson-Style Constructions (RE → ε-NFA)

- **Atom**: a ⇒ q —a→ r.
- **Union**: new start with ε to starts of R and S; new final with ε from their finals.
- **Concatenation**: connect final of R to start of S with ε.
- **Star**: new start/final; ε to R's start and to new final; ε from R's final back to R's start and to new final.

5.2 Subset Construction (NFA → DFA)

- Start set: ε-closure($\{q_0\}$).
- For each DFA state $T \subseteq Q\_NFA$ and symbol a, transition to ε-closure($\bigcup_{q \in T} \delta\_NFA(q,a)$).
- Accepting if T contains any accepting NFA state.

5.3 Complement / Intersection (DFA-level)

- **Complement**: Ensure DFA is **complete** (every state has all Σ-transitions). Then flip accepting status.
- **Intersection**: Product automaton with states (p,q), start (p_0,q_0), accepting $F_1 \times F_2$.

---

6) Comprehensive Examples

Example A: Even number of 0s over Σ={0,1}

- **RE**: (1*01*01*)*

- **DFA**:
  - Q={E,O}, start E, F={E}
  - δ(E,0)=O, δ(O,0)=E; δ(E,1)=E, δ(O,1)=O.

Example B: Strings over {a,b} with no substring abb

- Track the longest suffix that is also a prefix of "abb": states for ε, a, ab, and a dead state for reaching abb.
- Accept all except the dead state.

Example C: Binary numbers divisible by 3

- States: residues {0,1,2}; start 0; F={0}.
- $\delta(r, b) = (2r + b) \bmod 3$ for $b \in \{0,1\}$.

Example D: Ends with 01 over {0,1}

- **RE**: (0|1)*01
- **DFA**: states for how much of the suffix we've matched: qε, q0, q01(accept).

---

7) Equivalence Proof Outline (RE ↔ DFA)

1. **RE ⇒ ε-NFA** via structural induction (Thompson). Therefore L(RE) is accepted by some NFA.
2. **ε-NFA ⇒ DFA** via subset construction. Hence L(RE) is accepted by some DFA.
3. **DFA ⇒ RE**: Use **state-elimination** or **GNFA** to produce an equivalent RE. Thus both formalisms characterize the same languages.

---

8) Minimization Example (Sketch)

Given a DFA with states {A,B,C,D}, Σ={0,1}, F={C,D}:

1. Partition $P_0 = \{\{C,D\},\{A,B\}\}$.
2. Split using transitions until stable, e.g., if from A on 0 goes to C (accept) but from B on 0 goes to B (reject), then A and B are distinguishable.
3. Merge only truly indistinguishable states to obtain the minimal DFA.

---

9) Practical Tips for Exams/Design

- Start with **informal idea** of what needs to be remembered (parity, last k symbols, modulo state, forbidden pattern) and map that to states.
- Prefer **DFA** for counting and modular properties; prefer **RE** for local/positional constraints.
- When combining constraints, use **product** and then **minimize**.
- For simplification of REs, apply identities (idempotence, distributivity, Arden's lemma).

---

10) Short Exercise Set (Self-Check)

1. Give an RE for strings over {a,b} where the number of a's is even.
2. Design a DFA over {0,1} for strings that contain **001** as a substring.
3. Prove closure of regular languages under reversal.
4. Convert the RE (a|b)*abb to a DFA via ε-NFA and subset construction.
5. Use the pumping lemma to show {0^p : p is prime} is not regular.

Unit III Regular Languages

Non-Deterministic Finite Automata (NFA) – Relationship Between NFA and DFA – Transition Graphs (TG) – Properties of Regular Languages– The Relationship Between Regular Languages and Finite Automata–Kleene's Theorem

Regular Languages

A *regular language* is a formal language that can be described using a finite set of rules. It is recognized by finite automata and can be expressed using *regular expressions*. Regular languages are the simplest type of languages in the Chomsky hierarchy and are important because they describe many real-world problems such as searching for words in text, designing compilers, and validating input formats. For instance, the set of all binary strings that contain an even number of 0s is a regular language, since it can be represented using a simple automaton or expression.

One of the most important properties of regular languages is *closure*. They are closed under operations such as union, intersection, concatenation, complement, reversal, and Kleene star. This means that if we combine two regular languages using these operations, the result will still be a regular language. Another significant feature is that every regular language can be represented in multiple equivalent forms: as a DFA, an NFA, a transition graph, or a regular expression. However, regular languages also have limitations. They cannot represent languages requiring memory of past inputs, such as balanced parentheses or palindromes. This limitation helps in distinguishing regular languages from more complex ones like context-free or context-sensitive languages.

