**FLAT NOTES**

**UNIT : - 1**

* **Symbols:-**

A,b,c,d,e,f,aa,ab,ac,aac

* **Alphabets:**

An alphabet is a finite non empty set of symbols, which used to represent the input of a machine. Alphabets are typically thought of as represented by letters, characters, digits, signs, punctuation, etc. Conventionally we use the symbol ∑ for an alphabet. Common alphabets include:

* ∑ = {0, 1}: The binary alphabets.
* ∑ = {a, b, c, ……, z}: The set of all lower-case letters.
* ∑ = {The sets of all ASCII characters or the set of all printable ASCII characters}.

An infinite sequence of letters may be constructed from elements of an alphabet as well.

## Strings:

A string is a finite ordered sequence of symbols chosen form some set of alphabet or ∑. For example, ‘aababbbbaa’ is a valid string from the alphabet ∑ = {a, b}, similarly ‘001111000101’ is a valid string form the alphabet ∑ = {0, 1}.

### **Empty string:**

Every alphabet ∑ has a special string called empty string which means the string with zero occurrences of symbols. This string represented by λ, e or ε. It is the string that may be chosen from any alphabet whatsoever.

### **Length of a string:**

The finite occurrence of input symbols form ∑ present the length of a string. If s denotes the string over alphabet ∑ then length of a string is represented by |S|. For instance, ‘001110’ is a string from the alphabets ∑= {0, 1} has length 6. Similarly if ∑ = {a, b} and S = ‘aabbabbba’ then |S| = 9.

**Language:-** Language is a collection of string.

Like: aaa , aab , abb , aba , etc.

# **Grammar:-**

**Grammar :**  
It is a finite set of formal rules for generating syntactically correct sentences or meaningful correct sentences.

**Constitute Of Grammar :**  
Grammar is basically composed of two basic elements –

1. **Terminal Symbols –**  
   Terminal symbols are those which are the components of the sentences generated using a grammar and are represented using small case letter like a, b, c etc.
2. **Non-Terminal Symbols –**  
   Non-Terminal Symbols are those symbols which take part in the generation of the sentence but are not the component of the sentence. Non-Terminal Symbols are also called Auxiliary Symbols and Variables. These symbols are represented using a capital letter like A, B, C, etc.

**Formal Definition of Grammar :**  
Any Grammar can be represented by 4 tuples – <N, T, P, S>

* **N –**Finite Non-Empty Set of Non-Terminal Symbols.
* **T –**Finite Set of Terminal Symbols.
* **P –**Finite Non-Empty Set of Production Rules.
* **S –**Start Symbol (Symbol from where we start producing our sentences or strings).

# **Explain the concept of derivation:-**

Derivation is a sequence of production rules. It is used to get input strings. During parsing, we have to take two decisions, which are as follows

* We have to decide the non-terminal which is to be replaced.
* We have to decide the production rule by which the non-terminal will be replaced.

Two options to decide which non-terminal has to be replaced with the production rule are as follows −

* Left most derivation
* Right most derivation.

Let us understand these two options in detail.

Left Most Derivation

In the leftmost derivation, the input is scanned and then replaced with the production rule from left side to right. So, we have to read that input string from left to right.

Example

Production rules:

E=E+E rule1

E=E-E rule2

E=a|b rule3

Let the input be **a-b+a**

Now, when we perform the Left Most Derivation, the result will be as follows −

E=E+E

E=E-E+E from rule2

E=a-E+E from rule3

E=a-b+E from rule3

E=a-b+a from rule3

Finally, the given string is parsed

## Right Most Derivation

In Right most derivation, the input is scanned and replaced with the production rule right to left. So, we have to read the input string from right to left.

## Example

Production rule:

E=E+E rule1

E=E-E rule2

E=a|b rule3

Let the input be a-b+a

Now, when we perform the Right Most Derivation, we get the following result −

E=E-E

E=E-E+E from rule1

E=E-E+a from rule3

E=E-b+a from rule3

E=a-b+a from rule3

# **Explain the concept of Production:-**

**Production Rules :**  
A production or production rule in computer science is a rewrite rule specifying a symbol substitution that can be recursively performed to generate new symbol sequences. It is of the form α->  β where  α is a Non-Terminal Symbol which can be replaced by β which is a string of Terminal Symbols or Non-Terminal Symbols.

**Example-1 :**  
Consider Grammar G1 = <N, T, P, S>

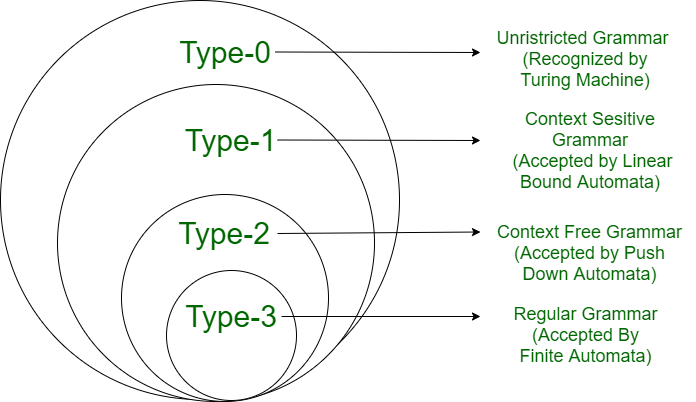
T= {a , b} #set of terminal symbols

P={A->Aa , A->Ab, A->a , A->b ,A->E(epsylan)}

# **Chomsky Hierarchy:-**

According to [Chomsky hierarchy](https://www.geeksforgeeks.org/toc-chomsky-hierarchy/), grammar is divided into 4 types as follows:

1. Type 0 is known as unrestricted grammar.
2. Type 1 is known as context-sensitive grammar.
3. Type 2 is known as a context-free grammar.
4. Type 3 Regular Grammar.



**Type 0: Unrestricted Grammar:-**

Type-0 grammars include all formal grammar. Type 0 grammar languages are recognized by turing machine. These languages are also known as the Recursively Enumerable languages.

In type 0 there must be at least one variable on the Left side of production.

Sab ---> ba

 A --> S

Here, Variables are S, A, and Terminals a, b.

**Type 1: Context-Sensitive Grammar:-**

Type-1 grammars generate 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

* First of all Type 1 grammar should be Type 0.
* Grammar Production in the form of

|\alpha |<=|\beta |

**Type 2: Context-Free Grammar:** Type-2 grammars generate context-free languages. The language generated by the grammar is recognized by a [Pushdown automata](https://www.geeksforgeeks.org/theory-of-computation-pushdown-automata/).  In Type 2:

* First of all, it should be Type 1.
* The left-hand side of production can have only one variable and there is no restriction on

|\alpha         | = 1.

For example:

S ---AB

A---a

B---b

**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 the 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)

For example:

S --->a

The above form is called strictly regular grammar.

There is another form of regular grammar called extended regular grammar.

### **Left Linear Grammar**

The production must be in the following form in the left linear grammar.

1. X → xY
2. X → x

Where

X, Y belongs to a variable (V), and 'x' belongs to T\*.

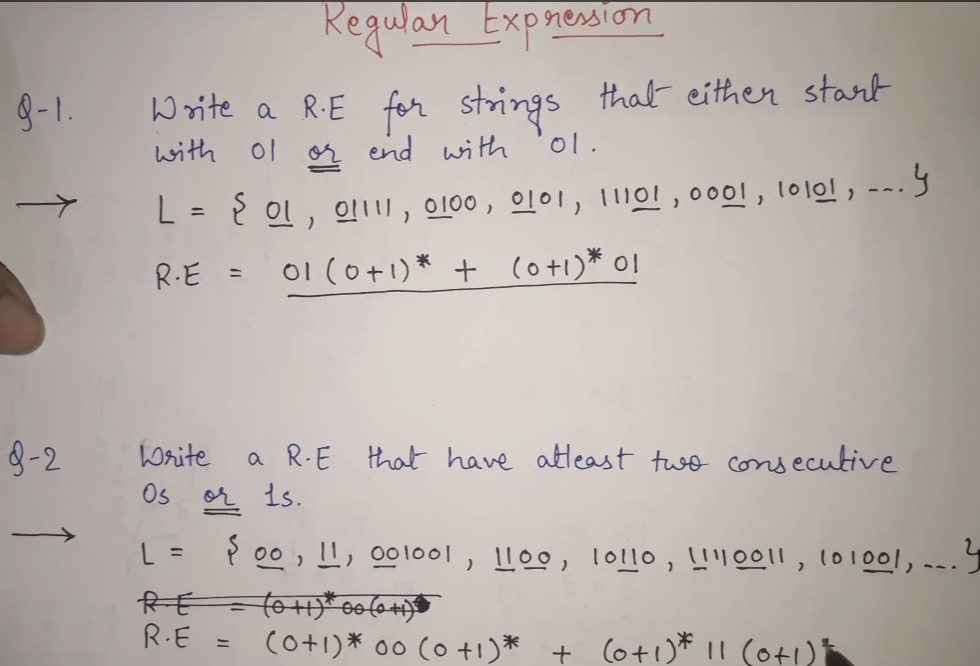
### **Right Linear Grammar**

The production must be in the following form in the right linear grammar.

