**Chomsky’s Hierarchy**

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Grammar** | **Language** | **Automata** |
| Type 0 | Unrestricted Grammar | Recursively Enumerable Languages | Turing Machine (TM) |
| Type 1 | Context-Sensitive Grammar (CSG) | Context-Sensitive Languages (CSL) | Linear-Bounded Automaton (LBA) |
| Type 2 | Context-Free Grammar (CFG) | Context-Free Languages (CFL) | Pushdown Automaton (PDA) |
| Type 3 | Regular Grammar | Regular Languages | Finite Automaton (FA) |

#### **Automaton (Automata)**

**Definition**:  
A mathematical model of computation that follows a predefined set of rules to accept or reject strings. It consists of **states**, **transitions**, and an **input alphabet**.

**Example**:

* **Finite Automaton (FA)**: Recognizes strings ending with 1 over {0, 1}.
  + **Input**: 101 → **Accepted** (ends with 1).
  + **Input**: 100 → **Rejected**.

#### **Alphabet (Σ)**

**Definition**:  
A finite, non-empty set of symbols used to construct strings.

**Example**:

* Σ = {a, b} (e.g., a, b, aa, ab, ...).
* Binary alphabet: Σ = {0, 1}.

#### **String**

**Definition**:  
A finite sequence of symbols from an alphabet.

**Example**:

* Over Σ = {a, b}:
  + Valid strings: a, ab, baa.
  + Empty string (ε): "" (length 0).

#### **Language**

**Definition**:  
A set of strings formed from an alphabet. It can be finite or infinite.

**Example**:

* Language of even-length strings over {a, b}:  
  L = {ε, aa, ab, ba, bb, aaaa, ...}.
* Language of valid C programs: All strings that compile without syntax errors.

#### **Grammar**

**Definition**:  
A set of production rules to generate strings in a language. Defined as G = (V, Σ, P, S), where:

* V: Set of Non-terminal symbols.
* Σ: Set of Terminal symbols.
* P: Set of rules.
* S: Start symbol.

#### **Need for Normalization of Grammar**

Normalization in grammars refers to converting a given grammar into a standardized form (like Chomsky Normal Form or Greibach Normal Form) to simplify analysis, parsing, and theoretical proofs

* **Remove Null Productions**: Eliminating ε-productions simplifies grammars by removing empty string derivations, ensuring all rules generate visible symbols.
* **Remove Useless/Unreachable Productions**: Deleting non-terminals that don’t contribute to valid strings optimizes grammars, reducing unnecessary complexity.
* **Eliminate Ambiguity & Redundant Rules**: Normalization removes multiple interpretations and duplicate rules, ensuring consistent language derivation.
* **Simplifies Parsing and Analysis**: Structured grammar forms make parsing algorithms faster and more predictable by enforcing rule constraints.
* **Enables Efficient Algorithms**: Normalized grammars allow optimized parsing techniques like CYK, reducing computational overhead.
* **Helps in Formal Proofs**: Standardized forms support theoretical analysis, such as proving language properties via the Pumping Lemma.
* **Easier Compiler Implementation**: Clean grammar rules integrate smoothly with lexers and parsers, improving compiler design efficiency.

#### **Right Linear Regular Grammar**

In this type of regular grammar, all the non-terminals on the right-hand side exist at the rightmost place, or at the right ends.

A ⇢ a, A ⇢ a**B**, A ⇢ ∈  
where,  
A and B are non-terminals,  
a is terminal, and  
∈ is empty string

S ⇢ 00**B** | 11**S**  
B ⇢ 0**B** | 1**B** | 0 | 1  
where,  
S and B are non-terminals, and  
0 and 1 are terminals

#### **Left Linear Regular Grammar**

In this type of regular grammar, all the non-terminals on the left-hand side exist at the leftmost place, or at the left ends.

A ⇢ a, A ⇢ Ba, A ⇢ ∈  
where,  
A and B are non-terminals,  
a is terminal, and  
∈ is empty string

S ⇢ B00 | S11  
B ⇢ B0 | B1 | 0 | 1  
where  
S and B are non-terminals, and  
0 and 1 are terminals

**DFA (Deterministic Finite Automaton)**

**Definition**: A finite automaton with **deterministic transitions**; accepts regular languages.

**Tuple** T=(Q,Σ,δ,q0,F):

|  |  |
| --- | --- |
| **Tuple Variable** | **Represents** |
| Q | Finite set of **states**. |
| Σ | Finite **input alphabet**. |
| δ | **Transition function**: Q×Σ→Q. |
| q0​ | **Initial state** (q0∈Q0​). |
| F | Set of **final/accepting states** (F⊆Q. |

**NFA (Non-deterministic Finite Automaton)**

**Definition**: A finite automaton with **non-deterministic transitions** (including ε-transitions); also accepts regular languages.