Introduction of Finite Automata

Finite automata are abstract machines used to recognize patterns in input sequences, forming the basis for understanding regular languages in computer science.

- Consist of states, transitions, and input symbols, processing each symbol step-by-step.
- If ends in an accepting state after processing the input, then the input is accepted; otherwise, rejected.
- Finite automata come in deterministic (DFA) and non-deterministic (NFA), both of which can recognize the same set of regular languages.
- Widely used in text processing, compilers, and network protocols.

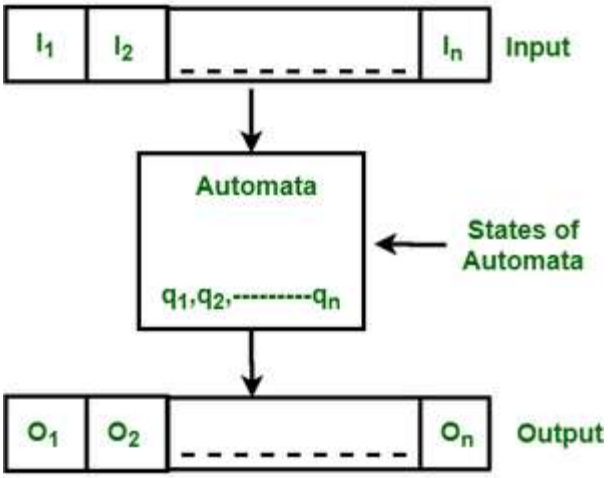


Figure: Features of Finite Automata

Features of Finite Automata

- **Input:** Set of symbols or characters provided to the machine.
- **Output:** Accept or reject based on the input pattern.
- **States of Automata:** The conditions or configurations of the machine.
- **State Relation:** The transitions between states.
- **Output Relation:** Based on the final state, the output decision is made.

Formal Definition of Finite Automata

- A finite automaton can be defined as a tuple:

- **Q** → Set of all states (finite, non-empty).
  Example: Q={q0,q1,q2}.
- **Σ (Sigma)** → Input alphabet (finite set of symbols).
  Example: Σ={0,1}.
- **q (or q_0)** → Initial/start state (an element of Q).
  Example: q_0 is the starting state.
- **F** → Set of final/accepting states (subset of Q).
  Example: F={q2}.
- **δ (delta)** → Transition function.
  - For DFA: $\delta: Q \times \Sigma \rightarrow Q$
  - For NFA: $\delta: Q \times \Sigma \rightarrow 2^Q$ (maps to a set of states).

So formally, a **finite automaton (FA)** is defined as a **5-tuple**:

M=(Q,Σ,δ,q0,F)M = (Q, Σ, δ, q_0, F)M=(Q,Σ,δ,q0,F)

Where each symbol has the meaning listed above.

Types of Finite Automata

There are two types of finite automata:

- ❖ Deterministic Finite Automata (DFA)
- ❖ Non-Deterministic Finite Automata (NFA)

1. Deterministic Finite Automata (DFA)

A DFA is represented as {Q, Σ, q, F, δ}. In DFA, for each input symbol, the machine transitions to one and only one state. DFA does not allow any null transitions, meaning every state must have a transition defined for every input symbol.

DFA consists of 5 tuples {Q, Σ, q, F, δ}.

Q : set of all states.

Σ : set of input symbols. (Symbols which machine takes as input)

q : Initial state. (Starting state of a machine)

F : set of final state.

δ : Transition Function, defined as δ : Q X Σ --> Q.

Example:

Construct a DFA that accepts all strings ending with 'a'.

Given:

Σ = {a, b},

Q = {q0, q1},

F = {q1}

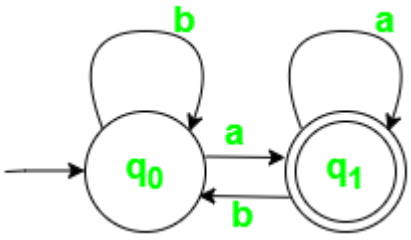


Fig 1. State Transition Diagram for DFA with Σ = {a, b}

State\Symbol	a	b
q0	q1	q0
q1	q1	q0

In this example, if the string ends in 'a', the machine reaches state q1, which is an accepting state.

2) Non-Deterministic Finite Automata (NFA)

A *Non-Deterministic Finite Automata (NFA)* is a computational model used to recognize regular languages. In an NFA, for a given state and input symbol, the machine can move to *zero, one, or multiple states*. This means the path taken by the automaton is not uniquely determined, unlike in a DFA. An important feature of NFAs is the presence of *ε-transitions*, which allow the automaton to change states without consuming any input. This property makes NFAs more flexible and easier to design than DFAs.

