**Introduction of Theory of Computation**

**Automata** theory (also known as **Theory Of Computation**) is a theoretical branch of Computer Science and Mathematics, which mainly deals with the logic of computation with respect to simple machines, referred to as automata.

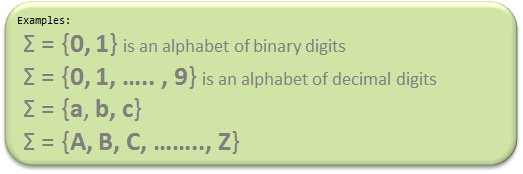
**Automata**\* enables scientists to understand how machines compute the functions and solve problems.

Now, let’s understand the basic terminologies, which are important and frequently used in the Theory of Computation.

**Symbol:** Symbol (often also called **character**) is the smallest building block, which can be any alphabet, letter, or picture.

Eg: a,b,c,0,1

**Alphabets**(**Σ**)**:** Alphabets are a set of symbols, which are always ***finite***.



**String:** String is a ***finite***sequence of symbols from some alphabet. A string is generally denoted as ***w*** and the length of a string is denoted as ***|w|***.

**Note:** Σ\* is a set of all possible strings (often power

set (need not be unique here or we can say multiset) of string) So this implies that ***language****is a subset of*Σ\*.

**Empty string is the string with zero occurrences of symbols, represented as ε.**

**Number of Strings (of length 2)**

**that can be generated over the alphabet {a, b}** is **aa,ab,ba,bb**

Length of String |w| = 2

Number of Strings = 4

**Conclusion:**

For alphabet {a, b} with length **n**, number of

Strings can be generated = **2n**.

**Language:** A language is a *set of strings*, chosen from some Σ\* or we can say- **‘**A language is a subset of Σ\* **‘**. A language that can be formed over ‘ Σ ‘ can be **Finite** or **Infinite**.

Example of Finite Language:

L1 = { set of string of 2 }

L1 = { xy, yx, xx, yy }

Example of Infinite Language:

L1 = { set of all strings starts with 'b' }

L1 = { babb, baa, ba, bbb, baab, ....... }

# Chomsky Hierarchy in Theory of Computation

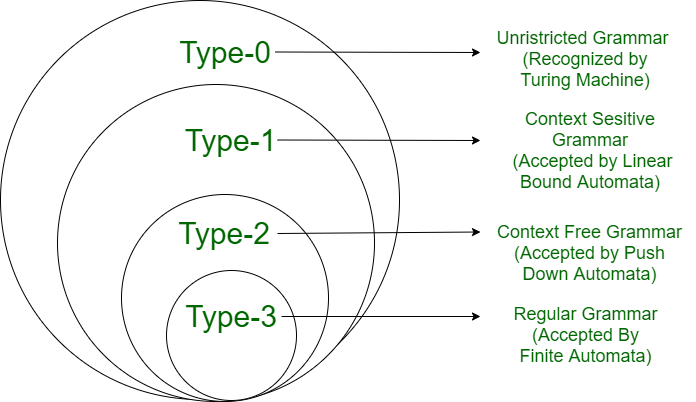
According to [Chomsky hierarchy](https://www.geeksforgeeks.org/toc-chomsky-hierarchy/), grammars are divided into 4 types:

Type 0 known as unrestricted grammar.

Type 1 known as context sensitive grammar.

Type 2 known as context free grammar.

Type 3 Regular Grammar.



|  |  |  |  |
| --- | --- | --- | --- |
| **Grammar Type** | **Grammar Accepted** | **Language Accepted** | **Automaton** |
| Type 0 | Unrestricted grammar | Recursively enumerable language | Turing Machine |
| Type 1 | Context-sensitive grammar | Context-sensitive language | Linear-bounded automaton |
| Type 2 | Context-free grammar | Context-free language | Pushdown automaton |
| Type 3 | Regular grammar | Regular language | Finite state automaton |

## Type - 0 Grammar

**Type-0 grammars** generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of **α → β** where **α** is a string of terminals and nonterminals with at least one non-terminal and **α** cannot be null. **β** is a string of terminals and non-terminals.

### Example

S → ACaB

Bc → acB

CB → DB

aD → Db

**Type 1: Context Sensitive Grammar)**

Type-1 grammars generate the context-sensitive languages. The language generated by the grammar is recognized by the [Linear Bound Automata](https://en.wikipedia.org/wiki/Linear_bounded_automaton)   
In Type 1   
I. First of all Type 1 grammar should be Type 0.   
II. Grammar Production in the form of

|α| <= |β|

i.e count of symbol in α is less than or equal to β  
    
For Example,   
S –> AB   
AB –> abc   
B –> b

## Type - 2 Grammar

**Type-2 grammars** generate context-free languages.

The productions must be in the form **A → γ**

where **A ∈ N** (Non terminal)

and **γ ∈ (T ∪ N)\*** (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

### Example

S → X a

X → a

X → aX

X → abc

X → ε

**Type 3: Regular Grammar:**

Type-3 grammars generate regular languages. These languages are exactly all languages that can be accepted by a finite state automaton.

Type 3 is most restricted form of grammar.   
Type 3 should be in the given form only:

**V –> VT / T**         (left-regular grammar)

**(or)**

**V –> TV /T**       (right-regular grammar)

example:

S –> a

# Finite Automata

* Finite automata are used to recognize patterns.
* It takes the string of symbol as input and changes its state accordingly. When the desired symbol is found, then the transition occurs.
* At the time of transition, the automata can either move to the next state or stay in the same state.
* Finite automata have two states, **Accept state** or **Reject state**. When the input string is processed successfully, and the automata reached its final state, then it will accept.

