THEORY OF COMPUTATION LECTURE NOTES

BCS 303 **<u>THEORY OF COMPUTATION (3-1-0)</u>** **Cr.-4**

Module – I **(10 Lectures)**

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

<u>Finite Automata</u>: 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

<u>Nondeterministic Finite Automata</u>: 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 – II **(10 Lectures)**

<u>Regular Expressions and Languages</u>: Regular Expressions: The Operators of regular Expressions, Building Regular Expressions, Precedence of Regular-Expression Operators, Precedence of Regular-Expression Operators

Finite Automata and Regular Expressions: From DFA's to Regular Expressions, ConvertingDFA's to Regular Expressions, Converting DFA's to Regular Expressions by Eliminating States, Converting Regular Expressions to Automata.

Algebraic Laws for Regular Expressions:

<u>Properties of Regular Languages</u>: The Pumping Lemma for Regular Languages, Applications of the Pumping Lemma Closure Properties of Regular Languages, Decision Properties of Regular Languages, Equivalence and Minimization of Automata,

<u>Context-Free Grammars and Languages</u>: Definition of Context-Free Grammars, Derivations Using a Grammars Leftmost and Rightmost Derivations, The Languages of a Grammar,

Parse Trees: Constructing Parse Trees, The Yield of a Parse Tree, Inference Derivations, and Parse Trees, From Inferences to Trees, From Trees to Derivations, From Derivation to Recursive Inferences,

Applications of Context-Free Grammars: Parsers, Ambiguity in Grammars and Languages: Ambiguous Grammars, Removing Ambiguity From Grammars, Leftmost Derivations as a Way to Express Ambiguity, Inherent Anbiguity

Module – III **(10 Lectures)**

Pushdown Automata: Definition Formal Definition of Pushdown Automata, A Graphical Notation for PDA's, Instantaneous Descriptions of a PDA,
Languages of PDA: Acceptance by Final State, Acceptance by Empty Stack, From Empty Stack to Final State, From Final State to Empty Stack
Equivalence of PDA's and CFG's: From Grammars to Pushdown Automata, From PDA's to Grammars
Deterministic Pushdown Automata: Definition of a Deterministic PDA, Regular Languages and Deterministic PDA's, DPDA's and Context-Free Languages, DPDA's and Ambiguous Grammars
Properties of Context-Free Languages: Normal Forms for Context-Free Grammars, The Pumping Lemma for Context-Free Languages, Closure Properties of Context-Free Languages, Decision Properties of CFL's

Module –IV (10 Lectures)

Introduction to Turing Machines: The Turing Machine: The Instantaneous Descriptions for Turing Machines, Transition Diagrams for Turing Machines, The Language of a Turing Machine, Turing Machines and Halting
Programming Techniques for Turing Machines, Extensions to the Basic Turing Machine, Restricted Turing Machines, Turing Machines and Computers,

Undecidability: A Language That is Not Recursively Enumerable, Enumerating the Binary Strings, Codes for Turing Machines, The Diagonalization Language
An Undecidable Problem That Is RE: Recursive Languages, Complements of Recursive and RE languages, The Universal Languages, Undecidability of the Universal Language
Undecidable Problems About Turing Machines: Reductions, Turing Machines That Accept the Empty Language. Post's Correspondence Problem: Definition of Post's Correspondence Problem, The "Modified" PCP, Other Undecidable Problems: Undecidability of Ambiguity for CFG's

Text Book:

1. Introduction to Automata Theory Languages, and Computation, by J.E.Hopcroft, R.Motwani & J.D.Ullman (3rd Edition) – Pearson Education
2. Theory of Computer Science (Automata Language & Computations), by K.L.Mishra & N. Chandrashekhar, PHI

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.

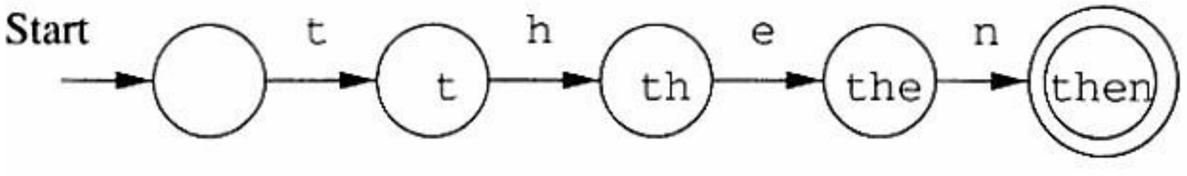


Figure 1.2: A finite automaton modeling recognition of **then**

Introduction to formal proof:

Basic Symbols used :

U – Union

∩- Conjunction

ϵ - Empty String

Φ – NULL set

7- negation

‘ – compliment

= > implies

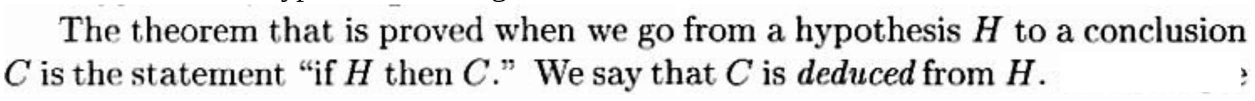
Additive inverse: a+(-a)=0
Multiplicative inverse: a*1/a=1
Universal set U={1,2,3,4,5}
Subset A={1,3}
A' ={2,4,5}
Absorption law: AU(A ∩B) = A, A∩(AUB) = A

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

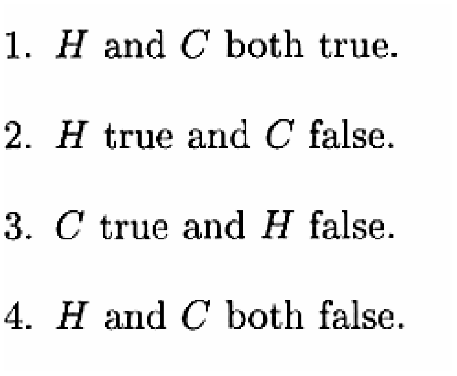
	Statement	Justification
1.	x is in $R \cup (S \cap T)$	Given
2.	x is in R or x is in $S \cap T$	(1) and definition of union
3.	x is in R or x is in both S and T	(2) and definition of intersection
4.	x is in $R \cup S$	(3) and definition of union
5.	x is in $R \cup T$	(3) and definition of union
6.	x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