Although NFAs appear more powerful than DFAs, both have the same expressive power. Every NFA can be converted into an equivalent DFA that accepts the same language. However, this conversion may cause an exponential increase in the number of states, making DFAs less compact. NFAs are mainly used for theoretical descriptions and for simplifying the construction of automata, whereas DFAs are preferred in practice for their determinism in implementation. Thus, NFAs play a key role in automata theory by serving as an intermediate step in designing automata from regular expressions.

NFA is similar to DFA but includes the following features:

- It can transition to multiple states for the same input.
- It allows null (ϵ) moves, where the machine can change states without consuming any input.

Example:

Construct an NFA that accepts strings ending in 'a'.

Given:

Σ = {a, b},
Q = {q0, q1},
F = {q1}

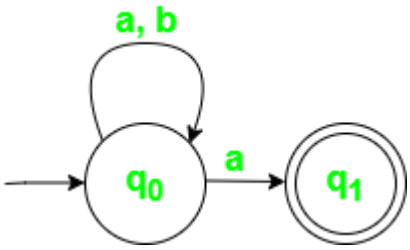


Fig 2. State Transition Diagram for NFA with Σ = {a, b}

State Transition Table for above Automaton,

State\Symbol	a	b
q0	{q0,q1}	q0
q1	φ	φ

In an NFA, if any transition leads to an accepting state, the string is accepted.

Relationship Between NFA and DFA

1. Basic Concept

- Both **Deterministic Finite Automata (DFA)** and **Non-deterministic Finite Automata (NFA)** are models of computation used to recognize **regular languages**.
- Even though NFAs appear more powerful because they allow *multiple transitions* (including ε-moves), **any NFA can be converted into an equivalent DFA** that recognizes the same language.
- Therefore, **NFA and DFA are equivalent in expressive power** — they both accept exactly the set of **regular languages**.

---

2. Structural Differences

- **DFA**:
  - For each state and input symbol, **exactly one transition** is defined.
  - No ε-moves are allowed.
  - Easier to implement in hardware/software because of determinism.
- **NFA**:
  - For a state and input, **zero, one, or multiple transitions** may be possible.
  - ε-moves (state changes without consuming input) are allowed.
  - Easier to design because of flexibility.

Transition Graphs (TG)

Transition Table :

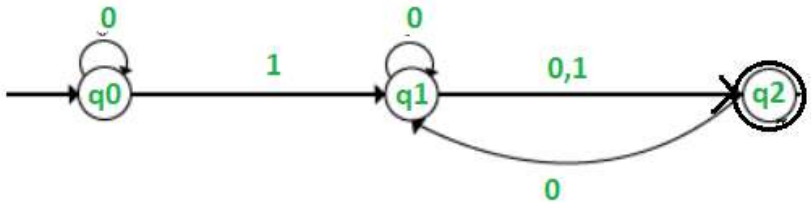
Transition function(∂) is a function which maps Q * ∑ into Q . Here 'Q' is set of states and '∑' is input of alphabets. To show this transition function we use table called transition table. The table takes two values a state and a symbol and returns next state.

A transition table gives the information about -

1. Rows represent different states.
2. Columns represent input symbols.
3. Entries represent the different next state.
4. The final state is represented by a star or double circle.
5. The start state is always denoted by an small arrow.

Example 1 -

This example shows transition table for NFA(non-deterministic finite automata) .



Transition Graph

Present State	Next State Of Input 0	Next State For Input 1
->q0	q0	q1
q1	q1, q2	q2
*q2	q1	Nill

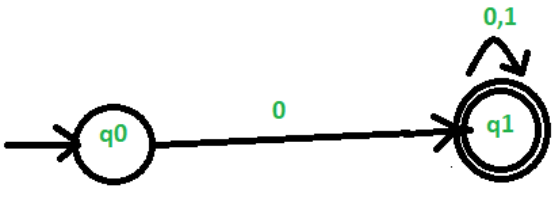
Transition Table

Explanation of above table -

1. First column indicates all the present states ,Next for input 0 and 1 respectively.
2. When the present state is q0, for input 0 the next state will become q0. For input 1 the next state is q1.
3. When the present state is q1, for input 0 the next state is q1 or q2, and for 1 input the next state is q2.
4. When the current state is q2 for input 0, the next state will become q1, and for 1 input the next state will become Nil.
5. The small straight arrow on q0 indicates that it is a start state and circle on to q3 indicates that it is a final state.

