THEORY OF COMPUTATION LECTURE NOTES

UNIT 1 **(8 Lectures)**

Introduction to Automata: The Methods Introduction to Finite Automata, Structural Representations, Automata and Complexity. Proving Equivalences about Sets, The Contrapositive, Proof by Contradiction, Inductive Proofs: General Concepts of Automata Theory: Alphabets Strings, Languages, Applications of Automata Theory.

Finite Automata: The Ground Rules, The Protocol, Deterministic Finite Automata: Definition of a Deterministic Finite Automata, How a DFA Processes Strings, Simpler Notations for DFA's, Extending the Transition Function to Strings, The Language of a DFA

Nondeterministic Finite Automata: An Informal View. The Extended Transition Function, The Languages of an NFA, Equivalence of Deterministic and Nondeterministic Finite Automata.

Finite Automata With Epsilon-Transitions: Uses of ∈-Transitions, The Formal Notation for an ∈-NFA, Epsilon-Closures, Extended Transitions and Languages for ∈-NFA's, Eliminating ∈-Transitions.

MODULE-I

What is TOC?

In theoretical computer science, the **theory of computation** is the branch that deals with whether and how efficiently problems can be solved on a model of computation, using an algorithm. The field is divided into three major branches: automata theory, computability theory and computational complexity theory.

In order to perform a rigorous study of computation, computer scientists work with a mathematical abstraction of computers called a model of computation. There are several models in use, but the most commonly examined is the Turing machine.

Automata theory

In theoretical computer science, **automata theory** is the study of abstract machines (or more appropriately, abstract 'mathematical' machines or systems) and the computational problems that can be solved using these machines. These abstract machines are called automata.

This automaton consists of

- **states** (represented in the figure by circles),
- and **transitions** (represented by arrows).

As the automaton sees a symbol of input, it makes a *transition* (or *jump*) to another state, according to its ***transition function*** (which takes the current state and the recent symbol as its inputs).

Uses of Automata: compiler design and parsing.

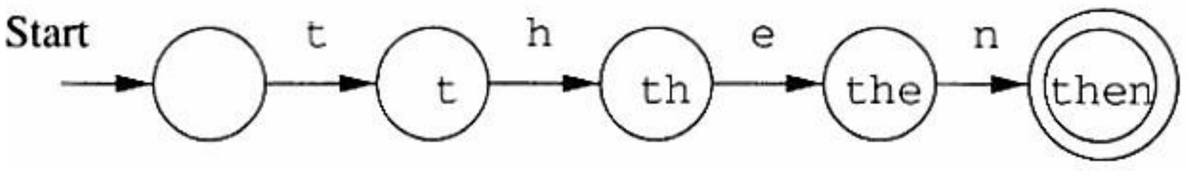


Figure 1.2: A finite automaton modeling recognition of **then**

Introduction to formal proof:

Basic Symbols used :

U – Union

∩- Conjunction

∈- Empty String

Φ – NULL set

7- negation

‘ – compliment

= > implies

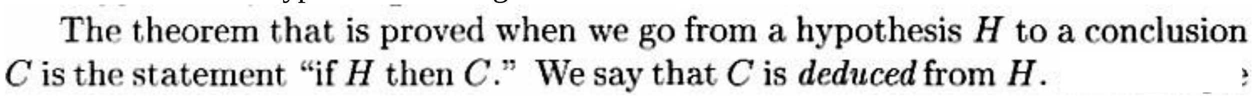
Additive inverse: a+(-a)=0
Multiplicative inverse: a*1/a=1
Universal set U={1,2,3,4,5}
Subset A={1,3}
A' ={2,4,5}
Absorption law: AU(A ∩B) = A, A∩(AUB) = A

De Morgan's Law:
(AUB)' =A' ∩ B'
(A∩B)' = A' U B'
Double compliment
(A')' =A
A ∩ A' = Φ

Logic relations:
a **€** b = > **7**a U b
7(a∩b)=**7**a U **7**b

Relations:
Let a and b be two sets a relation R contains aXb.
Relations used in TOC:
Reflexive: a = a
Symmetric: aRb = > bRa
Transition: aRb, bRc = > aRc
If a given relation is reflexive, symmentric and transitive then the relation is called equivalence relation.

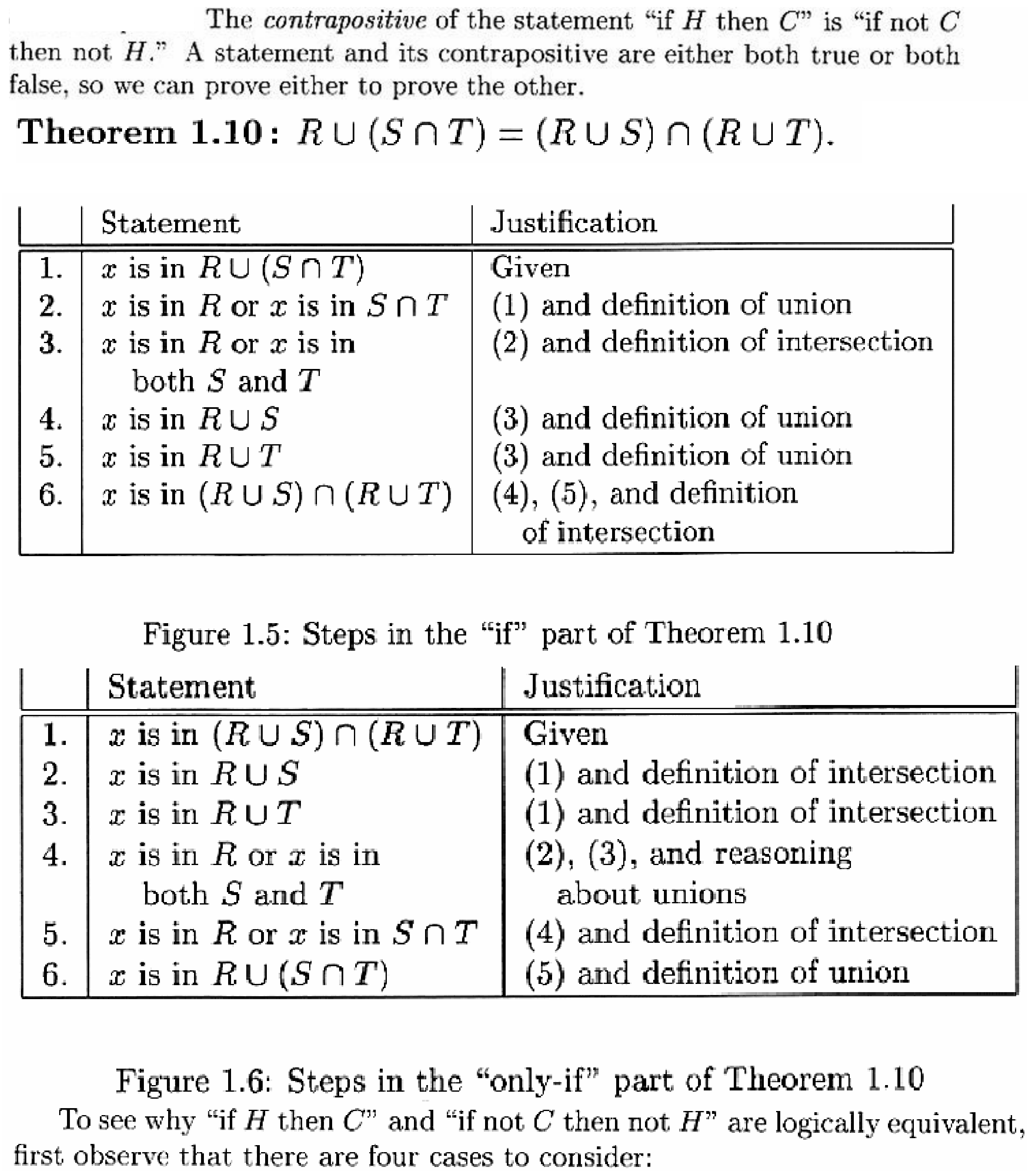
Deductive proof: Consists of sequence of statements whose truth lead us from some initial statement called the hypothesis or the give statement to a conclusion statement.


The theorem that is proved when we go from a hypothesis H to a conclusion C is the statement "if H then C." We say that C is *deduced* from H.

Additional forms of proof:
Proof of sets
Proof by contradiction
Proof by counter example

Direct proof (AKA) Constructive proof:
If *p* is true then *q* is true
Eg: if a and b are odd numbers then product is also an odd number.
Odd number can be represented as 2n+1
a=2x+1, b=2y+1
product of a X b = (2x+1) X (2y+1)
= 2(2xy+x+y)+1 = 2z+1 (odd number)

Proof by contrapositive:



The *contrapositive* of the statement "if H then C" is "if not C then not H." A statement and its contrapositive are either both true or both false, so we can prove either to prove the other.

Theorem 1.10: $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$.