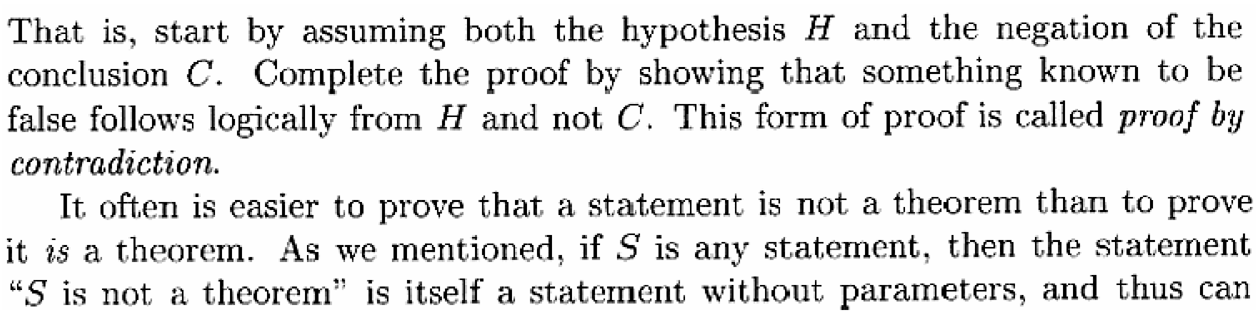
To see why "if H then C" and "if not C then not H" are logically equivalent, first observe that there are four cases to consider:



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:

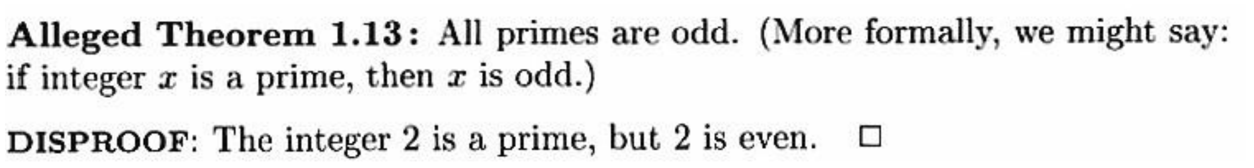
H and not C implies falsehood.



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.

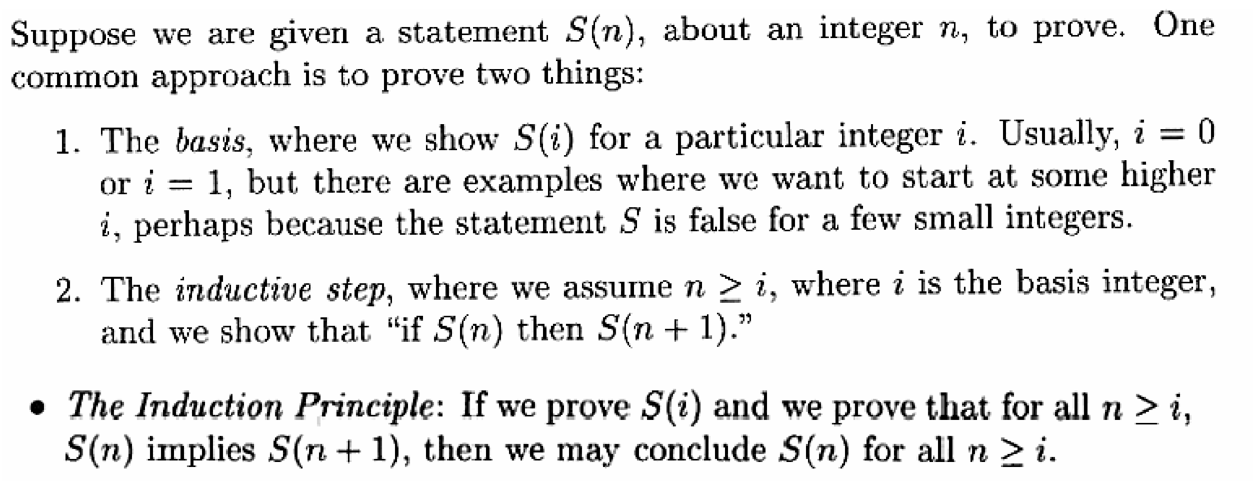


Alleged Theorem 1.13: All primes are odd. (More formally, we might say: if integer x is a prime, then x is odd.)

DISPROOF: The integer 2 is a prime, but 2 is even. $\square$

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 ≠ Φ
Then $\forall x,\ x\epsilon a \text{ and } x\epsilon c \Rightarrow x\epsilon b$
= > a∩b ≠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.

$$\Sigma = \{0,\ 1\}$$
$$\Sigma = \{a,\ b,\ c\}$$
$$\Sigma = \{a,\ b,\ c,\ \&,\ z\}$$

Example : $\Sigma = \{\#,\ \nabla,\ ♠,\ \beta\}$

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

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

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 de- fined manner determined by a transition function.

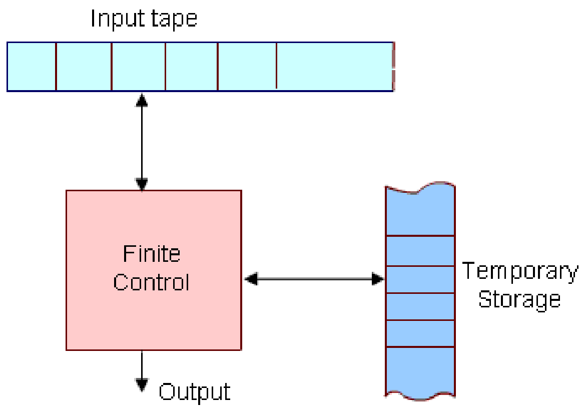


Figure 1: The figure above shows a diagrammatic representation of a generic automation.

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

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

$F = \{q_1\}$

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

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

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

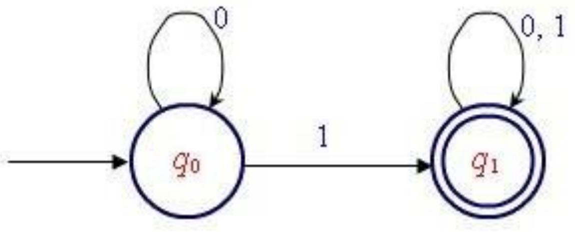
We can describe the same DFA by transition table or state transition diagram as following:

Transition Table :

	0	1
$\rightarrow q_0$	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").

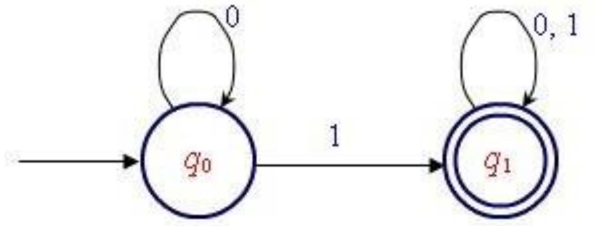
- Rows correspond to states,
- Columns correspond to input symbols,
- Entries correspond to next states
- The start state is marked with an arrow
- The accept states are marked with a star (*).

	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 s.tring $w \in \Sigma^*$ 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.

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}* | 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:

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

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

If we removed the $\in$- transition $\delta(p,\in)=q$ from the $\in$- *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 $\in$- *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 $\in$-*NFA* on the same input symbol. This process is stated formally in the following theories.