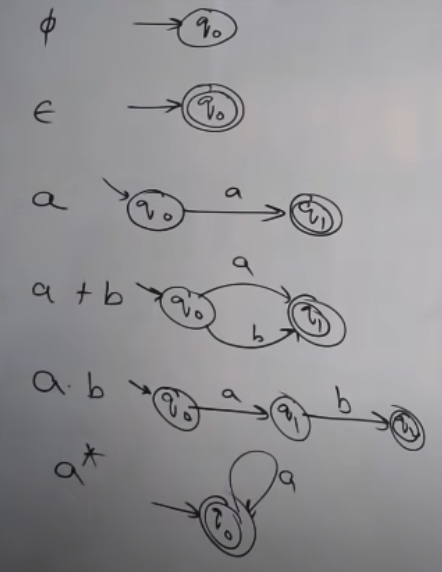
1. X → Yx
2. X →

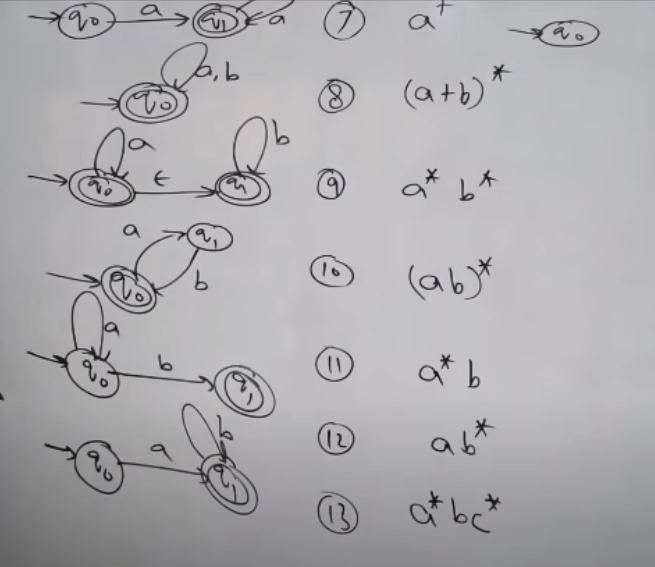
# **Regular Expression:-**

* Regular expression is use to describe the language . Regular expression is describe that language who is accepted by Finite automata.
* The languages accepted by some regular expression are referred to as Regular languages.



# **Conversion Regular Expression to Finite Automata:-**





* **Regular Languages:-**

A **regular language** is a language that can be expressed with a [regular expression](https://brilliant.org/wiki/regular-expressions/) or a deterministic or non-deterministic [finite automata](https://brilliant.org/wiki/finite-state-machines/) or state machine. A **language** is a set of [strings](https://brilliant.org/wiki/strings/) which are made up of characters from a specified alphabet, or set of symbols. Regular languages are a [subset](https://brilliant.org/wiki/sets-subsets/) of the set of all strings.

* **Automata:-**

Automaton is nothing but a machine which accepts the strings of a language L over an input alphabet Σ.

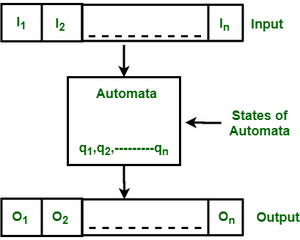
There are four different types of Automata that are mostly used in the theory of computation (TOC). These are as follows −

* Finite-state machine (FSM).
* Pushdown automata (PDA).
* Linear-bounded automata (LBA).
* Turing machine (TM).

When comparing these four types of automata, Finite-state machines are less powerful whereas Turing machines are more powerful.

# **Finite Automata:-**

Finite Automata(FA) is the simplest machine to recognize patterns. The finite automata or finite state machine is an abstract machine that has five elements or tuples. It has a set of states and rules for moving from one state to another but it depends upon the applied input symbol. Basically, it is an abstract model of a digital computer. The following figure shows some essential features of general automation.



* **Formal Definition of FA:-**

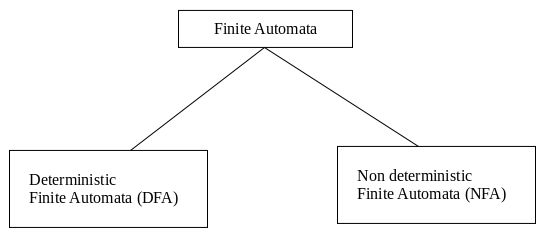
A finite automaton is a collection of 5-tuple (Q, ∑, δ, q0, F), where:

1. Q: finite set of states
2. ∑: finite set of the input symbol
3. q0: initial state
4. F: **final** state
5. δ: Transition function

* **Types of Finite Automata:**

There are two types of finite automata:

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



# **Deterministic Finite Automata(DFA):-**

DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. The finite automata are deterministic FA, if the machine reads an input string one symbol at a time.

In DFA, there is only one path input from the current state to the next state. It does not accept the null move, i.e. it cannot change state without any input. It can contain multiple final states. It is used in Lexical Analysis in compilers. Every DFA is an NFA.

Formal definition of different automata (DFA)

A Deterministic Finite automata (DFA) is a collection of defined as a 5-tuples and is as follows −

**M=(Q, Σ, δ,q0,F)**

Where,

* Q: Finite set called states.
* Σ: Finite set called alphabets.
* δ: Q × Σ → Q is the transition function.
* q0 ∈ Q is the start or initial state.
* F: Final or accept state.

Graphical Representation of DFA

A DFA can be represented by digraphs called state diagrams.

The following factors are considered in DFA −

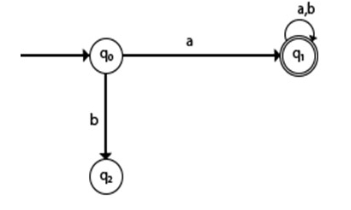
* The state is represented by vertices.
* The arc labelled with an input character shows the transitions.
* The initial state is represented with an arrow.
* The final state is represented by a double circle.

**Practice question:-**

<https://www.gatevidyalay.com/how-to-solve-dfa-problems-dfa-solved-examples/>

## Trap state in DFA

If a transition goes to a state from which it can never escape. Such a state is called a trap state. It is called the dead state.



In the above example, q2 is a trap or dead state because it can’t reach the final state

* **Application of DFA (Deterministic Finite Automata)**

The different applications of deterministic finite automata are as follows −

* Protocol analysis text parsing.
* Video game character behavior.
* Security analysis.
* CPU control units.
* Natural language processing Speech recognition, etc.

# **Non-Deterministic Finite Automata:-**

NFA stands for non-deterministic finite automata. It is easy to construct an NFA when compared to 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.

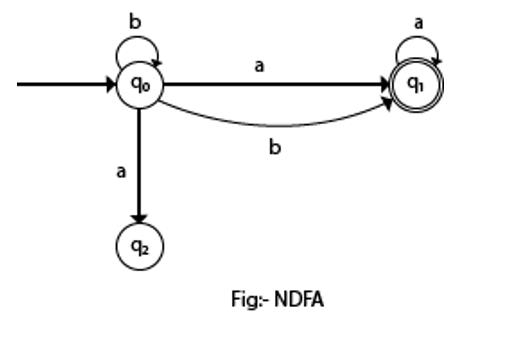
Each NFA can be translated into DFA but every NFA is Not DFA.

NFA is defined in the same way as DFA but with the following two exceptions, which are as follows –

* It contains multiple next states.
* It contains ε transitions.

Example:-

The transition diagram is as follows −



In the above NFA, it is clear that there exist many paths for specific input from the current state to the next state. (state q0 on input ‘b’ it goes to q0 itself and q1).

Non-deterministic finite automata also have five states which are same as DFA, but with different transition function, as shown follows −

δ: Q X Σ -> 2Q

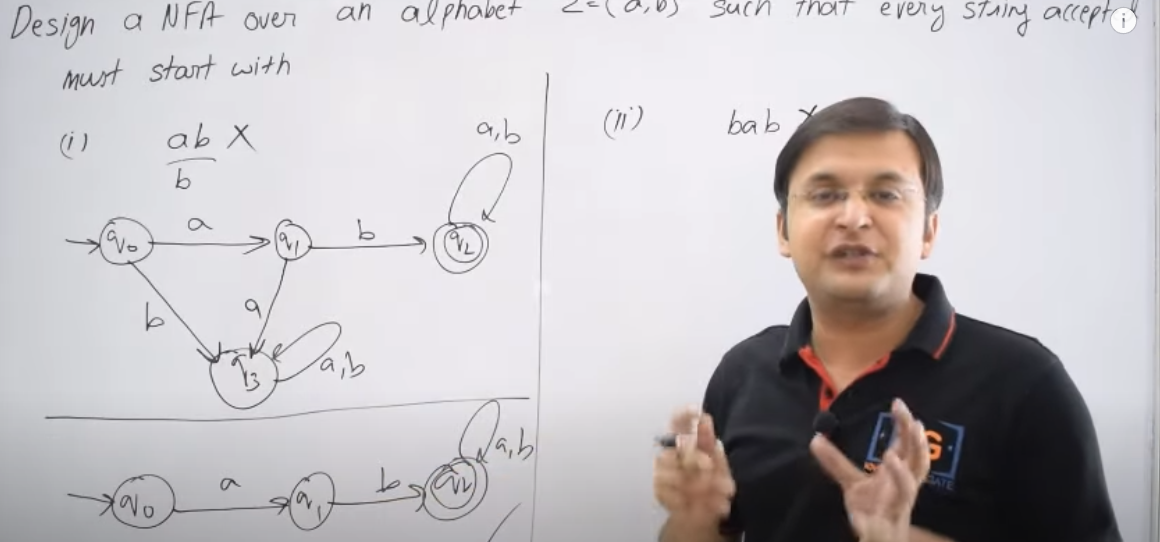
Non-deterministic finite automata is defined as a 5 tuple,

M=(Q, Σ, δ,q0,F)

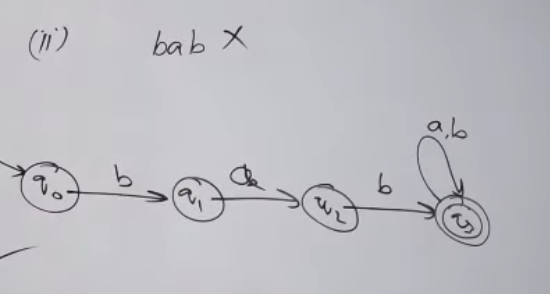
Where,

* Q: Finite set of states
* Σ: Finite set of the input symbol
* q0: Initial state
* F: Final state
* δ: Transition function: Q X Σ -> 2Q