	Statement	Justification
1.	x is in $R \cup (S \cap T)$	Given
2.	x is in R or x is in $S \cap T$	(1) and definition of union
3.	x is in R or x is in both S and T	(2) and definition of intersection
4.	x is in $R \cup S$	(3) and definition of union
5.	x is in $R \cup T$	(3) and definition of union
6.	x is in $(R \cup S) \cap (R \cup T)$	(4), (5), and definition of intersection

Figure 1.5: Steps in the "if" part of Theorem 1.10

	Statement	Justification
1.	x is in $(R \cup S) \cap (R \cup T)$	Given
2.	x is in $R \cup S$	(1) and definition of intersection
3.	x is in $R \cup T$	(1) and definition of intersection
4.	x is in R or x is in both S and T	(2), (3), and reasoning about unions
5.	x is in R or x is in $S \cap T$	(4) and definition of intersection
6.	x is in $R \cup (S \cap T)$	(5) and definition of union

Figure 1.6: Steps in the "only-if" part of Theorem 1.10

To see why "if H then C" and "if not C then not H" are logically equivalent, first observe that there are four cases to consider:

1. H and C both true.
2. H true and C false.
3. C true and H false.
4. H and C both false.

Proof by Contradiction:

H and not C implies falsehood.

That is, start by assuming both the hypothesis H and the negation of the conclusion C. Complete the proof by showing that something known to be false follows logically from H and not C. This form of proof is called *proof by contradiction.*

It often is easier to prove that a statement is not a theorem than to prove it *is* a theorem. As we mentioned, if S is any statement, then the statement "S is not a theorem" is itself a statement without parameters, and thus can

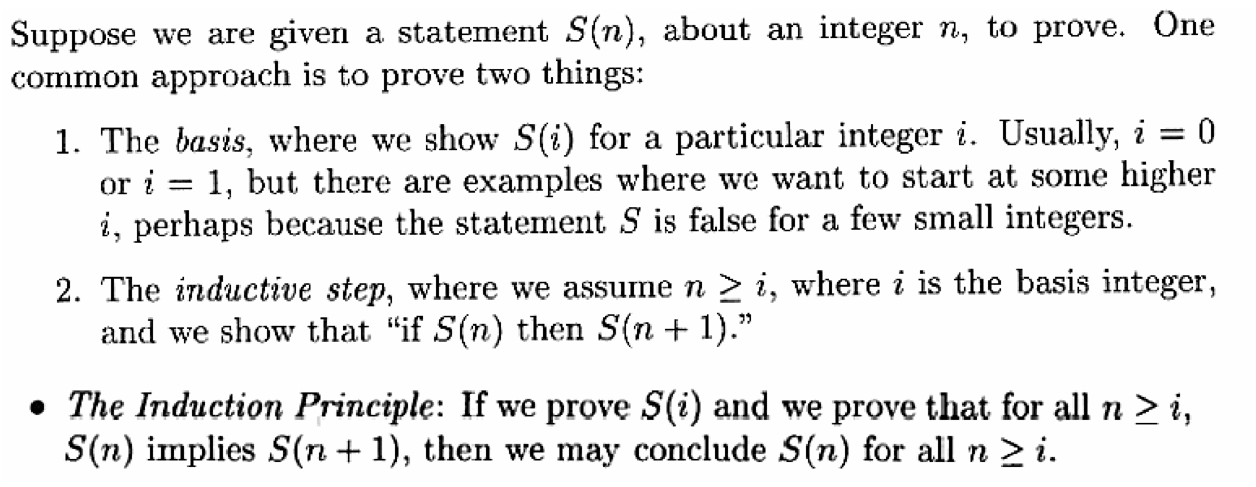
Be regarded as an observation than a theorem.

Alleged Theorem 1.13: All primes are odd. (More formally, we might say: if integer x is a prime, then x is odd.)

DISPROOF: The integer 2 is a prime, but 2 is even. □

For any sets a,b,c if a∩b = Φ and c is a subset of b the prove that a∩c =Φ
Given : a∩b=Φ and c subset b
Assume: a∩c $\neq$ Φ
Then $\forall x,\ x\epsilon a \text{ and } x\epsilon c \Rightarrow x\epsilon b$
= > a∩b $\neq$Φ = > a∩c=Φ(i.e., the assumption is wrong)

Proof by mathematical Induction:



Suppose we are given a statement $S(n)$, about an integer n, to prove. One common approach is to prove two things:

1. The *basis*, where we show $S(i)$ for a particular integer i. Usually, $i = 0$ or $i = 1$, but there are examples where we want to start at some higher i, perhaps because the statement S is false for a few small integers.

2. The *inductive step*, where we assume $n \geq i$, where i is the basis integer, and we show that "if $S(n)$ then $S(n+1)$."

- ***The Induction Principle*: If we prove $S(i)$ and we prove that for all $n \geq i$, $S(n)$ implies $S(n+1)$, then we may conclude $S(n)$ for all $n \geq i$.**

Languages :

The languages we consider for our discussion is an abstraction of natural languages. That is, our focus here is on formal languages that need precise and formal definitions. Programming languages belong to this category.

Symbols :

Symbols are indivisible objects or entity that cannot be defined. That is, symbols are the atoms of the world of languages. A symbol is any single object such as ♠, *a*, 0, 1, #, begin, or do.

Alphabets :

An alphabet is a finite, nonempty set of symbols. The alphabet of a language is normally denoted by Σ. When more than one alphabets are considered for discussion, then subscripts may be used (e.g. Σ_1, Σ_2 etc) or sometimes other symbol like G may also be introduced.

Example :

$$\Sigma = \{0,\ 1\}$$

$$\Sigma = \{a,\ b,\ c\}$$

$$\Sigma = \{a,\ b,\ c,\ \&,\ z\}$$

$$\Sigma = \{\#,\ \nabla,\ \spadesuit,\ \beta\}$$

Strings or Words over Alphabet :

A string or word over an alphabet Σ is a finite sequence of concatenated symbols of Σ.

Example : 0110, 11, 001 are three strings over the binary alphabet { 0, 1 } .

aab, abcb, b, cc are four strings over the alphabet { a, b, c }.

It is not the case that a string over some alphabet should contain all the symbols from the alphabet. For example, the string cc over the alphabet { a, b, c } does not contain the symbols a and b. Hence, it is true that a string over an alphabet is also a string over any superset of that alphabet.

Length of a string :
The number of symbols in a string w is called its length, denoted by |w|.

Example : | 011 | = 4, |11| = 2, | b | = 1

Convention : We will use small case letters towards the beginning of the English alphabet to denote symbols of an alphabet and small case letters towards the end to denote strings over an alphabet. That is, $a, b, c \in \Sigma$ (symbols) and u, v, w, x, y, z are strings.

Some String Operations :
Let $x = a_1a_2a_3 \in a_n$ and $y = b_1b_2b_3 \in b_m$ be two strings. The concatenation of x and y denoted by xy, is the string $a_1a_2a_3 \cdots a_nb_1b_2b_3 \cdots b_m$. That is, the concatenation of x and y denoted by xy is the string that has a copy of x followed by a copy of y without any intervening space between them.

Example : Consider the string 011 over the binary alphabet. All the prefixes, suffixes and substrings of this string are listed below.

Prefixes: ε, 0, 01, 011.
Suffixes: ε, 1, 11, 011.
Substrings: ε, 0, 1, 01, 11, 011.

Note that x is a prefix (suffix or substring) to x, for any string x and ε is a prefix (suffix or substring) to any string.

A string x is a proper prefix (suffix) of string y if x is a prefix (suffix) of y and x $\neq$ y.

In the above example, all prefixes except 011 are proper prefixes.

Powers of Strings : For any string x and integer $n \geq 0$, we use x^n to denote the string formed by sequentially concatenating n copies of x. We can also give an inductive definition of x^n as follows:
x^n = e, if n = 0 ; otherwise $x^n = xx^{n-1}$

Example : If $x = 011$, then $x^3 = 011011011$, $x^1 = 011$ and $x^0 = e$

Powers of Alphabets :

We write Σ^k (for some integer k) to denote the set of strings of length k with symbols from Σ. In other words,

Σ^k = { w | w is a string over Σ and | w | = k}. Hence, for any alphabet, Σ^0 denotes the set of all strings of length zero. That is, Σ^0 = { e }. For the binary alphabet { 0, 1 } we have the following.

$\Sigma^0 = \{e\}.$

$\Sigma^1 = \{0, 1\}.$

$\Sigma^2 = \{00, 01, 10, 11\}.$

$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

The set of all strings over an alphabet Σ is denoted by Σ^*. That is,

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \Sigma^n \cup \cdots$$
$$= \cup \Sigma^k$$

The set Σ^* contains all the strings that can be generated by iteratively concatenating symbols from Σ any number of times.

Example : If Σ = { a, b }, then Σ^* = { ε, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, …}.

Please note that if $\Sigma = F$, then Σ^* that is $\phi^* = \{e\}$. It may look odd that one can proceed from the empty set to a non-empty set by iterated concatenation. But there is a reason for this and we accept this convention

The set of all nonempty strings over an alphabet Σ is denoted by Σ^*. That is,

$$\Sigma^+ = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \Sigma^n \cup \cdots$$
$$= \cup \Sigma^k$$

Note that Σ^* is infinite. It contains no infinite strings but strings of arbitrary lengths.