Example 2 :

This example shows transition table of DFA(deterministic finite automata).



Present State	Next State Of Input 0	Next State For Input 1
->q0	q1	q1
*q1	q1	q1

Explanation of above table -

1. First column indicates all the present states, Next for input 0 and 1 respectively.
2. When the current state is q0, for input 0 the next state will become q1 and for input as 1 the next state is q1.

3. When the current state is q1, for input 0, the next state will become q1, and on 1 input the next state is q1.
4. The small straight arrow on q0 indicates that it is a start state and circle on to q3 indicates that it is a final state.

Properties of Regular Languages

Regular languages exhibit several important properties that make them very useful. The most important ones are *closure properties*. Regular languages are closed under union, concatenation, and Kleene star, which means that combining two regular languages using these operations results in another regular language. They are also closed under intersection, difference, complementation, and reversal, further expanding their usefulness. These properties make it easier to construct new regular languages from existing ones.

Another key property is that regular languages can be represented in multiple equivalent ways: as regular expressions, as DFAs or NFAs, or using transition graphs. However, regular languages have limitations—they cannot express constructs that require an unbounded memory. For example, the language of balanced parentheses `{ a^n b^n | n ≥ 0 }` is not regular, because a finite automaton cannot keep track of the exact number of `a`s to match with `b`s. To prove that a language is not regular, the *pumping lemma for regular languages* is often used. This property-based reasoning is fundamental in distinguishing regular languages from more powerful ones like context-free languages.

The Relationship between Regular Languages and Finite Automata

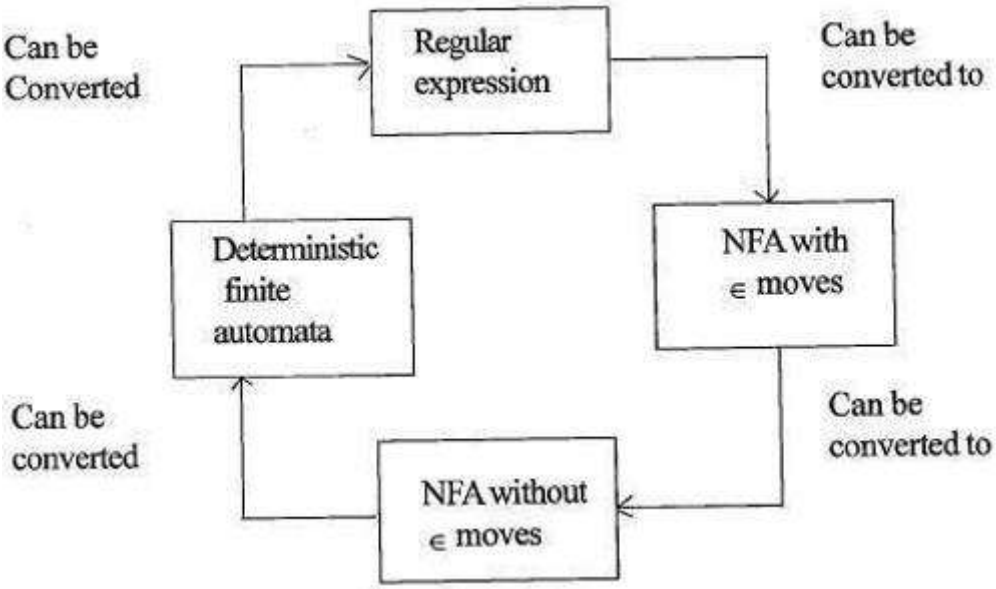
The relationship between regular languages and finite automata is central to automata theory. Every regular language can be recognized by some finite automaton, and every finite automaton corresponds to a regular language. This means that the two concepts are essentially equivalent. Regular expressions provide an algebraic way of describing languages, while finite automata provide a machine model that accepts them. This duality forms the foundation for both theoretical studies and practical applications.

In practical terms, finite automata are often used to implement regular languages in computer systems. For example, in lexical analysis (the first phase of a compiler), patterns for

tokens are described using regular expressions. These regular expressions are then converted into finite automata for efficient recognition of tokens. Thus, finite automata provide the computational mechanism, and regular languages provide the theoretical framework, establishing a strong link between the two.

Relationship

The relationship between FA and RE is as follows –



The above figure explains that it is easy to convert

- RE to Non-deterministic finite automata (NFA) with epsilon moves.
- NFA with epsilon moves to without epsilon moves.
- NFA without epsilon moves to Deterministic Finite Automata (DFA).
- DFA can be converted easily to RE.