**Practice of NFA Question link :-**  **https://www.javatpoint.com/examples-of-non-deterministic-finite-automata**

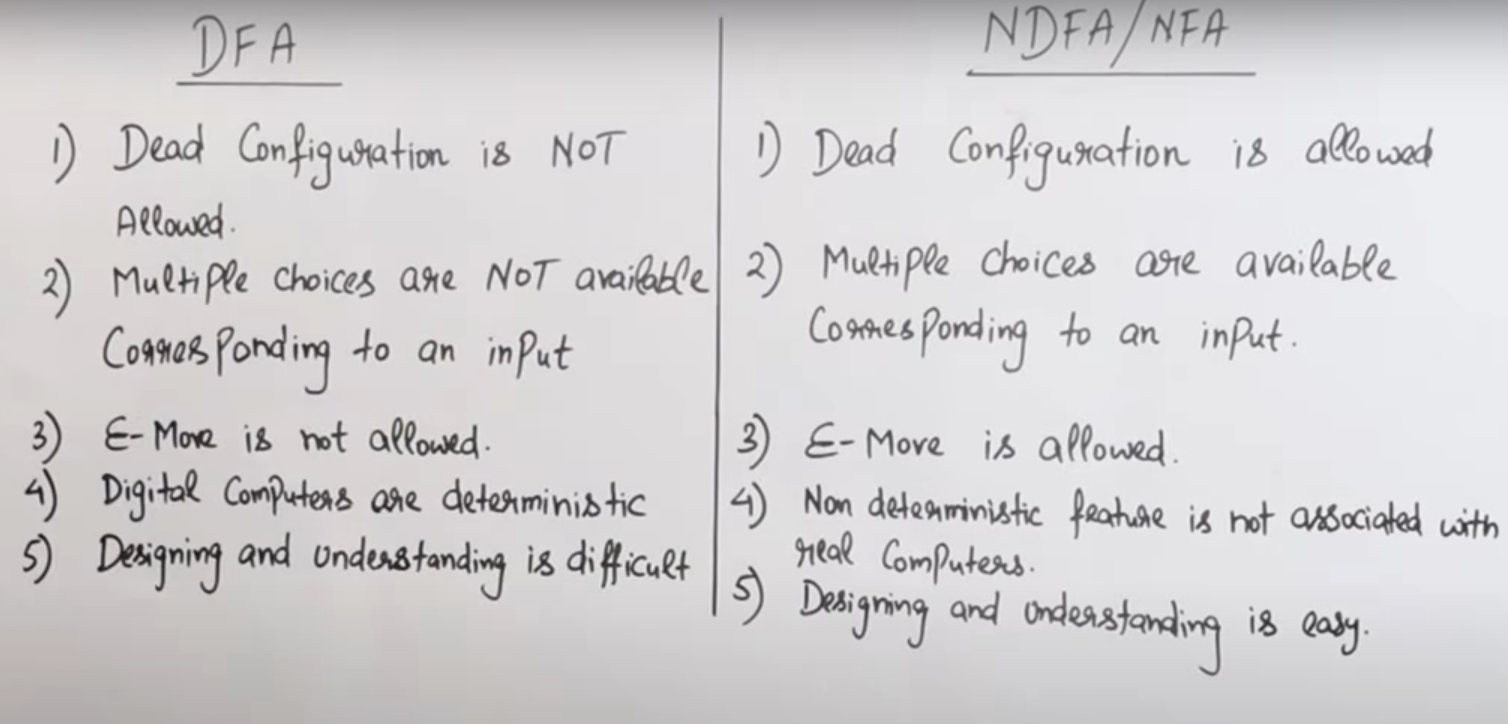




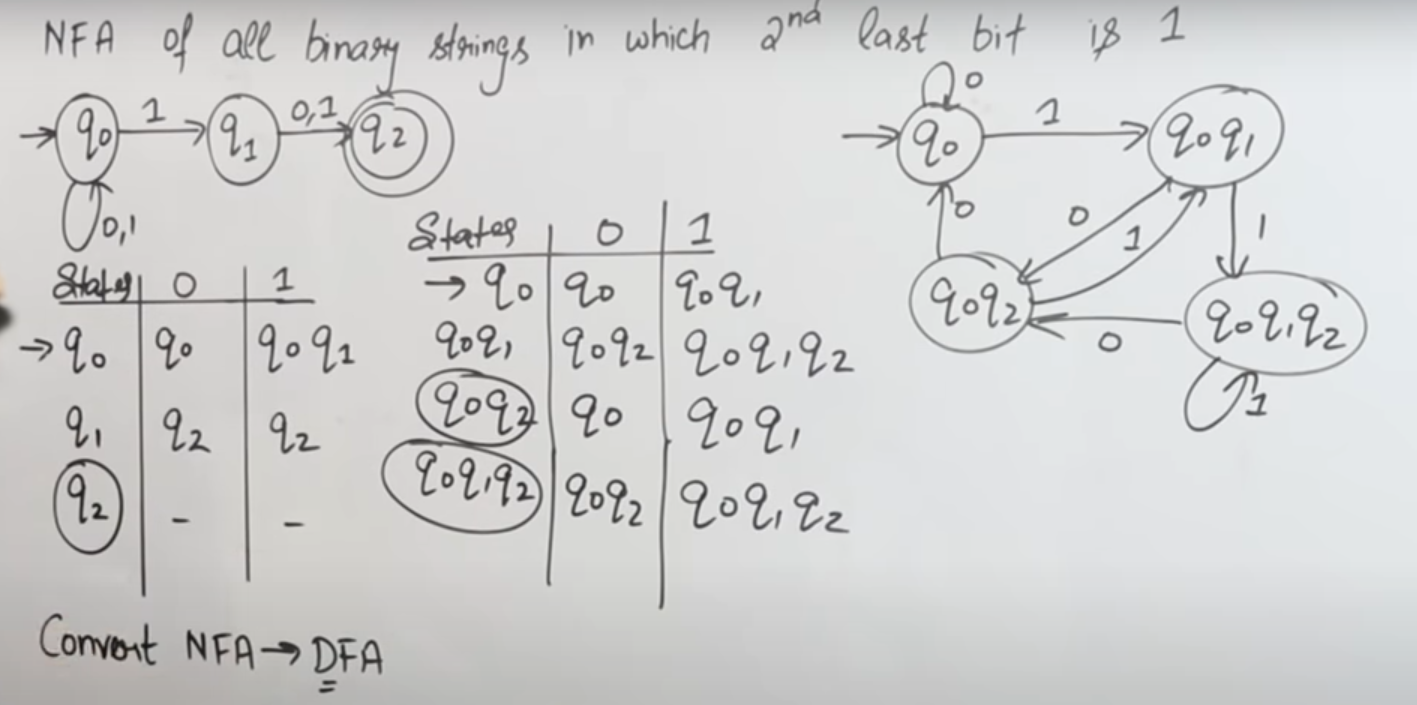




* **DIFFERENT BETWEEN DFA & NFA:-**



* **CONVERT NFA TO DFA:-**



# **Regular Grammar :-**

The regular languages can be generated by regular grammar. In regular grammar, the left-hand side always consists of a single non-terminal.

1. X → xY
2. X → x
3. X → Yx

**Where**

X and Y are used to indicate the Variable (V).

'x' is used to indicate the string of terminals (T\*)

## Types of Regular Grammar

There are two types of regular grammar, which are described as follows:

* Left Linear grammar (LLG)
* Right Linear Grammar (RLG)

### **Left Linear Grammar**

The production must be in the following form in the left linear grammar.

1. X → xY
2. X → x

Where

X, Y belongs to a variable (V), and 'x' belongs to T\*.

### **Right Linear Grammar:-**The production must be in the following form in the right linear grammar.

1. X → Yx
2. X → x

**Where**

X, Y belongs to a variable (V), and 'x' belongs to T\*.

# **Equivalence with Finite autometa:-**

The two finite automata (FA) are said to be equivalent if both the automata accept the same set of input strings .

**Two Automaton are equivalent if they satisfy the following conditions :**

1. The initial and final states of both the automatons must be same.  
2. Every pair of states chosen is from a different automaton only.

# **Properties of Regular languages:-**

In an automata theory, there are different closure properties for regular languages. They are as follows −

* Union
* Intersection
* concatenation
* Kleene closure
* Complement

Let see one by one with an example

## Union

If L1 and If L2 are two regular languages, their union L1 U L2 will also be regular.

## Example

L1 = {an | n > O} and L2 = {bn | n > O}

L3 = L1 U L2 = {an U bn | n > O} is also regular.

## Intersection

If L1 and If L2 are two regular languages, their intersection L1 ∩ L2 will also be regular.

Example

L1= {am bn | n > 0 and m > O} and

L2= {am bn U bn am | n > 0 and m > O}

L3 = L1 ∩ L2 = {am bn | n > 0 and m > O} are also regular.

## Concatenation

If L1 and If L2 are two regular languages, their concatenation L1.L2 will also be regular.

Example

L1 = {an | n > 0} and L2 = {bn | n > O}

L3 = L1.L2 = {am . bn | m > 0 and n > O} is also regular.

## Kleene Closure

If L1 is a regular language, its Kleene closure L1\* will also be regular.

Example

L1 = (a U b )

L1\* = (a U b)\*

## Complement

If L(G) is a regular language, its complement L'(G) will also be regular. Complement of a language can be found by subtracting strings which are in L(G) from all possible strings.

Example

L(G) = {an | n > 3} L'(G) = {an | n <= 3}

## Pumping Lemma:-

The language accepted by **Finite Automata** is known as **Regular Language**. Pumping Lemma is used to prove that a Language is not Regular. It cannot be used to prove that a language is Regular.

The term Pumping Lemma is made up of two words:-

**Pumping:** The word pumping refers to generating many input strings by pushing a symbol in an input string repeatedly.

**Lemma:**  The word Lemma refers to the intermediate theorem in a proof.

There are two Pumping Lemmas, that are defined for

1.) [Regular Languages](https://www.codingninjas.com/codestudio/library/regular-grammar), and

[2.) Context-Free Languages](https://www.codingninjas.com/codestudio/library/context-free-grammar-and-language)

## Pumping Lemma For Regular Languages:

## Pumping lemma is a technique to check that language is Regular language or not. This technique is use only for infinite Languages because Finite languages are always Regular language.

## To check language is finite or not There is some conditions. If all condition will be pass then we can not tell exact solution. May be it is regular or may be it is not regular language.

## But if condition will be fail then we can say that this is not regular language . Generally we use pumping lemma only to describe that this language is not regular language .