Reversal :

For any string $w = a_1 a_2 a_3 \cdots a_n$ the reversal of the string is $w^R = a_n a_{n-1} \cdots a_3 a_2 a_1$.

An inductive definition of reversal can be given as follows:

Languages :
A language over an alphabet is a set of strings over that alphabet. Therefore, a language L is any subset of Σ^*. That is, any $L \subseteq \Sigma^*$ is a language.

Example :

1. F is the empty language.
2. Σ^* is a language for any Σ.
3. {e} is a language for any Σ. Note that, $\phi \neq \{e\}$. Because the language F does not contain any string but {e} contains one string of length zero.
4. The set of all strings over { 0, 1 } containing equal number of 0's and 1's.
5. The set of all strings over {a, b, c} that starts with a.

Convention : Capital letters A, B, C, L, etc. with or without subscripts are normally used to denote languages.

Set operations on languages : Since languages are set of strings we can apply set operations to languages. Here are some simple examples (though there is nothing new in it).

Union : A string $x \in L_1 \cup L_2$ iff $x \in L_1$ or $x \in L_2$

Example : { 0, 11, 01, 011 } $\cup$ { 1, 01, 110 } = { 0, 11, 01, 011, 111 }

Intersection : A string, $x \in L_1 \cap L_2$ iff $x \in L_1$ and $x \in L_2$.

Example : { 0, 11, 01, 011 } $\cap$ { 1, 01, 110 } = { 01 }

Complement : Usually, Σ^* is the universe that a complement is taken with respect to.
Thus for a language L, the complement is L(bar) = { $x \in \Sigma^*$ | $x \notin L$ }.

Example : Let L = { x | |x| is even }. Then its complement is the language { $x \in \Sigma^*$ | |x| is odd }.
Similarly we can define other usual set operations on languages like relative complement, symmetric difference, etc.

Reversal of a language :
The reversal of a language L, denoted as L^R, is defined as: $L^R = \{w^R \mid w \in L\}$.

Example :

1. Let L = { 0, 11, 01, 011 }. Then L^R = { 0, 11, 10, 110 }.

2. Let L = { 1^n0^n | n is an integer }. Then L^R = { 1^n0^n | n is an integer }.

Language concatenation : The concatenation of languages L_1 and L_2 is defined as $L_1 L_2$ = { *xy* | $x \in L_1$ and $y \in L_2$ }.

Example : { *a*, *ab* }{ *b*, *ba* } = { *ab*, *aba*, *abb*, *abba* }.

Note that ,

1. $L_1 L_2 \neq L_2 L_1$ in general.
2. $L\Phi = \Phi$
3. $L\{\varepsilon\} = L = \{\varepsilon\}$

Iterated concatenation of languages : Since we can concatenate two languages, we also repeat this to concatenate any number of languages. Or we can concatenate a language with itself any number of times. The operation L^n denotes the concatenation of L with itself n times. This is defined formally as follows:

$L_0 = \{e\}$
$L^n = LL^{n-1}$

Example : Let L = { a, ab }. Then according to the definition, we have

$L_0 = \{e\}$
$L_1 = L\{e\} = L = \{a, ab\}$
$L_2 = L\,L_1 = \{a, ab\}\{a, ab\} = \{aa, aab, aba, abab\}$
$L_3 = L\,L_2 = \{a, ab\}\{aa, aab, aba, abab\}$
$= \{aaa, aaab, aaba, aabab, abaa, abaab, ababa, ababab\}$

and so on.

Kleene's Star operation : The Kleene star operation on a language L, denoted as L^* is defined as follows :

L^* = (Union n in N) L^n

$= L^0 \cup L^1 \cup L^2 \cup \cdots$

= { x | x is the concatenation of zero or more strings from L }

Thus L^* is the set of all strings derivable by any number of concatenations of strings in L. It is also useful to define

$L^+ = LL^*$, i.e., all strings derivable by one or more concatenations of strings in L. That is

L^+ = (Union n in N and n >0) L^n
$= L^1 \cup L^2 \cup L^3 \cup \cdots$

Example : Let L = { a, ab }. Then we have,

$L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$

$= \{e\} \cup \{a, ab\} \cup \{aa, aab, aba, abab\} \cup \ldots$

$L^+ = L^1 \cup L^2 \cup L^3 \cup \cdots$

$= \{a, ab\} \cup \{aa, aab, aba, abab\} \cup \ldots$

Note : ε is in L^*, for every language L, including .

The previously introduced definition of Σ^* is an instance of Kleene star.

$$\begin{tikzcd}
\text{Grammar} \arrow[r, "\text{(Generates)}"] & \text{Language} & \text{Automata} \arrow[l, "\text{(Recognizes)}"']
\end{tikzcd}$$

Automata: A algorithm or program that automatically recognizes if a particular string belongs to the language or not, by checking the grammar of the string.

An automata is an abstract computing device (or machine). There are different varities of such abstract machines (also called models of computation) which can be defined mathematically.

Every Automaton fulfills the three basic requirements.

- Every automaton consists of some essential features as in real computers. It has a mechanism for reading input. The input is assumed to be a sequence of symbols over a given alphabet and is placed on an input tape(or written on an input file). The simpler automata can only read the input one symbol at a time from left to right but not change. Powerful versions can both read (from left to right or right to left) and change the input.

- The automaton can produce output of some form. If the output in response to an input string is binary (say, accept or reject), then it is called an accepter. If it produces an output sequence in response to an input sequence, then it is called a transducer(or automaton with output).
- The automaton may have a temporary storage, consisting of an unlimited number of cells, each capable of holding a symbol from an alphabet (whcih may be different from the input alphabet). The automaton can both read and change the contents of the storage cells in the temporary storage. The accusing capability of this storage varies depending on the type of the storage.
- The most important feature of the automaton is its control unit, which can be in any one of a finite number of interval states at any point. It can change state in some defined manner determined by a transition function.

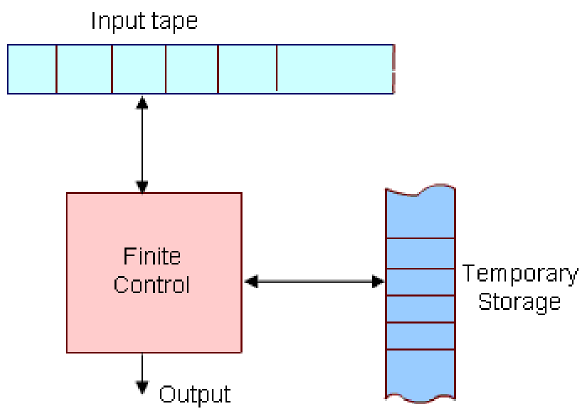


Figure 1: The figure above shows a diagrammatic representation of a generic automation.

Operation of the automation is defined as follows.
At any point of time the automaton is in some integral state and is reading a particular symbol from the input tape by using the mechanism for reading input. In the next time step the automaton then moves to some other integral (or remain in the same state) as defined by the transition function. The transition function is based on the current state, input symbol read, and the content of the temporary storage. At the same time the content of the storage may be changed and the input read may be modifed. The automation may also produce some output during this transition. The internal state, input and the content of storage at any point defines the configuration of the automaton at that point. The transition from one configuration to the next (as defined by the transition function) is called a *move*. Finite state machine or *Finite Automation* is the simplest type of abstract machine we consider. Any system that is at any point of time in one of a finite number of interval state and moves among these states in a defined manner in response to some input, can be modeled by a finite automaton. It doesnot have any temporary storage and hence a restricted model of computation.

Finite Automata

Automata (singular : automation) are a particularly simple, but useful, model of computation. They were initially proposed as a simple model for the behavior of neurons.

States, Transitions and Finite-State Transition System :

Let us first give some intuitive idea about *a state of a system* and *state transitions* before describing finite automata.

Informally, *a state of a system* is an instantaneous description of that system which gives all relevant information necessary to determine how the system can evolve from that point on.

Transitions are changes of states that can occur spontaneously or in response to inputs to the states. Though transitions usually take time, we assume that state transitions are instantaneous (which is an abstraction).

Some examples of state transition systems are: digital systems, vending machines, etc. A system

containing only a finite number of states and transitions among them is called a *finite-state transition system*.

Finite-state transition systems can be modeled abstractly by a mathematical model called *finite automation*

Deterministic Finite (-state) Automata

Informally, a DFA (Deterministic Finite State Automaton) is a simple machine that reads an input string -- one symbol at a time -- and then, after the input has been completely read, decides whether to accept or reject the input. As the symbols are read from the tape, the automaton can change its state, to reflect how it reacts to what it has seen so far. A machine for which a deterministic code can be formulated, and if there is only one unique way to formulate the code, then the machine is called deterministic finite automata.

Thus, a DFA conceptually consists of 3 parts:

1. A *tape* to hold the input string. The tape is divided into a finite number of cells. Each cell holds a symbol from Σ.
2. A *tape head* for reading symbols from the tape
3. A *control* , which itself consists of 3 things:
    - finite number of states that the machine is allowed to be in (zero or more states are designated as *accept* or *final* states),
    - a current state, initially set to a start state,

- a state transition function for changing the current state.