Theorem if *L* is accepted by an $\in$- *NFA N* , then there is some equivalent $NFA\ \ N'$ without $\in$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

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

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 $\forall w\in\Sigma^*$

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

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

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, $|w|$

<u>Basis :</u> $|w| = 1$, 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 $|w| \leq n$.

$$\begin{aligned}
\hat{\delta}'(q_0, w) &= \hat{\delta}'(q_0, xa) && \text{By definition of extension of } \hat{\delta}' \\
&= \delta'\left(\hat{\delta}'(q_0, x), a\right) && \text{By inductions hypothesis.} \\
&= \delta'\left(\hat{\delta}(q_0, x), a\right) && \text{Assuming that} \\
&= \delta'(R, a) && \hat{\delta}(q_0, x) = R, \text{where } R \subseteq Q \\
&= \bigcup_{p \in R} \delta'(p, a) && \\
&= \bigcup_{p \in R} \hat{\delta}(p, a) && \text{By definition of } \delta' \\
&= \hat{\delta}(q_0, xa) && \text{Since } R = \hat{\delta}(q_0, x) \\
&= \hat{\delta}(q_0, w)
\end{aligned}$$

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*.

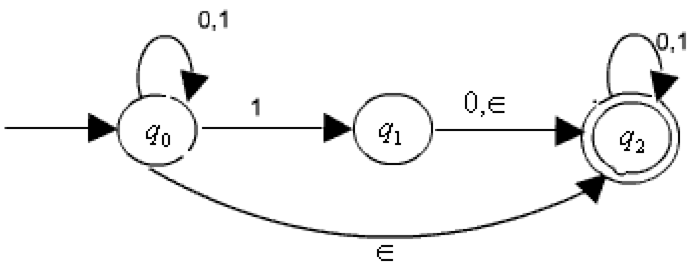
If $F' = F$ (and thus $\hat{\delta}(q_0,\in)$ is not in *F*), then $\forall w$ with $|w|=1$, w leads to an accepting state in *N′* iff it lead
to an accepting state in *N* (by the construction of *N′* and *N*).

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

If $F' = F \cup \{q_0\}$ (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′*.

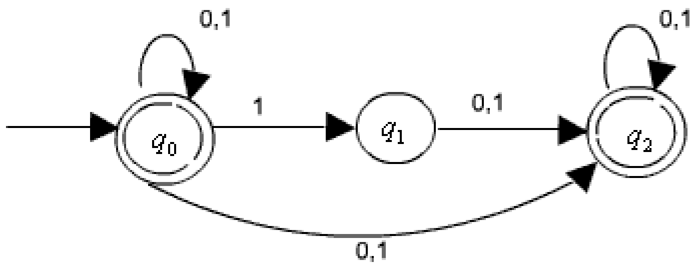
Ex: Consider the following *NFA* with $\in$- transition.



Transition Diagram δ

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

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

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

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

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

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

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

where $q_i \in Q,\ 1 \le i \le k.$

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

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

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

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

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

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

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 $|x| = n$ 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:

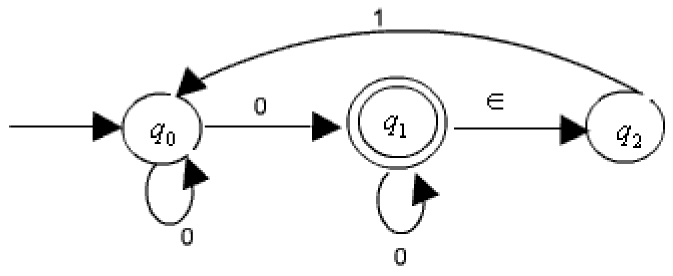
$$\delta^D\left(q^D,a\right)=\left\{q\in Q \middle| q\in\ \in-\text{closure}\left(\delta\left(q^D,a\right)\right)\right\}.$$

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	$\{q_0\}$
$F\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
$F\{q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_0\}$
$F\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$

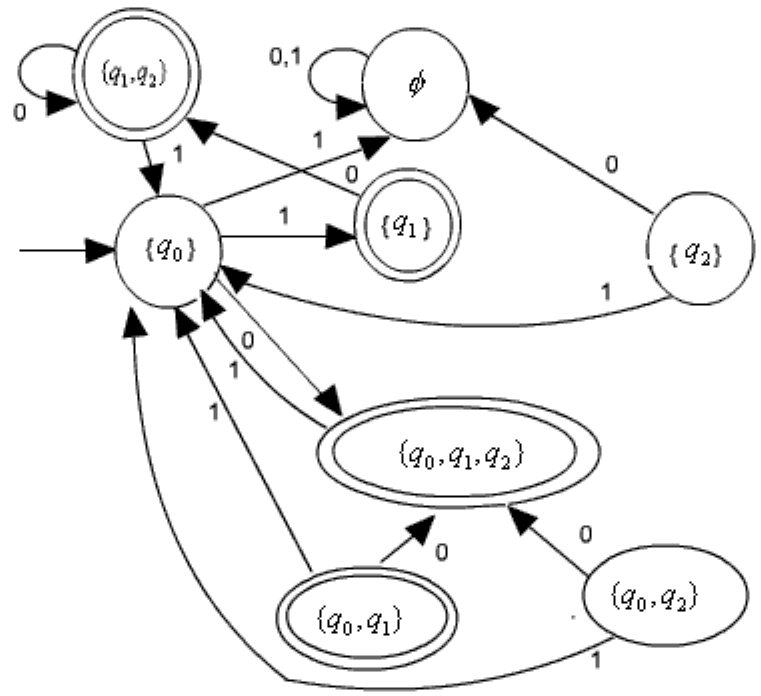
The start state of the *DFA* is $\in$- closures $(q_0) = \{q_0\}$

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

Let us compute one entry,

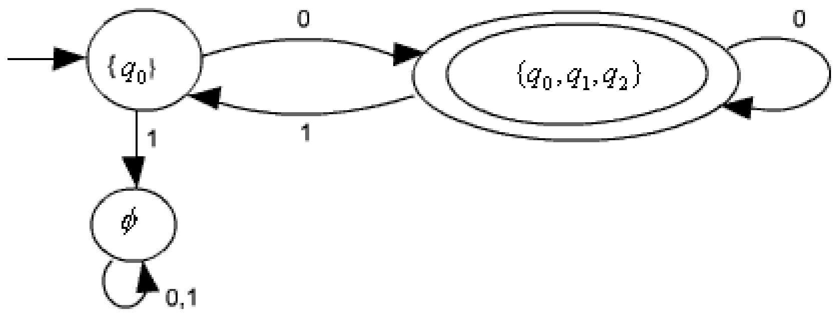
$$\begin{aligned}\delta^D(\{q_0, 0\}) &= \in -\text{closure}(\delta(q_0, 0)) \\ &= \in -\text{closure}(\{q_0, q_1\}) \\ &= \{q_0, q_1, q_2\}\end{aligned}$$

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 $\left(\delta^D(S,a)\right)$ because these set of states will also be accessible.