* **Theorem:**If A is a Regular Language, then A has a Pumping Length ‘P’ such that any string ‘S’ where |S ≥ P may be divided into three parts S = xyz such that the following conditions must be true:
* **1.) xyiz ∈ A for every i ≥ 0**
* **2.) |y| > 0**
* **3.) |xy| ≤ P**
* Pumping Lemma is used as proof of the **irregularity** of a language. Pumping lemma is negative tester . in pumping lemma , if language will not satisfy all condition of pumping lemma than that language will be not regular language. If language will satisfy all condition of pumping lemma even then(fir v) it may be regular language or non-regular language.
* We use the **CONTRADICTION**method to prove that a language is not Regular.
* **To prove that a language is not Regular using Pumping Lemma, follow the below steps:**
* **Step 1:**Assume that Language A is Regular.
* **Step 2:**It has to have a Pumping Length (say P).
* **Step 3:**All strings longer than P can be pumped |S| ≥ P.
* **Step 4:**Now, find a string ‘S’ in A such that |S| ≥ P.
* **Step 5:**Divide S into x y z strings.
* **Step 6:**Show that xyiz ∉ A for some i.
* **Step 7:**Then consider how S can be divided into x y z.
* **Step 8:**Show that none of the above strings satisfies all three pumping conditions simultaneously.
* **Step 9:**S cannot be pumped == CONTRADICTION.

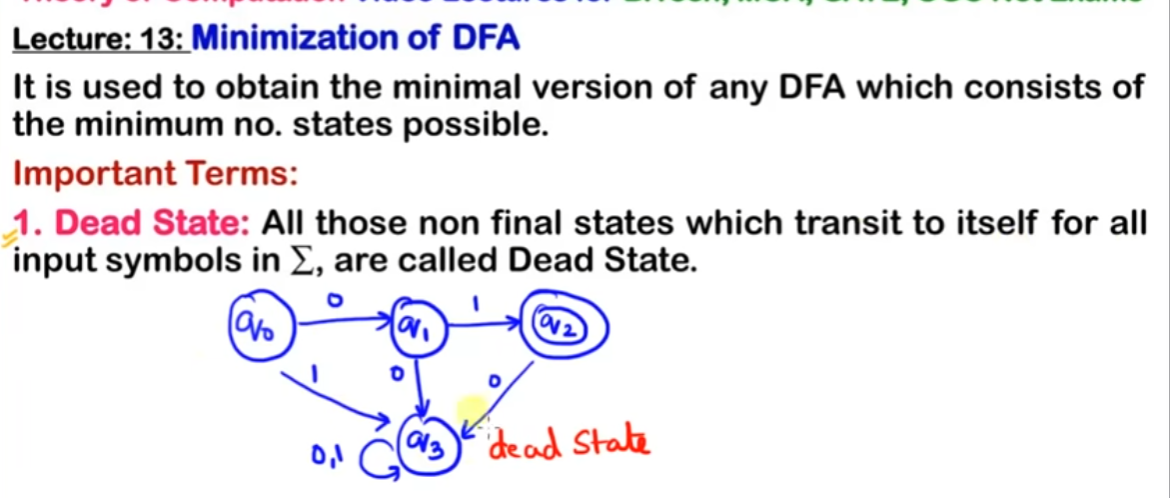
MINIMIZATION OF FINITE AUTOMATA:-

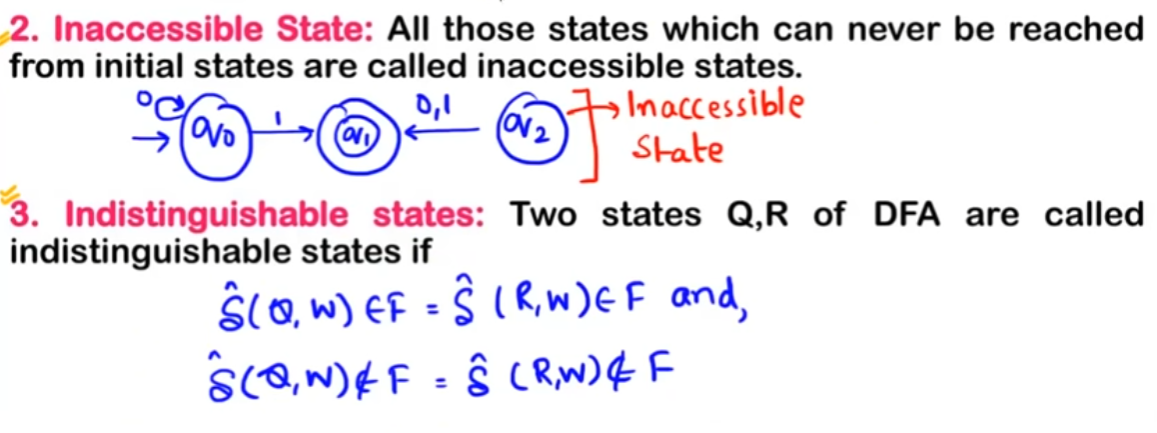
Minimization is use to minimize the states from automata(DFA , NFA).

It is minimize the state between initial state to finalization state.

In other word , Minimization of DFA means reducing the number of states from given FA(finite automata).

**There is some terms to minimize of DFA.**





**We have to follow the various steps to minimize the DFA. These are as follows:-**

**Step 1:** Remove all the states that are unreachable from the initial state via any set of the transition of DFA.

**Step 2:** Draw the transition table for all pair of states.

**Step 3:** Now split the transition table into two tables T1 and T2. T1 contains all final states, and T2 contains non-final states.

**Step 4:** Find similar rows from T1 such that:

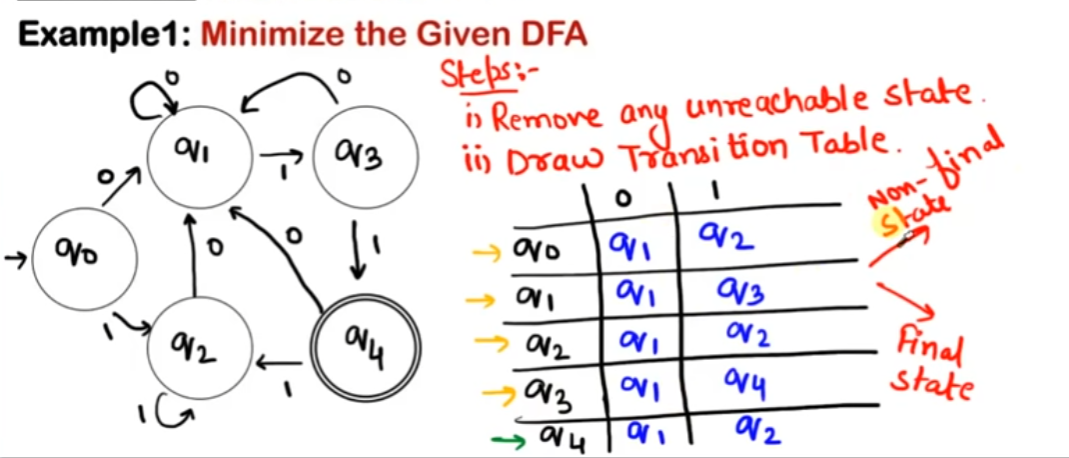
1. 1. δ (q, a) = p
2. 2. δ (r, a) = p

That means, find the two states which have the same value of a and b and remove one of them.

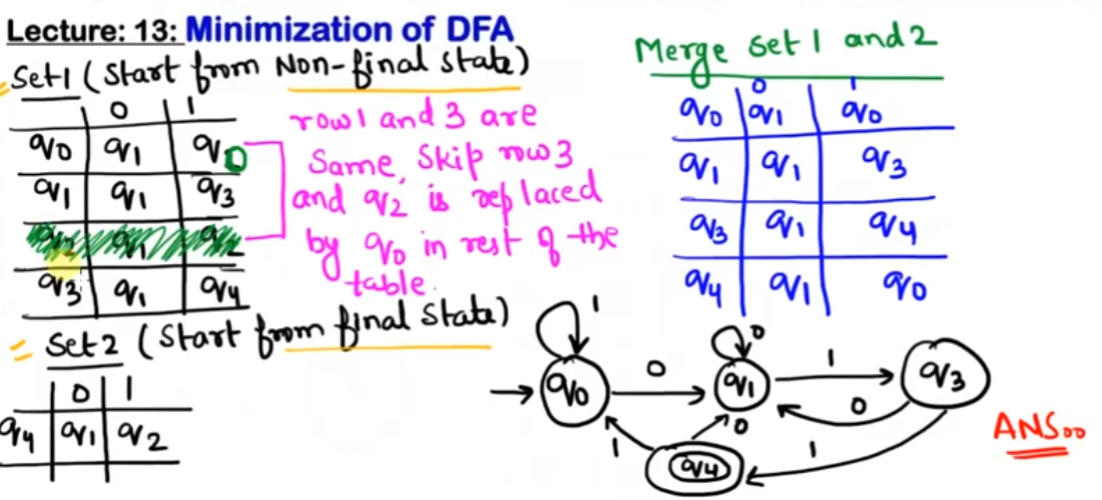
**Step 5:** Repeat step 3 until we find no similar rows available in the transition table T1.