Kleene's Theorem

Kleene's Theorem, named after **Stephen C. Kleene (1956)**, is a **fundamental result in Automata Theory**.
It establishes the **equivalence** between:

1. **Regular Expressions (RE)** – algebraic representation of patterns.
2. **Finite Automata (FA: DFA/NFA/ε-NFA)** – machine model for recognizing patterns.
3. **Regular Languages (RL)** – the set of languages described by regular expressions and accepted by finite automata.

A language is **regular** ⇔ it can be described by a **regular expression** ⇔ it can be accepted by a **finite automaton**.

Statement of the Theorem

Kleene's Theorem has **two parts**:

1. **Part I (RE → FA):**
   If a language can be described by a **regular expression**, then there exists a **finite automaton** (DFA or NFA) that accepts it.
2. **Part II (FA → RE):**
   If a language is accepted by a **finite automaton**, then there exists a **regular expression** describing it.

Thus, **Regular Expressions** and **Finite Automata** are equivalent in expressive power.

Proof Idea

Proof of Part I (RE → FA):

We use **structural induction** on regular expressions.

- Base cases:
  - RE = `a` → simple automaton with transition on `a`.
  - RE = ε → automaton with start = final state.
  - RE = ∅ → automaton with no accepting states.
- Induction steps (for complex REs):
  - If automata exist for RE1 and RE2, then:
    - RE1 + RE2 (union) → build automaton with ε-moves to both.
    - RE1·RE2 (concatenation) → connect automata in sequence with ε-transition.
    - RE1* (Kleene star) → add ε-loops from final to start state.

 Hence, any RE can beconverted into an NFA (later to DFA).

Proof of Part II (FA → RE):

We use the **state elimination method**:

- Start with a finite automaton (DFA/NFA).
- Gradually remove states, while replacing transitions with equivalent regular expressions.
- Continue until only two states remain (start and final).
- The resulting label on the transition is the regular expression describing the language.

 Hence, any FA can be converted into an equivalent RE.

Example

1. **RE → FA**
   RE = `(a+b)*ab`
   Construct NFA:
   - Start → loop on `a` or `b` (for `(a+b)*`).
   - Then accept strings ending with `ab`.
2. **FA → RE**
   Consider DFA for strings ending with "01":

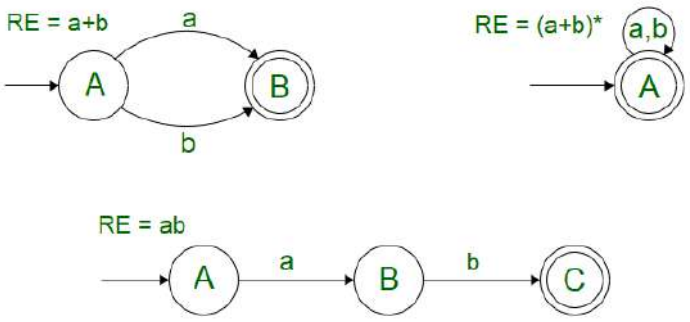
```
--> (q0) --0--> (q0)
    (q0) --1--> (q1)
    (q1) --0--> (q0)
    (q1) --1--> ((q2))
```

   Eliminate states to derive RE: `(0+1)*01`.

Importance of Kleene's Theorem

1. **Equivalence Proof** – Shows that regular expressions and finite automata are just two representations of the same class of languages.
2. **Compiler Design** – In lexical analysis, patterns of tokens are written as REs but implemented as automata.
3. **Mathematical Foundation** – Establishes the concept of regular languages and their closure properties.
4. **Practical Applications** – Search engines, text editors, and network protocols rely on this equivalence.

For certain expressions like:- (a+b), ab, (a+b)*; It's fairly easier to make the Finite Automata by just intuition as shown below. The problem arises when we are provided with a longer Regular Expression. This brings about the need for a systematic approach towards Finite Automata generation, which Kleene has put forward in Kleene's Theorem. For any Regular Expression r that represents Language L(r), there is a **Finite Automata** that accepts same language.