## Formal Definition of FA

A finite automaton is a collection of 5-tuple (Q, ∑, δ, q0, F), where:

Q: finite set of states

∑: finite set of the input symbol

q0: initial state

F: **final** state

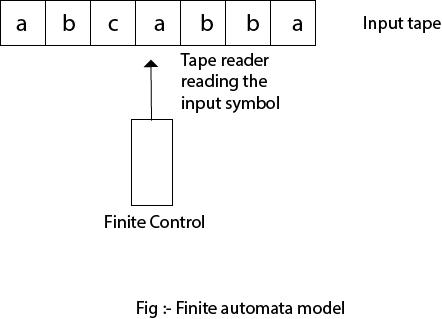
δ: Transition function

## Finite Automata Model:

Finite automata can be represented by input tape and finite control.

**Input tape:** It is a linear tape having some number of cells. Each input symbol is placed in each cell.

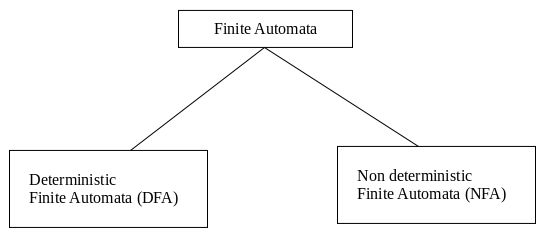
**Finite control:** The finite control decides the next state on receiving particular input from input tape. The tape reader reads the cells one by one from left to right, and at a time only one input symbol is read.



## Types of Automata:

There are two types of finite automata:

1. DFA(deterministic finite automata)
2. NFA(non-deterministic finite automata)



**1. DFA**

DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. In the DFA, the machine goes to one state only for a particular input character. DFA does not accept the null move.

**2. NFA**

NFA stands for non-deterministic finite automata. It is used to transmit any number of states for a particular input. It can accept the null move.

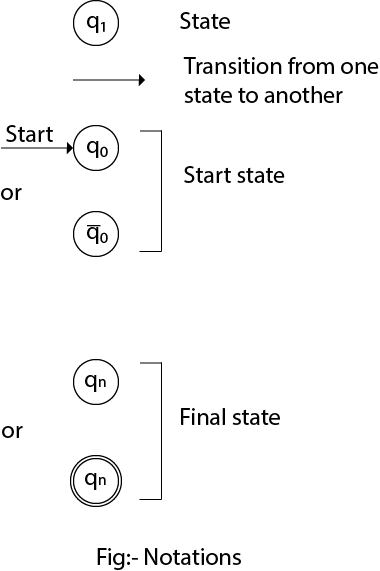
**Some important points about DFA and NFA:**

1. Every DFA is NFA, but NFA is not DFA.
2. There can be multiple final states in both NFA and DFA.
3. DFA is used in Lexical Analysis in Compiler.
4. NFA is more of a theoretical concept.

**Transition Diagram**

A transition diagram or state transition diagram is a directed graph which can be constructed as follows:

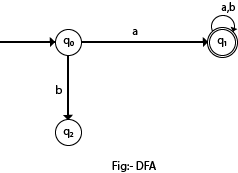
* There is a node for each state in Q, which is represented by the circle.
* There is a directed edge from node q to node p labeled a if δ(q, a) = p.
* In the start state, there is an arrow with no source.
* Accepting states or final states are indicating by a double circle.



**DFA (Deterministic finite automata)**

* DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. The finite automata are called deterministic finite automata if the machine is read an input string one symbol at a time.
* In DFA, there is only one path for specific input from the current state to the next state.
* DFA does not accept the null move, i.e., the DFA cannot change state without any input character.
* DFA can contain multiple final states. It is used in Lexical Analysis in Compiler.

In the following diagram, we can see that from state q0 for input a, there is only one path which is going to q1. Similarly, from q0, there is only one path for input b going to q2.



## Formal Definition of DFA

A DFA is a collection of 5-tuples same as we described in the definition of FA.

Q: finite set of states

∑: finite set of the input symbol

q0: initial state

F: **final** state

δ: Transition function

Transition function can be defined as:

δ: Q x ∑→Q

### Example 1:

Q = {q0, q1, q2}

∑ = {0, 1}

q0 = {q0}

F = {q2}

**Solution:**

Transition Diagram:

Deterministic finite automata

**Transition Table:**

|  |  |  |
| --- | --- | --- |
| **Present State** | **Next state for Input 0** | **Next State of Input 1** |
| →q0 | q0 | q1 |
| q1 | q2 | q1 |
| \*q2 | q2 | q2 |

**NFA (Non-Deterministic finite automata)**

* NFA stands for non-deterministic finite automata. It is easy to construct an NFA than DFA for a given regular language.
* The finite automata are called NFA when there exist many paths for specific input from the current state to the next state.
* Every NFA is not DFA, but each NFA can be translated into DFA.
* NFA is defined in the same way as DFA but with the following two exceptions, it contains multiple next states, and it contains ε transition.

## Formal definition of NFA:

NFA also has five states same as DFA, but with different transition function, as shown follows:

δ: Q x ∑ →2Q

where,

Q: finite set of states

∑: finite set of the input symbol

q0: initial state

F: **final** state

δ: Transition function

### Example 1:

Q = {q0, q1, q2}

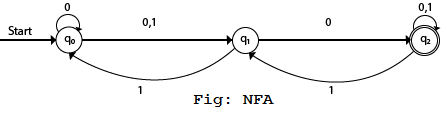
∑ = {0, 1}

q0 = {q0}

F = {q2}

**Solution:**

Transition diagram:



Transition Table:

|  |  |  |
| --- | --- | --- |
| **Present State** | **Next state for Input 0** | **Next State of Input 1** |
| →q0 | q0, q1 | q1 |
| q1 | q2 | q0 |
| \*q2 | q2 | q1, q2 |