**Step 6:** Repeat step 3 and step 4 for table T2 also.

**Step 7:** Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA.



**Divide the table in two part non-final state and final state:-**



* **Context-free language (CFL):-**

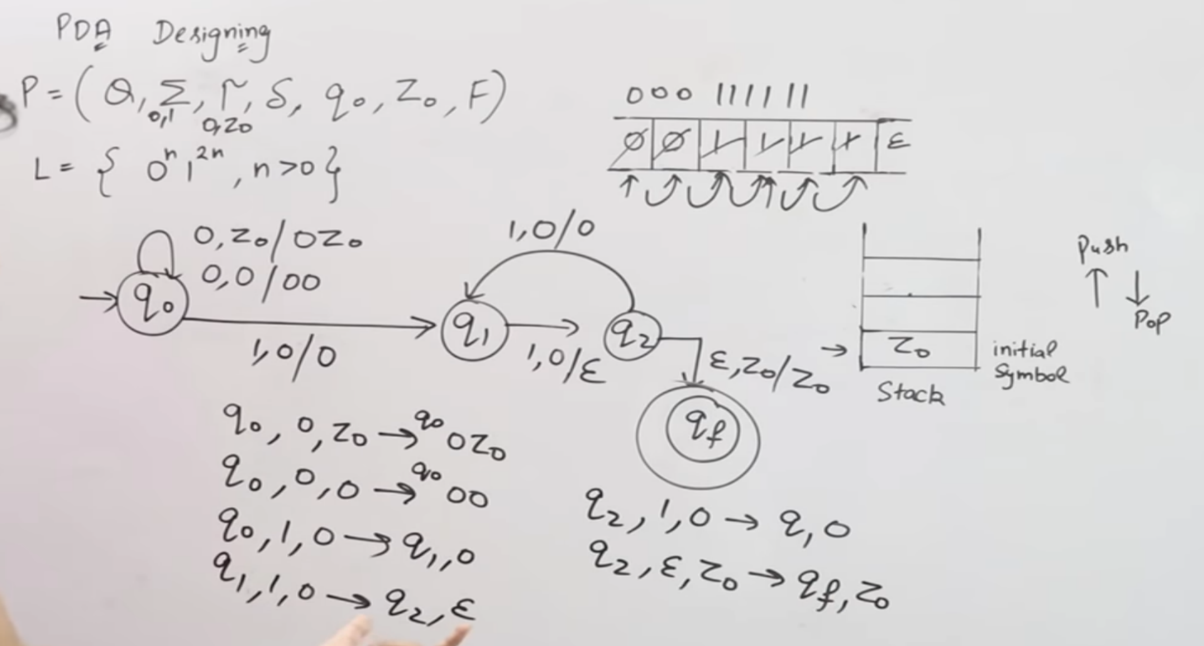
In [formal language theory](https://en.wikipedia.org/wiki/Formal_language_theory), a **context-free language** (**CFL**) is a [language](https://en.wikipedia.org/wiki/Formal_language) generated by a [context-free grammar](https://en.wikipedia.org/wiki/Context-free_grammar) (CFG).

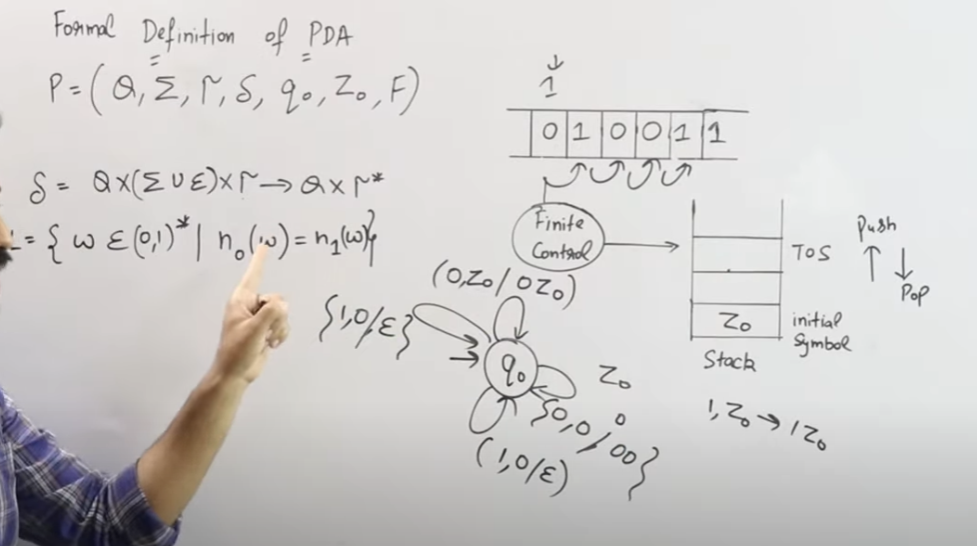
Context-free languages have many applications in [programming languages](https://en.wikipedia.org/wiki/Programming_languages), in particular, most arithmetic expressions are generated by context-free grammars.

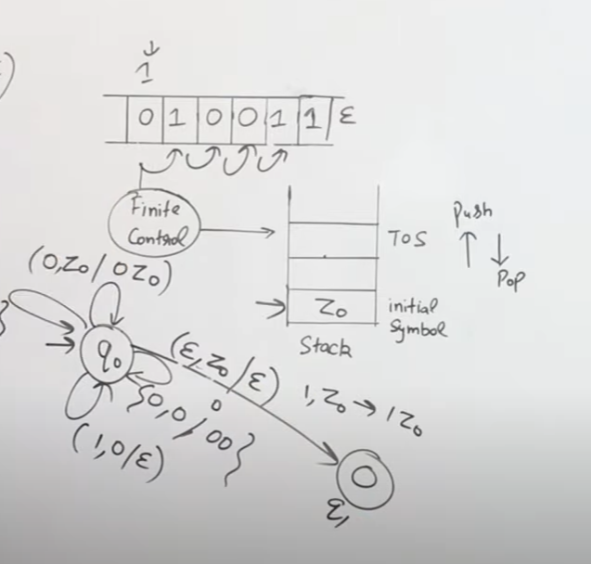
# **Pushdown Automata:-**

We have already discussed [finite automata](https://www.geeksforgeeks.org/introduction-of-finite-automata/). But finite automata can be used to accept only regular languages.   
Pushdown Automata is a finite automata with extra memory called stack which helps Pushdown automata to recognize Context Free Languages.   
    
A Pushdown Automata (PDA) can be defined as :

* Q is the set of states
* ∑is the set of input symbols
* Γ is the set of pushdown symbols (which can be pushed and popped from stack)
* q0 is the initial state
* Z is the initial pushdown symbol (which is initially present in stack)
* F is the set of final states
* δ is a transition function which maps Q x {Σ ∪ ∈} x Γ into Q x Γ\*. In a given state, PDA will read input symbol and stack symbol (top of the stack) and move to a new state and change the symbol of stack.
* **Question of Pushdown Automata:-**







# **Context-Free Grammar (CFG):-**

CFG stands for context-free grammar. It is a formal grammar which is used to generate all possible strings in a given formal language. Context-free grammar G can be defined by four tuples as:

**T** is the final set of a terminal symbol. It is denoted by lower case letters.

**V** is the final set of a non-terminal symbol. It is denoted by capital letters.

**P** is a set of production rules, which is used for replacing non-terminals symbols(on the left side of the production) in a string with other terminal or non-terminal symbols(on the right side of the production).

**S** is the start symbol which is used to derive the string. We can derive the string by repeatedly replacing a non-terminal by the right-hand side of the production until all non-terminal have been replaced by terminal symbols.

# **Chomsky's Normal Form (CNF):-**

CNF stands for Chomsky normal form. A CFG(context free grammar) is in CNF(Chomsky normal form) if all production rules satisfy one of the following conditions:

* Start symbol generating ε. For example, A → ε.
* A non-terminal generating two non-terminals. For example, S → AB.
* A non-terminal generating a terminal. For example, S → a.

# **Greibach Normal Form (GNF):-**

GNF stands for Greibach normal form. A CFG(context free grammar) is in GNF(Greibach normal form) if all the production rules satisfy one of the following conditions:

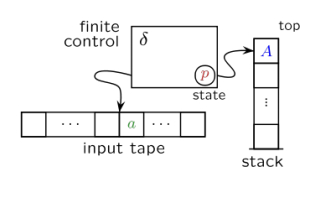
* A start symbol generating ε. For example, S → ε.
* A non-terminal generating a terminal. For example, A → a.
* A non-terminal generating a terminal which is followed by any number of non-terminals. For example, S → aASB.

# **Non-deterministic Pushdown Automata:-**

The non-deterministic pushdown automata is very much similar to NFA. We will discuss some CFGs which accepts NPDA.

The CFG which accepts deterministic PDA accepts non-deterministic PDAs as well.

The Non-deterministic Push down Automata (NPDAs) are like finite automata (FA), except they also have a stack memory where they can store an arbitrary amount of information.



Read/write stack memory works as LIFO: Last In, First Out

* **NPDA can be described as the following −**
* A finite set Q of states (& the start state & the set of accepting/final states).
* A finite set ∑ which is called the input alphabet.
* A finite set Γ which is called the stack alphabet (& the initial stack symbol $).
* A finite set of transition instructions (or a transition function T).
* **Recursive Language:-**

A language L is recursive (decidable) if L is the set of strings accepted by some Turing Machine (TM) that halts on every input.

If L is a recursive language then −

If w ∈ L then a TM halts in a final state,

If w ∉ L then TM halts in a non-final state.

**Example**

When a Turing machine reaches a final state, it halts. We can also say that a Turing machine M halts when M reaches a state q and a current symbol ‘a’ to be scanned so that δ(q, a) is undefined.

**Recursive Enumerable Language**

A language L is recursively enumerable if L is the set of strings accepted by some TM.

If L is a recursive enumerable language then −

If w ∈ L then a TM halts in a final state,

If w ∉ L then a TM halts in a non-final state or loops forever.

**Recursive Languages are also recursive enumerable**

# **Derivation tree/parse Tree:-**

It is a way to show how the derivation can be done through Tree. Derivation/parse tree is a graphical representation.

* **Properties:-**

The properties of the derivation/parse tree are given below −

* The root node is always the starting symbols.
* The derivation is read from left to right.
* The leaf node is always the terminal node.
* The interior nodes are always the non-terminal nodes.