Following these steps in the above example, we get the transition table given below

MODULE-II

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	$1+\phi$	Rule 2(i) & Results of Step 1, 2

4	$(1+\phi)$	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. 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) $r_1 r_2$ is a regular expression denoting the language $L(r_1 r_2)=L(r_1)\,L(r_2)$

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) $\cup$ L(1)

= $\{\in$, 0,00,000,........} {0} $\cup$ {1}

= $\{\in$, 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 $L(ab) \cup L(c)$ 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 $L(ab) \cup L(c)$ 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^3 denotes $r^2 r$, 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:

FA to regular expressions:

FA to RE (REs for Regular Languages) :

Lemma : If a language is regular, then there is a RE to describe it. i.e. if $L = L(M)$ for some DFA M, then there is a RE r such that $L = L(r)$.

Proof : We need to construct a RE *r* such that $L(r) = \{ w \mid w \in L(M) \}$. Since M is a DFA, it has a finite no of states. Let the set of states of M is Q = {1, 2, 3,..., n} for some integer n. [Note **:** if the n states of M were denoted by some other symbols, we can always rename those to indicate as 1, 2, 3,..., n]. The required RE is constructed inductively.

Notations : $r_{ij}^{(k)}$ is a RE denoting the language which is the set of all strings w such that w is the label of a path from state i to state j $(1 \le i, j \le n)$ in M, and that path has no intermediate state whose number is greater then k. (i & j (begining and end pts) are not considered to be "intermediate" so i and /or j can be greater than k)

We now construct $r_{ij}^{(k)}$ inductively, for all $i, j \in Q$ starting at $k = 0$ and finally reaching $k = n$.

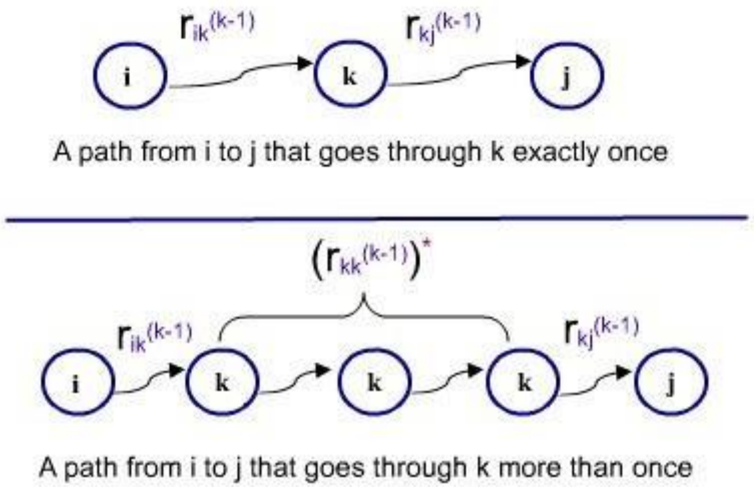
Basis : $k = 0$, $r_{ij}^{(0)}$ i.e. the paths must not have any intermediate state (since all states are numbered 1 or above). There are only two possible paths meeting the above condition **:**

1. A direct transition from state i to state j.
   - $r_{ij}^{(0)}$ = a if then is a transition from state i to state j on symbol the single symbol a.
   - $r_{ij}^{(0)} = a_1 + a_2 + \cdots + a_m$ if there are multiple transitions from state i to state j on symbols $a_1, a_2, \cdots, a_m$.
   - $r_{ij}^{(0)} = f$ if there is no transition at all from state i to state j.
2. All paths consisting of only one node i.e. when $i = j$. This gives the path of length 0 (i.e. the RE $\in$ denoting the string $\in$) and all self loops. By simply adding Î to various cases above we get the corresponding REs i.e.
   - $r_{ii}^{(0)} = \in + a$ if there is a self loop on symbol a in state i .
   - $r_{ii}^{(0)} = \in + a_1 + a_2 + \cdots + a_m$ if there are self loops in state i as multiple symbols $a_1, a_2, \cdots, a_m$.
   - $r_{ii}^{(0)} = \in$ if there is no self loop on state i.

Induction :

Assume that there exists a path from state i to state j such that there is no intermediate state whose number is greater than k. The corresponding Re for the label of the path is $r_{ij}^{(k)}$.
There are only two possible cases **:**

1. The path dose not go through the state k at all i.e. number of all the intermediate states are less than k. So, the label of the path from state i to state j is tha language described by the RE $r_{ij}^{(k-1)}$.
2. The path goes through the state k at least once. The path may go from i to j and k may appear more than once. We can break the into pieces as shown in the figure 7.



$r_{ik}^{(k-1)}$
$r_{kj}^{(k-1)}$
i
k
j
A path from i to j that goes through k exactly once
$(r_{kk}^{(k-1)})^*$
A path from i to j that goes through k more than once

Figure 7

1. The first part from the state i to the state k which is the first recurence. In this path, all intermediate states are less than k and it starts at iand ends at k. So the RE $r_{ik}^{(k-1)}$ denotes the language of the label of path.
2. The last part from the last occurence of the state k in the path to state j. In this path also, no intermediate state is numbered greater than k. Hence the RE $r_{kj}^{(k-1)}$ denoting the language of the label of the path.
3. In the middle, for the first occurence of k to the last occurence of k , represents a loop which may be taken zero times, once or any no of times. And all states between two consecutive k's are numbered less than k.

Hence the label of the path of the part is denoted by the RE $\left(r_{ij}^{(k-1)}\right)^*$ .The label of the path from state i to state j is the concatenation of these 3 parts which is

$r_{ik}^{(k-1)}\left(r_{kk}^{(k-1)}\right)^* r_{kj}^{(k-1)}$

Since either case 1 or case 2 may happen the labels of all paths from state i to j is denoted by the following RE

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} + r_{ik}^{(k-1)}\left(r_{kk}^{(k-1)}\right)^* r_{kj}^{(k-1)}$$

We can construct $r_{ij}^{(k)}$ for all $i, j \in \{1,2,..., n\}$ in increasing order of k starting with the basis $k = 0$ upto $k = n$ since $r_{ij}^{(k)}$ depends only on expressions with a small superscript (and hence will be available). WLOG, assume that state 1 is the start state and $j_1, j_2, \cdots, j_m$ are the m final states where $ji \in \{1, 2, ... , n\}$, $1 \leq i \leq m$ and $m \leq n$. According to the convention used, the language of the automatacan be denoted by the RE

$r_{1j_1}^{(n)} + r_{1j_2}^{(n)} + \cdots + r_{1j_m}^{(n)}$

Since $r_{1j_i}^{(n)}$ is the set of all strings that starts at start state 1 and finishes at final state j_i following the transition of the FA with any value of the intermediate state (1, 2, ... , *n*) and hence accepted by the automata.