**Tuple** T=(Q,Σ,δ,q0,F):

|  |  |
| --- | --- |
| **Tuple Variable** | **Represents** |
| Q | Finite set of **states**. |
| Σ | Finite **input alphabet** (includes ε for empty moves). |
| δ | **Transition function**: Q×(Σ∪{ϵ})→P(Q) |
| q0​ | **Initial state** (q0∈Q0). |
| F | Set of **final/accepting states** (F⊆Q). |

**Limitations FA**

### ****Limited Memory****

### ****Cannot Handle Context-Free or Higher Languages****

### ****No Stack or External Memory****

**Applications of FA**

* **Lexical Analysis in Compilers**: FA helps identify keywords, operators, and tokens in source code.
* **Pattern Recognition with Regular Expressions**: FA models regular expressions used for searching and matching patterns in text files.
* **Digital Circuit Design**: FA is used in the design of sequential circuits, such as Mealy and Moore machines.
* **Text Editors**: Used to find and replace patterns in large text files.
* **Spell Checkers**: FA can be used to recognize valid word forms in spelling applications.
* **Decision Making and Learning**: Can be modelled to help automate decision-making processes.

**Differentiate between DFA and NFA**

|  |  |
| --- | --- |
| **DFA** | **NFA** |
| For each symbolic representation of the alphabet, there is only one state transition in DFA. | No need to specify how does the NFA react according to some symbol. |
| DFA cannot use Empty String transition. | NFA can use Empty String transition. |
| DFA can be understood as one machine. | NFA can be understood as multiple little machines computing at the same time. |
| In DFA, the next possible state is distinctly set. | In NFA, each pair of state and input symbol can have many possible next states. |
| DFA is more difficult to construct. | NFA is easier to construct. |
| Time needed for executing an input string is less. | Time needed for executing an input string is more. |
| All DFA are NFA. | Not all NFA are DFA. |
| DFA requires more space. | NFA requires less space than DFA. |
| Conversion of Regular expression to DFA is difficult . | Conversion of Regular expression to NFA is simpler compared to DFA. |
| Epsilon move is not allowed in DFA | Epsilon move is allowed in NFA |
| In a DFA, there is only **one** possible transition for each input symbol from any given state. | In an NFA, there can be **multiple** transitions for a single input symbol from a given state. |

**Moore Machine**

**Definition**: A finite automaton where **outputs depend on the current state**.

**Tuple** T=(Q,Σ,Δ,δ,λ,q0)T=(Q,Σ,Δ,δ,λ,q0​):

|  |  |
| --- | --- |
| **Tuple Variable** | **Represents** |
| Q | Finite set of **states**. |
| Σ | Finite **input alphabet**. |
| Δ | Finite **output alphabet**. |
| δ | **Transition function**: Q×Σ→Q. |
| λ | **Output function**: Q→Δ(output tied to state). |
| q0​ | **Initial state** (q0∈Q). |

**Mealy Machine**

**Definition**: A finite automaton where **outputs depend on transitions** (current state + input).

**Tuple** T=(Q,Σ,Δ,δ,λ,q0):

|  |  |
| --- | --- |
| **Tuple Variable** | **Represents** |
| Q | Finite set of **states**. |
| Σ | Finite **input alphabet**. |
| Δ | Finite **output alphabet**. |
| δ | **Transition function**: Q×Σ→Q. |
| λ | **Output function**: Q×Σ→Δ. |
| q0​ | **Initial state** (q0∈Q). |

**Differentiate between Moore and Mealy Machines**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Moore Machine** | **Mealy Machine** |
| Output Dependency | Output depends only on the current state. | Output depends on the current state and input. |
| Output Placement | Output is associated inside the state (written inside the state node). | Output is associated with the transition (written on the edge). |
| Response Time | They react slower to inputs(One clock cycle later). | They react faster to inputs. |
| Number of states | Generally, requires more states for the same function. | Typically requires fewer states than Moore. |
| Synchronization | Better for synchronous systems due to delayed output. | Better for asynchronous systems due to immediate output. |
| Design | Easy to design. | It is difficult to design. |
| Hardware Requirement | More | Less |
| Example Use Case | Traffic light controller (output depends only on state). | Vending machine (output depends on state + input). |

**PDA (Push Down Automaton)**

**Definition**: A finite automaton with a **stack** for memory; recognizes context-free languages.

**Tuple** T=(Q,Σ,Γ,δ,q0,Z0,F):

|  |  |
| --- | --- |
| **Tuple Variable** | **Represents** |
| Q | Finite set of **states**. |
| Σ | Finite **input alphabet**. |
| Γ | Finite **stack alphabet** (symbols stored in the stack). |
| δ | **Transition function**: Q×(Σ∪{ϵ})×Γ→Q×Γ∗ |
| q0​ | **Initial state** (q0∈Q). |
| Z0 | **Initial stack symbol** (Z0∈Γ). |
| F | Set of **final states** (F⊆Q). |

**Features**

* **Recognizes Context-Free Languages (CFLs)**
* **More powerful than Finite Automata (FA)**
* Non-deterministic PDAs (NPDAs) can recognize languages that deterministic PDAs (DPDAs) cannot