## Example

The production rules for the derivation tree are as follows −

E=E+E

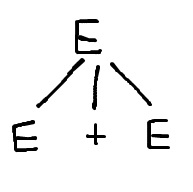
E=E\*E

E=a|b|c

Here, let the input be a\*b+c

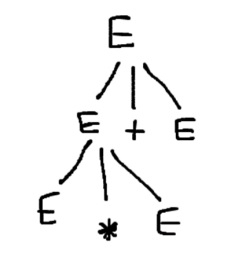
## Step 1

The step 1 is given below



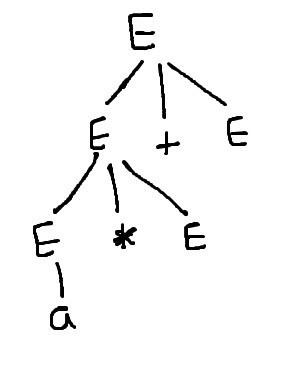
## Step 2

The step 2 is given below

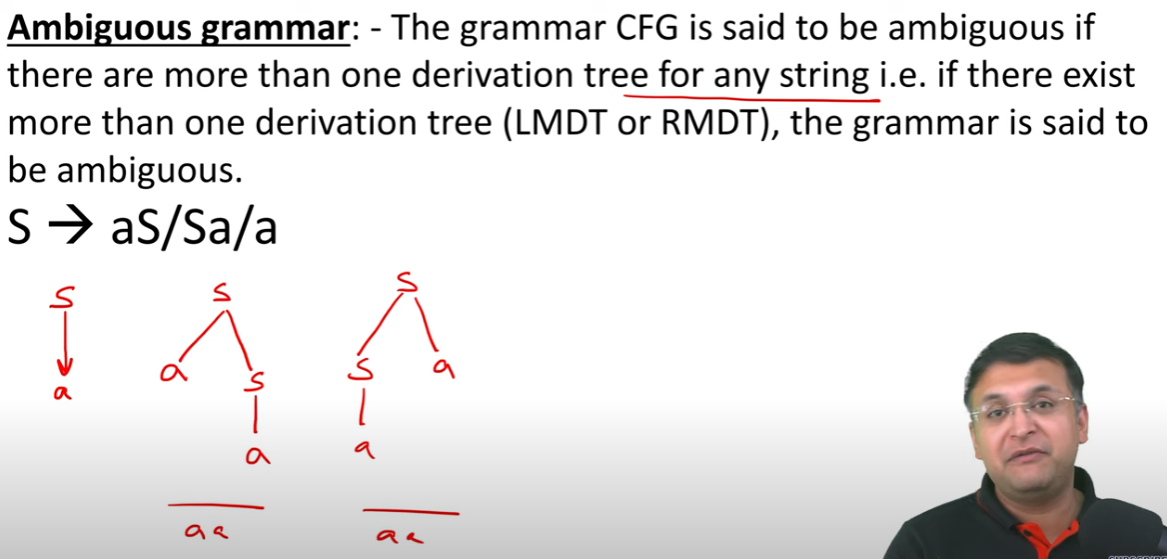


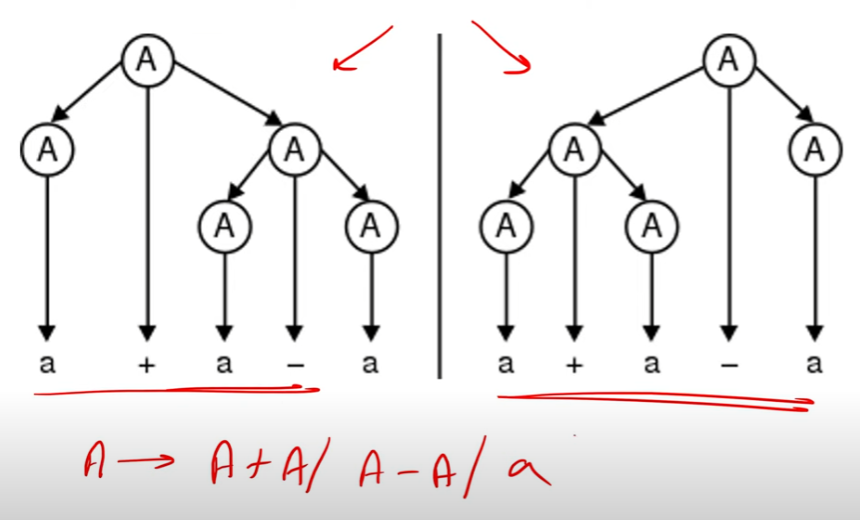
## Step 3

The step 3 is given below

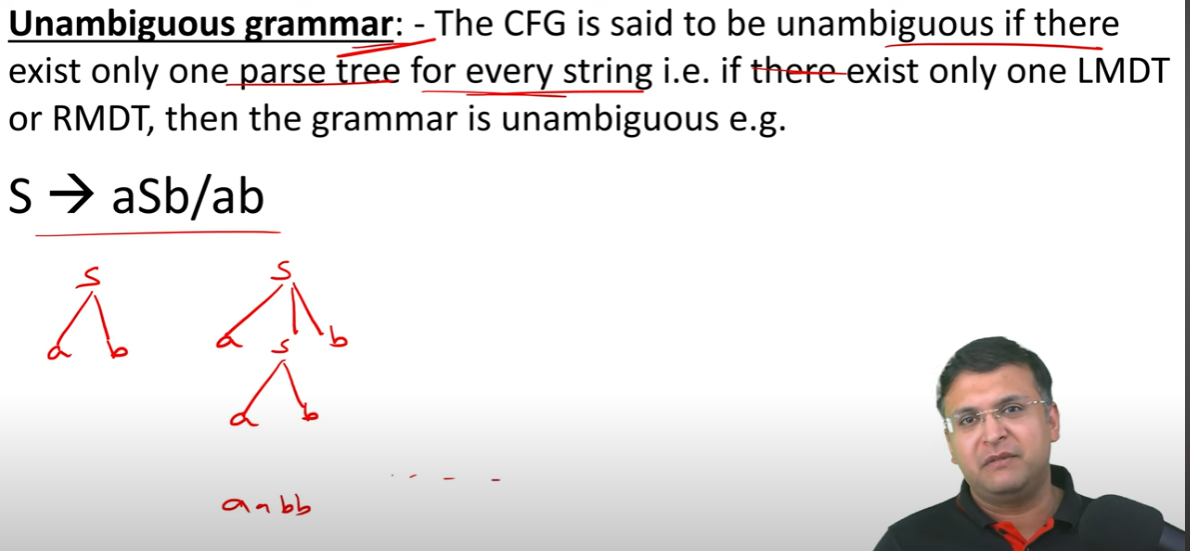


* **Ambiguity in context free grammar:-**





* **Unambiguity in context free grammar:-**



# **Pumping lemma for context free language:-**

Pumping lemma for context free language (CFL) is used to prove that a language is not a Context free language

Assume L is context free language

Then there is a pumping length n such that any string w εL of length>=n can be written as follows −

|w|>=n

We can break w into 5 strings, w=uvxyz, such as the ones given below

* |vxy| >=n
* |vy| # ε
* For all k>=0, the string uvkxyyz∈L

The steps to prove that the language is not a context free by using pumping lemma are explained below −

* Assume that L is context free.
* The pumping length is n.
* All strings longer than n can be pumped |w|>=n.
* Now find a string 'w' in L such that |w|>=n.
* Divide w into uvxyz
* Show that uvkxykz ∉L for some k
* Then, consider the ways that w can be divided into uvxyz.
* Show that none of these can satisfy all the 3 pumping conditions at same time.
* w cannot be pumped (contradiction)

# **Closure Property of CFL:-**

Context-free languages are **closed** under −

* Union
* Concatenation
* Kleene Star operation

## Union:-

Let L1 and L2 be two context free languages. Then L1 ∪ L2 is also context free.

## Concatenation:-

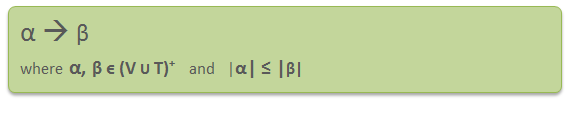
If L1 and L2 are context free languages, then L1L2 is also context free.

## Kleene Star:-

* If L is a context free language, then L\* is also context free.
* **Context-free languages are not closed under :−**
* **Intersection** − If L1 and L2 are context free languages, then L1 ∩ L2 is not necessarily context free.
* **Intersection with Regular Language** − If L1 is a regular language and L2 is a context free language, then L1 ∩ L2 is a context free language.
* **Complement** − If L1 is a context free language, then L1’ may not be context free.

# **Difference Between NPDA and DPDA:-**

| S. No | **DPDA(Deterministic Pushdown Automata)** | **NPDA(Non-deterministic Pushdown Automata)** |
| --- | --- | --- |
| 1. | It is less powerful than NPDA. | It is more powerful than DPDA. |
| 2. | It is possible to convert every DPDA to a corresponding NPDA. | It is not possible to convert every NPDA to a corresponding DPDA. |
| 3. | The language accepted by DPDA is a subset of the language accepted by NDPA. | The language accepted by NPDA is not a subset of the language accepted by DPDA. |
| 4. | The language accepted by DPDA is called DCFL(Deterministic Context-free Language) which is a subset of NCFL(Non-deterministic Context-free Language) accepted by NPDA. | The language accepted by NPDA is called NCFL(Non-deterministic Context-free Language). |

* **Context-Sensitive Grammar –**  
  A Context-sensitive grammar is an Unrestricted grammar in which all the productions are of form :–  
    
  Where α and β are strings of non-terminals and terminals.

Context-sensitive grammars are **more powerful** than context-free grammars because there are some languages that can be described by CSG but not by context-free grammars and CSL are less powerful than Unrestricted grammar. That’s why context-sensitive grammars are positioned between context-free and unrestricted grammars in the Chomsky hierarchy.

Context-sensitive grammar has 4-tuples. **G = {T, V, P, S}**, Where  
T = Set of non-terminal symbols  
V= Set of terminal symbols  
S = Start symbol of the production  
P = Finite set of productions  
All rules in P are of the form α1 A α2 –> α1 β α2

* **Context-sensitive Language:** The language that can be defined by context-sensitive grammar is called CSL. Properties of CSL are :
* Union, intersection and concatenation of two context-sensitive languages is context-sensitive.
* Complement of a context-sensitive language is context-sensitive.

**Example –**

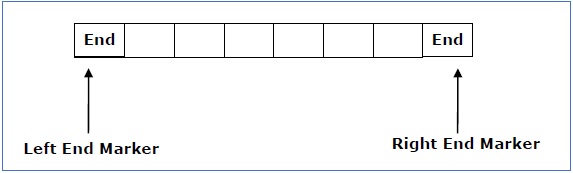
Consider the following CSG.  
S → abc/aAbc  
Ab → bA  
Ac → Bbcc  
bB → Bb  
aB → aa/aaA

# **Linear Bounded Automata:-**

Limited cell of tape in turing machine is called linear bound automata. In turing machine have unlimited cell of tape. In linear bound head can move both side left and right.