Regular Grammar:

A *grammar* $G = (N, \Sigma, P, S)$ is right-linear if each production has one of the following three forms:

- $A \rightarrow cB$,
- $A \rightarrow c$,
- $A \rightarrow \in$

Where A, $B \in N'$ (with $A = B$ allowed) and $c \in \Sigma$. A *grammar G* is left-linear if each production has once of the following three forms.

$A \rightarrow Bc$, $A \rightarrow c$, $A \rightarrow \in$

A right or left-linear grammar is called a regular grammar.

Regular grammar and Finite Automata are equivalent as stated in the following theorem.

Theorem : A language L is regular iff it has a regular grammar. We use the following two lemmas to prove the above theorem.

Lemma 1 : If L is a regular language, then L is generated by some right-linear grammar.

Proof : Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA that accepts L.

Let $Q = \{q_0, q_1, \cdots, q_n\}$ and $\Sigma = \{a_1, a_2, \cdots, a_m\}$.

We construct the right-linear grammar $G = (N, \Sigma, P, S)$ by letting

$N = Q$, $S = q_0$ and $P = \{A \rightarrow cB \mid \delta(A, c) = B\} \cup \{A \rightarrow c \mid \delta(A, c) \in B\}$

[Note: If $B \in F$, then $B \rightarrow \in P$]

Let $w = a_1 a_2 \ldots a_k \in L(M)$. For M to accept *w*, there must be a sequence of states $q_0, q_1, \ldots, q_k$ such that

$\delta(q_0, a_1) = q_1$

$\delta(q_1, a_2) = q_2$

$\cdots$

$\delta(q_{k-1}, a_k) = q_k$

and $q_k \in F$

By construction, the grammar G will have one production for each of the above transitions. Therefore, we have the corresponding derivation.

$$S = q_0 \underset{G}{\Rightarrow} a_1 q_1 \underset{G}{\Rightarrow} a_1 a_2 q_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} a_1 a_2 \cdots a_k q_k \underset{G}{\Rightarrow} a_1 a_2 \cdots a_k = w$$

Hence $w \in L(g)$.

Conversely, if $w = a_1 a_2 \ldots a_k \in L(G)$, then the derivation of w in G must have the form as given above. But, then the construction of G from M implies that

$\delta(q_0, a_1 a_2 \cdots a_k) = q_k$, where $q_k \in F$, completing the proof.

Lemma 2 : Let $G = (N, \Sigma, P, S)$ be a right-linear grammar. Then $L(G)$ is a regular language.

Proof: To prove it, we construct a FA M from G to accept the same language.

$M = (Q, \Sigma, \delta, q_0, F)$ is constructed as follows:

$Q = N \cup \{q_f\}$ (q_f is a special sumbol not in N)

$q_0 = S$, $F = \{q_f\}$

For any $q \in N$ and $a \in \Sigma$ and δ is defined as

$\delta(q, a) = \{p \mid q \rightarrow ap \in P\}$ if $q \rightarrow a \notin P$

and $\delta(q, a) = \{p \mid q \rightarrow ap \in P\} \cup \{q_f\}$, if $q \rightarrow a \in P$.

We now show that this construction works.

Let $w = a_1 a_2 \ldots a_k \in L(G)$. Then there is a derivation of w in G of the form

$$S \underset{G}{\Rightarrow} a_1q_1 \underset{G}{\Rightarrow} a_1a_2q_2 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} a_1a_2 \cdots a_{k-1}a_k \left(= w\right)$$

By contradiction of M, there must be a sequence of transitions

$$\delta\left(q_0, a_1\right) = q_1$$
$$\delta\left(q_1, a_2\right) = q_2$$
$$\cdots$$
$$\delta\left(q_{k-1}, a_k\right) = q_f$$

implying that $w = a_1a_2 \ldots a_k \in L(M)$ i.e. w is accepted by M.

Conversely, if $w = a_1a_2 \cdots a_k$ is accepted by M, then because q_f is the only accepting state of M, the transitions causing w to be accepted by M will be of the form given above. These transitions corresponds to a derivationof w in the grammar G. Hence $w \in L\left(G\right)$, completing the proof of the lemma.

Given any left-linear grammar G with production of the form $A \rightarrow cB \mid c \mid \in$, we can construct from it a right-linear grammar $\hat{G}$ by replacing every production of G of the form $A \rightarrow cB$ with $A \rightarrow Bc$

It is easy to prove that $L\left(G\right) = \left(L\left(\hat{G}\right)\right)^R$. Since $\hat{G}$ is right-linear, $L\left(\hat{G}\right)$ is regular. But then so are $\left(L\left(\hat{G}\right)\right)^R$ i.e. $L(G)$ because regular languages are closed under reversal.

Putting the two lemmas and the discussions in the above paragraph together we get the proof of the theorem-

A language L is regular iff it has a regular grammar

Example : Consider the grammar

$$G: S \rightarrow 0A \mid 0$$
$$A \rightarrow 1S$$

It is easy to see that G generates the language denoted by the regular expression (01)*0.
The construction of lemma 2 for this grammar produces the follwoing FA.
This FA accepts exactly (01)*1.

Decisions Algorithms for CFL

In this section, we examine some questions about CFLs we can answer. A CFL may be represented using a CFG or PDA. But an algorithm that uses one representation can be made to work for the others, since we can construct one from the other.

Testing Emptiness :

Theorem : There are algorithms to test emptiness of a CFL.

Proof : Given any CFL L, there is a CFG G to generate it. We can determine, using the construction described in the context of elimination of useless symbols, whether the start symbol is useless. If so, then $L(G) = \phi$; otherwise not.

Testing Membership :

Given a CFL L and a string x, the membership, problem is to determine whether $x \in L$?

Given a PDA P for L, simulating the PDA on input string x doesnot quite work, because the PDA can grow its stack indefinitely on $\in$ input, and the process may never terminate, even if the PDA is deterministic.

So, we assume that a CFG $G = (N, \Sigma, P, S)$ is given such that $L = L(G)$.

Let us first present a simple but inefficient algorithm.

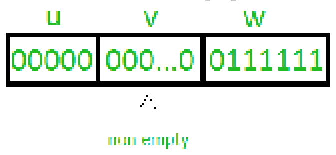
Convert G to $G' = (N', \Sigma', P', S')$ in CNF generating $L(G) - \{\in\}$. If the input string $x = \in$, then we need to determine whether $S \overset{*}{\underset{G}{\Rightarrow}} \in$ and it can easily be done using the technique given in the context of elimination of $\in$-production. If , $x \neq \in$ then $x \in L(G')$ iff $x \in L(G)$. Consider a derivation under a grammar in CNF. At every step, a production in CNF in used, and hence it adds exactly one terminal symbol to the sentential form. Hence, if the length of the input string x is n, then it takes exactly n steps to derive x (provided x is in $L(G')$).