An automaton processes a string on the tape by repeating the following actions until the tape head has traversed the entire string:

1. The tape head reads the current tape cell and sends the symbol s found there to the control. Then the tape head moves to the next cell.
2. he control takes s and the current state and consults the state transition function to get the next state, which becomes the new current state.

Once the entire string has been processed, the state in which the automation enters is examined.

If it is an accept state , the input string is accepted ; otherwise, the string is rejected . Summarizing all the above we can formulate the following formal definition:

Deterministic Finite State Automaton : A Deterministic Finite State Automaton (DFA) is a 5-tuple : $M = (Q, \Sigma , \delta, q_0, F)$

- *Q* is a finite set of states.
- Σ is a finite set of input symbols or alphabet
- $\delta : Q \times \Sigma \rightarrow Q$ is the "next state" transition function (which is total). Intuitively, is δ a function that tells which state to move to in response to an input, i.e., if M is in state q and sees input a, it moves to state $\delta(q,a)$.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set of accept or final states.

Acceptance of Strings :

A DFA accepts a string $w = a_1 a_2 \cdots a_n$ if there is a sequence of states $q_0, q_1, \cdots, q_n$ in *Q* such that

1. q_0 is the start state.
2. $\delta(q_i, q_{i+1}) = a_{i+1}$ for all $0 < i < n$.
3. $q_n \in F$

Language Accepted or Recognized by a DFA :

The language accepted or recognized by a DFA M is the set of all strings accepted by M , and is denoted by $L(M)$ i.e. $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$. The notion of acceptance can also be made more precise by extending the transition function δ.

Extended transition function :

Extend $\delta : Q \times \Sigma \to Q$ (which is function on symbols) to a function on strings, i.e. .
$\hat{\delta} : Q \times \Sigma^* \to Q$

That is, $\hat{\delta}(q, w)$ is the state the automation reaches when it starts from the state q and finish processing the string w. Formally, we can give an inductive definition as follows:
The language of the DFA M is the set of strings that can take the start state to one of the accepting states i.e.

$$L(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$$

$$= \{ w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F \}$$

Example 1 :

$M = (Q, \Sigma, \delta, q_0, F)$

$Q = \{q_0, q_1\}$

q_0 is the start state

$F = \{q_1\}$

$\delta(q_0, 0) = q_0 \qquad \delta(q_1, 0) = q_1$

$\delta(q_0, 1) = q_1 \qquad \delta(q_1, 1) = q_1$

It is a formal description of a DFA. But it is hard to comprehend. For ex. The language of the DFA is any string over { 0, 1} having at least one 1

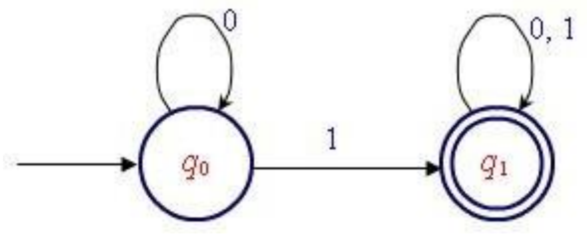
We can describe the same DFA by transition table or state transition diagram as following:

Transition Table :

	0	1
$\to q_0$	q_0	q_1

$*q_1$	q_1	q_1

It is easy to comprehend the transition diagram.



Explanation : We cannot reach find state q_1 w/0 or in the i/p string. There can be any no. of 0's at the beginning. (The self-loop at q_0 on label 0 indicates it). Similarly there can be any no. of 0's & 1's in any order at the end of the string.

Transition table :

It is basically a tabular representation of the transition function that takes two arguments (a state and a symbol) and returns a value (the "next state").

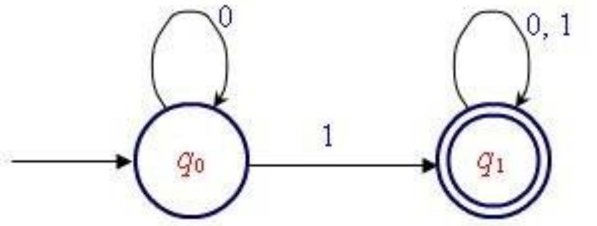
- Rows correspond to states,
- Columns correspond to input symbols,
- Entries correspond to next states
- The start state is marked with an arrow
- The accept states are marked with a star (*).

	0	1
$\rightarrow q_0$	q_0	q_1
$*q_1$	q_1	q_1

(State) Transition diagram :

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

1. For each state in Q there is a node.
2. There is a directed edge from node q to node p labeled a iff $\delta(q, a) = p$. (If there are several input symbols that cause a transition, the edge is labeled by the list of these symbols.)
3. There is an arrow with no source into the start state.
4. Accepting states are indicated by double circle.

5. 

6. Here is an informal description how a DFA operates. An input to a DFA can be any string $w \in \Sigma^*$ Put a pointer to the start state q. Read the input string w from left to right, one symbol at a time, moving the pointer according to the transition function, δ. If the next symbol of w is a and the pointer is on state p, move the pointer to $\delta(p, a)$. When the end of the input string w is encountered, the pointer is on some state, r. The string is said to be accepted by the DFA if $r \in F$ and rejected if $r \notin F$. Note that there is no formal mechanism for moving the pointer.
7. A language $L \in \Sigma^*$ is said to be regular if L = L(M) for some DFA M.

Regular Expressions: Formal Definition

We construct REs from primitive constituents (basic elements) by repeatedly applying certain recursive rules as given below. (In the definition)

Definition : Let S be an alphabet. The regular expressions are defined recursively as follows.

Basis :

i) ϕ is a RE

ii) $\in$ is a RE

iii) $\forall\ a \in S$, a is RE.

These are called primitive regular expression i.e. Primitive Constituents

Recursive Step :

If r_1 and r_2 are REs over, then so are

i) $r_1 + r_2$

ii) $r_1 r_2$

iii) r_1^*

iv) (r_1)

Closure : r is RE over only if it can be obtained from the basis elements (Primitive REs) by a finite no of applications of the recursive step (given in 2).

Example : Let Σ = { 0,1,2 }. Then (0+21)*(1+ F) is a RE, because we can construct this expression by applying the above rules as given in the following step.

Steps	**RE Constructed**	**Rule Used**
1	1	Rule 1(iii)
2	ϕ	Rule 1(i)
3	1+ϕ	Rule 2(i) & Results of Step 1, 2
4	(1+ϕ)	Rule 2(iv) & Step 3
5	2	1(iii)
6	1	1(iii)
7	21	2(ii), 5, 6
8	0	1(iii)
9	0+21	2(i), 7, 8
10	(0+21)	2(iv), 9
11	(0+21)*	2(iii), 10
12	(0+21)*	2(ii), 4, 11

Language described by REs : Each describes a language (or a language is associated with every RE). We will see later that REs are used to attribute regular languages.

Notation : If r is a RE over some alphabet then L(r) is the language associate with r . We can define the language L(r) associated with (or described by) a REs as follows.

1. ϕ is the RE describing the empty language i.e. $L(\phi) = \phi$.

2. $\in$ is a RE describing the language {$\in$} i.e. $L(\in) = \{\in\}$.

3. $\forall a \in S$, a is a RE denoting the language {a} i.e **.** L(a) = {a} .

4. If r_1 and r_2 are REs denoting language $L(r_1)$ and $L(r_2)$ respectively, then

i) $r_1 + r_2$ is a regular expression denoting the language $L(r_1 + r_2) = L(r_1) \cup L(r_2)$

ii) r_1r_2 is a regular expression denoting the language $L(r_1r_2)=L(r_1)\,L(r_2)$

iii) r_1^* is a regular expression denoting the language $L(r_1^*) = (L(r1))^*$

iv) (r_1) is a regular expression denoting the language $L((r_1)) = L(r_1)$

Example : Consider the RE (0*(0+1)). Thus the language denoted by the RE is

L(0*(0+1)) = L(0*) L(0+1)...................... by 4(ii)

= L(0)*L(0) ∪ L(1)

= {∈ , 0,00,000,. } {0}   {1}∪

= {∈ , 0,00,000,........} {0,1}

= {0, 00, 000, 0000,..........,1, 01, 001, 0001,...............}

Precedence Rule

Consider the RE $ab + c$. The language described by the RE can be thought of either $L(a)L(b+c)$ or $L(ab)$ $L(c)$ as provided by the rules (of languages described by REs) given already. But these two represents two different languages lending to ambiguity. To remove this ambiguity we can either

1) Use fully parenthesized expression- (cumbersome) or

2) Use a set of precedence rules to evaluate the options of REs in some order. Like other algebras mod in mathematics.

For REs, the order of precedence for the operators is as follows:

i) The star operator precedes concatenation and concatenation precedes union (+) operator.

ii) It is also important to note that concatenation & union (+) operators are associative and union operation is commutative.

Using these precedence rule, we find that the RE ab+c represents the language $L(ab)$
   $L(c)$ i.e. it should be grouped as ((ab)+c).
∪
We can, of course change the order of precedence by using parentheses. For example, the language represented by the RE $a(b+c)$ is $L(a)L(b+c)$.