A linear bounded automaton can be defined as an 8-tuple (Q, X, ∑, q0, ML, MR, δ, F) where −

* **Q** is a finite set of states
* **X** is the tape alphabet
* **∑** is the input alphabet
* **q0** is the initial state
* **ML** is the left end marker
* **MR** is the right end marker where MR ≠ ML
* **δ** is a transition function which maps each pair (state, tape symbol) to (state, tape symbol, Constant ‘c’) where c can be 0 or +1 or -1
* **F** is the set of final states

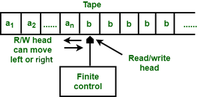


A deterministic linear bounded automaton is always **context-sensitive** and the linear bounded automaton with empty language is **undecidable.**.

* **TURING MACHINE:-**

Turing Machine was invented by Alan Turing in 1936 and it is used to accept Recursive Enumerable Languages (generated by Type-0 Grammar).

A Turing machine consists of a tape of infinite length. In turing machine tape read and writes operation can be performed. The tape consists of infinite cells. In turing machine tape, each cell either contains input symbol or blank symbol. It also consists of a head pointer which points to starting cell. it can move in both directions(left and right).

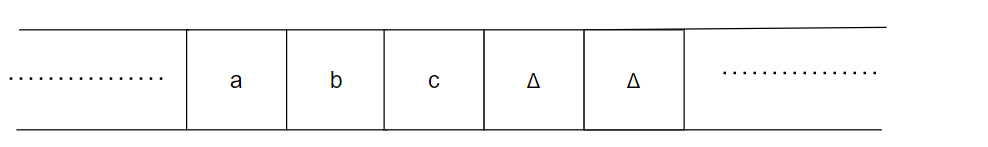


A TM is expressed as a 7-tuple (Q, T, B, ∑, δ, q0, F) where: 

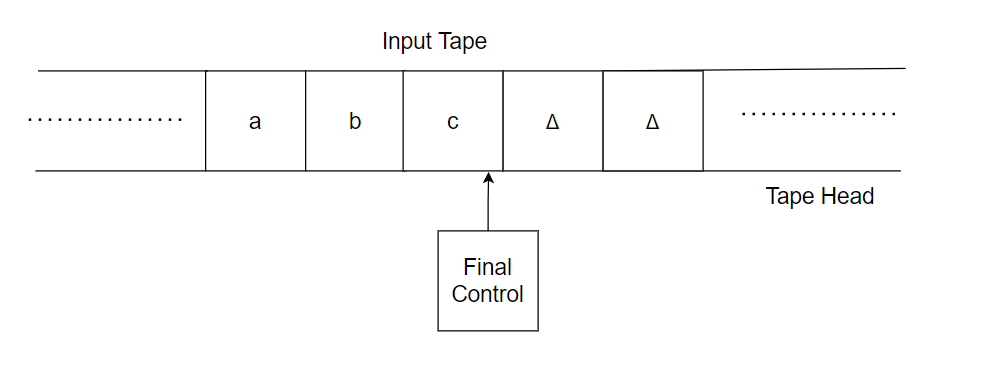
* **Q** is a finite set of states
* **T** is the tape alphabet (symbols which can be written on Tape)
* **B** is blank symbol (every cell is filled with B except input alphabet initially)
* **∑** is the input alphabet (symbols which are part of input alphabet)
* **δ**is a transition function which maps Q × T → Q × T × {L,R}. Depending on its present state and present tape alphabet (pointed by head pointer), it will move to new state, change the tape symbol (may or may not) and move head pointer to either left or right.
* **q0** is the initial state
* **F** is the set of final states. If any state of F is reached, input string is accepted.
* **The basic model for Turing machine:-**

The following representation can be used to model the turning machine.

1. The input tape has an infinite number of cells, each containing one input symbol, allowing the input string to be taped. The empty tape is filled with blank characters as shown below:



1. **the tape head which is responsible for reading the current input symbol. The tape head can move to left to right.**
2. It consists of a finite set of states that a machine must pass through.
3. **Finite set of symbols called external symbols which are used in building the logic of turing machine.**



# **Differentiate between recognizable and decidable in the Turing machine :-**

[More Detail](https://www.tutorialspoint.com/data-science-full-course-for-beginners/index.asp" \t "_blank)

When we talk about Turing machines (TM) it could accept the input, reject it or keep computing which is called loop.

* **Recognizable:-**

A language is recognizable if and only if a Turing machine accepts the string, when the provided input lies in the language.

Also, a language can be recognizable if the TM either terminates and rejects the string or doesn't terminate at all. This means that the TM continues with the computing when the provided input doesn't lie in the language.

* **Decidable:-**

The language is decidable if and only if there is a machine which accepts the string when the provided input lies in that language and rejects the string when provided input doesn't lie in that language.

**Example**

* A = {hM, wi | M is a DFA and w ∈ L(M)} is decidable.
* A = {hM, wi | M is a TM and w ∈ L(M)} is recognizable.

The major differences between a recognizable and a decidable in turning machine are as follows −

| **Sr. No** | **Turing Recognizable** | **Turing decidable** |
| --- | --- | --- |
| 1 | A language which is Turing Recognizable if there is a Machine that will halt and accept only the strings in that language and not in that language, then that TM either rejects, or does not halt at all. | A language is said to be Decidable if there is a Machine that will accept strings in the language and reject strings not in the language. |
| 2 | A Language is called Turing Recognizable if some Turing Machine recognizes it. | A Language is called Turing Decidable if some Turing Machine decides it. |
| 3 | If there exists a Turing Machine such that when encountering a string in that language, the machine terminates and accepts that string then we can say that type of language is a Turing recognizable. | If there exists a Turing Machine such that when encountering a string in that language, the machine terminates and accepts that string then we say that type of language is Turing decidable. |
| 4 | If there exists a Turing Machine such that when encountering a string not in that language, the machine either terminates and rejects that string or doesn’t terminate at all then we can say it is Turing-Recognizable. | If there exists a Turing Machine such that when encountering a string not in that language, the machine terminates and rejects that string then we can say it is Turing-Decidable. |
| 5 | It is not stronger condition than Turing Decidable | It is a stronger condition than Turing-Recognizable. |

* **Closure properties of various language:-**





* **Variant of Turing machine:-**

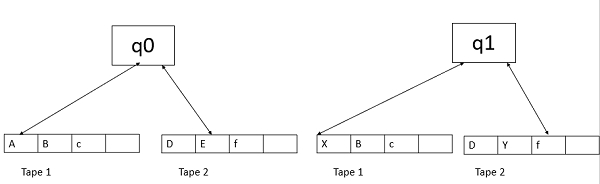
**Turing machines (TM) can also be deterministic or non-deterministic, but this does not make them any more or less powerful. We can use and implement various turing machine but it will not increase or decrease the power of turing machine.**

**Power means how many languages will be accepted by turing machine but when we use various type of turing machine that will be not effect on turing machine.**

* **Variations of turing machine:-**
* Multi-track
* Multi-tape
* Multi-head
* Multi-dimensional tape
* The off-line Turing machine
* **Multi-tape Turing Machine:-**

A Turing machine have several tapes then it called multi tape Turing machine.

Every tape’s have own Read/Write head.



* **Turing machine with stay option:-**

In stay option head will be stay on a particular position.

It will be not move left and right side.

**Turing machine with semi-finite state:-**

In semi-finite state , the tape will finite from one side and opposite side will be infinite but power will be not change.

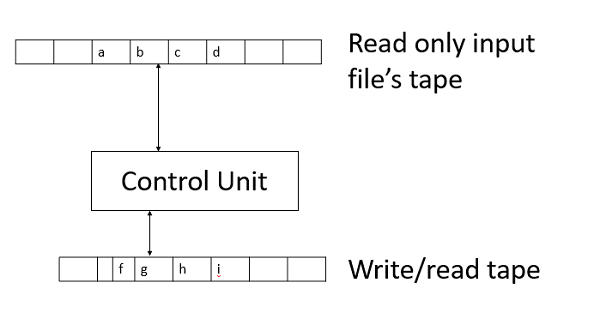


****



* **Ofline Turing machine:-**

**In ofline turing machine we inpute(read) the value in one tape and change(write / read) the Value of tape in second tape.**



**Multi-head Turing machine:-**

It has multiple heads in tape.

Each head independently reads/ writes symbols and moves left/right or keeps stationery.

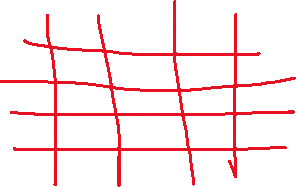
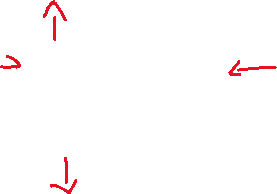


* **Non Deterministic Turing Machine:-**

**In non-deterministic turing machine head can move left , right , up , down.**



**In this we can change the value left side , right side also up and down side.**



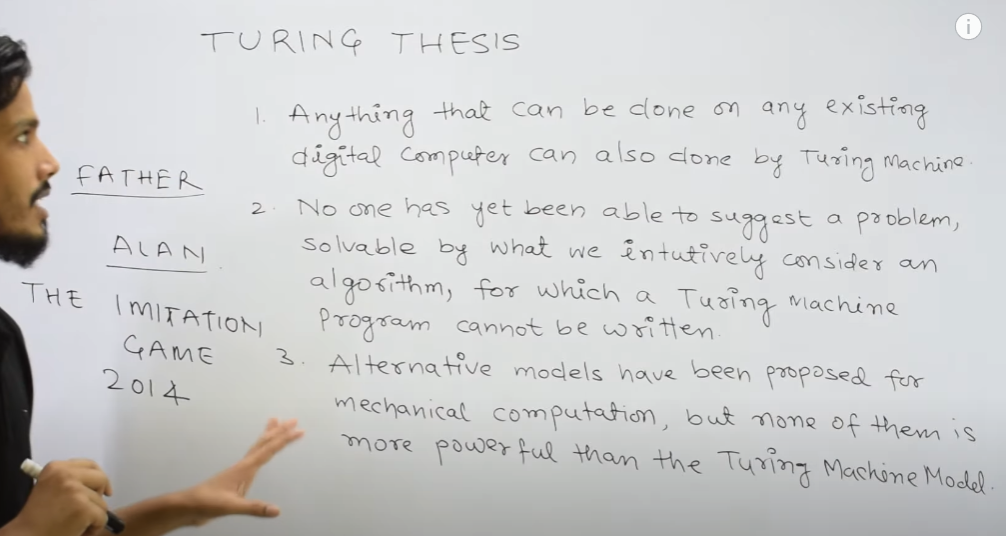
# **Unrestricted Grammar:-**

**Unrestricted Grammar** or **Phrase Structure Grammar** is the most general in the **Chomsky Hierarchy of classification**.  This is **type0** grammar, generally used to generate **Recursively Enumerable languages**.