Let the maximum number of productions for any nonterminal in G' is K. So at every step in derivation, there are atmost k choices. We may try out all these choices, systematically., to derive the string x in G'. Since there are atmost $K^{|x|}$ i.e. K^n choices. This algorithms is of exponential time complexity. We now present an efficient (polynomial time) membership algorithm.

Pumping Lemma:

Pumping Lemma in Theory of Computation There are two Pumping Lemmas, which are defined for 1. Regular Languages, and 2. Context – Free Languages Pumping Lemma for Regular Languages Theorem Let L be a regular language. Then there exists a constant ‘c’ such that for every string w in L − |w| ≥ c We can break w into three strings, w = xyz, such that − • |y| > 0 • |xy| ≤ c • For all k ≥ 0, the string xykz is also in L. In simple terms, this means that if a string y is ‘pumped’, i.e., if y is inserted any number of times, the resultant string still remains in L. Pumping Lemma is used as a proof for irregularity of a language. Thus, if a language is regular, it always satisfies pumping lemma. If there exists at least one string made from pumping which is not in L, then L is surely not regular. The opposite of this may not always be true. That is, if Pumping Lemma holds, it does not mean that the language is regular. Applications of Pumping Lemma Pumping Lemma is to be applied to show that certain languages are not regular. It should never be used to show a language is regular. • If L is regular, it satisfies Pumping Lemma. • If L does not satisfy Pumping Lemma, it is non-regular.

Method to prove that a language L is not regular • At first, we have to assume that L is regular. • So, the pumping lemma should hold for L. • Use the pumping lemma to obtain a contradiction − o Select w such that |w| ≥ c q0 q qf 1 x y z o Select y such that |y| ≥ 1 o Select x such that |xy| ≤ c o Assign the remaining string to z. o Select k such that the resulting string is not in L. For example, let us prove L01 = {0n1n | n ≥ 0} is irregular. Let us assume that L is regular, then by Pumping Lemma the above given rules follow. Now, let x ∈ L and |x| ≥ n. So, by Pumping Lemma, there exists u, v, w such that (1) – (3) hold. We show that for all u, v, w, (1) – (3) does not hold. If (1) and (2) hold then x = 0n1n = uvw with |uv| ≤ n and |v| ≥ 1. So, u = 0a , v = 0b , w = 0c1n where : a + b ≤ n, b ≥ 1, c ≥ 0, a + b + c = n But, then (3) fails for i = 0 uv0w = uw = 0a0c1n = 0a + c1n ∉ L, since a + c ≠ n



Limitations of Finite Automata and Non regular Languages :

The class of languages recognized by FA s is strictly the regular set. There are certain languages which are non regular i.e. cannot be recognized by any FA

Consider the language $L = \{a^n b^n \mid n \geq 0\}$

In order to accept is language, we find that, an automaton seems to need to remember when passing the center point between a's and b's how many a's it has seen so far. Because it would have to compare that with the number of b's to either accept (when the two numbers are same) or reject (when they are not same) the input string.

But the number of a's is not limited and may be much larger than the number of states since the string may be arbitrarily long. So, the amount of information the automaton need to remember is unbounded.

A finite automaton cannot remember this with only finite memory (i.e. finite number of states). The fact that FA s have finite memory imposes some limitations on the structure of the languages recognized. Inductively, we can say that a language is regular only if in processing any string in this language, the information that has to be remembered at any point is strictly limited. The argument given above to show that $a^n b^n$ is non regular is informal. We now present a formal method for showing that certain languages such as $a^n b^n$ are non regular

Properties of CFL's

Closure properties of CFL:

We consider some important closure properties of CFLs.

Theorem : If L_1 and L_2 are CFLs then so is $L_1 \cup L_2$

Proof : Let $G_1 = (N_1, \Sigma_1, P_1, S_1)$ and $G_2 = (N_2, \Sigma_2, P_2, S_2)$ be CFGs generating. Without loss of generality, we can assume that $N_1 \cap N_2 = \phi$. Let S_3 is a nonterminal not in N_1 or N_1. We construct the grammar $G_3 = (N_3, \Sigma_3, P_3, S_3)$ from G_1 and G_2, where

$N_3 = N_1 \cup N_2 \cup \{S_3\}$,

$\Sigma_3 = \Sigma_1 \cup \Sigma_2$

$P_3 = P_1 \cup P_2 \cup \{S_3 \to S_1 \mid S_2\}$

We now show that $L(G_3) = L(G_1) \cup L(G_2) = L_1 \cup L_2$

Thus proving the theorem.

Let $w \in L_1$. Then $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} w$. All productions applied in their derivation are also in G_2. Hence $S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_1 \overset{*}{\underset{G_3}{\Rightarrow}} w$ i.e. $w \in L(G_3)$

Similarly, if $w \in L_2$, then $w \in L(G_3)$

Thus $L_1 \cup L_2 \subseteq L(G_3)$.

Conversely, let $w \in L(G_3)$. Then $S_3 \overset{*}{\underset{G_3}{\Rightarrow}} w$ and the first step in this derivation must be either $S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_1$ or $S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_2$. Considering the former case, we have $S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_1 \overset{*}{\underset{G_3}{\Rightarrow}} w$

Since N_1 and N_1 are disjoint, the derivation $S_1 \overset{*}{\underset{G_3}{\Rightarrow}} w$ must use the productions of P_1 only (which are also in P_3) Since $S_1 \in N_1$ is the start symbol of G_1. Hence, $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} w$ giving $w \in L(G_1)$.

Using similar reasoning, in the latter case, we get $w \in L(G_2)$. Thus $L(G_3) \subseteq L_1 \cup L_2$.

So, $L(G_3) = L_1 \cup L_2$, as claimed

Theorem : If L_1 and L_2 are CFLs, then so is $L_1 L_2$.

Proof : Let $G_1 = (N_1, \Sigma_1, P_1, S_1)$ and $G_2 = (N_2, \Sigma_2, P_2, S_2)$ be the CFGs generating L_1 and L_2 respectively. Again, we assume that N_1 and N_1 are disjoint, and S_3 is a nonterminal not in N_1 or N_1. we construct the CFG $G_3 = (N_3, \Sigma_3, P_3, S_3)$ from G_1 and G_2, where

$N_3 = N_1 \cup N_2 \cup \{S_3\}$

$\Sigma_3 = \Sigma_1 \cup \Sigma_2$

$P_3 = P_1 \cup P_2 \cup \{S_3 \to S_1 S_2\}$

We claim that $L(G_3) = L(G_1) L(G_2) = L_1 L_2$

To prove it, we first assume that $x \in L_1$ and $y \in L_2$. Then $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} x$ and $S_2 \overset{*}{\underset{G_2}{\Rightarrow}} y$. We can derive the string *xy* in G_2 as shown below.

$$S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_1 S_2 \overset{*}{\underset{G_3}{\Rightarrow}} x S_2 \overset{*}{\underset{G_3}{\Rightarrow}} xy$$

since $P_1 \subseteq P$ and $P_2 \subseteq P$. Hence $L_1 L_2 \subseteq L(G_3)$.