Example : The RE ab^*+b is grouped as $((a(b^*))+b)$ which describes the language $L(a)(L(b))^* \cup L(b)$

Example : The RE $(ab)^*+b$ represents the language $(L(a)L(b))^* \cup L(b)$.

Example : It is easy to see that the RE (0+1)*(0+11) represents the language of all strings over {0,1} which are either ended with 0 or 11.

Example : The regular expression r =(00)*(11)*1 denotes the set of all strings with an even number of 0's followed by an odd number of 1's i.e. $L(r) = \{0^{2n}1^{2m+1} \mid n \geq 0, m \geq 0\}$

Note : The notation r^+ is used to represent the RE rr^*. Similarly, r^2 represents the RE rr, r^2 denotes r^2r, and so on.

An arbitrary string over Σ = {0,1} is denoted as (0+1)*.

Exercise : Give a RE r over {0,1} s.t. $L(r)$={ $\omega \in \Sigma^* \mid \omega$ has at least one pair of consecutive 1's}

Solution : Every string in $L(r)$ must contain 00 somewhere, but what comes before and what goes before is completely arbitrary. Considering these observations we can write the REs as (0+1)*11(0+1)*.

Example : Considering the above example it becomes clean that the RE (0+1)*11(0+1)*+(0+1)*00(0+1)* represents the set of string over {0,1} that contains the substring 11 or 00.

Example : Consider the RE 0*10*10*. It is not difficult to see that this RE describes the set of strings over {0,1} that contains exactly two 1's. The presence of two 1's in the RE and any no of 0's before, between and after the 1's ensure it.

Example : Consider the language of strings over {0,1} containing two or more 1's.

Solution : There must be at least two 1's in the RE somewhere and what comes before, between, and after is completely arbitrary. Hence we can write the RE as (0+1)*1(0+1)*1(0+1)*. But following two REs also represent the same language, each ensuring presence of least two 1's somewhere in the string

i) 0*10*1(0+1)*

ii) (0+1)*10*10*

Example : Consider a RE r over {0,1} such that

$L(r)$ = {$\omega \in \{0,1\}^* \mid \omega$ has no pair of consecutive 1's}

Solution : Though it looks similar to ex ……., it is harder to construct to construct. We observer that, whenever a 1 occurs, it must be immediately followed by a 0. This substring may be preceded & followed by any no of 0's. So the final RE must be a repetition of strings of the form: 00…0100….00 i.e. 0*100*. So it looks like the RE is (0*100*)*. But in this case the strings ending in 1 or consisting of all 0's are not accounted for. Taking these observations into consideration, the final RE is r = (0*100*)(1+ $\in$)+0*(1+$\in$).

Alternative Solution :

The language can be viewed as repetitions of the strings 0 and 01. Hence get the RE as r = (0+10)*(1+$\in$).This is a shorter expression but represents the same language.

Regular Expression and Regular Language :

Equivalence(of REs) with FA :

Recall that, language that is accepted by some FAs are known as Regular language. The two concepts : REs and Regular language are essentially same i.e. (for) every regular language can be developed by (there is) a RE, and for every RE there is a Regular Langauge. This fact is rather suprising, because RE approach to describing language is fundamentally differnet from the FA approach. But REs and FA are equivalent in their descriptive power. We can put this fact in the focus of the following Theorem.

Theorem : A language is regular iff some RE describes it.

This Theorem has two directions, and are stated & proved below as a separate lemma

RE to FA :

REs denote regular languages :

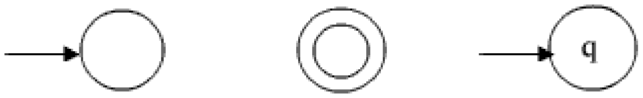
Lemma : If $L(r)$ is a language described by the RE r, then it is regular i.e. there is a FA such that $L(M) \cong L(r)$.

Proof : To prove the lemma, we apply structured index on the expression r. First, we show how to construct FA for the basis elements: ϕ, $\in$ and for any $a \in \Sigma$. Then we show how to combine these Finite Automata into Complex Automata that accept the Union, Concatenation, Kleen Closure of the languages accepted by the original smaller automata.

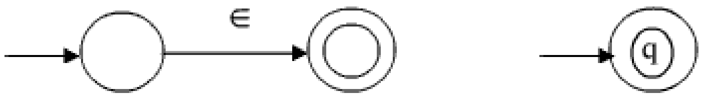
Use of NFAs is helpful in the case i.e. we construct NFAs for every REs which are represented by transition diagram only.

Basis :

- Case (i) **:** $r = \phi$. Then $L(r) = \phi$. Then $L(r) = \phi$ and the following NFA N recognizes $L(r)$. Formally $N = (Q, \{q\}, \Sigma, \delta, q, F, \phi)$ where $Q = \{q\}$ and $\delta(q,a) = \phi \; \forall \; a \in S, \; F = \phi$.

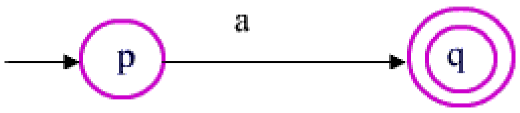


- Case (ii) **:** $r = \in$. $L(r) = \{\in\}$, and the following NFA N accepts L(r). Formally $N = (\{q\}, \Sigma, \delta, q, \{q\})$ where $\delta(q,a) = \phi \quad \forall a \in \Sigma$.



Since the start state is also the accept step, and there is no any transition defined, it will accept the only string $\in$ and nothing else.

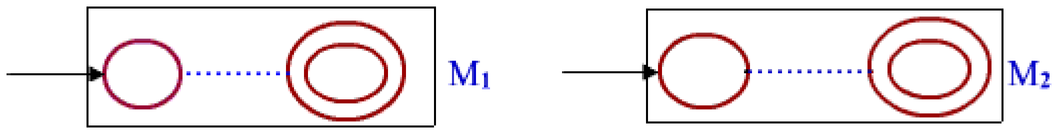
- Case (iii) : $r = a$ for some $a \in \Sigma$. Then $L(r) = \{a\}$, and the following NFA N accepts $L(r)$.



Formally, $N = (\{p,q\}, \Sigma, \delta, p, \{q\})$ where $\delta(p,q) = \{q\}, \; \delta(s, b) = \{\phi\}$ for $s \neq p$ or $b \neq a$

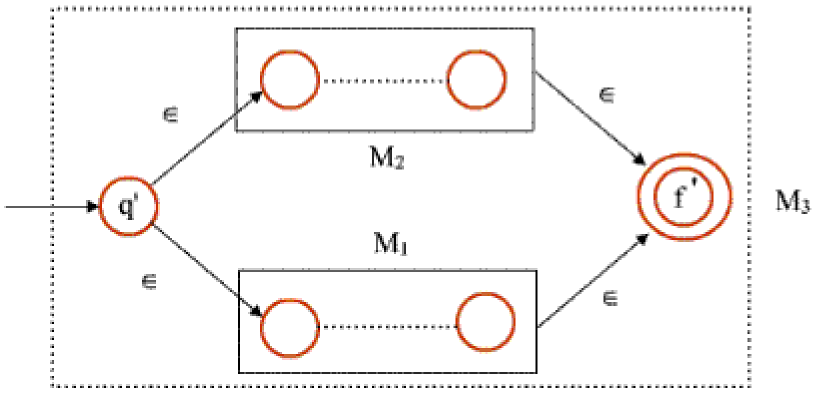
Induction :

Assume that the start of the theorem is true for REs r_1 and r_2. Hence we can assume that we have automata M_1 and M_2 that accepts languages denoted by REs r_1 and r_2, respectively i.e. $L(M_1) = L(r_1)$ and $L(M_2) = L(r_2)$. The FAs are represented schematically as shown below.



Each has an initial state and a final state. There are four cases to consider.

- Case (i) **:** Consider the RE $r_3 = r_1 + r_2$ denoting the language $L(r_1) \cup L(r_2)$. We construct FA M_3, from M_1 and M_2 to accept the language denoted by RE r_3 as follows **:**



Create a new (initial) start state q' and give $\in$- transition to the initial state of M_1 and M_2.This is the initial state of M_3.

- Create a final state f' and give $\in$-transition from the two final state of M_1 and M_2. f' is the only final state of M_3 and final state of M_1 and M_2 will be ordinary states in M_3.
- All the state of M_1 and M_2 are also state of M_3.