**Limitations**

* Cannot recognize non-context-free languages
* Only one stack, multiple nested dependencies can't be tracked (unlike a Turing Machine)
* Limited memory access, only the top of the stack is accessible at any time (LIFO structure)
* Deterministic PDAs (DPDAs) Are Weaker

**Applications of PDA**

* **Syntax Analysis in Compilers**: Helps parse programming language structures by using a stack to manage nested elements (e.g., parentheses).
* **Stack-Based Applications**: Used in scenarios where operations depend on the last inserted element, like evaluating arithmetic expressions.
* **Tower of Hanoi Problem**: Solves problems involving recursive and stack-based solutions.
* **Network Protocols**: PDA can validate message formats and enforce structured communication.
* **Natural Language Processing**: Used in tasks such as parsing sentences and generating syntax trees.
* **Cryptography**: Helps in designing algorithms for encryption and decryption.
* **Automatic Theorem Proving**: PDA is applied to verify the correctness of software models and systems.

**Turing Machine (TM)**

**Definition**: A computational model with an infinite tape and a read-write head; recognizes recursively enumerable languages.

Tuple T=(Q,Σ,Γ,δ,q0,B,F):

|  |  |
| --- | --- |
| **Tuple Variable** | **Represents** |
| Q | Finite set of **states**. |
| Σ | Finite **input alphabet** (symbols from the input string). |
| Γ | Finite **tape alphabet** (Σ⊆Γ, includes blank symbol B). |
| δ | **Transition function**: Q×Γ→Q×Γ×{L,R}. |
| q0 | **Initial state** (q0∈Q). |
| B | **Blank symbol** (B∈Γ, represents empty tape cells). |
| F | Set of **final (accepting) states** (F⊆Q). |

**Key Features**

* **Infinite Tape**: Divided into cells, each holding a symbol (from a finite alphabet).
* **Read-Write Head**: Moves left/right to read, write, or erase symbols.
* **Finite Control**: Governs transitions based on current state and tape symbol.

**Working of a Turing Machine**

**1. Input**: A string written on the tape (rest is blank).

**2. Execution**:

* Reads the symbol under the head.
* Based on current state and symbol:
  + - Writes a new symbol.
    - Moves head **Left (L)** or **Right (R)**.
    - Changes state.

**3. Termination**:

1. **Accepts**: Halts in a final state.
2. **Rejects**: Halts in a non-final state or loops infinitely.

### ****Variants of Turing Machines****

1. **Standard TM**:
   * Single infinite tape, deterministic transitions.
   * Recognizes **recursively enumerable languages**.
2. **Multi-Tape TM**:
   * Multiple tapes (each with independent heads).
   * **Equivalent to standard TM** but more efficient for some problems.
3. **Non-Deterministic TM (NTM)**:
   * Multiple possible transitions per state/symbol.
   * **Same power as deterministic TM** but may solve problems faster (e.g., NP problems).
4. **Universal TM (UTM)**:
   * Simulates other TMs (like a general-purpose computer).
   * Basis for modern computers.
5. **Alternating TM**:
   * Extends non-determinism with "AND" and "OR" states.
   * Models’ parallel computation.
6. **Quantum TM**:
   * Uses quantum bits (qubits) for superposition-based computation.
   * Theoretically solves certain problems (e.g., factorization) exponentially faster.

**Applications of TM**

* Solving Recursively Enumerable Problems: TM can solve any problem that is recursively enumerable.
* Artificial Intelligence: Forms the foundation of AI algorithms, including decision-making and machine learning.
* Robotics: Used to model robot actions and control systems.
* Neural Networks: TM can model complex neural networks.
* Complexity Theory: Used to analyse the computational complexity of algorithms.
* Computational Biology: Applied to model biological processes and systems.
* Quantum Computing: Provides insights into the relationship between classical and quantum computing models.
* Digital Circuit Design: Used to model and verify the behaviour of complex digital circuits.

**Halting Problem**

* The Halting Problem asks whether a given program or algorithm will eventually halt (terminate) or continue running indefinitely for a particular input.
* ‘Halting’ means that the program will either accept or reject the input and then terminate, rather than going into an infinite loop.
* Alan Turing in 1936 proved that the halting problem is undecidable, meaning it is impossible to design a generalized algorithm that can accurately determine whether any arbitrary program will halt.
* The only way to know if a specific program will halt is to run it and observe the outcome. This makes the Halting Problem an **undecidable problem**.

**Assumption**: Suppose a function H(P,I) exists that returns ‘halts’ if P halts on I, and ‘does not halt’ otherwise.

**Construct a Program D** that takes a program P as input:

* D runs H(P,P).
* If H says P halts on input P, D enters an infinite loop.
* If H says P does not halt on P, D halts immediately.

**Contradiction**:

* Run D(D).
  + If H(D,D) returns ‘halts’, then D(D) loops forever.
  + If H(D,D) returns ‘does not halt’, then D(D) halts.
* Both outcomes contradict H’s correctness, proving H cannot exist.