For the converse, let $w \in L(G_3)$. Then the derivation of *w* in G_3 will be of the form

$S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_1S_2 \overset{*}{\underset{G_3}{\Rightarrow}} w$ i.e. the first step in the derivation must see the rule $S_3 \rightarrow S_1S_2$. Again, since N_1 and N_1 are disjoint and $S_1 \in N_1$ and $S_2 \in N_2$, some string *x* will be generated from S_1 using productions in P_1 (which are also in P_3) and such that $xy = w$.

Thus $S_3 \Rightarrow S_1S_2 \Rightarrow xS_2 \Rightarrow xy = w$

Hence $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} x$ and $S_2 \overset{*}{\underset{G_2}{\Rightarrow}} y$.

This means that *w* can be divided into two parts *x*, *y* such that $x \in L_1$ and $y \in L_2$. Thus $w \in L_1L_2$ .This completes the proof

Theorem : If L is a CFL, then so is L^*.

Proof : Let $G = (N, \Sigma, P, S)$ be the CFG generating L. Let us construct the CFG $G' = (N, \Sigma, P', S)$ from G where $P' = P \cup \{S \rightarrow SS | \in\}$.

We now prove that $L(G') = (L(G))^* = L^*$, which prove the theorem.

G' can generate $\in$ in one step by using the production $S \rightarrow \in$ since $P \subseteq P'$, G' can generate any string in L.

Let $w \in L^n$ for any n >1 we can write $w = w_1w_2 \cdots w_n$ where $w_i \in L$ for $1 \le i \le n$. *w* can be generated by G' using following steps.

$$S \overset{n-1}{\underset{G}{\Rightarrow}} SS \cdots S \overset{*}{\underset{G}{\Rightarrow}} w_1ss \cdots s \overset{*}{\underset{G}{\Rightarrow}} w_1w_2ss \cdots s \overset{*}{\underset{G}{\Rightarrow}} w_1w_2 \cdots w_n = w$$

First $(n-1)$-steps uses the production $S \rightarrow SS$ producing the sentential form of n numbers of S 's. The nonterminal S in the i-th position then generates w_i using production in P (which are also in P')

It is also easy to see that G can generate the empty string, any string in L and any string $w \in L^n$ for *n* >1 and none other.

Hence $L(G') = (L(G))^* = L^*$

Theorem : CFLs are not closed under intersection

Proof : We prove it by giving a counter example. Consider the language $L_1 = \{a^ib^ic^j \mid i, j \ge 0\}$ .The following CFG generates L_1 and hence a CFL

$S \rightarrow XC$
$X \rightarrow aXb \mid \in$
$C \rightarrow cC \mid \in$

The nonterminal X generates strings of the form $a^n b^n,\ n \geq 0$ and C generates strings of the form c^m , $m \geq 0$. These are the only types of strings generated by X and C. Hence, S generates L_1.

Using similar reasoning, it can be shown that the following grammar $L_2 = \{a^i b^j c^j \mid i, j \geq 0\}$ and hence it is also a CFL.

$S \rightarrow AX$
$A \rightarrow aA \mid \in$
$X \rightarrow bXc \mid \in$

But, $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$ and is already shown to be not context-free.

Hence proof.

Theorem : A CFL's are not closed under complementations

Proof : Assume, for contradiction, that CFL's are closed under complementation. SInce, CFL's are also closed under union, the language $\overline{\overline{L_1} \cup \overline{L_2}}$, where L_1 and L_2 are CFL's must be CFL. But by DeMorgan's law

$\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$

This contradicts the already proved fact that CFL's are not closed under intersection.

But it can be shown that the CFL's are closed under intersection with a regular set.

Theorem : If L is a CFL and R is a regular language, then $L \cap R$ is a CFL.

Proof : Let $P = (Q_p, \Sigma, \Gamma, \delta_p, q_p, z_0, F_p)$ be a PDA for L and let $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$ be a DFA for R.

We construct a PDA M from P and D as follows

$M = (Q_p \times Q_D, \Sigma, \Gamma, \delta_M, (q_p, q_D), z_0, F_p \times F_D)$

where δ_M is defined as

$\delta_M((p,q), a, X)$ contains $((r,s), \alpha)$ iff

$\delta_D(q,a) = s$ and $\delta_p(p,a,X)$ contains (r,α)

The idea is that M simulates the moves of P and D parallely on input w, and accepts w iff both P and D accepts. That means, we want to show that

$$L(M) = L(P) \cap L(D) = L \cap R$$

We apply induction on n, the number of moves, to show that

$$((q_p, q_D), w, z_0) \overset{n}{\underset{M}{\mapsto}} ((p,q), \in, \gamma) \text{ iff}$$

$$(q_p, w, z_0) \overset{n}{\underset{P}{\mapsto}} (p, \in, \gamma) \text{ and } \hat{\delta}(q_D, w) = q$$

Basic Case is n=0. Hence $p = q_p$, $q = q_D$, $\gamma = z_0$ and $w = \in$. For this case it is trivially true

Inductive hypothesis : Assume that the statement is true for n -1.

Inductive Step : Let $w = xa$ and

Let $((q_p, q_D), x_a, z_0) \overset{n-1}{\underset{M}{\mapsto}} ((p', q'), a, \alpha) \overset{1}{\underset{M}{\mapsto}} ((p,q), \in, \gamma)$

By inductive hypothesis, $(q_p, x, z_0) \overset{n-1}{\underset{P}{\mapsto}} (p', \in, \alpha)$ and $\hat{\delta}_D(q_D, x) = q'$

From the definition of δ_M and considering the n-th move of the PDA M above, we have

$$\delta_p(p', a, \alpha) = (p, \in, \gamma) \text{ and } \delta_D(q', a) = q$$

Hence $(q_p, xa, z_0) \overset{n-1}{\underset{p}{\mapsto}} (p', a, \alpha) \overset{1}{\underset{p}{\mapsto}} (p, \in, \gamma)$ and $\hat{\delta}_D(q_D, w) = q$

If $p \in F_p$ and $q \in F_D$, then $p, q \in F_p \times F_D$ and we got that if M accepts w, then both P and D accepts it.
We can show that converse, in a similar way. Hence $L \cap R$ is a CFL (since it is accepted by a PDA M)
This property is useful in showing that certain languages are not context-free.
Example : Consider the language

$$L = \{ w \in \{a,b,c\}^* \mid w \text{ contains equal number of } a\text{'s}, b\text{'s and } c\text{'s} \}$$

Intersecting L with the regular set $R = a^* b^* c^*$, we get

$$L \cap R = L \cap a^* b^* c^*$$
$$= \{ a^n b^n c^n \mid n \geq 0 \}$$
Which is already known to be not context-free. Hence L is not context-free

Theorem : CFL's are closed under reversal. That is if L is a CFL, then so is L^R

Proof : Let the CFG $G = (N, \Sigma, P, S)$ generates L. We construct a CFG $G' = (N, \Sigma, P', S)$ where $P = \{ A \rightarrow \alpha \mid A \rightarrow \alpha^R \in P \}$. We now show that $L(G') = L^R$, thus proving the theorem.
We need to prove that

$A \overset{n}{\underset{G}{\Rightarrow}} \alpha$ iff $A \overset{n}{\underset{G'}{\Rightarrow}} \alpha^R$.