- All the moves of M_1 and M_2 are also moves of M_3. [Formal Construction]

It is easy to prove that $L(M_3) = L(r_3)$

Proof: To show that $L(M_3) = L(r_3)$ we must show that

= $L(r_1) \cup L(r_2)$

= $L(M_1) = L(M_2)$ by following transition of M_3

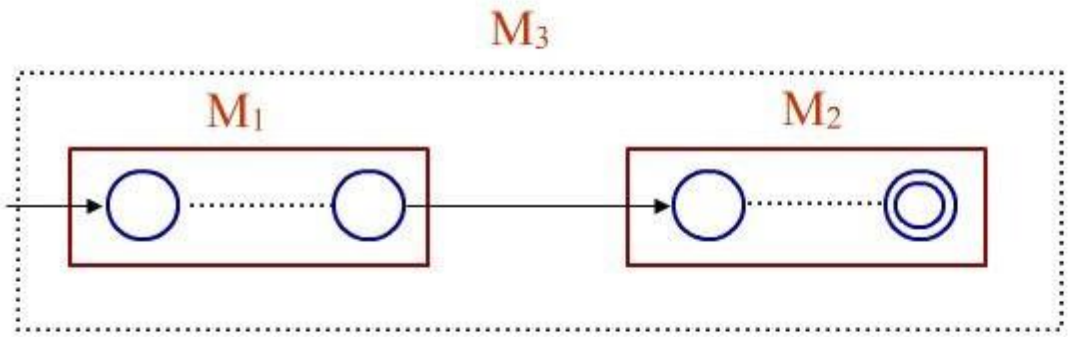
Starts at initial state q' and enters the start state of either M_1 or M_2 follwoing the transition i.e. without consuming any input. WLOG, assume that, it enters the start state of M_1. From this point onward it has to follow only the transition of M_1 to enter the final state of M_1, because this is the only way to enter the final state of M by following the e-transition.(Which is the last transition & no input is taken at hte transition). Hence the whole input w is considered while traversing from the start state of M_1 to the final state of M_1. Therefore M_1 must accept w_3.

Say, $w \in L(M_1)$ or $w \in L(M_2)$.

WLOG, say $w \in L(M_1)$

Therefore when M_1 process the string w , it starts at the initial state and enters the final state when w consumed totally, by following its transition. Then M_3 also accepts w, by starting at state q' and taking $\in$-transition enters the start state of M_1-follows the moves of M_1 to enter the final state of M_1 consuming input w thus takes $\in$-transition to f'.
Hence proved

- Case(ii) **:** Consider the RE $r_3 = r_1 r_2$ denoting the language $L(r_1)L(r_2)$. We construct FA M_3 from M_1 & M_2 to accept $L(r_3)$ as follows **:**



Create a new start state q' and a new final state

1. Add $\in$- transition from
   - q' to the start state of M_1
   - q' to f'
   - final state of M_1 to the start state of M_1
2. All the states of M_1 are also the states of M_3. M_3 has 2 more states than that of M_1 namely q' and f'.
3. All the moves of M_1 are also included in M_3.

By the transition of type (b), M_3 can accept $\in$.
By the transition of type (a), M_3 can enters the initial state of M_1 w/o any input and then follow all kinds moves of M_1 to enter the final state of M_1 and then following $\in$-transition can enter f'. Hence if any $w \in \Sigma^*$ is accepted by M_1 then w is also accepted by M_3. By the transition of type (b), strings accepted by M_1 can be repeated by any no of times & thus accepted by M_3. Hence M_3 accepts $\in$ and any string accepted by M_1 repeated (i.e. concatenated) any no of times. Hence $L(M_3) = (L(M_1))^* = (L(r)_1)^* = r_1^*$

Case(iv) **:** Let r_3 =(r_1). Then the FA M_1 is also the FA for (r_1), since the use of parentheses does not change the language denoted by the expression

Non-Deterministic Finite Automata
Nondeterminism is an important abstraction in computer science. Importance of nondeterminism is found in the design of algorithms. For examples, there are many problems with efficient nondeterministic solutions but no known efficient deterministic solutions. (Travelling salesman, Hamiltonean cycle, clique, etc). Behaviour of a process is in a distributed system is also a good example of nondeterministic situation. Because

the behaviour of a process might depend on some messages from other processes that might arrive at arbitrary times with arbitrary contents.
It is easy to construct and comprehend an NFA than DFA for a given regular language. The concept of NFA can also be used in proving many theorems and results. Hence, it plays an important role in this subject.
In the context of FA nondeterminism can be incorporated naturally. That is, an NFA is defined in the same way as the DFA but with the following two exceptions:

- multiple next state.
- $\in$- transitions.

Multiple Next State :

- In contrast to a DFA, the next state is not necessarily uniquely determined by the current state and input symbol in case of an NFA. (Recall that, in a DFA there is exactly one start state and exactly one transition out of every state for each symbol in Σ).
- This means that - in a state q and with input symbol a - there could be one, more than one or zero next state to go, i.e. the value of $\delta(q,a)$ is a subset of Q. Thus $\delta(q,a) = \{q_1, q_2, \cdots, q_k\}$ which means that any one of $q_1, q_2, \cdots, q_k$ could be the next state.
- The zero next state case is a special one giving $\delta(q,a) = \phi$, which means that there is no next state on input symbol when the automata is in state q. In such a case, we may think that the automata "hangs" and the input will be rejected.

$\in$- transitions :

In an -transition, the tape head doesn't do anything- it doesnot read and it doesnot move. However, the state of the automata can be changed - that is can go to zero, one or more states. This is written formally as $\delta(q,\in) = \{q_1, q_2, \cdots, q_k\}$ implying that the next state could by any one of $q_1, q_2, \cdots, q_k$ w/o consuming the next input symbol.

Acceptance :

Informally, an NFA is said to accept its input ω if it is possible to start in some start state and process ω, moving according to the transition rules and making choices along the way whenever the next state is not uniquely defined, such that when ω is completely processed (i.e. end of ω is reached), the automata is in an accept state. There may be several possible paths through the automation in response to an input ω since the start state is not determined and there are choices along the way because of multiple next states. Some of these paths may lead to accpet states while others may not. The automation is said to accept ω if at least one computation path on input ω starting from at least one start state leads to an accept state- otherwise, the automation rejects input ω. Alternatively, we can say that, ω is accepted iff there exists a path with label ω from some start state to some accept state. Since there is no mechanism for determining which state to start in or which of the possible next moves to take (including the ω -transitions) in response to an input symbol we can think that the automation is having some "guessing" power to chose the correct one in case the input is accepted

Example 1 : Consider the language $L = \{\omega \in \{0, 1\}^* \mid$ The 3rd symbol from the right is 1$\}$. The following four-state automation accepts L.

The m/c is not deterministic since there are two transitions from state q_1 on input 1 and no transition (zero transition) from q_0 on both 0 & 1.

For any string ω whose 3rd symbol from the right is a 1, there exists a sequence of legal transitions leading from the start state q, to the accept state q_4. But for any string ω where 3rd symbol from the right is 0, there is no possible sequence of legal tranisitons leading from q_1 and q_4. Hence m/c accepts L. How does it accept any string $\omega \in L$?

Formal definition of NFA :

Formally, an NFA is a quituple $M = (Q, \Sigma, \delta, q_0, F)$ where Q, Σ, q_0, and F bear the same meaning as for a DFA, but δ, the transition function is redefined as follows:

$$\delta : Q \times (\Sigma \cup \{\in\}) \to P(Q)$$

where $P(Q)$ is the power set of Q i.e. 2^Q.

The Langauge of an NFA :

From the discussion of the acceptance by an NFA, we can give the formal definition of a language accepted by an NFA as follows :

If $N = (Q, \Sigma, \delta, q_0, F)$ is an NFA, then the langauge accepted by N is writtten as $L(N)$ is given by $L(N) = \{\omega \mid \hat{\delta}(q_0, \omega) \cap F = \phi\}$.

That is, $L(N)$ is the set of all strings w in Σ^* such that $\hat{\delta}(q_0, \omega)$ contains at least one accepting state.

Removing ϵ-transition:
∈- transitions do not increase the power of an *NFA* . That is, any ∈- *NFA* (*NFA* with ∈transition), we can always construct an equivalent *NFA* without ∈-transitions. The equivalent *NFA* must keep track where the ∈*NFA* goes at every step during computation. This can be done by adding extra transitions for removal of every ∈- transitions from the ∈- *NFA* as follows.

If we removed the ∈- transition $\delta(p,\in)=q$ from the ∈- *NFA* , then we need to moves from state *p* to all the state r on input symbol $q\in\Sigma$ which are reachable from state q (in the ∈- *NFA*) on same input symbol *q*. This will allow the modified *NFA* to move from state *p* to all states on some input symbols which were possible in case of ∈-*NFA* on the same input symbol. This process is stated formally in the following theories.

Theorem if *L* is accepted by an ∈- *NFA N* , then there is some equivalent $NFA\ \ N'$ without ∈transitions accepting the same language *L*
Proof:

Let $N=(Q,\Sigma,\delta,q_0,F)$ be the given $\in -NFA$ with

We construct $N'=(Q,\Sigma,\delta',q_0,F')$

Where, $\delta'(q,a)=\{p \mid p\in\hat{\delta}(q,a)\}$ for all $q\in Q,$ and $a\in\Sigma,$ and

$$F'=\begin{cases} F \\ F \end{cases} U\{q_0\} \text{ if } \hat{\delta}(q_0,\in)\cap F\neq\phi \text{ otherwise.}$$

Other elements of *N'* and *N*

We can show that $L(N)=L(N')$ i.e. *N'* and *N* are equivalent.