A **Turning Machine** can simulate **Unrestricted Grammar** and **Unrestricted Grammar** can simulate **Turning Machine** configurations.

**N** - A finite set of **non-terminal** symbols or **variables**,  
**Σ** - It is a set of terminal symbols or the alphabet of the language being described, where **N ∩ Σ = φ**,  
**P** - It is a finite set of "**productions**" or "**rules**",  
**S** - It is a **start variable** or **non-terminal** symbol.

* **Turing thesis:-**



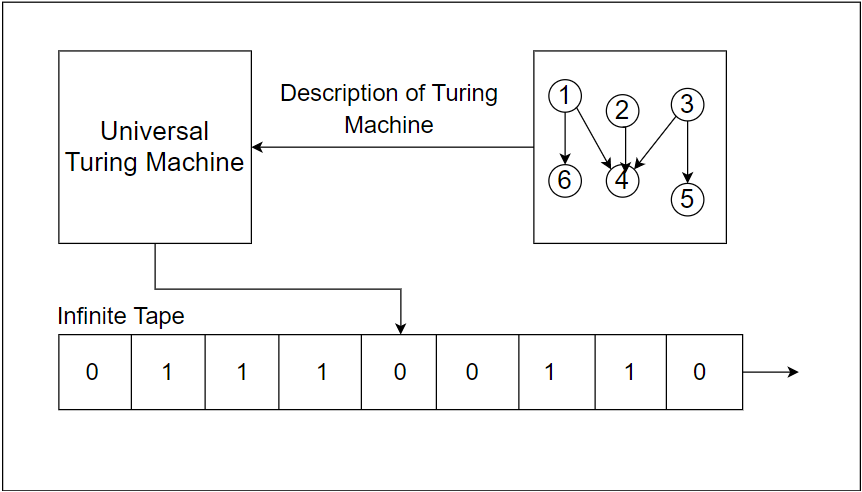
* **Universal turing machine(UTM):-**

Turing machine is power-full turing machine .

Suppose we want to adder , subtract , multiple than we have to make different turing machine for these.

But in universal turing machine we can describe all these things in single turing machine . like adder , subtraction , multiple etc.

**Universal Turning Machine** stimulates a Turning Machine. Universal Turing Machine can be considered as a subset of all the Turing machines, it can match or surpass other Turing machines including itself. Universal Turing Machine is like a single Turing Machine that has a solution to all problems that is computable. It contains a Turning Machine description as input along with an input string, runs the Turning Machine on the input and returns a result.



# **Difference between Turing machine and Universal Turing machine:-**

The difference between Turing Machine and Universal Turing Machine are as follows:

|  |  |  |
| --- | --- | --- |
|  | **Turing Machine** | **Universal Turing Machine** |
| 1. | It is a mathematical model of computation it manipulates symbols on the tape according to the rules defined | Universal Turing Machine is like a single Turing Machine that has a solution to all problem that is computable |
| 2. | A program can be compared to a Turing Machine | Programmable Turing Machine is called Universal Turing Machine |
| 3. | Turing machine’s temporary storage is tape. The infinite cells of the Turing machine can contain input symbols and blanks. | Universal Turing Machine contains Turning Machine description as input along with an input string, runs the Turning Machine on the input and returns the result. |
| 4. | Turing machines help us understand the fundamental limitations of mechanical computation power | Although developed for theoretical reasons, it helped in the development of stored program computers |
| 5. | A Turing machine is a formal model of a computer with a fixed program | Universal Turing Machine provides a solution to problems that are computable |
| 6. | It does not minimize the space complexity | It minimizes space complexity |

* **Diagonalization language:-**
* We have seen one language, the diagonalization language, that is not accepted by any Turing machine. This proves the diagonalization language is not recursively enumerable.
* **Universal Language :-**
* We shall now define a language *Lu*, the universal language, that can be accepted by a Turing machine but is still undecidable. *Lu* is recursively enumerable but not recursive.
* *Lu* is the set of binary strings that consist of encoded pairs (*M*, *w*) such that *M* is an encoding of a Turing machine and *w* is an encoding of a binary input string accepted by that Turing machine.

# **Rice Theorem:-**

Rice theorem states that any non-trivial property of a recursive enumerable language which is recognized by a Turing machine is undecidable. A property, P, is the language of all Turing machines that satisfy that property.

Trivial means , if all property of language is mapped/satisfied with only ‘0’ or only ‘1’ that is called trivial property.

Non-Trivial means , if all property of language is mapped/satisfied with ‘0’ and ‘1’ that is called non-trivial property.

In non-triviel properties some properties accepted by ‘0’ and some properties accepted by ‘1’.

## Formal Definition

If P is a non-trivial property, and the language holding the property, Lp , is recognized by Turing machine M, then Lp = {<M> | L(M) ∈ P} is undecidable.

## Description and Properties

* P is a Property of languages. If any language belongs from P (L ∈ P), so that is called that Language satisfies the property P(property of language).
* A property is called to be trivial if it is not satisfied by any recursively enumerable languages,

**Proof:-**

Suppose, a property P is non-trivial and φ ∈ P.

Since, P is non-trivial, at least one language satisfies P, i.e., L(M0) ∈ P , ∋ Turing Machine M0.

Let, w be an input in a particular instant and N is a Turing Machine which follows −

On input x

* Run M on w
* If M does not accept (or doesn't halt), then do not accept x (or do not halt)
* If M accepts w then run M0 on x. If M0 accepts x, then accept x
* **Decidable and Undecidable problems about language:-**
* **Decidable Problems:-**  
  A problem is decidable if we can construct a Turing machine which will halt in finite amount of time for every input and give answer as ‘yes’ or ‘no’. A decidable problem has an algorithm to determine the answer for a given input.

**Examples**

* **Equivalence of two regular languages:** Given two regular languages, there is an algorithm and Turing machine to decide whether two regular languages are equal or not.
* **Finiteness of regular language:** Given a regular language, there is an algorithm and Turing machine to decide whether regular language is finite or not.
* **Emptiness of context free language:** Given a context free language, there is an algorithm whether CFL is empty or not.
* **Undecidable Problems:-**  
  A problem is undecidable if there is no Turing machine which will always halt in finite amount of time to give answer as ‘yes’ or ‘no’. An undecidable problem has no algorithm to determine the answer for a given input.

**Examples**

* **Ambiguity of context-free languages:** Given a context-free language, there is no Turing machine which will always halt in finite amount of time and give answer whether language is ambiguous or not.
* **Equivalence of two context-free languages:** Given two context-free languages, there is no Turing machine which will always halt in finite amount of time and give answer whether two context free languages are equal or not.
* **Everything or completeness of CFG:** Given a CFG and input alphabet, whether CFG will generate all possible strings of input alphabet (∑\*)is undecidable
* **Reducibility and Undecidability**:-  
  Language A is reducible to language B (represented as A≤B) if there exists a function f which will convert strings in A to strings in B as:

w ɛ A <=> f(w) ɛ B

Theorem 1: If A≤B and B is decidable then A is also decidable.  
Theorem 2: If A≤B and A is undecidable then B is also undecidable.

**P class problem:-**

A problem which can be solved in polynomial time is know as P-class problem

Like:- sorting , searching etc.

**NP-Class Problem:-**

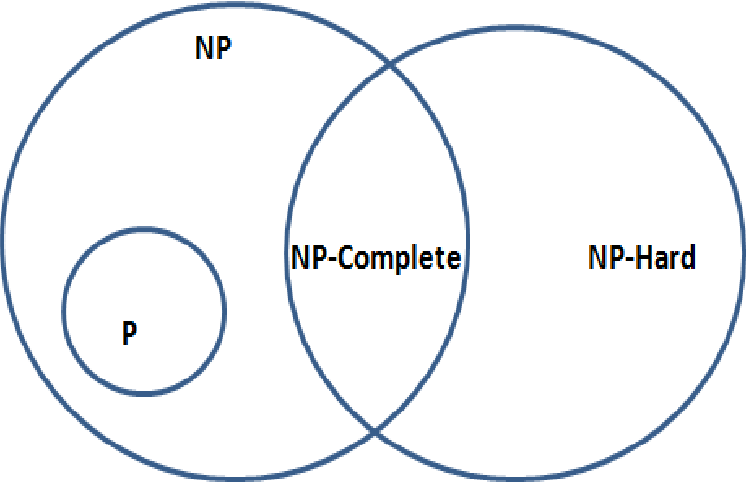
A problem which can not solve in polynomial time is know as NP-Class problem . It can be verify in polynomial time like:-Knapsack problem , su-do-ku , TSP, etc.

* **Tractability:-**

P-class problem is called tractability.

* **Intractability:-**

Np problem is called Intractability.



**CO-NP:-**

CO\_NP Means NP-Complete Which problem that is come under NP and NP-Hard that problem is called NP-complete.

**Co-NP Class**

Co-NP stands for the complement of NP Class. It means if the answer of a problem in Co-NP is No, then there is proof that can be in polynomial time.

**Features:**

1. If a problem X is in NP, then its complement X’ is also is in CoNP.
2. For an NP and CoNP problem, there is no need to verify all the answers at once in polynomial time,

**NP-Hard:-**