The proof is by induction on n, the number of steps taken by the derivation. We assume, for simplicity (and of course without loss of generality), that G and hence G' are in CNF.
The basis is n=1 in which case it is trivial. Because α must be either $a \in \Sigma$ or BC with $B, C \in N$.

Hence $A \overset{1}{\underset{G}{\Rightarrow}} a$ iff $A \overset{1}{\underset{G'}{\Rightarrow}} a$

Assume that it is true for (n-1)-steps. Let $A \overset{n}{\underset{G}{\Rightarrow}} \alpha$. Then the first step must apply a rule of the form $A \rightarrow BC$ and it gives

$A \overset{1}{\underset{G}{\Rightarrow}} BC \overset{n-1}{\underset{G}{\Rightarrow}} \beta\gamma = \alpha$ where $B \overset{*}{\underset{G'}{\Rightarrow}} \beta^R$ and $C \overset{*}{\underset{G'}{\Rightarrow}} \gamma^R$

By constructing of G', $A \rightarrow CB \in P'$
Hence

$A \overset{1}{\underset{G'}{\Rightarrow}} CB \overset{n-1}{\underset{G'}{\Rightarrow}} \gamma^R \beta^R = \alpha^R$

The converse case is exactly similar

Substitution :

$\forall a \in \Sigma$, let L_a be a language (over any alphabet). This defines a function S, called substitution, on Σ which is denoted as $s(a) = L_a$ - for all $a \in \Sigma$
This definition of substitution can be extended further to apply strings and langauge as well.
If $w = a_1 a_2 \cdots a_n$, where $a_i \in \Sigma$, is a string in Σ^*, then
$s(w) = s(a_1 a_2 \cdots a_n) = s(a_1) s(a_2) \cdots s(a_n)$.
Similarly, for any language L,
$s(L) = \{ s(w) \mid w \in L \}$
The following theorem shows that CFLs are closed under substitution.

Thereom : Let $L \subseteq \Sigma^*$ is a CFL, and s is a substitution on Σ such that $s(a) = L_a$ is a CFL for all $a \in \Sigma$, thus $s(L)$ is a CFL

Proof : Let $L = L(G)$ for a CFG $G = (N, \Sigma, P, S)$ and for every $a \in \Sigma$, $L_a = L(G_a)$ for some $G_a = (N_a, \Sigma_a, P_a, S_a)$. Without loss of generality, assume that the sets of nonterminals N and N_a's are disjoint.

Now, we construct a grammar G', generating $s(L)$, from G and G_a's as follows **:**

- $G' = (N', \Sigma', P', S)$
- $N' = N \cup \bigcup_{a_i \in \Sigma} N_{a_i}$
- $\Sigma' = \bigcup_{a_i \in \Sigma} \Sigma_{a_i}$
- P' consists of
  1. $\bigcup_{a_i \in \Sigma} P_{a_i}$ and
  2. The production of P but with each terminal a in the right hand side of a production replaced by S_a everywhere.

We now want to prove that this construction works i.e. $w \in L(G')$ iff $w \in s(L)$.

If Part : Let $w \in s(L)$ then according to the definition there is some string $x = a_1 a_2 \cdots a_n \in L$ and $x_i \in S(a_i)$ for $i = 1, 2, \cdots, n$ such that $w = x_1 x_2 \cdots x_n \left(= s(a_1) s(a_2) \cdots s(a_n)\right)$

We will show that $S \overset{*}{\underset{G}{\Rightarrow}} w$.

From the construction of G', we find that, there is a derivation $S \overset{*}{\underset{G}{\Rightarrow}} S_{a_1} S_{a_2} \cdots S_{a_n}$ corresponding to the string $x = a_1 a_2 \cdots a_n$ (since G' contains all productions of G but every ai replaced with S_{a_i} in the RHS of any production).

Every S_{a_i} is the start symbol of G_{a_i} and all productions of G_{a_i} are also included in G'.
Hence

$$S \overset{*}{\underset{G}{\Rightarrow}} S_{a_1} S_{a_2} \cdots S_{a_n}$$
$$\overset{*}{\underset{G}{\Rightarrow}} x_1 S_{a_2} \cdots S_{a_n}$$
$$\overset{*}{\Rightarrow} x_1 x_2 \cdots x_n = w$$

Therefore, $w \in L(G')$

(Only-if Part) Let $w \in L(G')$. Then there must be a derivative as follows :

$S \overset{*}{\underset{G}{\Rightarrow}} S_{a_1} S_{a_2} \cdots S_{a_n}$ (using the production of G include in G' as modified by (step 2) of the construction of P'.)

Each S_{a_i} ($i = 1, 2, \cdots, n$) can only generate a string $x_i \in L_{a_i}$, since each N_{a_i}'s and N are disjoin. Therefore, we get

$$S \overset{*}{\underset{G}{\Rightarrow}} S_{a_1} S_{a_2} \cdots S_{a_n}$$

$\overset{*}{\underset{G}{\Rightarrow}} x_1 S_{a_2} \cdots S_{a_n}$ since $S_{a_1} \overset{*}{\underset{G_{a_1}}{\Rightarrow}} x_1$

$\overset{*}{\underset{G}{\Rightarrow}} x_1 x_2 S_{a_3} \cdots S_{a_n}$ since $S_{a_2} \overset{*}{\underset{G_{a_2}}{\Rightarrow}} x_2$

$\overset{*}{\underset{G}{\Rightarrow}} x_1 x_2 \cdots x_n$

$= w$

The string $w = x_1 x_2 \cdots x_n$ is formed by substituting strings x_i for each a_i's and hence $w \in s(L)$.

Theorem : CFL's are closed under homomorphism

Proof : Let $L \subseteq \Sigma^*$ be a CFL, and *h* is a homomorphism on Σ i.e $h : \Sigma \rightarrow \Delta^*$ for some alphabets Δ. consider the following substitution S:Replace each symbol $a \in \Sigma$ by the language consisting of the only string h(a), i.e. $s(a) = \{h(a)\}$ for all $a \in \Sigma$. Then, it is clear that, $h(L) = s(L)$. Hence, CFL's being closed under substitution must also be closed under homomorphism.