UNIT – IV NOTES : Non-Regular Languages and Context Free Grammars

Non-Regular Languages and Context Free Grammars Pumping Lemma for Regular Grammars – Context-Free Grammars (CFG)

1. Non-Regular Languages

Non-regular languages are those languages that cannot be recognized by any finite automaton because they require more computational power than finite memory can offer. Regular languages are limited to simple, repeating, or fixed-pattern structures, whereas non-regular languages often involve counting, matching, or balancing different symbols—tasks that require unlimited memory. A typical example is the language $\mathbf{L = \{a^n b^n \mid n \geq 0\}}$, which consists of strings where the number of *a*'s must exactly match the number of *b*'s. A finite automaton cannot verify this condition because it cannot store the count of *a*'s before reading the corresponding *b*'s. Other examples include $\mathbf{\{0^n 1^n\}}$, $\mathbf{\{ww \mid w \in \{0,1\}^*\}}$, and $\mathbf{\{a^n b^n c^n\}}$, all of which require tracking and comparing portions of the string. Because of such characteristics, these languages cannot be expressed using regular expressions or regular grammars, making them fundamentally non-regular.

Characteristics

- Requires memory to count or match symbols (FA has no memory).
- Often involves **balanced**, **nested**, or **matched** patterns.

Common examples of Non-Regular Languages

1. $\mathbf{L = \{ a^n b^n \mid n \geq 0 \}}$
   (same number of a's followed by same number of b's)
2. $\mathbf{L = \{ 0^n 1^n \mid n \geq 0 \}}$
3. $L = \{ ww \mid w \in \{0,1\} \}^*$
   (string repeated twice)
4. $\mathbf{L = \{ a^n b^n c^n \mid n \geq 0 \}}$

2. Pumping Lemma for Regular Languages

The Pumping Lemma for Regular Languages is a fundamental theoretical tool used to prove that a given language is not regular. The lemma states that for any regular language there exists a constant called the pumping length such that every sufficiently long string in the language can be divided into three parts, x, y, and z, in such a way that repeating the middle portion y any number of times will still produce strings within the language. These conditions hold for all regular languages because finite automata must eventually repeat

states when processing long inputs, causing a loop that can be "pumped." The lemma is most commonly used as a proof-by-contradiction: we assume a language is regular, apply the pumping conditions, and demonstrate that pumping leads to a string that violates the language definition. If such a contradiction occurs, the language cannot be regular. Thus, the pumping lemma serves as a mathematical criterion to distinguish regular languages from non-regular ones.

Statement

If **L** is a regular language, then there exists a constant **p** (pumping length), such that any string **s** in L with $|s| \geq p$ can be divided into **three parts**:

s = xyz, satisfying:

1. $|xy| \leq p$
2. $|y| \geq 1$
3. For all $i \geq 0$, $\mathbf{xy^iz \in L}$

Purpose

- Mainly used **to prove non-regularity by contradiction**.

Standard Steps in Pumping Lemma Proof

1. **Assume** L is regular.
2. Let **p** be pumping length.
3. Choose a string $\mathbf{s \in L}$ with $|s| \geq p$.
4. Split s = xyz satisfying lemma conditions.
5. Show that some pumped string $\mathbf{xy^iz \notin L}$ for some i (usually i = 0 or 2).
6. **Contradiction** ⇒ L is **not regular**.

Example: Prove L = { a^n b^n | n ≥ 0 } is not regular

To show how the pumping lemma works in practice, consider the language **L = {a^nb^n | n ≥ 0}**, which requires equal numbers of *a*'s and *b*'s. Assuming L is regular, the pumping lemma gives us a pumping length *p*. We choose the string **s = a^pb^p**, which is definitely in the language. According to the lemma, this string can be divided into three parts *xyz*, where *y* lies entirely within the segment of *a*'s and contains at least one *a*. When we pump the string by removing *y* (i = 0), the number of *a*'s decreases while the number of *b*'s remains unchanged, making the resulting string invalid for L. This contradiction proves that the language cannot be regular. This method is widely used to demonstrate non-regularity of languages that require counting or matching patterns.

1. Assume L is regular.
2. Let p be pumping length.

3. Choose **s = a^p b^p**.
4. y is part of first p symbols → consists only of **a's**.
5. Pump down (i = 0):
   $xy^0z = a^k b^p$ where k < p
6. Number of a's ≠ number of b's ⇒ not in L
7. Contradiction ⇒ L is **not regular**.