We need to prove that $\forall w\in\Sigma^*$

$w\in L(N)\ \text{ iff }\ w\in L(N')$ i.e.

$$\forall w\in\Sigma^*\quad \hat{\delta}'(q_0,w)\in F'\ \text{ iff }\ \hat{\delta}(q_0,w)\in F$$

We will show something more, that is,

$$\forall w\in\Sigma^*\quad \hat{\delta}'(q_0,w)=\hat{\delta}(q_0,w)$$

We will show something more, that is, $|w|$

Basis : $|w| = 1$, then $x = a \in \Sigma$

But $\hat{\delta}'(q_0, a) = \hat{\delta}(q_0, a)$ by definition of δ'.

Induction hypothesis Let the statement hold for all $w \in \Sigma^*$ with $|w| \leq n$.

$\hat{\delta}'(q_0, w) = \hat{\delta}'(q_0, xa)$ By definition of extension of $\hat{\delta}'$

$= \delta'\left(\hat{\delta}'(q_0, x), a\right)$ By inductions hypothesis.

$= \delta'\left(\hat{\delta}(q_0, x), a\right)$ Assuming that

$= \delta'(R, a)$ $\hat{\delta}(q_0, x) = R, \text{where } R \subseteq Q$

$= \bigcup_{p \in R} \delta'(p, a)$ By definition of δ'

$= \bigcup_{p \in R} \hat{\delta}(p, a)$

$= \hat{\delta}(q_0, xa)$ Since $R = \hat{\delta}(q_0, x)$

$= \hat{\delta}(q_0, w)$

To complete the proof we consider the case

When $|w| = 0$ i.e. $w = \in$ then

$\delta'(q_0,\in)=\{q_0\}$ and by the construction of F', $q_0 \in F'$ wherever $\hat{\delta}(q_0,\in)$ constrains a state in *F*.

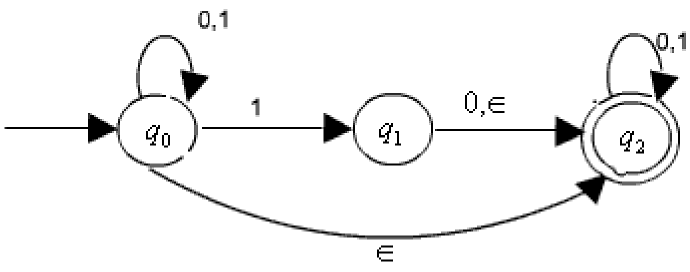
If $F' = F$ (and thus $\hat{\delta}(q_0,\in)$ is not in *F*), then $\forall w$ with $|w|=1$, w leads to an accepting state in *N′* iff it lead
to an accepting state in *N* (by the construction of *N′* and *N*).

Also, if ($w=\in$, thus *w* is accepted by N' iff *w* is accepted by *N* (iff $q_0 \in F$)

If $F' = F \bigcup \{q_0\}$ (and, thus in *M* we load $\hat{\delta}(q_0,\in)$ in *F*), thus $\in$ is accepted by both *N′* and *N* .

Let $|w| \geq 1$. If *w* cannot lead to q_0 in *N* , then $w \in L(N)$. (Since can add $\in$ transitions to get an accept state). So there is no harm in making q_0 an accept state in *N′*.

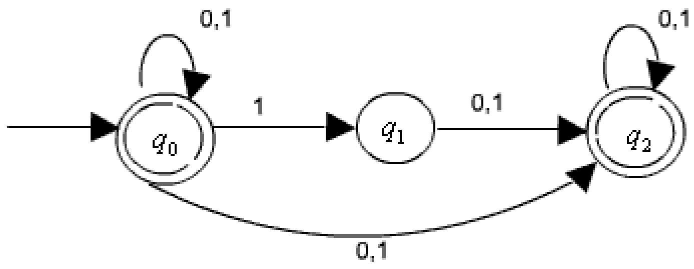
Ex: Consider the following *NFA* with $\in$- transition.



Transition Diagram δ

	0	1	$\in$
$\rightarrow q_0$	$\{q_0\}$	$\{q_0,q_1\}$	$\{q_2\}$
q_1	$\{q_2\}$	ϕ	$\{q_2\}$
$F\ \ q_2$	$\{q_2\}$	ϕ	$\{q_2\}$

Transition diagram for δ ' for the equivalent *NFA* without $\in$- moves



	0	1
$\xrightarrow[F]{} q_0$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$
q_1	$\{q_2\}$	$\{q_2\}$
$F \quad q_2$	$\{q_2\}$	$\{q_2\}$

Since $\delta(q_0, \in) = q_2 \in -NFA$ the start state q_0 must be final state in the equivalent *NFA* .

Since $\delta(q_0, \in) = q_2$ and $\delta(q_2, 0) = q_2$ and $\delta(q_2, 1) = q_2$ we add moves $\delta(q_0, 0) = q_2$ and $\delta(q_0, 1) = q_2$ in the equivalent *NFA* . Other moves are also constructed accordingly.

$\in$-closures:

The concept used in the above construction can be made more formal by defining the $\in$-closure for a state (or a set of states). The idea of $\in$-closure is that, when moving from a state *p* to a state *q* (or from a set of states S_i to a set of states S_j) an input $a \in \Sigma$, we need to take account of all $\in$-moves that could be made after the transition. Formally, for a given state *q*,

$\in$-closures: $(q) = \{p \mid p \text{ can be reached from } q \text{ by zero or more } \in \text{-moves}\}$

Similarly, for a given set $R \subseteq Q$

$\in$-closures:
$(R) = \{p \in Q \mid p \text{ can be reached from any } q \in R \text{ by following zero or more } \in \text{-moves}\}$

So, in the construction of equivalent *NFA N'* without $\in$-transition from any *NFA* with $\in$ moves. the first rule can now be written as $\delta'(q, a) = \in \text{-closure}(\delta(q, a))$

Equivalence of *NFA* and *DFA*

It is worth noting that a *DFA* is a special type of *NFA* and hence the class of languages accepted by *DFA* s is a subset of the class of languages accepted by *NFA* s. Surprisingly, these two classes are in fact equal. *NFA* s appeared to have more power than *DFA* s because of generality enjoyed in terms of $\in$-transition and multiple next states. But they are no more powerful than *DFA* s in terms of the languages they accept.

Converting *DFA* to *NFA*

Theorem: Every *DFA* has as equivalent *NFA*

Proof: A *DFA* is just a special type of an *NFA* . In a *DFA* , the transition functions is defined from $Q \times \Sigma \text{ to } Q$ whereas in case of an *NFA* it is defined from $Q \times \Sigma \text{ to } 2^Q$ and $D = (Q, \Sigma, \delta, q_0, F)$ be *a DFA* . We construct an equivalent *NFA* $N = (Q', \Sigma, \delta', q_0, F)$ as follows.

$\{q_i\} \in Q', \forall q_i \in Q$

$\delta'(\{p\}, a) = \{\delta(p, a)\},$ i. e

If $\delta(p, a) = q,$ and $\delta'(\{p\}, a) = \{q\}.$

All other elements of *N* are as in *D*.

If $w = a_1 a_2 \cdots, a_n \in L(D)$ then there is a sequence of states $q_0, q_1, q_2 \cdots, q_n$ such that $\delta(q_{i-1}, a_i) = q_i \text{ and } q_n \in F$

Then it is clear from the above construction of *N* that there is a sequence of states (in *N*) $\{q_0\}, \{q_1\}, \{q_2\}, \cdots, \{q_n\}$ such that $\delta'(\{q_{i-1}\}, a_i) = \{q_i\}$ and $\{q_n\} \in F$ and hence $w \in L(N).$

Similarly we can show the converse.

Hence , $L(N) = L(D)$

Given any *NFA* we need to construct as equivalent *DFA* i.e. the *DFA* need to simulate the behaviour of the *NFA* . For this, the *DFA* have to keep track of all the states where the NFA could be in at every step during processing a given input string.

There are 2^n possible subsets of states for any *NFA* with *n* states. Every subset corresponds to one of the possibilities that the equivalent *DFA* must keep track of. Thus, the equivalent *DFA* will have 2^n states.

The formal constructions of an equivalent *DFA* for any *NFA* is given below. We first consider an *NFA* without $\in$ transitions and then we incorporate the affects of $\in$ transitions later.

Formal construction of an equivalent *DFA* for a given *NFA* without $\in$ transitions.

Given an $N = (Q, \Sigma, \delta, q_0, F)$ without $\in$- moves, we construct an equivalent *DFA*

$D = \left(Q^D, \Sigma, \delta^D, q_0^D, F^D\right)$ as follows

$Q^D = P(Q)$ i.e. $Q^D = \{S \mid S \subseteq Q\}$,

$q_0^D = \{q_0\}$,

$F^D = \left\{q^D \in Q^D \mid q^D \cap F \neq \phi\right\}$ (i.e. every subset of *Q* which as an element in *F* is considered as a final stat
in *DFA D*)

$\delta^D\left(\{q_1, q_2, \cdots, q_k\}, a\right) = \delta(q_1, a) \cup \delta(q_2, a) \cup \cdots \cup \delta(q_k, a)$

for all $a \in \Sigma$ and $q^D = \{q_1, q_2, \cdots, q_k\}$

where $q_i \in Q, \; 1 \leq i \leq k.$

That is, $\delta^D\left(q^D, a\right) = \bigcup_{q_i \in q^D} \delta(q_i, a)$

To show that this construction works we need to show that *L(D)=L(N)* i.e.