3. Context-Free Grammars (CFG)

A Context-Free Grammar (CFG) is a formal system used to generate and describe context-free languages, which are more powerful than regular languages. A CFG consists of a set of variables (non-terminals), terminals (symbols of the language), production rules, and a starting symbol. The production rules can replace a single non-terminal with a string of terminals and non-terminals, allowing the grammar to build complex, hierarchical patterns. This gives CFGs the ability to describe languages with nested or balanced structures, such as mathematical expressions, programming language syntax, and well-formed parentheses. For example, the language **{$a^n b^n$}** can be generated by the grammar **S → aSb | ab**, where each application of a production rule maintains balance between the number of *a*'s and *b*'s. Thus, CFGs provide a structured method for defining languages that cannot be expressed using regular grammars.**Definition**

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

- **V** – Variables (non-terminals)
- **T** – Terminals
- **S** – Start symbol
- **P** – Productions of the form A → α
  (A ∈ V, α ∈ (V ∪ T)*)

CFG Examples

1. Grammar for L = { a^n b^n | n ≥ 1 }

```
S → aSb | ab
```

2. Grammar for Balanced Parentheses

```
S → SS | (S) | ε
```

3. Grammar for Palindromes over {0,1}

```
S → 0S0 | 1S1 | 0 | 1 | ε
```

4. Grammar for { a^n b^m | n,m ≥ 1 }

```
S → aS | A
A → bA | b
```

---

4. Context-Free Languages (CFL)

Context-Free Languages (CFLs) are the class of languages generated by Context-Free Grammars. They are recognized by Pushdown Automata, which are similar to finite automata but equipped with an auxiliary stack that gives them additional memory for handling recursive and nested structures. CFLs are powerful enough to describe programming constructs, expression parsing, and hierarchical data formats. Important closure properties of CFLs include closure under union, concatenation, and Kleene star, which means that combining CFLs in these ways will still result in a CFL. However, CFLs are not closed under intersection, complement, or difference, meaning combining two CFLs using these operations may lead to a non-context-free language. These properties help classify languages and determine whether they can be processed using pushdown automata or generated using context-free grammars.

- Closed under:
  - ✔ Union
  - ✔ Concatenation
  - ✔ Kleene closure
- Not closed under:
  - ✘ Intersection
  - ✘ Complement
  - ✘ Difference

5. Pumping Lemma for Context-Free Languages (Just Introduction)

The Pumping Lemma for Context-Free Languages provides a method for proving that a language is not context-free. Similar to the regular pumping lemma, it states that long enough strings in a context-free language can be decomposed and "pumped," but the decomposition involves five parts: **u, v, x, y, z**, with certain constraints. Pumping the *v* and *y* segments simultaneously (either removing them or repeating them) must still produce strings within the language. This lemma is particularly useful for proving that languages requiring simultaneous counting of three or more symbols, such as **$\{a^n b^n c^n\}$**, are not context-free. Unlike regular languages, the stack in a pushdown automaton allows tracking of only one type of nesting or counting at a time, so languages requiring multiple parallel counts fail to satisfy the lemma. Thus, the pumping lemma is an essential tool for identifying the limitations of context-free languages.**Statement**

If L is context-free, any string s with $|s| \geq p$ can be written as:
s = uvxyz, such that

1. $|vxy| \leq p$
2. $|vy| \geq 1$
3. $\forall\, i \geq 0$, $\mathbf{uv^i xy^i z \in L}$

Example of Non-CFL

$L = \{ a^n b^n c^n \mid n \geq 0 \}$
Cannot be generated by CFG.

---

6. Difference Between Regular and Context-Free Grammars

Feature	Regular Grammar	CFG
Power	Less powerful	More powerful
Automata	Finite Automata	Pushdown Automata
Memory	No stack	One stack
Handles	Simple patterns	Nested, balanced structures

Unit V: PDA and Context-Free Languages

Deterministic And Non-Deterministic Pushdown Automata (PDA) – Parse Trees – Leftmost Derivation – Pumping Lemma for CFL – Properties Of CFL

---

1. Pushdown Automata (PDA) and Context-Free Languages (CFL)

A **Pushdown Automaton (PDA)** is an abstract computational model used to recognize **Context-Free Languages (CFLs)**. PDAs are similar to finite automata but are more powerful because they use an extra storage called a **stack**. This stack allows the machine to keep track of nested structures such as parentheses, recursive patterns, and other non-regular constructs. A language is said to be **context-free** if it can be generated by a **Context-Free Grammar (CFG)**. PDAs and CFGs are equivalent in power: for every CFG there exists a PDA that accepts the same language, and vice versa. The stack plays a crucial role in managing the variable-length memory needed to process CFLs.

- PDA is a finite automaton equipped with an additional memory called a **stack**.
- PDA is used to recognize **Context-Free Languages (CFL)**.
- Stack allows handling of recursive patterns and nested structures.
- Every CFL can be recognized by some PDA.
- PDA and CFG are **equivalent in computational power**.
- PDA operations: **push**, **pop**, and **read** input symbol.

---

2. Deterministic PDA (DPDA)

A **Non-Deterministic PDA (NPDA)** is a PDA where multiple transitions may be possible for the same state, input symbol, and stack symbol. NPDAs are more powerful than DPDAs because they can "guess" the correct path to accept a string. Many natural CFLs like **palindromes**, **anbncn**, and **balanced parentheses** are recognized by NPDAs.
A **Deterministic PDA (DPDA)** has at most one possible move for every combination of the current input symbol, stack top, and state. DPDAs cannot use ε-transitions freely if they cause ambiguity. DPDAs accept only a proper subset of CFLs called **Deterministic Context-Free Languages (DCFL)**. DCFLs are recognized by LR parsers (used in compilers). Every DPDA is also an NPDA, but the reverse is not true.

- At most **one transition** is possible for each combination of:
  (state, input symbol, stack top).
- Does NOT allow ambiguous ε-moves if they create multiple choices.
- Accepts **Deterministic Context-Free Languages (DCFL)**.
- DCFL ⊂ CFL (proper subset).
- Used in LR parsing (bottom-up parsing).
- Less powerful than NPDA.

---

3. Non-Deterministic PDA (NPDA)

A **parse tree** (or derivation tree) is a hierarchical tree representation of how a string is generated using a CFG. The root of the tree is the start symbol, internal nodes are non-terminals, and leaf nodes are terminals. Parse trees illustrate the structure of a string and show how grammar rules are applied step by step. They help detect ambiguities: if a string has two different parse trees, the grammar is **ambiguous**. Parse trees are widely used in syntax analysis in compilers and natural language processing.

- Can have **multiple transitions** for the same input and stack conditions.
- More powerful than DPDA.
- Can "guess" the correct path of computation.
- Can use ε-transitions freely.
- Able to accept all CFLs.
- Example languages accepted:
  - $\{a^n b^n\}$
  - Balanced parentheses
  - Palindromes

---

4. Parse Trees

A **leftmost derivation** is a method of generating a string from a grammar where in every step, the **leftmost non-terminal** in the current string is expanded first. This creates a predictable order of applying production rules and avoids confusion during parsing. Leftmost derivation is used by **top-down parsing methods** such as LL parsers. A grammar is called **unambiguous** if every string in the language has **only one possible leftmost derivation**; otherwise, it is ambiguous.

- A tree representation of derivation of a string in a grammar.
- Root node → Start symbol.

- Internal nodes → Non-terminals.
- Leaf nodes → Terminals.
- Shows structure and order of derivations.
- Helps detect **ambiguity** in grammar.
- Forms the basis of syntax analysis in compilers.

---

5. Leftmost Derivation

- Derivation where the **leftmost non-terminal** is expanded first.
- Used in **top-down parsing** (e.g., LL parsers).
- Helps maintain a unique order of production applications.
- If a string has more than one leftmost derivation → grammar is **ambiguous**.
- Produces a corresponding parse tree.

---

6. Pumping Lemma for CFL

The **Pumping Lemma for CFLs** is a theoretical tool used to prove that certain languages are **not** context-free. It states that for every context-free language, there exists a constant **p** such that any string **s** with length ≥ p can be split into five parts **s = uvxyz**, satisfying:

1. **v and y are the "pumping" parts**,
2. $|vxy| \leq p$ (the middle portion is short),
3. $|vy| > 0$ (something must be pumped),
4. For all $i \geq 0$, $\mathbf{uv^ixy^iz}$ must also belong to the language.

   By showing that a language violates these conditions, we can prove it is **not context-free**. Examples of non-CFLs include $\{a^nb^nc^n\}$ and $\{a^nb^nc^{2n}\}$.

- Used to **prove that languages are NOT context-free**.
- States: For any CFL, long strings (length ≥ p) can be split into **uvxyz**.
- Conditions:
  - $|vxy| \leq p$
  - $|vy| > 0$
  - For all $i \geq 0$: $\mathbf{uv^ixy^iz} \in L$
- v and y are the "pumping segments."
- Useful to show languages like $\{a^nb^nc^n\}$ are not CFL.

7. Properties of CFL

Closure Properties

✔ Closed under:

- Union
- Concatenation
- Kleene star
- Reversal
- Intersection with Regular Languages

✘ Not closed under:

- Intersection
- Complement
- Difference

Decidability Properties

- Emptiness → decidable
- Membership → decidable (CYK algorithm)
- Finiteness → decidable
- Equivalence of CFLs → **undecidable**
- Ambiguity of a grammar → **undecidable**