$\forall w \in \Sigma^* \quad \hat{\delta}^D\left(q_0^D, w\right) \in F^D \text{ iff } \hat{\delta}(q_0, w) \cap F \neq \phi$

Or, $\forall w \in \Sigma^* \quad \hat{\delta}^D\left(\{q_0\}, w\right) \cap F \neq \text{ iff } \hat{\delta}(q_0, w) \cap F \neq \phi$

We will prove the following which is a stranger statement thus required.

$$\forall w \in \Sigma^*,\ \hat{\delta}^D(\{q_0\}, w) = \hat{\delta}(q_0, w)$$

Proof : We will show by inductions on $|w|$

Basis If $|w|$=0, then *w* =$\in$

So, $\delta^D(\{q_0\},\in) = \{q_0\} = \delta(q_0,\in),$ by definition.

Inductions hypothesis : Assume inductively that the statement holds $\forall w \in \Sigma^*$ of length less than or equal to *n*.

Inductive step

Let $|w| = n+1$, then $w = xa$ with $|x| = n \text{ and } a \in \Sigma.$

Now,

$$\begin{aligned}
\hat{\delta}^D(\{q_0\}, w) &= \hat{\delta}^D(\{q_0\}, xa) \\
&= \delta^D\left(\hat{\delta}^D(\{q_0\}, x), a\right), \text{ by inductive extension of } \delta^D \\
&= \delta^D\left(\hat{\delta}(q_0, x), a\right), \text{by induction hypothesis} \\
&= \bigcup_{q_i \in \delta(q_0, x)} \delta(q_i, a), \text{ by definition of } \delta^D \\
&= \delta(q_0, xa) \quad \text{by definition of } \hat{\delta} \text{ (extension of } \delta) \\
&= \delta(q_0, w)
\end{aligned}$$

Now, given any *NFA* with $\in$-transition, we can first construct an equivalent *NFA* without $\in$-transition and then use the above construction process to construct an equivalent *DFA* , thus, proving the equivalence of *NFA* s and *DFA* s..

It is also possible to construct an equivalent *DFA* directly from any given *NFA* with $\in$-transition by integrating the concept of $\in$-closure in the above construction.

Recall that, for any $S \subseteq Q,$

$\in$- closure :
$(S) = \{q \in Q \mid q \text{ can be reached from any } p \in S \text{ by following zero or more} \in -\text{transitions}\}$

In the equivalent *DFA* , at every step, we need to modify the transition functions δ^D to keep track of all the states where the *NFA* can go on $\in$-transitions. This is done by replacing $\delta(q,a)$ by $\in$-closure $(\delta(q,a))$, i.e. we now compute $\delta^D(q^D,a)$ at every step as follows:

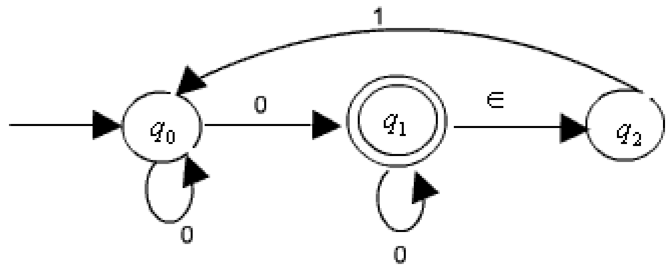
$$\delta^D\left(q^D,a\right)=\left\{q\in Q \,\middle|\, q\in \ \in\text{-closure}\left(\delta\left(q^D,a\right)\right)\right\}.$$

Besides this the initial state of the *DFA D* has to be modified to keep track of all the states that can be reached from the initial state of *NFA* on zero or more -transitions.

This can be done by changing the initial state q_0^D to $\in$-closure (q_0^D) .
It is clear that, at every step in the processing of an input string by the *DFA D* , it enters a state that corresponds to the subset of states that the *NFA N* could be in at that particular point. This has been proved in the constructions of an equivalent NFA for any $\in$-*NFA*

If the number of states in the *NFA* is *n* , then there are 2^n states in the *DFA* . That is, each state in the *DFA* is a subset of state of the *NFA* .

But, it is important to note that most of these 2^n states are inaccessible from the start state and hence can be removed from the *DFA* without changing the accepted language. Thus, in fact, the number of states in the equivalent *DFA* would be much less than 2^n .
Example : Consider the NFA given below.



	0	1	$\in$
$\rightarrow q_0$	$\{q_0,q_1\}$	ϕ	ϕ
$F\ q_1$	$\{q_1\}$	ϕ	$\{q_2\}$
q_0	ϕ	ϕ	$\{q_0\}$

Since there are 3 states in the NFA

There will be $2^3 = 8$ states (representing all possible subset of states) in the equivalent *DFA* . The transition table of the *DFA* constructed by using the subset constructions process is produced here.

	0	1
ϕ	ϕ	ϕ
$\rightarrow q_0$	$\{q_0, q_1, q_2\}$	ϕ
$F\{q_1\}$	$\{q_1, q_2\}$	$\{q_0\}$
$\{q_0\}$	ϕ	$\{q_0\}$
$F\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
$F\{q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_0\}$
$F\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$

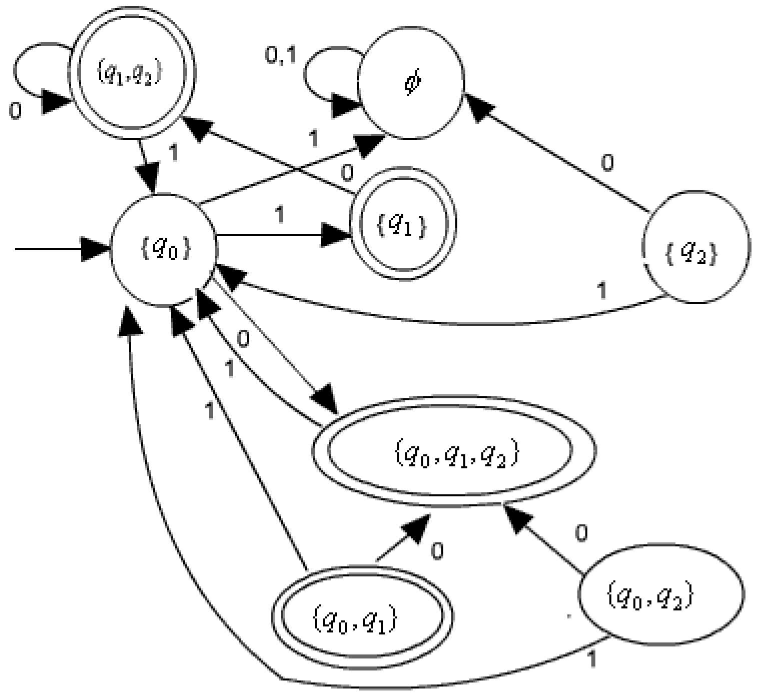
The start state of the *DFA* is $\in$- closures $(q_0) = \{q_0\}$

The final states are all those subsets that contains q_1 (since $q_1 \in F$ in the *NFA*).

Let us compute one entry,

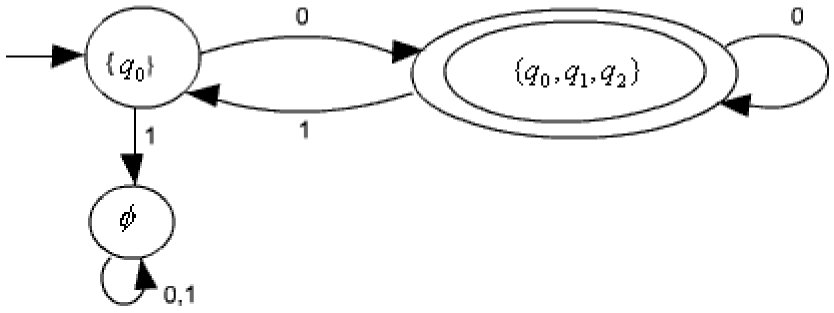
$$\begin{aligned}\delta^D(\{q_0, 0\}) &= \in -\text{closure}(\delta(q_0, 0)) \\ &= \in -\text{closure}(\{q_0, q_1\}) \\ &= \{q_0, q_1, q_2\}\end{aligned}$$

Similarly, all other transitions can be computed



	0	1
$\rightarrow\{q_0\}$	$\{q_0, q_1, q_2\}$	ϕ
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$

Corresponding Transition fig. for DFA. Note that states $\{q_1\}, \{q_2\}, \{q_1 q_2\}, \{q_0, q_2\}$ and $\{q_0, q_1\}$ are not accessible and hence can be removed. This gives us the following simplified *DFA* with only 3 states.



It is interesting to note that we can avoid encountering all those inaccessible or unnecessary states in the equivalent *DFA* by performing the following two steps inductively.

1. If q_0 is the start state of the NFA, then make $\in$- closure (q_0) the start state of the equivalent *DFA* . This is definitely the only accessible state.
2. If we have already computed a set δ of states which are accessible. Then $\forall a \in \Sigma$. compute $\left(\delta^D\left(S,a\right)\right)$ because these set of states will also be accessible.